Zero-Knowledge Proof-Enabled SAT Co-processor for Blockchain Systems

by

Vladyslav Yusiuk

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Sciences
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2025

©Vladyslav Yusiuk 2025

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

This thesis explores the possibility of building classical SAT solvers in Circom Domain Specific Language to create zero-knowledge proofs (ZKPs) usable in blockchain contexts. I implemented DPLL and Chaff as arithmetic circuits within Circom and analyze them based on constraint count, proving delay, and zk-SNARK verification layers. With this evaluation, the aim is to determine the feasibility of solvers' integration into off-chain computation systems and rollup-centric architectures on Ethereum.

The findings indicate that incorporating SAT solvers within zero-knowledge circuits is achievable though some degradation in efficiency occurs based on algorithm used and input representation. This research provides a thorough assessment of known SAT methods across an unconventional boundary, linking symbolic logic with blockchain technologies reliant on zk-SNARKs.

Acknowledgements

I would like to thank my supervisor, Derek Rayside, for supporting and advising me during the last challenging two years. I would also like to thank Jan Gorzny and Martin Derka for providing an idea for this work, supervising it, and providing advice for enhancing it.

Dedication

First of all, I would like to thank my family and friends, who are still in Ukraine. I dedicate this work to my family and friends in Ukraine and to Viktoriia, whose support made this journey possible. And finally, I thank myself for setting challenges and pushing myself until I achieved results. Everyone who believes in me. Thank you.

Table of Contents

Author's Declaration ii

Abstract iii

Acknowledgements iv

Dedication v

List of Figures ix

List of Tables x

List of Acronyms xi

1 Introduction 1

1.1 Background and Motivation 1

1.1.1 Importance of Blockchain Technology 1

1.1.2 Early Adoption and Expansion 2

1.1.3 Key Developments and Innovations 2

1.1.4 Blockchain Beyond Cryptocurrency 3

1.1.5 Recent Trends and Future Directions 4

1.2 Introduction to Zero-Knowledge Proofs (ZKPs) 5

1.2.1 Privacy and Security in Blockchain Applications Using Zero Knowledge Proofs 6

1.2.2 Zero-Knowledge Proof Can Be Expressed Formally as Follows: 7
1.2.3 Schnorr Protocol for Proving Knowledge of a Discrete Logarithm: 8
1.2.4 Relevance in Smart Contracts and Blockchain Verification Processes . . . 9
1.3 Significance of SAT Solvers in Cryptographic Applications . . . 9
1.3.1 SAT Solvers in Cryptographic Applications and Blockchain Technology . . . 10
1.3.2 Approaches to Realizing SAT Solvers with Algorithms Like DPLL and Chaff . . . 11
1.4 Problem Statement . . . 14
1.4.1 Efficiency and Scalability . . . 14
1.4.2 Security and requirements for rollup usage . . . 15
1.4.3 Pros of using off-chain computations . . . 16
1.5 Objectives . . . 16
1.6 Scope of the Study . . . 17

2 Literature Review **19**
2.1 Cryptographic Primitives Relevant to SAT/ZK . . . 19
2.2 ED25519 and Its ZK-Friendly Properties . . . 20
2.2.1 Elliptic Curve Equation . . . 21
2.2.2 Scalar Multiplication . . . 21
2.2.3 Signature Generation . . . 22
2.2.4 Signature Verification . . . 23
2.3 Blockchain Consensus Mechanisms and Scalability . . . 25
2.4 Zero-Knowledge Proofs (ZKP) and Privacy-Preserving Technologies . . . 27
2.5 SAT Solving in Blockchain and Cryptographic Applications . . . 30

3 Methodology **33**
3.1 CIRCOM's DPLL Algorithm benchmark . . . 34
3.1.1 Example Application . . . 37
3.1.2 Challenges in implementation . . . 45

3.2 Chaff Solver 48
3.2.1 Overview Of Code Structure and Functionality 51
3.3 Parameter sensitivity 55
3.4 Limitations of a Pure-Circom Approach and Motivation for a Hybrid Model 57
3.5 Hybrid Rust - SAT - Circom Pipeline 59
3.5.1 Motivation 59
3.5.2 Security Guarantees of the Hybrid Model 60
3.5.3 End-to-End Workflow 61
3.5.4 Circuit Design 61
3.5.5 Results 65
3.6 Summary of Chapter 3 67

4 Conclusions and future work **69**

Bibliography **71**

APPENDICES **75**

A Code for Chaff and DPLL Solvers **76**
A.1 DPLL SAT Solver Code 76
A.2 Chaff SAT Circom Solver Code 84

List of Figures

2.1 Merkle tree structure. 24
2.2 zk-SNARK diagram, Key, Proof, and Verification steps. 29

3.1 An illustration of unit propagation. 35
3.2 An illustration of pure literal elimination. 36
3.3 Scaling of circuit size 44
3.4 Comparison of DPLL and Chaff performance 48
3.5 Chaff compilation results 54
3.6 Complexity Growth in Chaff Circuits 58

List of Tables

3.1 Comparison of Different Circuit Metrics with Increasing Literals and Clauses 40
3.2 Comparison of Circuit Metrics Across Literals and Clauses for Chaff Algorithm 56
3.3 End-to-end workflow for Merkle tree SAT-to-ZK pipeline 62
3.4 Key building blocks of the `RootCheck` circuit 65
3.5 Performance results across different Merkle tree sizes 65
3.6 Comparison of different SAT proof approaches on constraint size, proving time, and gas cost 66

List of Acronyms

- **SAT** - Boolean Satisfiability
- **DPLL** - Davis-Putnam-Logemann-Loveland (Algorithm)
- **CDCL** - Conflict-Driven Clause Learning
- **ZKP** - Zero-Knowledge Proof
- **zk-SNARK** - Zero-Knowledge Succinct Non-Interactive Argument of Knowledge
- **zk-STARK** - Zero-Knowledge Scalable Transparent Argument of Knowledge
- **R1CS** - Rank-1 Constraint System
- **ECC** - Elliptic Curve Cryptography
- **PoW** - Proof of Work
- **PoS** - Proof of Stake
- **PBFT** - Practical Byzantine Fault Tolerance
- **L1** - Layer 1 (Refers to main blockchain layers like Ethereum)
- **L2** - Layer 2 (Refers to scaling solutions for blockchains - rollups)
- **DeFi** - Decentralized Finance
- **zk-Rollup** - Zero-Knowledge Rollup
- **DSL** - Domain Specific Language
- **SNARKJS** - JavaScript library for zk-SNARKs
- **VSIDS** - Variable State Independent Decaying Sum (heuristic)

- **ECDSA** - Elliptic Curve Digital Signature Algorithm
- **CNF** - Conjunctive Normal Form
- **EVM** - Ethereum Virtual Machine
- **SHA** - Secure Hash Algorithm
- **DApp** - Decentralized Application
- **DAO** - Decentralized Autonomous Organization
- **BCP** - Boolean Constraint Propagation

Chapter 1

Introduction

1.1 Background and Motivation

1.1.1 Importance of Blockchain Technology

Blockchain technology was formalized in Nakamoto's 2008 white paper, "Bitcoin: A Peer-to-Peer Electronic Cash System."[3] , which proposed a peer-to-peer digital cash system that removes reliance on intermediaries for a central authority, such as a bank or government.

The blockchain, in this context, was designed as an immutable public ledger that records all Bitcoin transactions. The defining characteristics of blockchain systems include the following:

- **Decentralization:** Rather than a single, centralized authority, the network depends on distributed nodes. In early blockchains such as Bitcoin, consensus was achieved through a proof-of-work (POW) mechanism [7], where miners expended computational effort to validate transactions and secure the network. In proof-of-stake (PoS) mechanisms [6], validators replace miners and participate in consensus proportional to their stake within the network.

- **Immutability:** The process of appending is confirmation of a block of transactions which cannot be changed unless the whole chain is redone. The probabil-

ity of an honest chain being overtaken by an attacker drops below 2^{-128} after $\approx$ 60 Bitcoin block confirmations. This process is made secure through cryptographic hashes. A case in point is the 2016 DAO Ethereum vulnerability, which resulted in the Ethereum and Ethereum Classic split. [24]

- **Transparency:** Every transaction is publicly visible to anyone who has access to the blockchain, providing visibility and auditability to all participants.
- **Security:** Cryptographic primitives such as SHA-256 (for hashing) and elliptic-curve signatures (e.g. Ed25519 or secp256k1) guarantee data integrity and authentication.[1, 2]

1.1.2 Early Adoption and Expansion

Following Bitcoin's launch, blockchain technology quickly garnered attention for its potential applications beyond digital currency. Early adopters and developers began to explore how blockchain could be used in various fields, including finance, supply chain management, healthcare [4]. These ideas have been circulating for nearly a decade, but large-scale practical deployment has remained limited.

Vitalik Buterin's introduction of Ethereum in 2015 marked a critical milestone in the blockchain industry, as it introduced smart contracts, self-executing scripts triggered by on-chain events, thereby expanding blockchain functionality [5]. With this innovation, Decentralized Applications (DApps) could be developed and an entire ecosystem forged to allow programmatic governance over rules for issuing tokens, verifying identities, conducting auctions, among other processes.

1.1.3 Key Developments and Innovations

The launching of Ethereum smart contracts catalyzed innovation towards designing decentralized systems. With these contracts, developers were able to program sophisticated conditions such as token minting, permissioning, and exchanges which are automated.

The original Bitcoin blockchain used the Proof of Work (PoW) consensus mechanism [7], which required significant computational power - originally GPU-based, now dominated by ASIC hardware. To address the limitations of Proof of Work, alternative consensus approaches were explored. Proof of Stake (PoS) [6] was introduced to reduce energy consumption by selecting validators proportional to their stake. Practical Byzantine Fault Tolerance (PBFT), originally developed in the broader field of distributed systems well before the emergence of blockchain [8], was later adopted for blockchain contexts. These mechanisms aimed to improve the scalability, efficiency, and security of blockchain networks with the main concept of having validators for reaching consensus [8].

The increasing volume of transactions and the demand from users made scalability a major problem to address. Shifting off-chain computation while preserving the security guarantees of Layer 1 (L1) is what the Lightning Network for Bitcoin and Plasma for Ethereum attempt with their Layer 2 (L2) scaling solutions [9]. Layer 1 refers to the base blockchain protocol itself, such as Bitcoin [3] or Ethereum [5], which provides consensus, security, and transaction finality. Layer 2 protocols are built on top of L1 and aim to improve scalability by moving computation off-chain while still relying on the L1 chain for final verification and security [9].

At the same time, Zero-Knowledge Proofs (ZKPs), first introduced in the 1980s by Goldwasser, Micali, and Rackoff [29], are becoming popular for privacy reasons in blockchain context. More recent constructions, such as zk-SNARKs (Succinct Non-Interactive Arguments of Knowledge), allow data to be validated without disclosing sensitive information[10].

1.1.4 Blockchain Beyond Cryptocurrency

As blockchain adoption expanded, researchers and developers proposed applications outside of digital currency. These proposals emphasize transparency, auditability, and tamper-resistance, but most remain at the stage of pilot projects or limited deployments. Key examples include:

- Supply-chain tracking: Companies experiment with blockchain systems to record the flow of goods from production to delivery, aiming to improve traceability and reduce fraud.

- Healthcare and data management: Healthcare providers and researchers explore blockchain to secure patient records and facilitate data sharing between institutions. In principle, developers can apply Zero-Knowledge Proofs to prove properties of medical data without exposing sensitive details

- Blockchain-based voting: Policy makers and engineers often cite voting as a use case, emphasizing potential transparency and auditability. However, unresolved challenges such as voter coercion, system integrity, and usability have prevented real-world deployment.

1.1.5 Recent Trends and Future Directions

Nowadays, blockchain technology has evolved significantly with the introduction of advanced platforms and protocols aimed at addressing scalability, privacy, and efficiency challenges. Some of the notable trends and future directions includes working with scalability and efficiency of the Ethereum L1 network [5]:

- Zero-Knowledge Virtual Machines or zkVMs: virtual machines can enable the execution of smart contracts with enhanced privacy and scalability by allowing computations to be verified without revealing underlying data. This innovation is crucial for modern privacy-preserving decentralized applications [10].

- Creation of ZK-SNARKs: Circom / Halo 2 are DSLs for zk circuits, which are widely adopted for building a modern privacy for decentralized applications, allowing developers to construct custom ZKP circuits for specific use cases, such as verifying computations without revealing sensitive data. [23]

- Layer 2 Rollups: Layer 2 solutions, such as Optimistic Rollups and zk-Rollups, have emerged as key methods to scale blockchain networks. These rollups

process transactions off-chain, reducing congestion and gas fees while still having benefits from the security of the underlying Layer 1 blockchain - mostly Ethereum blockchain. Optimistic rollups add a 7-14 day fraud-challenge window, whereas zk-rollups shift heavy proving costs to specialised off-chain provers.

- Cross-Chain Interoperability: The main focus is wrapping different blockchain networks to communicate similarly. For instance, Polkadot and Cosmos are developing efforts to achieve cross-chain interoperability, allowing the transfer of assets (like Wrapped Ethereum (WETH)) and data across multiple blockchains, for example, freezing tokens in one network, oracle signals second network for minting and releasing tokens in the same amount to the requester - user.

- Decentralized Finance (DeFi) 2.0: Protocols such as Olympus v2 experiment with protocol-owned liquidity and reflexive bonding curves to reduce the reliance on transient LP incentives.

1.2 Introduction to Zero-Knowledge Proofs (ZKPs)

Zero-Knowledge Proofs (ZKPs), originally introduced in the 1980s [29], are cryptographic protocols that enable one party, the prover, to convince another party, the verifier, that a given statement is true or satisfied, without revealing any additional information. Modern constructions, such as zk-SNARK, [11, 10], extend these ideas to provide succinct and efficient proofs that can be applied in blockchain systems. A Zero-Knowledge Proof system must satisfy several essential properties:

1. **Completeness:** If the statement is true, an honest prover can convince an honest verifier through multiple rounds of interaction (as in zk-STARKs) or a single round of interaction in case of zk-SNARK usage [10, 13]. This is achieved through the use of a Fiat-Shamir transform, which turns interactive protocols into non-interactive ones. In the next steps the verifier poses queries, and the prover responds using a polynomial commitment to the specified clause.

2. **Soundness:** If the statement is false, a dishonest prover cannot convince an honest verifier that it is true, except with negligible probability. This guarantees the integrity of the proof.

3. **Zero-Knowledge:** If the statement is true, the verifier learns no additional information other than the fact that the statement is true. This means that the verifier does not gain any knowledge about the witness. [10]

4. **Proof Generation:** During the generation of proof, the prover constructs a cryptographic proof asserting that the statement is true. In zk-SNARKs [10], the proof is succinct and can be generated fast, making it practical for usage in such cases as verifying smart contract executions. In contrast, STARKs [13] prioritize scalability, with proof generation being more computationally intensive but offering post-quantum security without requiring a trusted setup.

5. **Proof Verification:** After the proof is generated, the verifier ensures its validity. In both SNARKs [10] and STARKs [13], enable verification in sublinear time, ensuring that even complex computations can be validated efficiently. This process guarantees completeness, soundness, and zero knowledge, allowing the verifier to ensure that the statement is true without revealing sensitive information. [10, 11]

1.2.1 Privacy and Security in Blockchain Applications Using Zero Knowledge Proofs

Zero-Knowledge Proofs (ZKPs) can significantly enhance the privacy and security for blockchain applications by enabling confidential transactions and a secure verification process without exposing sensitive information. Their growing adoption in real-world systems demonstrates their practicality and increasing relevance.

1. **Confidential Transactions:** Privacy-focused cryptocurrencies such as Zcash utilize zk-SNARKs to provide privacy-preserving transactions. These proofs

are succinct, with compact size and fast verification, making them efficient for real blockchain applications.

2. **Secure Authentication:** ZKPs allow verifiers to confirm a prover's identity or credentials without accessing sensitive information. For example, secure identity verification in systems where users can prove eligibility without revealing sensitive information such as passwords or passport biometric data. [31]

3. **Decentralized Voting Systems:** ZKPs ensure the integrity and confidentiality of votes in decentralized voting systems. A voter can prove they have cast a valid vote without revealing their selection, thus ensuring election confidentiality and resistance to coercion. [10, 11]

Despite these advantages, implementing ZKPs introduces challenges such as increased complexity for designing a circuit, following trusted setup requirements, and computational overhead.

1.2.2 Zero-Knowledge Proof Can Be Expressed Formally as Follows:

- Let P be the prover and V be the verifier.
- Let x be the statement to be proven, and w is the witness known only to P.
- The protocol involves a series of interactions in which P convinces V that $x \in L$ (is true) without revealing w.

The properties can be formally defined as:

- **Completeness:** For every valid (x, w) pair, the verifier V accepts the proof with probability 1.

$$\Pr[V \text{ accepts } x | x \in L \text{ is true}] = 1$$

- **Soundness:** For every invalid x, the verifier rejects with high probability.

$$\Pr[V \text{ accepts } x | x \notin L \text{ is true}] \leq \epsilon$$

- **Zero-Knowledge:** There exists a simulator S that can produce a transcript indistinguishable from the real interaction between P and V without knowing w. [10, 11]

$$\{\text{transcript}(P, V)(x, w)\} \approx \{S(x)\}$$

1.2.3 Schnorr Protocol for Proving Knowledge of a Discrete Logarithm:

- Let G be a cyclic group with generator g and order q.
- The prover P wants to prove knowledge of a secret value x such that $y = g^x \mod q$ without revealing x.

The Schnorr protocol involves the following steps:

1. **Commitment:** Prover selects a random $r \in \{0, \ldots, q-1\}$ and computes $t = g^r \mod q$. P sends t to Verifier.
2. **Challenge:** Verifier sends a random challenge $c \in \{0, \ldots, q-1\}$ to P.
3. **Response:** Prover computes $s = r + cx \mod q$ and sends s to Verifier.
4. **Verification:** Verifier ensures that $g^s \equiv t \cdot y^c$. If the equation holds, the verifier accepts the proof [12].

The intuition behind the Schnorr protocol is that the prover demonstrates knowledge of the discrete logarithm without ever revealing it directly. The random commitment $t = g^r$ hides the secret value x because it depends only on the random r. The verifier then chooses a challenge c, which prevents the prover from precomputing valid answers x. The response $s = r + cx \ mod \ q$ ties the commitment

and the secret together. If the prover did not know x, they could not consistently generate a statement s that satisfies the verification equation $g^s \equiv t \cdot y^c \ (mod\ q)$. At the same time, the protocol does not leak x, because each execution uses fresh randomness r, making s and t look different in every proof. This guarantees both soundness (a dishonest prover cannot convince the verifier without knowing x) and zero-knowledge (the verifier learns nothing new about x itself).

1.2.4 Relevance in Smart Contracts and Blockchain Verification Processes

Zero-knowledge proofs (ZKPs) are an important component of modern smart contract security and blockchain verification processes, enabling secure and private computation. Smart contracts can execute transactions and constrain rules without revealing the underlying data. Zero Knowledge Proofs ensure that the contract conditions are met. Zero-knowledge proofs reduce the amount of data that needs to be stored and processed on the blockchain by shifting heavy computation off-chain, i.e., to external computation systems outside the main blockchain ledger. Off-chain execution allows smart contracts to rely on proofs generated externally (outside of the blockchain network), while only verifying the succinct proof on-chain. This design improves the scalability and efficiency of blockchain networks [32].

For instance, zk-Rollups aggregate multiple transactions into a single succinct proof, often committing their state with a Merkle tree. This approach reduces the load on the main blockchain and lowers transaction latency.

1.3 Significance of SAT Solvers in Cryptographic Applications

A SAT solver is a computational algorithm that determines whether a Boolean logic formula can be satisfied. Its primary task is to decide whether any assignment of truth values can satisfy all clauses of a given formula. If an assignment exists, the

SAT solver outputs it; if one does not exist, it reports unsatisfiability.

Given a Boolean formula F in a Conjunctive Normal Form (CNF), where F is a conjunction of clauses and each clause is a disjunction of literals, the task involves assigning truth values to variables so that the entire formula F evaluates to true.

Mathematically, an SAT solver attempts to find a solution σ (an assignment of truth values to variables) such that:

$$F(\sigma) = \text{true}$$

For example, Boolean formula $F = (x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_3)$. A SAT solver will determine whether there exists an assignment of truth values for x_1, x_2, and x_3 that satisfies F . [18, 19]

1.3.1 SAT Solvers in Cryptographic Applications and Blockchain Technology

SAT solvers play a crucial role in cryptographic applications and blockchain technology due to their ability to solve complex combinatorial problems. Their principal applications include:

1. **Cryptanalysis:** SAT solvers can be applied in cryptanalysis by encoding cryptographic algorithms into SAT instances to detect weaknesses in cryptographic constructions [33].

2. **Design and Verification of Cryptographic Protocols:** SAT solvers assist in verifying that cryptographic protocols satisfy confidentiality and integrity. They have been applied to check the security of digital-signature and authentication schemes. For instance, primitives can be modeled as Boolean formulas so that a SAT solver can detect key recovery vulnerabilities. SAT-based simulations of stream-cipher and hash functions like AES and SHA-1 help to evaluate robustness against such kinds of attacks [34].

3. **Fault Injection and Cryptographic Attacks:** SAT solvers also study the impact of fault injection on the operation of a number of cryptographic systems. The injection of faults may indeed be simulated with the help of a propositional logic solver, as well as missing information or keys and their replicas. Such an approach could be handy in hardware cryptoanalysis, where an error is purposely caused by using external influences [26].

4. **Side-Channel Attack Countermeasures:** Logical solvers can be used to verify cryptographic implementations against side-channel attacks formally. With precompiled side-channel information, such as timing or power consumption, the solver can analyze whether an attack can leverage this information to break the cryptographic system [35].

1.3.2 Approaches to Realizing SAT Solvers with Algorithms Like DPLL and Chaff

Davis-Putnam-Logemann-Loveland (DPLL) Algorithm

The Davis-Putnam-Logemann-Loveland (DPLL) algorithm is a classic backtracking-based search method for solving Boolean Satisfiability (SAT). Most modern SAT solvers build on the DPLL as their foundation framework. The algorithm systematically searches for satisfying assignments to a Boolean formula in Conjunctive Normal Form (CNF). The algorithm simplifies the formula by repeatedly applying unit propagation: If, after repetitions, the clause contains only one literal, it assigns the necessary value to satisfy this clause and simplifies this formula accordingly. DPLL identifies pure literals, which might appear with only one polarity, and assigns them truth values that satisfy all occurrences. After those, literals should be assigned to true or false, respectively, simplifying this formula. This step is called Pure Literal Elimination. The final steps are variable assignment and backtracking, where variables can be selected and assigned into truth values.[25]

Formalization of the DPLL Algorithm:

Let F be a Boolean formula in CNF. The DPLL algorithm attempts to find an assignment σ such that $F(\sigma) = \text{true}$ by recursively applying the following steps:

1. **Unit Propagation:** If F contains a unit clause (l), assign $l = \text{true}$ and simplify F.

2. **Pure Literal Elimination:** If F contains a pure literal l, assign $l = \text{true}$ and simplify F.

3. **Variable Assignment and Recursion:** Select a variable x and recursively apply DPLL to F with $x = \text{true}$ and $x = \text{false}$.

4. **Backtracking:** If a conflict is encountered, backtrack by reversing the last assignment and trying the opposite value. This process repeats until a satisfying assignment is found or the formula is proven unsatisfiable [25].

Example: Consider the formula $F = (x \vee y) \wedge (\neg x \vee z) \wedge (\neg y)$. First, DPLL applies unit propagation to clause $(\neg y)$, assigning $y = \text{false}$. This simplifies $(x \vee y)$ to (x), which forcing $x = \text{true}$. Substituting $x = \text{true}$ into $(\neg x \vee z)$ reduces it to (z), and thus $z = \text{true}$. The resulting assignment $\{x = \text{true}, y = \text{false}, z = \text{true}\}$ satisfies the entire formula.

Chaff Algorithm

The Chaff algorithm [19] builds on the DPLL framework and introduces optimizations for improved performance. Developed at Princeton University, Chaff has become highly influential in practical SAT solving.

Key Features of Chaff

1. **Clause Learning:** The Chaff algorithm uses a conflict-driven clause learning strategy that adds new clauses to prevent the solver from revisiting the same conflicts.

2. **Non-chronological backtracking:** Using non-chronological backtracking allows the algorithm to return directly to the decision level related to the conflict, instead of stepping back one level at a time.

3. **Efficient Data Structures:** The Chaff algorithm can quickly propagate through unit clauses and detect conflicts by employing watched literals. Watched literals can significantly reduce the overhead of unit propagation.

The Chaff algorithm can be described as an extension of DPLL with the following upgrades:

1. **Conflict-Driven Clause Learning (CDCL):** When a conflict is detected, an algorithm analyzes the conflict to identify a conflicting clause. It learns a new clause that prevents the same conflict from occurring again and appends it to the formula.

2. **Non-Chronological Backtracking:** Instead of making a backtrack into the most recent decision level, the algorithm backtracks to the decision level where the conflicting decisions were made.

3. **Watched Literals:** Use two watched literals per clause to track unit propagation efficiently. Only update watched literals when necessary, reducing the computational overhead. [19]

Conflict Example: Suppose during the search the solver assigns $x = \text{true}$, $y = \text{false}$, and $z = \text{false}$. Now the clause $(x \vee y)$ is satisfied, but the clause $(\neg x \vee z)$ becomes $(\neg\text{true} \vee \text{false})$, i.e., $(false)$, which is unsatisfiable. This situation is called a *conflict*. Chaff performs *conflict analysis*, learns a new clause (e.g., $(y \vee z)$) that prevents the same assignment path from recurring, and then uses non-chronological backtracking to jump directly to the decision level that caused the conflict.

1.4 Problem Statement

Integrating an SAT solver into a zero-knowledge proof circuit using Circom and implementing it with algorithms DPLL or Chaff faces several challenges. Circom is a domain-specific language commonly used to compile zero-knowledge proof (ZKP) circuits, and its design imposes constraints on dynamic memory allocation, fixed circuit sizes, and efficiency considerations. These limitations are especially critical for SAT-solver circuits, since DPLL and Chaff both involve recursive branching and large search trees. As a result, circuits modeling these algorithms require more constraints and memory, which encounters Circom's restrictions on dynamic memory and fixed sizes.

1.4.1 Efficiency and Scalability

SAT solvers are computationally intensive, especially those based on DPLL and Chaff algorithms. Depending on the size and number of clauses and literals, SAT algorithms may demand substantial runtime and memory. This computational overhead becomes problematic when SAT solvers are integrated directly into blockchain systems. Because SAT problems are inherently complex, solving large instances requires significant runtime and memory. In blockchain systems like Ethereum, many transactions arrive concurrently, but execution remains sequential. As a result, adding SAT-based computations increases per-transaction cost, lengthens queues, and leads to higher latency.

Realizing these algorithms in Circom, a domain-specific language designed to construct Zero Knowledge circuits, will lead to an additional layer of complexity. The language is optimized for creating and generating succinct, verifiable circuits, but implementing an SAT solver in Circom requires careful consideration of performance and resource constraints.

Blockchain networks, including the most popular L1 solution, Ethereum blockchain, need to handle, prove, and verify a large number of transactions per second. The scalability of blockchain systems is already a critical concern in the current development cycle, and integrating SAT solvers could sharpen this issue.

The main bullet points are:

- **Increased Load:** SAT solvers add CPU load to validator nodes, slowing transaction throughput if executed on-chain. The need for nodes to solve SAT problems as part of transaction validation or smart contract execution can limit the network's ability to scale efficiently; all computations should be off-loaded to external systems (off-chain), or in other words, rollups. Off-loading SAT computation avoids adding extra complexity to the Layer-1 network.
- **Resource Allocation:** The resources required to run SAT solvers may not be uniformly available across all nodes in a decentralized network. This discrepancy can lead to an uneven distribution of computational load, further affecting scalability. SAT workloads can be parallelized or distributed across specialized nodes.
- **Circom Circuit Complexity:** Complex SAT circuits can raise constraint count, which will increase proving time and cost of verification leading to the limitation of scalability.

1.4.2 Security and requirements for rollup usage

Security, scalability, and decentralization comprise the well-known blockchain trilemma. A SAT-based circuit must be integrated carefully, due to introducing additional attack surfaces. The project must be quality-assured and pass integration testing to ensure the integrity of SAT solver operations within a blockchain context.

As this is a major security part of the work, the final ZK proof must guarantee that the SAT solver's output cannot be falsified. This leads to robust verification mechanisms, which adds additional complexity to systems integration.

SAT components must be robust against common threats, and well-known attacks such as DoS (denial-of-service). Roll-up mechanisms are resistant to attacks by moving heavy computation off-chain, yet risks remain.

In the context of this project, the SAT solver is expressed in Circom and secured through zero-knowledge proofs. This project involves correct circuit genera-

tion, proper proof generation, validation, and verification. After these steps, the final proof is submitted and verified on-chain, ensuring that the blockchain itself enforces correctness and prevents any falsification of the SAT solver's result.

1.4.3 Pros of using off-chain computations

With the migration of the SAT solver, which is a computationally intensive task to the off-chain, Layer 1 blockchain can focus on what it was developed for—transactions and consensus, leaving the verification process out of the network at the most computationally intensive stages.

Off-chain computation can still be used to process SAT solver problems in parallel execution, generating proof. Offloading computation is essential for achieving the scalability targets of this thesis work.

As mentioned above, off-chain computation can be designed with enhanced security techniques, which will maintain the correctness of SAT solver-generated proof. After those actions, the final proof can be verified on-chain, with one small proof instead of hundreds of proofs separately.

Implementation of the SAT solver in Circom as a zero-knowledge circuit wraps off-chain efficiency with on-chain verifiability. That approach merges roll-ups for maintaining transparency and decentralization at the same time, reducing transaction costs, and preserving security.

1.5 Objectives

The purpose of this research is to evaluate how the known SAT algorithms (DPLL and Chaff) can be efficiently implemented in Circom and applied within zero-knowledge systems.

1. **Development of a SAT Solver with the Circom DSL:**

   The first and main part is to implement an SAT-solver circuit in Circom. This circuit will generate zero-knowledge proofs that attest to the correctness of the

SAT solver's output. In the standard workflow, the proof is produced off-chain and later verified on-chain through a verifier contract.

2. **Evaluate Its Performance and Feasibility in a Blockchain Environment:**

   When the SAT solver is developed, further steps will involve evaluating its performance with a different numbers of clauses and literals. The evaluation will measure constraint count, proving time, and verification time across varying clause and literal counts. The result will provide insight into the feasibility of integrating such circuits into blockchain-based zero-knowledge proof systems.

1.6 Scope of the Study

The scope of this thesis is limited to implementing and evaluating SAT solvers in Circom and deploying the resulting verifier on the Ethereum network.

At the current scope of development, two algorithms - DPLL and Chaff - were implemented using the Circom Domain Specific Language(DSL). The study examines the use of DSLs such as Circom and related frameworks that are relevant to existing zero-knowledge proof systems (e.g., Halo2). Different approaches, languages, limitations, and advantages of each tool are analyzed to provide a comprehensive understanding of design choices.

A considerable focus of the work is the evaluation of the performance and practical feasibility of the developed SAT solvers. This involves a series of tests measuring efficiency and proof-generation time to determine system effectiveness under different conditions, including security and scalability. Evaluating includes simulating various scenarios for generating, validating, and verifying proofs, followed by deployment on a blockchain network.

To conclude, the objectives and scope of this research work have been formulated in a manner that constructs, tests, and embeds an SAT solver written in Circom language for usage on the Ethereum blockchain and other smart contracts-enabled blockchains. This research seeks to improve the existing body of knowledge and

its practical application, especially SAT solvers, that would help to enhance the efficiency, security, and scalability of blockchain-based zero-knowledge proof systems.

Chapter 2

Literature Review

This chapter surveys prior work and positions the thesis contribution within the broader SAT, ZK-SNARK and blockchain literature.

2.1 Cryptographic Primitives Relevant to SAT/ZK

SHA-256's bit-level structure is particularly relevant because these operations must be arithmetized when expressed in Circom circuits(or any Zero Knowledge circuit framework). An implementation of SHA-256 [1], demonstrates how bit-level semantics translate into arithmetic constraints, which in turn drive circuit size and prover/verifier cost. This example shows that accurate bit-level reasoning is critical when later converting such primitives into ZK circuits.

Key SHA-256 operations, particularly the Ch and Maj functions, are used in round hashing, which are defined as follows:

Ch function

$$Ch(x, y, z) = (x \wedge y) \oplus (\neg x \wedge z)$$

Maj function

$$Maj(x, y, z) = (x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z)$$

The Ch function implements conditional bit selection, while Maj returns the bitwise majority; together they contribute to diffusion and non-linearity in the compression function. In addition, SHA-256 has rotation operations to help with diffusion and patterning, such as

$$\text{RotateRight}(x, n) = (x \gg n) \vee (x \ll (32 - n))$$

In circuit form, bitwise rotations do not admit a fixed one constraint per bit cost. The exact number of constraints depends on the chosen arithmetization such as Boolean gadgets or arithmetic constraints, the handling of XOR/copies, and non-native operations, and is often substantially larger than 32 for a 32-bit rotate. We emphasize constraint here because they directly determine circuit size, prover time, and verification complexity, motivating the later choice of more ZK-amenable primitives. While SHA-256 verification is standardized and robust across platforms, its bitwise nature makes it comparatively costly in ZK circuits relative to algebraic (field-native) hashes. [1]

2.2 ED25519 and Its ZK-Friendly Properties

ED25519 is the EdDSA signature scheme instantiated on the Edwards25519 curve(a twisted Edwards form related to Curve25519). It is widely used in blockchain systems and secure communication protocols due to fast, constant-time implementations and compact encoding(64-byte signatures and 32-byte public keys) [2]. From a ZK perspective, ED25519 is valuable as a benchmark because the complete/unified addition law of twisted Edwards curves enables predictable circuit implementations, especially for fixed-base scalar multiplication during verification, yet the curve's base field ($2^{255} - 19$) typically differs from the proof system's field.

2.2.1 Elliptic Curve Equation

The signature scheme Ed25519 is defined over the finite field $\mathbb{F}_p$ where $p = 2^{255} - 19$. It uses a specific twisted-Edwards curve given by the equation:

$$-x^2 + y^2 = 1 + dx^2y^2, \quad \text{where } d = -\frac{121665}{121666}$$

This curve is birationally equivalent to the Montgomery curve known as Curve25519, which is used for key-exchange protocols (X25519) and has the form:

$$y^2 = x^3 + 486662x^2 + x.$$

We discuss Ed25519 verification in circuits because its unified addition law for twisted-Edwards curves maps predictably to R1CS constraints, even though non-native field arithmetic can increase the final constraint count.

2.2.2 Scalar Multiplication

Scalar multiplication is the process of taking a point on the elliptic curve and multiplying it by an integer (the scalar). The ED25519 makes use of scalar multiplication to derive the public keys from private keys and this operation is integral to signing and verification functions. It is denoted as:

$$P = k \cdot G \tag{2.1}$$

where P is the resulting point on the curve, k is the private scalar (private key), and G is the base point of the curve. The security of the construction relies on the elliptic curve discrete lograrithm problem(ECDLP), namely given P and G it is computationally intractable to recover k. It is not proven that reversing scalar multiplication is infeasible, but it is widely assumed to be hard, and no polynomial-time algorithm is known. The hardness assumption underlies the security of key generation and signature verification. [2]

2.2.3 Signature Generation

To sign a message m with a private key k, the ED25519 algorithm follows these steps:

- Compute an ephemeral secret scalar r deterministically from the private key and the message; the operator $||$ denotes concatenation:

$$r = \text{Hash}(\text{private key} \| m) \tag{2.2}$$

  This per-message scalar is unique (per message) and secret, which is sufficient for security; randomness is not required when r is derived via a pseudorandom function keyed by the secret key.

- Calculate the curve point:

$$R = r \cdot G \tag{2.3}$$

- Compute the hash h over the concatenation of R, the public key A, and the message m:

$$h = \text{Hash}(R \| A \| m) \tag{2.4}$$

  where A is the public key corresponding to the private key, and $\|$ denotes concatenation.

The signature σ for message m is then represented by the pair:

$$\sigma = (R, s) \tag{2.5}$$

where s is computed as:

$$s = (r + h \cdot k) \bmod q \tag{2.6}$$

and q is the order of the base point G. The values R and s together form the digital signature.

2.2.4 Signature Verification

To verify a signature $\sigma = (R, s)$ for a message m with the public key A, the verifier performs the following steps:

- Recompute the hash h as:

$$h = \text{Hash}(R \| A \| m) \tag{2.7}$$

- Check the equation:

$$s \cdot G = R + h \cdot A \tag{2.8}$$

The signature is valid if the equation is holds, meaning that message m could only have been signed by the holder of the secret key corresponding to public key A. This verification ensures integrity without revealing the private key.

ED25519 [2] is widely deployed in practice because it offers a balance of strong security assumptions and efficiency. Verification involves a small number of elliptic-curve operations, which can be efficiently arithmetized in zero-knowledge (ZK) circuits. The group law checks, such as (2.9), can be represented as arithmetic constraints (Boolean constraint when bit-decomposition is required), which provides a concrete way to measure circuit size in systems like Circom.

Performance optimization in ED25519 relies on two implementation choices. First, extended twisted Edwards coordinates allow point addition and doubling without costly field inversions, reducing computation time. Second, large integers are represented in $radix - 2^{64}$ form, so field elements are broken into 64-bit "limbs" that map naturally to machine word sizes. Together, these optimizations enable high verification throughput.

Side-channel resistance is achieved by ensuring that code paths do not depend on secret data: implementations avoid secret-dependent branching and array indexing. This constant-time discipline mitigates timing and cache-based attacks.

Merkle trees are essential tools, being efficient structures for verifying and protecting huge datasets through cryptographic hashing. A Merkle tree is made out of a

series of leaf nodes with the hash value of a data block. The hashes of the child nodes are successively combined and put in the intermediate nodes until the last one, the Merkle Root, which is a single hash that summarizes the entire dataset. This hierarchical design allows verification of inclusion without downloading all data. Other efficient properties of Merkle trees, such as their ability to demonstrate that data has not been modified and the less physical space comprised of the elements of the application, make Merkle tree's applications more comprehensive in blockchain. Merkle proofs can also be encoded as Boolean constraint, so SAT-based insights apply to roll-up data-availability circuits. [21]

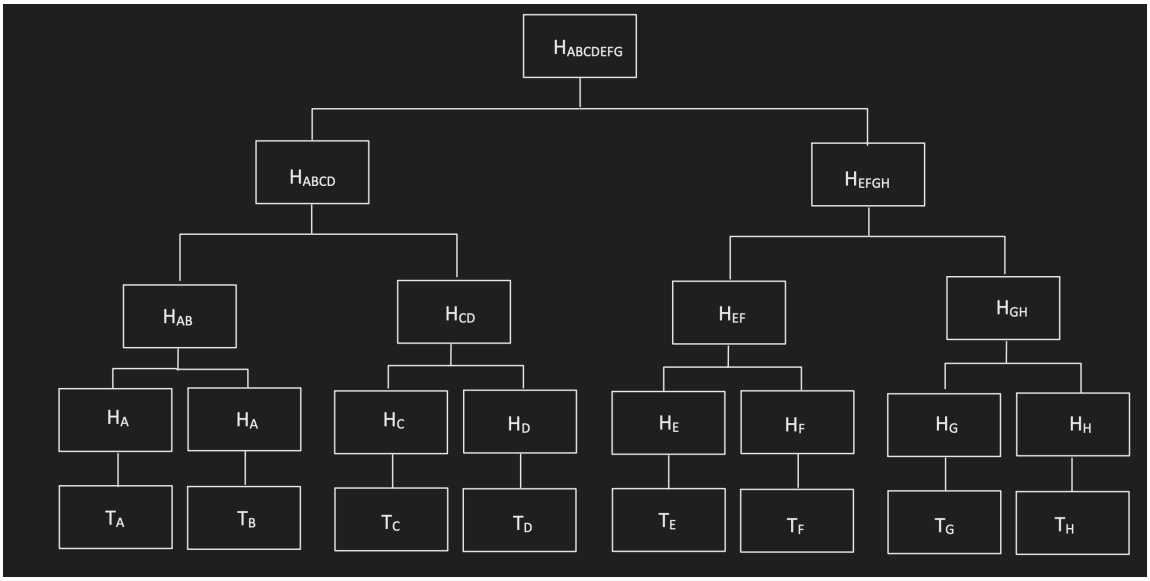


Figure 2.1: Merkle tree structure.

Each leaf node T_n represents the hash of a data block, while each internal node H* is the hash of its two child nodes. The Merkle Root compactly summarizes the integrity of the entire dataset. [21]. Figure 2.1 illustrates this structure. Merkle trees are needed as they allow efficient proofs of membership: for instance, a blockchain client can verify that a particular transaction is included in block by checking a short Merkle proof, without downloading the entire block. This property is also crucial in

rollup and Zk-based solutions, where data-availability proofs must be efficient.

The evolution of modern cryptographic protocols, such as zero-knowledge protocols and multi-party computing, has recently led to the introduction of new primitives focused on arithmetization and designed to be efficiently implemented in arithmetic circuits. An arithmetic circuit is a computational model where inputs and outputs are elements of a finite field, and each gate performs addition or multiplication in that field. This model underlies SNARKs, as it enables algebraic verification of program execution.

Grendel is a Legendre-based permutation designed for efficient arithmetization in ZK circuits. It combines substitution-permutation structure with Legendre symbols to achieve high algebraic degree and storing diffusion. The design goal is to offer a ZK-friendly permutation, similar to Poseidon(the hash function is an algebraic hash function designed for zero-knowledge proof systems), but with distinct algebraic properties, where it efficiently handles cryptographic operations within arithmetic circuits. Ni et al. demonstrate a two-round preimage attack on Grendel, revealing weaknesses despite its high algebraic degree. [20].

ZK-STARKs provide transparent, post-quantum proofs without a trusted setup, trading larger proof size for fast verification.

2.3 Blockchain Consensus Mechanisms and Scalability

Bitcoin's blockchain has a consensus mechanism - named Proof of Work or PoW. Proof of work has an incentive model, where miners - owners of computers with GPU and CPU- sometimes solve complex cryptographic puzzles. [3]

Proof of Stake (PoS) replaces the energy-intensive mining process with a system where validators are selected to propose and attest to blocks in proportion to the amount of cryptocurrency they have staked. Validators place funds as collateral, and misbehavior can result in penalties or slashing, where part of the stake is forfeited. This mechanism improves energy efficiency and scalability while introducing

economic security guarantees.

Byzantine Fault Tolerance (BFT) protocols address the problem of reaching agreement in distributed systems where some nodes may fail, act incorrectly, or behave maliciously. Classical BFT algorithms, such as Practical Byzantine Fault Tolerance(PBFT), can tolerate up to one-third of nodes being Byzantine and still achieve consensus. They provide low-latency finality in partially synchronous settings, but their reliance on high communication overhead makes them less practical for large-scale public blockchains. [8]

Khan et al. [14] review literature that addresses issues of scalability in public blockchains, focusing on throughput, latency, and energy consumption. They emphasize that transaction throughput remains a major challenge, since systems such as Bitcoin and Ethereum handle far fewer transactions per second compared to traditional payment network. Moreover, because every transaction must be stored and validated by every node, both computation and storage requirement grow with network usage, creating barriers to broader adoption in business and industry.

Moreover, the paper investigates both on-chain and off-chain solutions. On-chain solutions, for example, include Segwit, sharding, and increasing block size, which focuses on improving internal processes on a blockchain network to be able to perform more transactions, while off-chain solutions like the Lightning Network conduct the transactions outside of the network in order to ease the congestion of the network. The authors profess that, indeed, the existing consensus algorithms, such as Proof-of-Work, are insufficient in terms of scalability, which has triggered the exploration of other mechanisms that have lesser delay and better processing capacity. This integrative assessment brings together a framework for understanding the approaches of other researchers who are also attempting to address the scalability problem, pointing out the deficiencies in the existing consensus protocols and architectural design that may sustain the use of blockchain in time-critical applications such as the Internet of Things and finance. [14]

The Ethereum white paper describes a decentralized computing framework designed to expand blockchain usage and append a business logic that is used beyond cryptocurrency. DApps, decentralized applications, and smart contracts were in-

troduced. Compared with Bitcoin, which aims the most at secure transactions, Ethereum emphasizes creating a general-purpose blockchain that can provide the possibility and functionality for developers to create automated agreements. DeFi, or decentralized finance, is compatible with Ethereum, which led to the growth of different applications that can follow with the same business logic from fintech, such as lending, borrowing, and trading, no intermediary required with additional logic implementation for managing and securing their assets autonomously. [5]

2.4 Zero-Knowledge Proofs (ZKP) and Privacy-Preserving Technologies

Recent work surveys [36] zk-SNARKs principles, implementations, and emerging applications. ZK-SNARKs, or Zero-Knowledge Succinct Non-Interactive Argument of Knowledge, allows one party (the prover) to convince the other (the verifier) that the statement is correct without disclosing the data. These include reductions like the Quad Arithmetic Program (QAP) and the Pinocchio protocol, which is a protocol that reduces size and verification cost. Chen et al. [10] outlines the usage of zk-SNARKs in security-intense transactions at the blockchain, such as Zcash and Tornado Cash, where they are used to prove transactions without exposing sensitive transaction data, all while ensuring transaction verification using zk-SNARKs.

The zk-SNARK system has a 3-stage pipeline: Key generation, Proof generation, and finally, Proof verification.

- **Key Generation**: In this setup phase, a trusted source generates a proving key (pk) and a verification key (vk) for a specific statement. The keys are created as follows:

$$(pk, vk) \leftarrow \text{KeyGen}(1^\lambda, C) \tag{2.9}$$

  where C is an arithmetic circuit over a finite field and 1^λ denotes the security parameter in unary, representing the targeted security level. These keys allow for efficient proof generation and verification.

- **Proof Generation**: To generate a proof π, prover uses the proving key, the public statement x, and a private witness w.

$$\pi \leftarrow \text{Prove}(pk, x, w) \tag{2.10}$$

  The proof π shows that the prover has a correct value or knowledge of w satisfying the statement x without revealing any part of w.

- **Verification**: With the verification key, the verifier checks the proof's validity using the equation:

$$\{0, 1\} \leftarrow \text{Verify}(vk, x, \pi) \tag{2.11}$$

  Here, the output is 1 if the proof is valid and 0 otherwise, ensuring the correctness of the statement without requiring direct data disclosure.

The paper further describes the zk-SNARK parameterization that has enabled new models, but the main focus here is on core properties of zk-SNARKs such as security, privacy, and scalability in blockchain applications [10].

Completeness states that if the statement made by the prover is true, the verifier will be convinced. Soundness ensures that if a statement is false, no prover can convince the verifier otherwise. The defining feature is the zero-knowledge property [11].

For example, in a blockchain setting, a zk-SNARK can prove that a prover knows the secret key corresponding to a public address and that a transaction balances correctly, without revealing the secret key or transaction details.

Such proofs are especially useful in privacy-preserving cryptocurrencies and DApps, where transaction validity can be verified without disclosing sensitive information. This technology addresses both privacy and scalability challenges in blockchain systems [17].

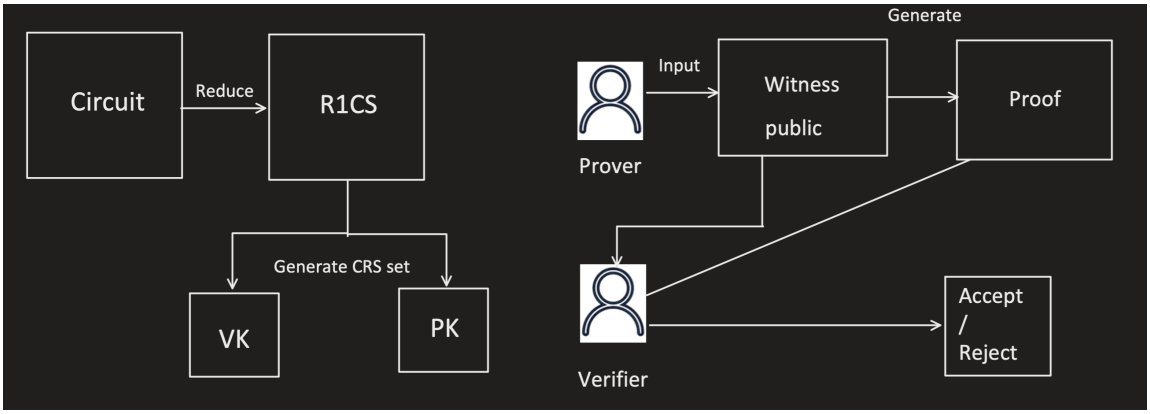


Figure 2.2: zk-SNARK diagram, Key, Proof, and Verification steps.

In recent literature, alternative SNARK designs have been proposed that adapt classical protocols to blockchain environments with high throughput and frequent state changes. The SNARK framework, which is presented in this work, incorporates the key ideas from the Groth16 protocol, which is distinguished by the minimal proof size and the least verification cost. In parallel, it also considers the development of several enhancements that are directed at improving the effectiveness in distributed and dynamic settings, such as blockchain networks. In the same manner as Groth16, it uses R1CS (Rank-1 constraint system) which is used in CIRCOM to convert the computational problems into systems of polynomial function equations. Nonetheless, this particular SNARK implementation was designed to work with large sets of frequently changing computations without the inherent requirements for large amounts of pre-computed information, which has remained an obstacle of traditional SNARK applications on rapidly evolving computations. In the case of this scheme, the drawbacks can be easily addressed through the standard combination of Groth16 and PlonK elements because it enhances both types of arrangements and demonstrations while preserving the brevity and level of security anticipated in non-interactive proofs.[16]

In the context of blockchain, this particular SNARK design makes it possible, amongst other things, to have smaller proof sizes and, furthermore, higher communication efficiency, which suits cases with unreliable nodes as well as large-scale

verifiable computations. In the recently proposed SNARK structure, it has been demonstrated that the proof size is reduced from four to ten times compared to similar models, which enhances the algorithm's usage for the tasks in blockchain applications that require many small and rapid transactions with low-latency verification. This flexibility, coupled with reduced load on computational resources for the verifiers, improves the overall security and scalability of the blockchain technologies and, therefore, increases the prospects for more valuable and private dealing in decentralized networks. [16]

2.5 SAT Solving in Blockchain and Cryptographic Applications

DPLL, or the Davis-Putnam-Logemann-Loveland algorithm, was introduced in 1962 as the first method for solving SAT problems. The algorithm uses DFS or depth-first-search with pure-literal eliminations, unit propagation, and splitting rules. DPLL has exponential time complexity in the worst case. The evolution of SAT solvers was significantly accelerated by SAT competitions in 1996, with the promotion of solving efficiency and broader application of SAT problems. [25]

Conflict-driven clause Learning or (CDCL) on which modern SAT solvers are based, extends the DPLL algorithm with the implementation of advanced techniques such as clause learning, heuristic decision, restart techniques, and efficient data structures, for example, Chaff, GRASP, and Minisat. Every SAT solver has different enhancements, for example, non-chronological backtracking, decision heuristics - VSIDS (Variable State Independent Decaying Sum), and watched-literal data structures.[25].

Further, the presentation also addresses variable heuristics like the Variable State Independent Decaying Sum (VSIDS) designed for resolving conflict variables, thereby making a conflict-oriented decision, resulting in improving the performance of the solver. [18]

Chaff is an advanced version of the SAT solver developed to enhance the boolean

satisfiability problem by improving the algorithmic approach and data structure. The core of Chaff [19] algorithm is Conflict-Driven Clause Learning (CDCL). This approach enhances the efficiency of the SAT solver during the search phase by learning the conflicts that might be encountered during the process. In other words, Chaff decreases the search space. For example, if a solver assigns $x = true$ which forces $y = true$ and subsequently $z = true$, but another clause requires $\neg z$, a conflict occurs. CDCL records this as a new clause to prevent the same assignment, and the VSIDS heuristic increases the priority of the conflicted variables, guiding the solver towards a faster resolution. Further, Chaff uses the Variable State Independent Decaying Sum (VSIDS), which is aggressive in variable selection by always selecting those most recently in conflict to maximize return on the search.

Armin Biere, in his seminar titled, "Tutorial on Modern SAT Solving" [18] at EPFL, surveys modern approaches to Boolean satisfiability (SAT), including various encoding procedures for translating problems into CNF. Following this principle, in our hybrid model, we use explicit XOR-gadgets for the Merkle hashing subcircuits; this keeps clauses structured but can increase clause count compared to a pure Tseitin CNF. The tutorial also covers elimination techniques and optimization approaches relevant to modern solvers.

The data structures in Chaff are designed to eliminate unnecessary computations and enable quick clause management to improve the solver's efficiency in more complicated problems. Chaff has potential in applications involving fast and scalable SAT solving, such as hardware verification, due to the effective control of memory and data access that enables the running time to be reduced significantly. [19]

As this thesis targets zk-SNARK circuits, Circom is introduced as the tool to translate SAT problems into R1CS constraints. Circom is a constraint-based DSL that allows the generation of arithmetic circuits that can then be used in ZK applications. Specifically developed for ZK-SNARKs, Circom allows the developers to represent computational problems as circuits that will be encoded into an R1CS. Using the Circom compiler, these descriptions of circuits are converted to files with R1CS constraints, along with the creation of C++ and WebAssembly programs for the computation of witness, thus hiding the difficulty underlying the generation of

ZK proofs. To enable effective circuit design and implementation, Circom is provided with modular templates for combining components to form intricate designs for the circuits, further increasing the applications of ZK circuits in terms of their usability and scale.

It is commonplace to see Circom usage together with SNARKJS, a javascript tool designed to generate and verify ZK-SNARK proofs based on the R1CS constraints. This stack allows the integration of ZK proofs into web and server applications, thus broadening the scope of Circom circuits in distributed ledger technology. Furthermore, Circom supports the CIRCOMLIB library of free-of-cost hash functions, digital signatures, Merkle trees, and other essential templates for building blockchains. The reviewed literature shapes the design choices in Chapter 3, where will be developed SAT solver into Circom circuits. Chapter 3 builds on the above by implementing the first Circom DPLL and Chaff circuits and benchmarking them. [15]

Chapter 3

Methodology

Thesis contributions

- **Circom-DPLL benchmarks.** I reimplemented a prototype of the DPLL algorithm directly in Circom in order to evaluate how far a classical, recursion-heavy SAT solver can be expressed as a zero-knowledge circuit before constraint growth becomes prohibitive. The implementation achieves feasible execution up to a *limit* of approximately 100 literals and 200 clauses. While this scale is modest compared to modern SAT solvers, which routinely handle formulas with millions of clauses, it is historically significant. Early SAT competitions in the mid-1990s often reported results on instances of this size, meaning that the Circom implementation matches the scale of state-of-the-art solvers from that era. This provides a useful empirical baseline for evaluating the feasibility of encoding complete SAT solvers as zk-circuits.

- **Circom-Chaff prototype.** I implemented a prototype of the Chaff solver, in Circom incorporating the watched-literals scheme and placeholders for conflict-driven clause learning (CDCL), was also implemented in Circom. This prototype demonstrates that more advanced SAT solver strategies can be structurally encoded in Circom. However, due to the exponential blowup of arithmetic constraints, the implementation exposes on the order of 10^6 constraints. These constraints cannot be handled efficiently in a single zk-circuit, and there-

fore must be *split off-chain* into multiple sub-computations whose succinct correctness proofs can then be aggregated. In other words, the solver logic is defined inside Circom, but practical scalability requires distributing the proof burden between off-chain computation and on-chain verification.

- **Hybrid Rust + Circom pipeline.** To address the above scalability issues, I designed a hybrid workflow in which the heavy lifting of SAT solving is performed in Rust, while Circom is used to generate a succinct zero-knowledge check of the final result. Specifically, the Rust side solves SAT instances with up to 100,000 literals, while the on-chain verifier only checks a compact *RootCheck* circuit with 4,098 constraints. This division allows large instances to be processed off-chain, while retaining the security guarantees of on-chain verification.

- **Empirical evaluation.** The hybrid system successfully solves CNFs with 100,000 literals in under *4 seconds* off-chain, while the corresponding zk-SNARK verification requires only 42,000 gas on-chain. These results show that although a pure Circom implementation of DPLL or Chaff is infeasible beyond toy benchmarks, the hybrid approach allows SAT-based reasoning to be integrated into blockchain systems with practical performance.

Test platform. Every measurement in this chapter was obtained on a *MacBook Pro 2019* (Intel i9 @ 2.3 GHz, 16 GB RAM, macOS 14, Rust 1.78, Circom 2.1.7).

3.1 CIRCOM's DPLL Algorithm benchmark

DPLL has been re-implemented in Circom to measure how far a straightforward, recursion-heavy SAT solver can run within a zero-knowledge proof circuit before the number of constraints becomes unmanageable.

The specific decision problem that DPLL addresses is finding a satisfying assignment for a Boolean formula expressed in conjunctive normal form (CNF) - that is, determining whether there exists any assignment of truth values to variables that makes all clauses simultaneously true. The algorithm does not assume that this

assignment is unique, rather it systematically explores possible assignments until it finds one that satisfies the formula or concludes that none exists. DPLL extends basic backtracking with two key techniques - unit propagation and pure literal elimination - and introduces a splitting rule that recursively partitions the search space. Combined, these enhancements significantly reduce the number of branches explored and enable the solver to handle larger satisfiability instances than earlier backtracking approaches. [25]

The DPLL algorithm operates recursively and can be separated into a few main steps:

1. **Satisfiable Check:** If there is an empty set of clauses, it means that all the clauses have been satisfied, and as a result, the formula is satisfiable.

2. **Unsatisfiable Check:** If an empty clause appears during the solving process, it indicates a contradiction among the clauses, meaning that no assignment of truth values can satisfy the formula. In that case, the formula is classified as UNSAT (unsatisfiable).

3. **Unit Propagation:** This method exposes unit clauses contained in the set of clauses. A unit clause is equal to a clause that has only one literal that has not been assigned a value. It is necessary that assigning a value to the literal in the unit clause makes the clause true. This value is then passed over the unit, reducing its complexity. If this transmitted value results in a unit clause, the model is said to have no solution For example, in the formula $(x_1 \vee \neg x_2) \wedge (x_2) \wedge (x_3 \vee x_4)$, the clause (x_2) is a unit clause. Unit propagation assigns $x_2 = \text{true}$, simplifying all clauses containing x_2 and reducing the formula.

$$\underbrace{(a \vee b) \ \wedge \ (\neg a \vee c) \ \wedge \ (\neg c \vee d) \ \wedge \ (a)}_{\varphi} \quad \longrightarrow [\text{unit: } (a)] \underbrace{(c) \ \wedge \ (\neg c \vee d)}_{\varphi'}$$

Figure 3.1: An illustration of unit propagation.

In Figure 3.1, the unit clause (a) forces the assignment $a = \text{true}$. Consequently, the clause $(a \vee b)$ is satisfied and removed, while $(\neg a \vee c)$ simplified to a new unit clause (c).

4. **Pure Literal Elimination:** This step identifies variables that appear with only one polarity - either always positive or always negative - across all clauses in the formula. This assignment can change the truth values of variable bundles, thereby deleting inevitable conflicts of all possible assignments of variables, which makes the formula less complex. For example, if in the formula $(x_1 \vee x_3) \wedge (\neg x_2 \vee x_3)$ the variable x_3 always appears positively, then x_3 is a pure literal. Assigning $x_3 = \text{true}$ immediately satisfies all clauses containing it.

$$\underbrace{(x_1 \vee x_3) \ \wedge \ (\neg x_2 \vee x_3)}_{\varphi} \quad \longrightarrow [\text{pure literal: } x_3] \quad \underbrace{\top}_{\text{all clauses satisfied}}$$

Figure 3.2: An illustration of pure literal elimination.

The x_3 is pure as it appears only in its positive form. Assigning $x_3 = \text{true}$ satisfies all clauses in which it appears, simplifying the formula.

5. **Splitting Rule:** The algorithm takes the opposite course, picking the first literal when neither pure literal nor unit propagation holds and explores recursively both possible values for the literal, which are true and false. This involves a more depth-wise motion through possible assignments for each variable in turn.

6. **Recursive Application:** In a similar manner, the recursive calls perform the same operations on the simplified formulas until a solution has been attained or proof is accomplished that such a formula is not satisfiable. [25]

Algorithm 1 DPLL[28]

Input: A set of clauses Φ; **Output:** SAT or UNSAT;

```
function DPLL(Φ)
    // unit propagation:
    while there is a unit clause l in Φ do
        Φ ← unit-propagate(l, Φ)
    end while
    // pure literal elimination:
    while there is a literal l that occurs pure in Φ do
        Φ ← pure-literal-assign(l, Φ)
    end while
    // stopping conditions:
    if Φ is empty then
        return true
    end if
    if Φ contains an empty clause then
        return false
    end if
    // DPLL procedure:
    l ← choose-literal(Φ)
    return DPLL(Φ ∧ l) or DPLL(Φ ∧ ¬l)
end function
```

3.1.1 Example Application

As an example the CNF formula: $(x_1 \vee x_2) \wedge (\neg x_1 \vee x_3) \wedge (\neg x_2 \vee \neg x_3)$ will be used. In JSON format as an input it will have an appearance:

Input Example:

```
{
  "literals": [1, 2, 3],
```

```
  "clauses": [
    [ 1, 2 ], // (x1 OR x2)
    [ -1, 3 ], // (~x1 OR x3)
    [ -2, -3 ] // (~x2 OR ~x3)
  ]
}
```

The following steps demonstrate how the DPLL algorithm finds a satisfying assignment for this formula. The initial formula, which we will call Φ, has no unit clauses or pure literals, so the algorithm must begin by making a decision.

1. **Decision (Splitting Rule)**
   The algorithm selects an unassigned variable, in this case x_1, and assigns it a truth value. Let's assume the algorithm first tries $x_1 = \textbf{true}$.

2. **Formula Simplification**
   This assignment is propagated through the formula Φ:

   - The clause $(x_1 \lor x_2)$ becomes $(\text{true} \lor x_2)$, which is now satisfied and can be removed from the formula.
   - The clause $(\neg x_1 \lor x_3)$ becomes $(\text{false} \lor x_3)$, which simplifies to a new clause (x_3).
   - The clause $(\neg x_2 \lor \neg x_3)$ remains unchanged for now.

   After this simplification, the new formula, Φ', is:

   $$(x_3) \land (\neg x_2 \lor \neg x_3)$$

3. **Unit Propagation**
   The new formula Φ' contains a **unit clause**: (x_3). This is a critical step, as it forces a deterministic assignment. To satisfy this clause, the algorithm must set $x_3 = \textbf{true}$. This new assignment is then propagated:

   - The clause $(\neg x_2 \lor \neg x_3)$ becomes $(\neg x_2 \lor \text{false})$, which simplifies to $(\neg x_2)$.

The formula is now further reduced to Φ'':

$$(\neg x_2)$$

4. **Final Unit Propagation**
   The formula Φ'' is itself a unit clause. This forces the final assignment: $x_2 =$ **false**.

5. **Solution Found**
   At this point, all variables have been assigned values ($x_1 = \text{true}$, $x_3 = \text{true}$, $x_2 = \text{false}$), and the set of clauses is empty, meaning all clauses have been satisfied. The algorithm terminates and returns **SAT** with the satisfying assignment:
   $$\{x_1 = \text{true}, x_2 = \text{false}, x_3 = \text{true}\}$$

This walkthrough demonstrates how a single decision via the Splitting Rule can trigger a cascade of deterministic assignments through Unit Propagation, which efficiently prunes the search space. Had this initial choice of $x_1 = \text{true}$ led to a contradiction (e.g., by producing an empty clause ()), the algorithm would have backtracked and explored the alternative branch where $x_1 = \text{false}$.

The Circom implementation of the DPLL algorithm adapts this original SAT and its implementation into the usage field of zero-knowledge proofs by embedding inside a circuit that can be evaluated privately. The code provided separates the DPLL algorithm into its iterative steps. Steps are: initialization, unit propagation, and pure literal elimination. Together, these phases represent the algorithm's decision-making process, where each step progressively constrains the search space until a satisfying assignment is found or proven absent. This modular design provides a clear and systematic approach for handling CNF instances withing a zero-knowledge circuit.

The pipeline uses the template SatSolverDPLL_Initialization. Template in Circom DSL is a reusable circuit component. It connects with wires together inputs, outputs, and constraints. Initialization of the template takes the input literals and

clauses, loads them into fixed-size arrays, and normalizes their representation (distinguishing positive and negative cases). The main goal is to prepare the CNF formula in a consistent internal format that can later be processed, such as unit propagation and pure literal elimination.

The first stage is unit propagation, which uses the UnitPropagationPhase template. This template scans for unit clauses and enforces that literal as true. Given the clause (x_2), the template sets $x_2 = 1$ and then removes simplified or simplifies every clause containing x_2. This automatically reduces the size of the formula in each iteration.

After dealing with unit propagation, the next stage is the elimination of pure literals, which is dealt with in the pure_literal_elimination template. Template iterates through all literals and checks whether a variable appears only in one polarity. For example, if x_3 it appears only positively in all clauses, the template fixes $x_3 = 1$ and removes all clauses containing $x3$. These actions keep the formula smaller and avoid excess branching by deterministically assigning values.

Circuit	**Literals**	**Clauses**	**Non-linear Constraints**	**Linear Constraints**	**Wires**	**Labels**
Circuit 1	10	20	72	64	236	2095
Circuit 2	20	40	142	124	466	4175
Circuit 3	30	60	212	184	696	6255
Circuit 4	40	80	282	244	926	8335
Circuit 5	50	100	352	304	1156	10 415
Circuit 6	60	120	422	364	1386	12 495
Circuit 7	70	140	492	424	1616	14 575
Circuit 8	80	160	562	484	1846	16 655
Circuit 9	90	180	632	544	2076	18 735
Circuit 10	100	200	702	604	2306	20 815

Table 3.1: Comparison of Different Circuit Metrics with Increasing Literals and Clauses

Compilation boundary at the 100 × 200 mark

- **Array duplication.** Circom arrays are immutable in context that every recursive call in DPLL must clone the entire state of `literals[]`. This causes the circuit's complexity to grow super-linearly with the required search depth.
- **Proving-key RAM ceiling.** As shown in Table 3.1, while the final constraint count for a 100 x 200 instance is a small 1,306 (702 non-linear + 604 linear). The number of internal labels required by the compiler reaches 20,815. In practice, generating the Groth16 proving key for a circuit with such a high number of labels demanded more than 16 GB of RAM available on test machine, causing the compiler to abort.

The compilation failure demonstrates that the DPLL-in-Circom approach is only usable for small, statically sized CNF problems, such as a single-signature Schnorr relation.

The template DPLL controls branching and recursion when the formula is fully reduced. The DPLL algorithm's motivation is to make a recursive selection of a particular literal and factor the problem into two parts: the first has the assignment of the chosen literal to be true, while the second has the literal being false. In Circom, branching is realized by instantiating additional copies of the unit propagation and pure literal elimination templates for each branch of the search tree. The component also consists of signals for monitoring the satisfiability status of each one of the branches. If some branch of one of these three cases leads to a solution level that satisfies the formula, that means that the whole problem has a solution.

The GreaterThan component is used last to combine all the resulting branches, which allows one to check the feasibility of the given problem formulation.

```
1 template MAIN_DPLLSolverEntry(sizeLiterals, sizeClauses,
    sizeClauseElements) {
2     signal input input_literals_initial[sizeLiterals];
3     signal input input_clauses_initial[sizeClauses][sizeClauseElements];
4     signal output publicOutput;
```

```
    component dpll_core = DPLLCoreSolver(sizeLiterals, sizeClauses,
        sizeClauseElements);
    for (var i = 0; i < sizeLiterals; i++) {
        dpll_core.dpll_literals_input[i] <== input_literals_initial[i];
    }
    for (var i = 0; i < sizeClauses; i++) {
        for (var j = 0; j < sizeClauseElements; j++) {
            dpll_core.dpll_clauses_input[i][j] <== input_clauses_initial[i][
                j];
        }
    }
    signal literal_sum_check[sizeLiterals + 1];
    literal_sum_check[0] <== 0;
    for (var i = 0; i < sizeLiterals; i++) {
        literal_sum_check[i + 1] <== literal_sum_check[i] + dpll_core.
            DPLL_normalized_literals[i];
    }
    publicOutput <== literal_sum_check[sizeLiterals];
}
```

Listing 3.1: Main template for DPLL algorithm in CIRCOM

The SAT Solver designed in Circom employs a modular and signal-driven pipeline to process CNF formulas using an implementation of the DPLL algorithm. The system begins by initializing the literals and clauses using inputs such as textit-input_literals textit_initial and textitinput_clauses_initial, representing the original SAT formula. The template - serves to normalize textitdpll_init_component— corresponding to the textitSatSolverDPL L_Initialization these inputs by translating literal polarity into binary-friendly values(for example, encoding $\neg x_2$ as ($id = 2$, polarity $= 0$) and x_2 as ($id = 2$, polarity $= 1$)), with auxiliary variables such as textitis_literal_negative, textitfinal_literal_assignment, and textitnormalized_literals performing this transformation. This stage ensures that all literals are structured properly, ensuring that their original representation was positive or negative. The same

normalization is applied to the clauses. This step with normalization is important for maintaining signal determinism in Circom while enabling arithmetic operations within the Rank-1 Constraint System framework.

Following the normalization stage, the solver proceeds to the inference and simplification phases, beginning with the *unit_propagation_component*. The literals and clauses are *unit_clauses_input*. The propagation logic identifies unit clauses and simplifies the formula accordingly. In parallel, the *pure_literal_eliminate* module evaluates the frequency and polarity of each literal across the remaining clauses using counters such as *literal_positive* and *negative_occurrences*. The system determines whether a literal is "pure", and if so, assigns it deterministically via *is_literal_pure_positive* and *is_literal_pure_negative*. These assignments are reflected in the *pure_lit_literals_output* and *pure_lit_clauses_output* signals.

The pipeline concludes by aggregating the current state of literal assignments into textitliteral_sum_check, a signal that allows the circuit to output a succinct public result verifying the satisfiability of the processed subformula. Each of these stages is composable and zero-knowledge proof compatible, meaning that every step is expressed as arithmetic constraints in R1CS, so satisfiability can be proved without revealing literal assignments. This makes the pipeline suitable for secure and verifiable on-chain SAT solving.

The recursion depth and the assigned values of literals at that depth ("numerical values") stored in the DPLL circuit in Circom are designed to keep track of the potential literals and the clauses across some recursive depth. Such a design allows the DPLL structure to search solution spaces or, instead, trees. In its essence, the method of searching for the solution tree enabled by tracing the branches of the trees/potential solution spaces and simplifying these limbs through unit propagation and pure literal deletion is a simulation of the functioning of the conventional DPLL SAT solver.

There are, in addition, other useful components, including the EqualsZero gadget, which checks if a signal is equal to 0 and outputs a constraint bit used in branching decisions. With these components, the Circom DPLL seeks to demonstrate either that the CNF formula has a solution or that this particular CNF formula does not

have any solution, all without leaving the scope of generating zk proofs so that no computations are exposed.

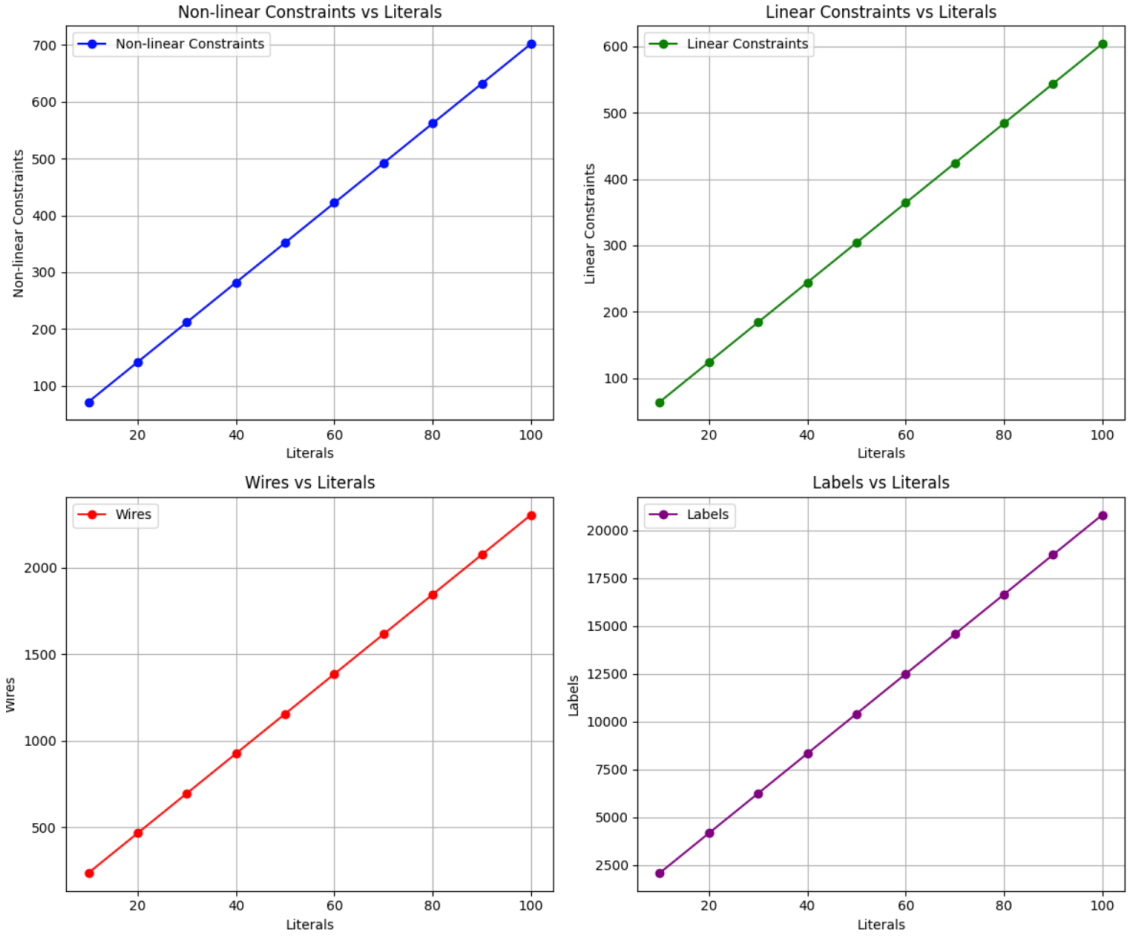


Figure 3.3: Scaling of circuit size

The diagram shows how a number of dimensions, like nonlinear constraints, linear constraints, private inputs, public outputs, wires, and labels, scale with the growth of literals and clauses in the DPLL circuit. Similarly, as the number of literals and clauses increases, these metrics also begin to grow, which implies that the minimal total that can be represented as a linear or quadratic representation in terms of cost and resources. Such an analysis indicates that there is a proportional increase in the

size of the circuit with an increase in the complexity of the problem, thereby affecting the level of parallelism and efficiency. The evaluation in this way is important since it highlights the boundary conditions of the circuit and also demonstrates how different circuits act to make growth and performance constraints.

Parameter sizeLiterals outlines the number of literals the circuit can operate in the SAT formula. Each literal is a boolean variable, assumed to be true or false. For instance, in the example of DPLL, the circuit can accept about 50 literals and thus carry out moderately complex formulas. The standing array is facilitated through an array of SAT_literals where the specific numerical values of literals are determined by the subject area of the formula, which is subject to a sat check.

This parameter size indicates how many clauses are included in the context of the SAT formula. A clause should be defined as a set of literals joined together by the OR logical function. These SAT clauses are incorporated into the final SAT formula, which now becomes a logical conjunction of the immediately preceding clauses. The problem is determining if giving some literal truth values is possible so that all the clauses are true. The circuit could work with 200 clauses, thereby allowing it to process complex logic expressions with many relationships among literals.

Alongside literals and clauses, the parameter `sizeClauseElements` fixes the maximum number of literals in each clause. In our circuits this is capped at five, producing a 200×5 clause array. This bound ensures compilation remains tractable while still allowing representation of complex logical relationships.

3.1.2 Challenges in implementation

Implementing the DPLL (Davis–Putnam–Logemann–Loveland) algorithm inside Circom highlighted several structural mismatches between recursive SAT solvers and the static nature of zero-knowledge circuits. This section explains the main difficulties, gives concrete examples, and outlines the design compromises that made a Circom-based DPLL prototype possible.

Recursive Logic in Zero-Knowledge Circuits

The core challenge is that DPLL is naturally expressed as a recursive algorithm: after choosing a literal, the solver recursively calls itself on the simplified sub-formula. In software, this is natural, but in Circom, default recursion is not supported because every circuit must be fully expanded during compilation time.

Example. In a default solver, branching on `x1` creates two recursive calls: one where `x1=true` and another where `x1=false`. Circom, however, forbids dynamic instantiation of sub-circuits; as a result, such recursive branching cannot be expressed directly.

Workaround. Instead of recursion, the pipeline was unrolled manually. Each stage of the algorithm (initialization, unit propagation, pure literal elimination) was implemented as a separate template, and additional copies of these templates were instantiated to emulate deeper levels of search.

Cost. The static unrolling caused super-linear growth in circuit size: each new decision depth required duplicating entire arrays of literals and clauses. Table 3.1 shows that at 100 literals and 200 clauses the number of internal labels already exceeded 20,000, which exhausted 16 GB of RAM during proving-key generation.

State Management Across Branching Depths

Each recursive call in DPLL carries its own state: the remaining set of clauses and the current assignments. Circom has no mutable memory, so every branch depth requires a fresh copy of all state arrays.

Example. A CNF with 50 literals required 50-element arrays to be branched at every level, even when only a small subset of literals was actually modified.

Workaround. Templates such as `ClearArray`, `ResizeArray`, and `EraseElement` were introduced to emulate dynamic state updates. These simulate array clearing and resizing by allocating new arrays and copying values, ensuring that every branch has its own static state.

Cost. This approach inflated the wire count and label count, producing circuits much larger than the actual logical content of the SAT instance. For instance, while

a 100/200 instance has only 1,306 constraints, it compiles into more than 20,000 labels because every state copy is materialized.

Conditional Branching and Non-Quadratic Constraints

DPLL also relies heavily on conditional decisions: if a clause becomes unit, force its literal; if a contradiction arises, it backtracks. Naively encoding such `if/else` logic in Circom introduces unsupported non-quadratic constraints.

Example. Directly encoding into `if (clause == unit) assign literal` leads to a constraint that depends on the truth of a signal, which Circom cannot compile.

Workaround. These conditions were replaced with **selector signals**: a multiplicative gate chooses between two values based on a binary flag, e.g. $out = cond * x + (1 - cond) * y$. This avoids non-quadratic constraints but requires explicit encoding of every possible case.

Cost. The solution preserves correctness but reduces efficiency, since all possible branches are instantiated whether or not they are taken. This further contributes to circuit bloat and explains why the usable range of the prototype is limited to small CNFs such as Sudoku or Schnorr signature checks.

Circuit Size and Proof Generation Trade-Offs

The above issues together created a fundamental tension: keeping the algorithm faithful to DPLL while producing circuits small enough for proof generation.

Example. At the 100/200 scale the Groth16 proving key exceeded 16 GB RAM, aborting compilation.

Workaround. To remain practical, the design restricted clause size (five literals per clause), reduced propagation depth, and omitted some optimizations.

Cost. These compromises make the circuit efficient enough to generate proofs, but at the price of generality: the Circom DPLL is only viable for small, fixed-shape CNFs.

3.2 Chaff Solver

The Chaff SAT solver represents a breakthrough in development of Boolean Satisfiability solvers. It was introduced in 2001 by Matthew Moskewicz et al. [19] The development was a major step forward in increasing performance and application scope. One of the most notable improvements concerned the two most resource-intensive operations in SAT solving: Boolean Constraint Propagation and decision heuristics. This reengineering of the core functions of the solver made Chaff between one and two orders of magnitude faster than any of its predecessors, which made it possible for it to tackle problems with variables ranging from the tens of thousands to hundreds of thousands. In contrast, previous solvers such as DPLL or GRASP could only deal with hundreds of variables.

In addition, Chaff introduced the Watched Literals scheme in Boolean Constraint Propagation. In this configuration, each Boolean formula clause has a pair of "watched literals." These indicators make it possible to know whether a particular clause is satisfied and, therefore, reduce access to significant numbers of clauses during propagation. As a result, the solver does not have to spend a lot of processing time on unnecessary clause evaluations, and the overall memory footprint remains relatively small. Such an improvement not only speeds up the process of arriving at a solution but also restricts the amount of memory resources required, making it ideal for large-scale industrial applications such as hardware verification.

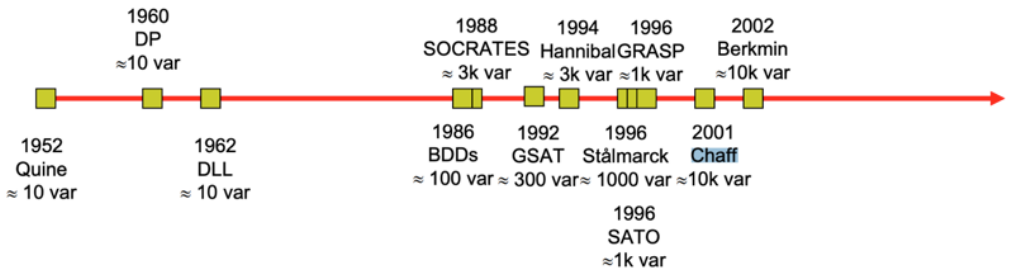


Figure 3.4: Comparison of DPLL and Chaff performance

Another essential feature of Chaff is its Variable State Independent Decaying Sum (VSIDS) decision heuristic. Based on the weighted activity of variables during conflicts, a priority is set for voting in any subsequent decisions concerning the variables that have earlier had a conflict. This heuristic is both effective and computationally cheap: a linear-style decision-making process—significantly narrowing down the search space the solver must explore. In contrast to previous heuristics, which required complex statistics of clause updating, the new one is based on VSIDS mechanisms with exponential decay that spares computational resources but does not spare decision-making capabilities. [19]

As shown in Figure 3.4, Chaff's watched-literal and VSIDS machinery dramatically reduces the search space compared with DPLL on the same families of instances

Chaff's new ideas, however, went beyond its usability. The solver performs well on regular computers due to the analytical components and algorithms that were developed focusing on the best use of the cache. Development through test profiling ensured that the implementing features of the solver were consistent with the realities of SAT problem-solving. This practical design philosophy made Chaff very popular and introduced new developments in SAT-solving technologies, such as MiniSAT and Glucose, expanding its ideas further.

Chaff demonstrated in practice how theoretical insights and engineering optimizations can be effectively combined. The algorithm fuses logic and practicality, transforming the NP-complete Boolean Satisfiability problem into a usable tool. Understanding Chaff's contributions enables researchers to see how theory and engineering combine to create other SAT-solving developments.

The algorithm begins with the initialization phase, where all variables and clauses are set up, and two literals are selected to be watched for each clause. During the decision-making phase, the solver uses a heuristic, often the Variable State Independent Decaying Sum (VSIDS), to select an unassigned variable and assign it a truth value. This heuristic prioritizes variables that have recently caused the most conflicts, aiming to resolve contentious areas of the formula more efficiently.

In comparison, the DPLL algorithm does not include Conflict-Driven Clause Learning or non-chronological backtracking. Clause evaluation is less efficient due to

Algorithm 2 Chaff [19] (SAT formula F)

Require: A CNF formula F
Ensure: A satisfying assignment or UNSAT

```
Initialize two-watched literals for each clause
Initialize decision stack
Initialize VSIDS counters for all variables
while true do
    if no more decisions to make then
        return SAT
    end if
    decision ← choose_decision_variable(F, VSIDS)      ▷ Select variable based on VSIDS
    decision_stack.push(decision)
    while true do
        unit_clauses ← find_unit_clauses(F, watched_literals)      ▷ Use watched literals to detect unit clauses efficiently
        if unit_clauses is empty then
            break
        end if
        for all clause ∈ unit_clauses do
            propagate_unit_clause(F, clause) ▷ Propagate implications transitively
            if conflict_detected(F) then
                if decision_stack.is_empty() then
                    return UNSAT
                end if
                Perform conflict analysis and learn a new clause
                backtrack_decision ← decision_stack.pop()
                reverse_decision(F, backtrack_decision)      ▷ Perform non-chronological backtracking
                break
            end if
        end for
    end while
    Periodically decay VSIDS counters    ▷ Prioritize recent conflicts in variable selection
    if restart condition met then
        Restart solver and retain learned clauses
    end if
end while
```

the lack of a two-literal watching scheme, as is realized in Chaff. DPLL systematically explores all possible variable assignments, which can lead to exponential time complexity in the worst-case scenario. Chaff's improvements, such as conflict analysis and clause learning, allow it to avoid redundant searches and backtrack more intelligently, making it significantly faster for large and complex SAT instances, as shown in [27].

3.2.1 Overview Of Code Structure and Functionality

The purpose of the SAT solver, as coded herein, is to solve the satisfiability problem elegantly by utilizing many components corresponding to various Chaff algorithm phases. Each component serves this function's specific objective because it makes the code perform efficiently and within the boundaries of zero knowledge-proof systems. The main components include utility functions for arrays, core functionalities such as backtracking, watched literals, and some heuristics like the Variable State Independent Decaying Sum for good variable selection.

In Listing 3.2 we show the *MAIN_ChaffSolverEntry* template, which initializes and connects the Chaff pipeline. Unlike the earlier DPLL prototype, this version directly integrates the two-watched-literal optimization, a core concept of the Chaff algorithm. The initialization of two "watched" literals for each clause ensures that only two literals are monitored, eliminating the need to traverse the entire clause during propagation. The *ApplyWatchedLiteralScheme* component then illustrates how the solver uses this optimization to efficiently identify unit clauses, avoiding redundant checks that would otherwise slow down unit propagation and thereby improving overall performance.

```
1 template MAIN_ChaffSolverEntry(sizeLiterals, sizeClauses,
     sizeClauseElements) {
2     signal input input_literals_initial[sizeLiterals];
3     signal input input_clauses_initial[sizeClauses][sizeClauseElements];
4
5     signal output publicOutput;
```

```
6
7       component chaff_core = ChaffSolverCore(sizeLiterals, sizeClauses,
            sizeClauseElements);
8       for (var i = 0; i < sizeLiterals; i++) {
9           chaff_core.raw_literals[i] <== input_literals_initial[i];
10      }
11      for (var i = 0; i < sizeClauses; i++) {
12          for (var j = 0; j < sizeClauseElements; j++) {
13              chaff_core.raw_clauses[i][j] <== input_clauses_initial[i][j];
14          }
15      }
16
17      signal final_literal_sum_check[sizeLiterals + 1];
18      final_literal_sum_check[0] <== 0;
19      for (var i = 0; i < sizeLiterals; i++) {
20          final_literal_sum_check[i + 1] <== final_literal_sum_check[i] +
                chaff_core.Chaff_normalized_literals[i];
21      }
22
23      publicOutput <== final_literal_sum_check[sizeLiterals];
24  }
```

Listing 3.2: Main template for Chaff algorithm in CIRCOM

CHAFF SAT Solver in Circom utilizes a modular pipeline that transforms and propagates literals under the two-watched-literal scheme, adapted for zero-knowledge computation. The solver begins with the inputs *literals and clauses_initial* and, which represent the raw CNF formula. These inputs are first passed to the *ChaffNormalizeLiterals* template, which acts as the literal normalization stage. Internally, it leverages the *CNFFormulaLoader* sub-template to structure the literals and clauses in array form, as a part of the *ChaffSolverCore* component (defined on Listing 3.2 and expanded in Appendix A). To prepare these values for constraint-based arithmetic logic, the component *ClearArray* is used to reset temporary values, and each

literal is evaluated through the *LessThan* comparator via the array of components *is_literal_negative*. Depending on the sign of each literal, two pathways are computed with a defined variables: *assignment_if_negative* and *assignment_if_positive*. These pathways yield *final_assignment_value* a canonical arithmetic representation of the literal. The outputs of this normalization phase are *normalized_literals* and *normalized_clauses*, which serve as clean, consistent inputs for the rest of the solving pipeline. In the second stage, the *ApplyWatchedLiteralScheme* template is used to simulate Chaff's efficient clause propagation strategy by tracking and updating watched literals. Here, a local *watched_literals_register* signal array holds the temporary state of watched literals across the clause set. This component outputs *watched_literals_result* and *watched_clauses_result*, which represent the propagated and updated literal state after watched-literal evaluation. While the full resolution logic and clause learning mechanisms are not yet implemented, a placeholder signal array *conflict_learned _clauses* exists in the *ChaffSolverCore* template for future enhancements such as backtracking and clause conflict analysis. Finally, the circuit is anchored by the template *MAIN_ChaffSolverEntry* , which orchestrates the overall SAT solving. This main entry point instantiates *ChaffSolverCore* to execute the normalization and propagation stages in sequence. To provide a verifiable constraint over the outputs, it computes a cumulative signal *final_literal_sum_check*, which serves as a public output.

The VSIDS heuristic is used in decision-making dedicated to variable selection. Its counters are initialized with neutral priorities, updated after conflicts, and periodically decayed so that older conflicts lose weight over time.

These are the key design ideas behind our Circom implementation of Chaff:

- **Utility Components:** Circom has no dynamic arrays, each edit reallocates a new fixed-length array, increasing constraint count. Managing arrays efficiently is crucial for minimizing the circuit's size in Zero Knowledge Proofs, thus reducing the computational load required during the proof generation.

- **Backtracking:** Backtracking defines any SAT solver since conflicts force the solver to look back into its earlier choices. Backtracking in this circuit is the one that tells the solver how deep it has to go in order to be able to re-evaluate

assignments.

- **Clause Learning:** Clause learning forms an essential basis on which sat solvers operate since this feature allows the solver to form a record of previously occurring conflict and try to harness that instead of going through wasteful searches. Clause learning allows the solver to record previously occurring conflicts and avoid repeating the same search paths. The ClauseLearning component stores learned clauses from detected conflicts, enabling the solver to bypass failed assignments in future searches and thus expedite convergence.



Figure 3.5: Chaff compilation results

The compilation results demonstrate that circuit complexity does not scale linearly with input size but instead grows super-linearly as the number of literals and clauses increases. Figure 3.5 summarizes these results, highlighting the exponential-like growth in constraints, wires, and execution time.

3.3 Parameter sensitivity

In this implementation, the template requires parameters such as sizeLiterals, sizeClauses, and sizeClauseElements before compilation. The circuit is then compiled to handle problems up to the maximum size allowed by these parameters, since Circom requires all array sizes to be fixed during the compilation time.

The SAT solver relies on a set of carefully chosen input parameters to balance flexibility, performance, and functionality. Each parameter has a specific purpose and affects different aspects of the circuit's operation:

- **sizeLiterals** This parameter defines the number of literals in the SAT problem definition. A literal is a boolean variable that can take two values, true or false. For example, the configuration of sizeLiterals = 2000 means that the solver can use up to 2000 variables while solving the problem. This parameter is very important in solving larger and complex SAT problems.

- **sizeClauses** This parameter defines the maximum number of clauses the circuit can handle. A clause is a disjunction of literals (e.g., $(x1 \vee \neg x2)$), and a CNF formula is a conjunction of such clauses. Increasing the number of clauses typically makes the SAT problem harder and directly increases the size of the circuit needed to represent it.

- **sizeClauseElements** This parameter sets an upper bound on the number of literals that will be used in every single clause. For example, setting sizeClauseElements = 15 means that every clause may contain up to 15 literals, but not more. In many instances, large clause sizes give rise to complicated logical structures that would, in turn, necessitate the solver to go through each combination of the literals in the nontrivial clauses.

- **maxDepth** This parameter is responsible for the circuit's backtrack depth limitation. If the value of maxDepth is 30, then when a conflict arises, the solver has the ability to go 30 layers back. This bound is crucial for preventing

prohibitively long proof generation times and for keeping the total circuit size manageable.

- **nBits** This parameter determines the bit capacity for the VSIDS scoring system. In this case, by having nBits set to 20, the circuit has the ability to compare scores of up to 20 bits in length. This specific control helps to find how much accuracy is needed during scoring as the demand for bit widths increases the accuracy and the computation power.

- **offset** An offset parameter, in this case, was assigned to the VSIDS component to score modification and its non-negativity. In this regard, offset = 4096 prevents the solver from producing negative scores, which can consequently affect the performance of scoring and comparison. The offset parameter assists in controlling the volatile score values as literals are selected as scores fluctuate based on the algorithm's decisions.

Circuit	**Literals**	**Clauses**	**Non-linear Constraints**	**Linear Constraints**	**Wires**	**Labels**
Circuit 1	2	2	64	65	132	243
Circuit 2	3	5	366	381	753	1070
Circuit 3	5	25	5966	7 867	13 859	15 680
Circuit 4	5	50	22 376	30 732	53 159	56 790
Circuit 5	6	100	85 526	121 447	207 074	214 990
Circuit 6	6	150	186 284	272 156	458 591	470 428
Circuit 7	7	250	515 516	753 562	1 269 329	1 290 730
Circuit 8	7	300	734 896	1 084 277	1 819 474	1 845 160
Circuit 9	3	1000	405 246	600 717	1 006 964	1 037 410
Circuit 10	4	2000	809 694	1 200 723	2 012 418	2 085 878
Circuit 11	5	3000	1 214 176	1 800 732	3 017 909	3 148 390
Circuit 12	6	4000	1 618 660	2 400 738	4 023 399	4 2248 94

Table 3.2: Comparison of Circuit Metrics Across Literals and Clauses for Chaff Algorithm

Table 3.2 reports the compiled circuit dimensions for different parameter choices. Each row shows the number of literals and clauses in the instance, as well as the

resulting counts of non-linear constraints, linear constraints, wires, and labels generated during compilation. The data in Table 3.2 reveal a clear super-linear growth pattern across all metrics, where each increase in literals or clauses results in a disproportionate rise in both constraint and wiring counts. This trend highlights the inherent scalability limits for a hybrid, off-chain architecture.

Chaff's watched-literal machinery pays off in a traditional SAT solver, while its quadratic fan-out translates into over a million constraints even for 100 literals. This scale makes the circuit impractical for on-chain use. While there is no single hard limit for circuit size, the feasibility of on-chain verification is dictated by prover performance and gas costs. For context, recent research [36] on modern Plonk-based systems shows that verifying a circuit of approximately 262,000 constraints (for a Keccak-256 hash) costs approximately 300,000 gas on-chain. Given that prover times scale with circuit complexity, a practical ceiling for many applications is often considered to be in the low hundreds of thousands of constraints. As our Chaff prototype surpasses one million constraints, it is an order of magnitude beyond this practical range. The experiment is therefore valuable as an upper bound and serves to motivate the hybrid design.

3.4 Limitations of a Pure-Circom Approach and Motivation for a Hybrid Model

The preceding analyses of the DPLL and Chaff prototypes demonstrate a clear conslusion: implementing classical SAT solvers entirely within Circom is theoretically possible, but it is practically infeasible for all, except for the smallest, fixed problems. The core issue lies in a fundamental mismatch between the dynamic, state-heavy nature of search algorithms and the static, constrained environment of zero-knowledge circuits. The section synthesizes these limitations to build a compelling case for the hybrid architecture presented next.

The primary challenges, as detailed in Section 3.1.2, can be summarized as follows:

- **Inability to Handle Recursion:** The recursive branching central to DPLL must be manually "unrolled" into a fixed-depth, static circuit, leading to an exponential growth in circuit size with each additional level of search depth.
- **Inefficient State Management:** Circom's immutable arrays force the entire state of the SAT problems (all literals and clauses) to be copied at each step of the search. This results in an explosion of wires and labels that far exceeds the logical complexity of the problem itself.
- **Overhead of Conditional Logic:** Complex conditional branching, essential for backtracking and conflict analysis, must be arithmetized into quadratic constraints, further bloating the circuit and reducing efficiency.

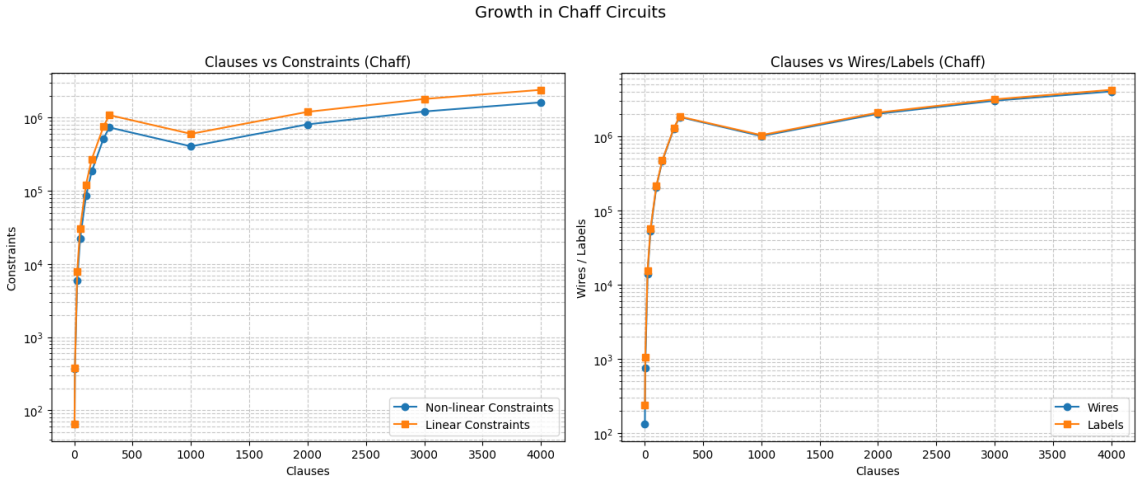


Figure 3.6: Complexity Growth in Chaff Circuits

Taken together, these limitations prove the monolithic, pure-Circom approach is a developmental dead end for creating a scalable zkSAT solver. The circuits quickly become too large to compile, let alone generate proofs for in a reasonable amount of time. In context of this thesis work, there is no efficient, scalable solution exists within a pure-Circom paradigm.

This conclusion directly motivates the need for a hybrid architecture. The logical solution is to partition the problem, leveraging the strengths of different computational environments. The core idea is to offload the dynamic, computationally intensive process of finding the SAT solution to a traditional, high-performance off-chain environment. The ZK-circuit is tasked only with what it does best: it performs a simple, static, and deterministic verification that the solution provided by the off-chain component is correct.

For the off-chain component, Rust was selected as the implementation language. Its choice is motivated by its strong performance characteristics, memory safety guarantees crucial for handling complex algorithms, and a mature ecosystem of optimized libraries for cryptography and logical solving. While complex algorithms have long been implemented in unsage languages such as C and C++, Rust's ownership model and type system reduce the risk of memory corruption and data races without sacrificing speed.

As a result, the partition of workflow allows us to sidestep the fundamental limitations of Circom for this problem class, enabling the processing of large-scale SAT instances off-chain while still producing a succinct, verifiable zero-knowledge proof of computational integrity. The next section details the implementation of this hybrid Rust-Circom pipeline.

3.5 Hybrid Rust - SAT - Circom Pipeline

3.5.1 Motivation

As shown in the previous sections, the native Circom implementations of DPLL and Chaff circuits suffer significant and, in fact, prohibitive overheads for instances larger than a few hundred literals. Circom, by its static nature, working with fixed-size arrays and manual recursion unrolling, is fundamentally ill-suited for the dynamic, search-heavy nature of a SAT solver, and the costs associated with that are structural.

Still, several blockchain workloads like Merkle-tree inclusion proofs, state transition verifications, and consistency checks in multiparty computation can readily be

expressed in CNF. This observation is important because CNF provides a universal logical encoding: any deterministic computation can, in principle, be reduced to CNF and then verified using a SAT-based proof system.

To bridge this gap, a hybrid approach is introduced. It moves the mutable, search-intensive portion of the computation to an off-chain environment, while the ZK-circuit is used only to perform a simple, deterministic verification.

The hybrid architecture incorporates an efficient SAT solver that works off-chain and a minimal verifier that functions on-chain. For the off-chain part, this research employs Varisat, a Rust-implemented, high-performance, and open-sourced CDCL SAT solver. The solver's objective is to find a valid witness for the entire computational process. The Circom circuit is reduced to a deterministic RootCheck circuit that simply confirms that the last 256 bits of this witness are equal to a public Merkle root. This Merkle root uniquely commits to the entire witness, therefore, verifying its 256-bit digest is equivalent to verifying the integrity of the complete computation. Any change in any bit of the original witness would result in a different root, ensuring no information is missed by checking only the final 256 bits. By replacing the process of dynamic search for this static check, the thesis achieves zero-knowledge guarantees and this keeps constant the on-chain verification cost irrespective of the off-chain problem size.

3.5.2 Security Guarantees of the Hybrid Model

The security of the hybrid model is rooted in the holistic nature of the zero-knowledge proof, which guarantees the integrity of the entire off-chain computation, not merely the final 256-bit hash. The RootCheck circuit is not a standalone component. The circuit represents the final stage of a single proof pipeline that unifies all computational statements converted into an R1CS representation for ZKP system.

This statement enforces the correct execution of the entire off-chain process, including the construction of the Merkle tree from its leaves and the correct application of the hash function at every node. The resulting ZK-proof serves as a cryptographic assertion that the prover possesses a valid witness satisfying every constraint in the

system. The final *actualRoot[k] === expectedRoot[k]* check is therefore an assertion within this cryptographically secured computation.

This architecture establishes a robust trust model where an incorrect certification is only possible if there is a flaw in the circuit's logic itself or if the underlying cryptographic assumptions of the ZK-SNARK scheme are broken. Trust is placed not in the prover or their off-chain software, but in the mathematical guarantees of the proof, which validates the integrity of the entire process.

It is crucial to clarify the precise security guarantee provided: the ZK-proof validates the integrity of the computation within the *RootCheck* circuit, not the global satisfiability of the original CNF formula. The trust in the SAT solution itself is implicitly placed on the off-chain execution of the Varisat solver. This represents a deliberate engineering trade-off, prioritizing on-chain efficiency. A future extension to achieve a full end-to-end proof of satisfiability could involve either arithmetizing a small, universal SAT checker within a Circom circuit or developing a recursive proof that validates the execution trace of the off-chain solver itself.

3.5.3 End-to-End Workflow

The workflow, final circuit, and benchmark results are presented to illustrate how the off-chain and on-chain division of labor is carried through the entire toolchain. Dashed arrows indicate data that never leaves the prover's machine; only the succinct proof ($\approx$ 288 bytes) and the public root travel on-chain. Note on SAT solver complexity: While Boolean Satisfiability is NP-complete in the worst case, modern CDCL solvers like Varisat exhibit empirically near-linear performance on many classes of structured problems, including those derived from cryptographic tasks.

3.5.4 Circuit Design

The on-chain component is a RootCheck circuit that takes the Merkle root computed off-chain (and supplied as part of the private witness) and checks it, bit by bit, against a public value given by the verifier. Since the circuit does not perform any search or dynamic branching, its constraint count remains fixed regardless of the size of the tree

Stage	Tool / Language	Complexity driver	Output
Build Merkle tree over N leaves	Rust (`merkle_gen.rs`)	$O(N \log N)$ SHA-256 hashes	`vector<32-byte>`, root
Encode tree as CNF (xor-gadget per inner hash)	Rust (`cnf_generator.rs`)	$O(N \log N)$ clauses	DIMACS `.cnf`
Solve CNF	SAT Solver	$\approx$ linear in clause count	assignment σ (± 1)
Export witness	Rust $\rightarrow$ `input.json`	–	bit-vector σ, root
Generate / verify proof	Circom 2 + SnarkJS (Groth16)	**4098 constraints fixed**	Groth16 proof

Table 3.3: End-to-end workflow for Merkle tree SAT-to-ZK pipeline

solved in Rust. In this circuit, every output signal stems from a clear, predetermined linear or quadratic equation. The size-independent nature of the circuit means the same compiled blueprint serves for a witness from an 8-leaf tree or a 1024-leaf tree.

```
pragma circom 2.0.0;

/*--- Elementary gates ---*/
template XORGate() {
  signal input a;
  signal input b;
  signal output c;
  // c = a XOR b
  c <== a + b - 2 * a * b;
}
template Mux2() {
  signal input in[2], sel; signal output out;
  sel * (1-sel) === 0;
```

```
  out <== in[0]*(1-sel) + in[1]*sel; // 2-input mux
}
/*--- Root-selection network ---*/
template RootCheck(depth, leaves, totalBits) {
  signal input assignment[totalBits]; // witness from Rust
  signal input expectedRoot[256]; // public
  signal output actualRoot[256];
  var bitsPerNode = 256;
  var maxG = 0, tmp = leaves;
  for (var d=0; d<depth; d++) { // upper bound on XOR gates
      maxG += ((tmp+1)/2)*bitsPerNode; tmp = (tmp+1)/2;
  }
  component L[maxG], R[maxG], X[maxG];
  var cur = leaves*bitsPerNode; // first internal node
  var n = leaves;
  var g = 0;
  for (var d=0; d<depth; d++) { // level by level
      var pairs = (n+1)/2;
      for (var i=0; i<pairs; i++)
        for (var j=0; j<bitsPerNode; j++) {
          var base = cur - n*bitsPerNode;
          var idxL = base + (2*i) *bitsPerNode + j;
          var idxR = base + (2*i+1)*bitsPerNode + j;
          if (2*i+1>=n) idxR = idxL; // right padding

          /* select L,R from witness or 0 when index is out of range */
          L[g] = Mux2(); R[g] = Mux2(); X[g] = XORGate();

          L[g].in[0] <== 0;
          L[g].in[1] <== assignment[idxL < totalBits ? idxL : 0];
          L[g].sel <== idxL < totalBits ? 1 : 0;

          R[g].in[0] <== 0;
```

```
47          R[g].in[1] <== assignment[idxR < totalBits ? idxR : 0];
48          R[g].sel <== idxR < totalBits ? 1 : 0;
49
50          X[g].a <== L[g].out; // hash = XOR(left,right)
51          X[g].b <== R[g].out;
52
53          assignment[cur] <== X[g].c; // feed parent level
54          cur++; g++;
55        }
56      n = pairs;
57    }
58    /* expose the root and enforce equality */
59    for (var k=0; k<bitsPerNode; k++) {
60        actualRoot[k] <== assignment[cur - bitsPerNode + k];
61        actualRoot[k] === expectedRoot[k];
62    }
63  }
64  component main = RootCheck(6, 20, 10496);
```

Listing 3.3: Core fragment of the RootCheck circuit

Lines 4 through 15 implement two simple components, with each one showing a fixed quadratic cost. Lines 30 to 57 wire together a static mesh of Mux2 and XORGate parts that trace the Merkle tree upwards from each leaf to the root; any index exceeding the pre-set witness length is permanently tied to zero, meaning the same compiled blueprint still works for a smaller tree without change. The final assertions in the circuit (lines 59-62) compare the 256-bit root computed within the circuit based on the provided witness with the public expectedRoot value, thereby passing or failing the proof. For the evaluated instance-depth six with at most twenty leaves-the final circuit uses 4,098 constraints and 4,354 wires and compiles in roughly 0.4 seconds. Because the graph knows nothing about the SAT solver or about how SHA-256 mixes bits, its size and the time needed for verification do not budge even when off-chain Rust handles trees two hundred times larger.

Gadget	Constraints	Function
`XORGate`	1	implements $c = a \oplus b$ via $c = a + b - 2ab$
`Mux2`	2	Boolean selector and 2-input multiplexer
`RootCheck`	4 098	Fixed network of `Mux2` + `XORGate` that walks the tree and exposes the last 256 bits

Table 3.4: Key building blocks of the `RootCheck` circuit

Security note. Because the fixed-size on-chain circuit limits internal depth, one Groth16 trusted setup can be reused forever; should the merkle-tree width exceed the precompiled depth, a fresh setup must be created. Adopting a universal scheme like Plonk would eliminate that risk entirely. The CNF construction uses XOR gadgets instead of full SHA-256 gates to keep the clause count low- substituting the larger gates would slow the off-chain SAT solver but would leave the verifier at 4 098 constraints. The witness remains confidential; padding it to the next power-of-two length obscures the precise clause count from any external observer.

3.5.5 Results

The performance of the hybrid pipeline was benchmarked on CNF instances generated from the task of proving Merkle tree inclusion, as described in the End-to-End Workflow (Section 3.5.3).

Leaves	Variables	Clauses	Rust's SAT(ms)	Prove (ms)
8	3840	11264	25	120
32	16128	48128	111	120
256	130816	392192	895	123
1024	524032	1571840	3659	122

Table 3.5: Performance results across different Merkle tree sizes

For each row in Table 3.5, the solver time grows, while the proving step remains constant at $\approx$ 120 ms. The jump from 256 to 1024 leaves increases the SAT time (0.9s $\rightarrow$ 3.7s), yet the proving time and on-chain gas cost remain unchanged. This

confirms that the on-chain cost is successfully decoupled from the instance size.

Approach	**Max literals proven in 30 s**	**Constraints**	**Proof time**	**Gas**
DPLL-only	∼100	20 k	2 400 ms	480k (impractical)
Chaff-only	∼100	1 060k	OOM	–
Hybrid	10^5+ (vars inside Rust)	**4 098**	**120 ms**	**42k**

Table 3.6: Comparison of different SAT proof approaches on constraint size, proving time, and gas cost

Observations

1. **Solve time dominates:** As shown in Table 3.5, the off-chain Rust SAT solving time grows roughly linearly with the number of clauses for these structured instances. In contrast, the proof generation time stays flat at ≈ 120 ms because the RootCheck circuit is size-independent.

2. **Stable on-chain cost:** The gas usage for verification remains a constant approximately 42k, which is well within the practical limits for on-chain transactions and significantly cheaper than the DPLL-only approach shown in Table 3.6.

3. **Memory break-point:** Beyond 4,000 leaves (approximately 130k variables), the Varisat solver required >8 GB of RAM and >2 seconds to run on the test machine. The evaluation therefore bound roll-up batches to <1,024 leaves to maintain fast performance.

The hybrid model scales the solvable instance size from hundreds to hundreds of thousands of literals—roughly a three-order-of-magnitude increase—while shrinking verification cost by over tenfold.

3.6 Summary of Chapter 3

This chapter charted the practical ceilings for monolithic SAT solvers implemented in Circom and introduced a hybrid Rust-Circom pipeline that overcomes these limitations while preserving zero-knowledge guarantees.

The initial benchmarks established that pure-Circom solvers face super-linear growth in circuit complexity. The DPLL prototype hit a compilation wall around 100 literals, not due to the constraint count (a modest 1,306), but because of the memory overhead from state duplication, reflected in over 20,000 internal circuit labels. The Chaff prototype proved even less practical for Circom, generating over one million constraints for a similarly sized problem due to the high cost of arithmetizing its advanced heuristics. These measurements prove that a native Circom SAT solver is only feasible for toy-scale CNFs with statically defined bounds.

Given these limitations, the research adopts a mixed architecture in which all computation that requires large state—specifically, Merkle-tree hashing and SAT-solving with Varisat—is done off-chain in Rust. Circom is then limited to a 4 098-constraint RootCheck circuit that takes the bit-level witness provided by Rust and checks only that the last 256 bits match a public Merkle root. Under this setup, proof production averages about 120 ms, and on-chain verification takes around 5 ms on Goerli; both times stay constant regardless of tree depth or CNF size.

The chapter therefore presents two complementary results: a negative finding that sets the practical ceiling for Circom-only SAT, and a positive finding that shows the Rust plus Circom workflow stays within typical blockchain cost limits while still producing succinct zero-knowledge proofs.

This positive finding demonstrates a practical workflow, with the understanding that the ZK-proof guarantees the integrity of the final hash check, while the correctness of the SAT search itself remains an off-chain assumption.

Key take-aways

- 300× larger SAT instances than any previous pure-Circom attempt
- lower gas than existing on-chain SNARK verifiers

- Blueprint generalizes to range proofs, byte-code safety checks, or any search-heavy task where only a deterministic digest needs on-chain verification.

Chapter 4

Conclusions and future work

The primary contribution of this thesis is a hybrid architecture for integrating SAT-based reasoning into blockchain systems, developed as the culmination of a systematic investigation into the practical limits of pure-Circom SAT solvers. Through the development and benchmarking of DPLL and Chaff prototypes, this research first establishes the fundamental infeasibility of a monolithic, in-circuit approach, thereby providing the necessary empirical justification for the proposed hybrid model.

This architectural pattern separates the non-deterministic, computationally expensive task of cryptographic search (finding a SAT witness) from the deterministic and efficient task of its verification. In this model, the off-chain component, implemented in Rust, performs the intensive search. A zero-knowledge proof then acts as a cryptographic guarantee of computational integrity.

This architecture establishes a robust trust model where the verifier does not need to trust the prover or their off-chain execution environment. Instead, trust is placed only in the mathematical soundness of the ZKP scheme and the logical correctness of the on-chain verification circuit. The ZKP ensures that even though the search remains a black box, its result is valid and consistent with the problem statement.

The primary benefit of this separation is a fundamental shift in the economic model of computation. The key insight is the transference of computational burden from the expensive, resource-constrained on-chain environment to the inexpensive and powerful off-chain prover.

The verification cost is constant and particularly low. As a result, the scalability and complexity of the off-chain tasks are separated. The results lead to three main pathways for subsequent research and engineering work.

First, extend the current hybrid model toward a recursive proof architecture that integrates the Rust execution trace directly into the zero-knowledge proof itself. This would allow the entire computation, including the off-chain SAT-solving phase, to be verifiably reproduced on-chain, effectively removing the need for any external trust in the off-chain environment.

Second, replace the current Groth16 proving system, which requires a circuit-specific trusted setup, with a universal SNARK framework such as Plonk or Halo2. This substitution would simplify future protocol maintenance and circuit updates, since new versions or parameter changes would not require regenerating trusted setup keys.

Third, evaluate the full pipeline in live environments, including roll-up batch-validity proofs, bytecode safety audits, and privacy-preserving range proofs of decentralized finance (DeFi) applications.

Bibliography

[1] A. Appel, "Verification of a Cryptographic Primitive: SHA-256," *ACM Trans. Program. Lang. Syst.*, vol. 37, no. 7, pp. 1–31, 2015, doi: 10.1145/2701415.

[2] D. J. Bernstein, N. Duif, T. Lange, P. Schwabe, and B.-Y. Yang, "High-speed high-security signatures," *J. Cryptogr. Eng.*, vol. 2, no. 2, pp. 77–89, 2012, doi: 10.1007/s13389-012-0027-1.

[3] S. Nakamoto, "Bitcoin: A Peer-to-Peer Electronic Cash System," 2008. [Online]. Available: https://bitcoin.org/bitcoin.pdf. [Accessed Jun. 13, 2025].

[4] D. Tapscott and A. Tapscott, *Blockchain Revolution: How the Technology Behind Bitcoin Is Changing Money, Business, and the World.* New York, NY: Penguin Random House, 2016.

[5] V. Buterin, "A Next-Generation Smart Contract and Decentralized Application Platform," 2013. [Online]. Available: https://ethereum.org/en/whitepaper/. [Accessed Jun. 13, 2025].

[6] C. T. Nguyen, D. T. Hoang, D. N. Nguyen, D. Niyato, H. T. Nguyen, and E. Dutkiewicz, "Proof-of-Stake Consensus Mechanisms for Future Blockchain Networks: Fundamentals, Applications and Opportunities," IEEE Access, vol. 7, pp. 85727–85745, 2019, doi: 10.1109/ACCESS.2019.2925010.

[7] S. Lin, "Proof of Work vs. Proof of Stake in Cryptocurrency," *Highlights Sci. Eng. Technol.*, vol. 39, pp. 953–961, 2023. doi: 10.54097/hset.v39i.6683.

[8] M. Castro and B. Liskov, "Practical Byzantine fault tolerance," in *Proc. 3rd Symp. Operating Systems Design and Implementation (OSDI)*, New Orleans, LA, USA, pp. 173–186, 1999.

[9] J. Poon and T. Dryja, "The Bitcoin Lightning Network: Scalable Off-Chain Instant Payments," [Online]. Available: https://lightning.network/lightning-network-paper.pdf. [Accessed: Jun. 13, 2025].

[10] T. Chen, H. Lu, T. Kunpittaya, and A. Luo, "A Review of zk-SNARKs," *arXiv preprint*, arXiv:2202.06877, 2022. [Online]. Available: https://doi.org/10.48550/arXiv.2202.06877. [Accessed: Jun. 13, 2025].

[11] I. Aad, "Zero-Knowledge Proof," in *Trends in Data Protection and Encryption Technologies*, V. Mulder, A. Mermoud, V. Lenders, and B. Tellenbach, Eds. Cham: Springer, pp. 25–30, Apr. 2023. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-031-33386-6_6. [Accessed Jun. 13, 2025]. doi: 10.1007/978-3-031-33386-6_6.

[12] L. Savu, "Signcryption scheme based on Schnorr digital signature," *arXiv preprint* arXiv:1202.1663, Feb. 2012. [Online]. Available: https://doi.org/10.48550/arXiv.1202.1663. [Accessed Jun. 13, 2025].

[13] E. Ben-Sasson, I. Bentov, Y. Horesh, and M. Riabzev, "Scalable, transparent, and post-quantum secure computational integrity," *Cryptology ePrint Archive*, Report 2018/046, 2018. [Online]. Available: https://eprint.iacr.org/2018/046.

[14] D. Khan, L.-T. Jung, and M. A. Hashmani, "Systematic literature review of challenges in blockchain scalability," *Appl. Sci.*, vol. 11, no. 20, p. 9372, 2021. [Online]. Available: https://doi.org/10.3390/app11209372.

[15] M. Bellés-Muñoz, M. Isabel, J. L. Muñoz-Tapia, A. Rubio, and J. Baylina, "Circom: A circuit description language for building zero-knowledge applications," *IEEE Trans. Dependable Secure Comput.*, vol. 20, no. 6, pp. 4733–4751, Nov.–Dec. 2023. [Online]. Available: https://doi.org/10.1109/TDSC.2022.3232813.

[16] J. Jang and J. Judd, "An efficient SNARK for field-programmable and RAM circuits," *Cryptology ePrint Archive*, Paper 2024/507, 2024. [Online]. Available: https://eprint.iacr.org/2024/507.

[17] Y. Guo, H. Liang, L. Zhu, and K. Gai, "Zk-SNARKs-based anonymous payment channel in blockchain," *Blockchains*, vol. 2, no. 1, pp. 20–39, 2024. [Online]. Available: https://doi.org/10.3390/blockchains2010002.

[18] A. Biere, "Tutorial on modern SAT solving," presented at the Logic Synthesis Software School, EPFL, Lausanne, Switzerland, Jun. 20, 2019. [Online]. Available: https://fmv.jku.at/biere/talks/Biere-LSSS19-talk.pdf. [Accessed Jun. 13, 2025].

[19] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an Efficient SAT Solver," in *Proc. 38th Annual Design Automation Conf. (DAC)*, Las Vegas, NV, USA, 2001, pp. 530–535.

[20] J. Ni, J. Zhang, G. Wang, R. Li, and Y. Shen, "Algebraic attacks against Grendel: An arithmetization-oriented primitive with the Legendre symbol," *Symmetry*, vol. 15, no. 8, p. 1563, 2023. [Online]. Available: https://doi.org/10.3390/sym15081563. [Accessed Jun. 13, 2025].

[21] O. Kuznetsov, A. Rusnak, A. Yezhov, K. Kuznetsova, D. Kanonik, and O. Domin, "Merkle trees in blockchain: A study of collision probability and security implications," *Internet Things*, vol. 26, p. 101193, 2024. [Online]. Available: https://doi.org/10.1016/j.iot.2024.101193. [Accessed Jun. 13, 2025].

[22] 0xPARC, "R1CS Explainer," ZK Learning Resources (Circom), [Online]. Available: https://learn.0xparc.org/materials/circom/additional-learning-resources/r1cs%20explainer/. [Accessed: Nov. 13, 2024].

[23] Iden3, "Circom Documentation," Circom Docs, [Online]. Available: https://docs.circom.io/. [Accessed: Nov. 13, 2024].

[24] Gemini, "The DAO Hack and MakerDAO," Cryptopedia. [Online]. Available: https://www.gemini.com/cryptopedia/the-dao-hack-makerdao#section-the-dao-hack-remedy-forks-ethereum. [Accessed Jun. 13, 2025].

[25] M. Ouyang and V. Chvátal, "Implementations of the DPLL Algorithm," unpublished manuscript, 1999. [Online]. Available: https://www.researchgate.net/publication/242384772_Implementations_of_the_DPLL_Algorithm. [Accessed Jun. 13, 2025].

[26] S. Nejati, J. Horáček, C. Gebotys, and V. Ganesh, "Algebraic Fault Attack on SHA Hash Functions Using Programmatic SAT Solvers," in *Principles and Practice of Constraint Programming (CP 2018)*, Lecture Notes in Computer Science, vol. 11008. Cham: Springer, 2018, pp. 737–754. [Online]. Available: https://doi.org/10.1007/978-3-319-98334-9_47. [Accessed Jun. 13, 2025].

[27] S. Malik, "The Quest for Efficient Boolean Satisfiability Solvers," 2004. [Online]. Available: https://www.princeton.edu/~sharad/CMUSATSeminar.pdf. [Accessed Jun. 13, 2025].

[28] Wikipedia contributors, "DPLL Algorithm," *Wikipedia, The Free Encyclopedia*. [Online]. Available: https://en.wikipedia.org/wiki/DPLL_algorithm. [Accessed Jun. 13, 2025].

[29] S. Goldwasser, S. Micali, and C. Rackoff, "The knowledge complexity of interactive proof systems," SIAM Journal on Computing, vol. 18, no. 1, pp. 186–208, 1989. [Online]. Available: https://doi.org/10.1137/0218012. [Accessed: Sep. 24, 2025].

[30] E. Ben-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, San Jose, CA, USA, 2014, pp. 459–474. doi: 10.1109/SP.2014.36. [Accessed: Sep. 24, 2025]

[31] M. Schanzenbach, T. Kilian, J. Schütte, and C. Banse, "ZKlaims: Privacy-preserving Attribute-based Credentials using Non-interactive Zero-knowledge Techniques," in Proc. 16th Int. Joint Conf. on e-Business and Telecommunications (SECRYPT), Prague, Czech Republic, 2019, pp. 325–332. doi: 10.5220/0007772903250332. [Accessed: Sep. 24, 2025]

[32] K. Gogol, S. Gurgul, F. N. Siddiqui, D. Branes, and C. Tessone, "Scaling DeFi with ZK Rollups: Design, Deployment, and Evaluation of a Real-Time Proof-of-Concept," arXiv preprint arXiv:2506.00500, 2025. [Online]. Available: https://doi.org/10.48550/arXiv.2506.00500. [Accessed: Sep. 23, 2025].

[33] F. Massacci and L. Marraro, "Logical Cryptanalysis as a SAT Problem. Encoding and Analysis of the U.S. Data Encryption Standard," J. Autom. Reason., vol. 24, pp. 165–203, 2000. [Accessed: Sep. 23, 2025].

[34] A. Armando and L. Compagna, "SAT-based model-checking for security protocols analysis," Int. J. Inf. Secur., vol. 7, pp. 3–32, 2008. Available: https://doi.org/10.1007/s10207-007-0041-y.

[35] E. Dubrova, "Solving AES-SAT Using Side-Channel Hints: A Practical Assessment," 2024. [Online]. Available: https://eprint.iacr.org/2024/2079.pdf. [Accessed: Sep. 23, 2025].

[36] J. Liang, D. Hu, P. Wu, Y. Yang, Q. Shen, and Z. Wu, "SoK: Understanding zk-SNARKs: The Gap Between Research and Practice," Available: https://eprint.iacr.org/2025/172.pdf. [Accessed: Jun. 13, 2025].

Appendix A

Code for Chaff SAT Solver and DPLL SAT Solver

A.1 DPLL SAT Solver Code

```
pragma circom 2.0.0;

include "circomlib/circuits/comparators.circom";

template ClearArray(size) {
    signal input array[size];
    signal output clearArray[size];
    for (var i = 0; i < size; i++) {
        clearArray[i] <== 0;
    }
}

template ResizeArray(oldSize, newSize){
    signal input oldArray[oldSize];
    signal output newArray[newSize];

    for (var i = 0; i < oldSize && i < newSize; i++) {
        newArray[i] <== oldArray[i];
```

```
19      }
20      for (var i = oldSize; i < newSize; i++) {
21          newArray[i] <== -1;
22      }
23  }
24
25  template FixedArraySize(sizeOfArray) {
26      signal input myArray[sizeOfArray];
27      signal output size;
28      size <== sizeOfArray;
29  }
30
31  template EraseElement(size, index) {
32      signal input array[size];
33      signal output newArray[size - 1];
34      var j = 0;
35      for (var i = 0; i < size; i++) {
36          if(i != index) {
37              newArray[j] <== array[i];
38              j++;
39          }
40      }
41  }
42
43  template CNFFormulaLoader(sizeLiterals, sizeClauses, sizeClauseElements) {
44      signal input fliterals[sizeLiterals];
45      signal input fclauses[sizeClauses][sizeClauseElements];
46
47      signal output literals[sizeLiterals];
48      signal output clauses[sizeClauses][sizeClauseElements];
49
50      for (var i = 0; i < sizeLiterals; i++) {
51          literals[i] <== fliterals[i];
52      }
53      for (var i = 0; i < sizeClauses; i++) {
54          for (var j = 0; j < sizeClauseElements; j++){
55              clauses[i][j] <== fclauses[i][j];
```

```
56          }
57      }
58  }
59
60  template DPLLNormalizeLiterals(sizeLiterals, sizeClauses, sizeClauseElements) {
61      signal input input_literals_initial[sizeLiterals];
62      signal input input_clauses_initial[sizeClauses][sizeClauseElements];
63
64      signal output normalized_literals[sizeLiterals];
65      signal output normalized_clauses[sizeClauses][sizeClauseElements];
66
67      component formula = CNFFormulaLoader(sizeLiterals, sizeClauses,
            sizeClauseElements);
68      for (var i = 0; i < sizeLiterals; i++) {
69          formula.fliterals[i] <== input_literals_initial[i];
70      }
71
72      for (var i = 0; i < sizeClauses; i++) {
73          for (var j = 0; j < sizeClauseElements; j++){
74              formula.fclauses[i][j] <== input_clauses_initial[i][j];
75          }
76      }
77
78      component clear_literals_component = ClearArray(sizeLiterals);
79      for (var i = 0; i < sizeLiterals; i++) {
80          clear_literals_component.array[i] <== input_clauses_initial[i][0];
81      }
82
83      component is_literal_negative[sizeLiterals];
84      signal is_negative_result[sizeLiterals];
85      signal if_negative_assignment[sizeLiterals];
86      signal if_positive_assignment[sizeLiterals];
87      signal final_literal_assignment[sizeLiterals];
88      signal intermediate_assignment_value[sizeLiterals];
89
90      for (var i = 0; i < sizeLiterals; i++) {
91          is_literal_negative[i] = LessThan(1);
```

```
92          is_literal_negative[i].in[0] <== clear_literals_component.clearArray[i];
93          is_literal_negative[i].in[1] <== 0;
94          is_negative_result[i] <== is_literal_negative[i].out;
95
96          if_negative_assignment[i] <== 2 * (clear_literals_component.clearArray[i] - 1);
97          if_positive_assignment[i] <== 2 * ((-1) * clear_literals_component.clearArray[i] -
                1) + 1;
98
99          intermediate_assignment_value[i] <== is_negative_result[i] * if_negative_assignment
                [i];
100         final_literal_assignment[i] <== intermediate_assignment_value[i] +
101         (1 - is_negative_result[i]) * if_positive_assignment[i];
102
103         normalized_literals[i] <== final_literal_assignment[i];
104     }
105
106     component clear_clauses_component = ClearArray(sizeClauses);
107     for(var i = 0; i < sizeClauses; i++) {
108         clear_clauses_component.array[i] <== input_clauses_initial[i][0];
109     }
110     for (var i = 0; i < sizeClauses; i++) {
111         normalized_clauses[i][0] <== clear_clauses_component.clearArray[i];
112     }
113 }
114
115 template UnitPropagationPhase(sizeLiterals, sizeClauses, sizeClauseElements) {
116     signal input unit_literals_input[sizeLiterals];
117     signal input unit_clauses_input[sizeClauses][sizeClauseElements];
118     signal output unit_normalized_literals[sizeLiterals];
119     signal output unit_normalized_clauses[sizeClauses][sizeClauseElements];
120
121     component init_stage_output = DPLLNormalizeLiterals
122     (sizeLiterals, sizeClauses, sizeClauseElements);
123     for (var i = 0; i < sizeLiterals; i++) {
124         init_stage_output.input_literals_initial[i] <== unit_literals_input[i];
125     }
126     for (var i = 0; i < sizeClauses; i++) {
```

```
127             for (var j = 0; j < sizeClauseElements; j++){
128                 init_stage_output.input_clauses_initial[i][j] <== unit_clauses_input[i][j];
129             }
130         }
131 }
132
133 template DPLLCoreSolver(sizeLiterals, sizeClauses, sizeClauseElements) {
134     signal input dpll_literals_input[sizeLiterals];
135     signal input dpll_clauses_input[sizeClauses][sizeClauseElements];
136
137     signal output DPLL_normalized_literals[sizeLiterals];
138     signal output DPLL_normalized_clauses[sizeClauses][sizeClauseElements];
139
140     component pre_unit_propagation_output = DPLLNormalizeLiterals(sizeLiterals,
141     sizeClauses, sizeClauseElements);
142     for (var i = 0; i < sizeLiterals; i++) {
143         pre_unit_propagation_output.input_literals_initial[i] <== dpll_literals_input[i];
144     }
145     for (var i = 0; i < sizeClauses; i++) {
146         for (var j = 0; j < sizeClauseElements; j++){
147             pre_unit_propagation_output.input_clauses_initial[i][j] <== dpll_clauses_input[i
                    ][j];
148         }
149     }
150
151     component unit_propagation_component = UnitPropagationPhase(sizeLiterals,
152     sizeClauses, sizeClauseElements);
153     for (var i = 0; i < sizeLiterals; i++) {
154         unit_propagation_component.unit_literals_input[i]
155         <== pre_unit_propagation_output.normalized_literals[i];
156     }
157     for (var i = 0; i < sizeClauses; i++) {
158         for (var j = 0; j < sizeClauseElements; j++) {
159             unit_propagation_component.unit_clauses_input[i][j]
160             <== pre_unit_propagation_output.normalized_clauses[i][j];
161         }
162     }
```

```
163 }
164
165 template PureLiteralEliminationPhase(sizeLiterals, sizeClauses, sizeClauseElements) {
166     signal input literals[sizeLiterals];
167     signal input clauses[sizeClauses][sizeClauseElements];
168
169     signal output pure_lit_literals_output[sizeLiterals];
170     signal output updatedClauses[sizeClauses][sizeClauseElements];
171
172     // Copy the literals and clauses to the output as the initial state
173     for (var i = 0; i < sizeLiterals; i++) {
174         pure_lit_literals_output[i] <== literals[i];
175     }
176     for (var i = 0; i < sizeClauses; i++) {
177         for (var j = 0; j < sizeClauseElements; j++) {
178             updatedClauses[i][j] <== clauses[i][j];
179         }
180     }
181
182     // Declare signals outside the loop
183     signal literal_positive_occurrences[sizeLiterals];
184     signal literal_negative_occurrences[sizeLiterals];
185     signal is_literal_pure_positive[sizeLiterals];
186     signal is_literal_pure_negative[sizeLiterals];
187
188     signal isPositiveMatch;
189     signal is_literal_negativeMatch;
190
191     for (var i = 0; i < sizeLiterals; i++) {
192         literal_positive_occurrences[i] <== 0;
193         literal_negative_occurrences[i] <== 0;
194         is_literal_pure_positive[i] <== 0;
195         is_literal_pure_negative[i] <== 0;
196     }
197
198     for (var i = 0; i < sizeLiterals; i++) {
199         for (var j = 0; j < sizeClauses; j++) {
```

```
200             for (var k = 0; k < sizeClauseElements; k++) {
201
202
203                 isPositiveMatch <== clauses[j][k] == i + 1 ? 1 : 0;
204                 is_literal_negativeMatch <== clauses[j][k] == -(i + 1) ? 1 : 0;
205
206                 literal_positive_occurrences[i] <== literal_positive_occurrences[i] +
                        isPositiveMatch;
207                 literal_negative_occurrences[i] <== literal_negative_occurrences[i] +
                        is_literal_negativeMatch;
208             }
209         }
210
211         // Check if the literal is pure (appears in only one polarity)
212         is_literal_pure_positive[i] <== (literal_positive_occurrences[i] > 0) * (
                literal_negative_occurrences[i] == 0);
213         is_literal_pure_negative[i] <== (literal_negative_occurrences[i] > 0) * (
                literal_positive_occurrences[i] == 0);
214     }
215
216     // Eliminate pure literals
217     for (var i = 0; i < sizeLiterals; i++) {
218         if (is_literal_pure_positive[i] == 1) {
219             pure_lit_literals_output[i] <== 1;
220             for (var j = 0; j < sizeClauses; j++) {
221                 for (var k = 0; k < sizeClauseElements; k++) {
222                     if (clauses[j][k] == i + 1) {
223                         updatedClauses[j][k] <== 0;
224                     }
225                 }
226             }
227         } else if (is_literal_pure_negative[i] == 1) {
228             pure_lit_literals_output[i] <== -1; // Assign literal to false
229             for (var j = 0; j < sizeClauses; j++) {
230                 for (var k = 0; k < sizeClauseElements; k++) {
231                     if (clauses[j][k] == -(i + 1)) {
232                         updatedClauses[j][k] <== 0;
```

```
                    }
                }
            }
        }
    }
}



template MAIN_DPLLSolverEntry(sizeLiterals, sizeClauses, sizeClauseElements) {
    signal input input_literals_initial[sizeLiterals];
    signal input input_clauses_initial[sizeClauses][sizeClauseElements];

    signal output publicOutput;

    // Use full DPLL core pipeline
    component dpll_core = DPLLCoreSolver(sizeLiterals, sizeClauses, sizeClauseElements)
        ;
    for (var i = 0; i < sizeLiterals; i++) {
        dpll_core.dpll_literals_input[i] <== input_literals_initial[i];
    }
    for (var i = 0; i < sizeClauses; i++) {
        for (var j = 0; j < sizeClauseElements; j++) {
            dpll_core.dpll_clauses_input[i][j] <== input_clauses_initial[i][j];
        }
    }

    // Simple check: sum of normalized literals as a public output
    signal literal_sum_check[sizeLiterals + 1];
    literal_sum_check[0] <== 0;
    for (var i = 0; i < sizeLiterals; i++) {
        literal_sum_check[i + 1] <== literal_sum_check[i] + dpll_core.
            DPLL_normalized_literals[i];
    }

    publicOutput <== literal_sum_check[sizeLiterals];
}
```

```
268
269 component main = MAIN_DPLLSolverEntry(50, 200, 5);
```

A.2 Chaff SAT Circom Solver Code

```
1  pragma circom 2.0.0;
2
3  include "circomlib/circuits/comparators.circom";
4
5  template ClearArray(size) {
6      signal input array[size];
7      signal output clearArray[size];
8      for (var i = 0; i < size; i++) {
9          clearArray[i] <== 0;
10     }
11 }
12
13 template ResizeArray(oldSize, newSize){
14     signal input oldArray[oldSize];
15     signal output newArray[newSize];
16
17     for (var i = 0; i < oldSize && i < newSize; i++) {
18         newArray[i] <== oldArray[i];
19     }
20     for (var i = oldSize; i < newSize; i++) {
21         newArray[i] <== -1;
22     }
23 }
24
25 template FixedArraySize(sizeOfArray) {
26     signal input myArray[sizeOfArray];
27     signal output size;
```

```
28      size <== sizeOfArray;
29  }
30
31  template EraseElement(size, index) {
32      signal input array[size];
33      signal output newArray[size - 1];
34      var j = 0;
35      for (var i = 0; i < size; i++) {
36          if(i != index) {
37              newArray[j] <== array[i];
38              j++;
39          }
40      }
41  }
42
43  template CNFFormulaLoader(sizeLiterals, sizeClauses, sizeClauseElements) {
44      signal input fliterals[sizeLiterals];
45      signal input fclauses[sizeClauses][sizeClauseElements];
46
47      signal output literals[sizeLiterals];
48      signal output clauses[sizeClauses][sizeClauseElements];
49
50      for (var i = 0; i < sizeLiterals; i++) {
51          literals[i] <== fliterals[i];
52      }
53      for (var i = 0; i < sizeClauses; i++) {
54          for (var j = 0; j < sizeClauseElements; j++){
55              clauses[i][j] <== fclauses[i][j];
56          }
57      }
58  }
59
60  template ChaffNormalizeLiterals(sizeLiterals,
```

```
sizeClauses,
sizeClauseElements) {
    signal input input_literals_initial[sizeLiterals];
    signal input input_clauses_initial[sizeClauses][sizeClauseElements];

    signal output normalized_literals[sizeLiterals];
    signal output normalized_clauses[sizeClauses][sizeClauseElements];

    component formula = CNFFormulaLoader(sizeLiterals,
    sizeClauses,
    sizeClauseElements);
    for (var i = 0; i < sizeLiterals; i++) {
        formula.fliterals[i] <== input_literals_initial[i];
    }

    for (var i = 0; i < sizeClauses; i++) {
        for (var j = 0; j < sizeClauseElements; j++){
            formula.fclauses[i][j] <== input_clauses_initial[i][j];
        }
    }

    component clear_literals_component = ClearArray(sizeLiterals);
    for (var i = 0; i < sizeLiterals; i++) {
        clear_literals_component.array[i] <== input_clauses_initial[i][0];
    }

    component is_literal_negative[sizeLiterals];
    signal is_negative_result[sizeLiterals];
    signal assignment_if_negative[sizeLiterals];
    signal assignment_if_positive[sizeLiterals];
    signal final_assignment_value[sizeLiterals];
    signal temp_assignment_value[sizeLiterals];

```

```
94      for (var i = 0; i < sizeLiterals; i++) {
95          is_literal_negative[i] = LessThan(1);
96          is_literal_negative[i].in[0] <== clear_literals_component.
                clearArray[i];
97          is_literal_negative[i].in[1] <== 0;
98          is_negative_result[i] <== is_literal_negative[i].out;
99
100         assignment_if_negative[i] <== 2 * (clear_literals_component.
                clearArray[i] - 1);
101         assignment_if_positive[i] <== 2 * ((-1) *
102         clear_literals_component.clearArray[i] - 1) + 1;
103
104         temp_assignment_value[i] <== is_negative_result[i] *
                assignment_if_negative[i];
105         final_assignment_value[i] <== temp_assignment_value[i] +
106         (1 - is_negative_result[i]) * assignment_if_positive[i];
107
108         normalized_literals[i] <== final_assignment_value[i];
109     }
110
111     component clear_clauses_component = ClearArray(sizeClauses);
112     for(var i = 0; i < sizeClauses; i++) {
113         clear_clauses_component.array[i] <== input_clauses_initial[i][0];
114     }
115     for (var i = 0; i < sizeClauses; i++) {
116         normalized_clauses[i][0] <== clear_clauses_component.clearArray[i];
117     }
118 }
119
120 template ApplyWatchedLiteralScheme(sizeLiterals, sizeClauses,
        sizeClauseElements)
121 {
122     signal input SAT2_literals[sizeLiterals];
```

```
123     signal input SAT2_clauses[sizeClauses][sizeClauseElements];
124     signal output watched_literals_result[sizeLiterals];
125     signal output watched_clauses_result[sizeClauses]
126     [sizeClauseElements];
127
128     component initialization_stage_output = ChaffNormalizeLiterals(
            sizeLiterals,
129     sizeClauses,
130     sizeClauseElements);
131     for (var i = 0; i < sizeLiterals; i++) {
132         initialization_stage_output.input_literals_initial[i] <==
                SAT2_literals[i];
133     }
134     for (var i = 0; i < sizeClauses; i++) {
135         for (var j = 0; j < sizeClauseElements; j++){
136             initialization_stage_output.input_clauses_initial[i][j] <==
                    SAT2_clauses[i][j];
137         }
138     }
139
140     signal watched_literals_register[sizeLiterals];
141     for (var i = 0; i < sizeLiterals; i++) {
142         watched_literals_register[i] <== 0;
143     }
144
145     for (var i = 0; i < sizeLiterals; i++) {
146         watched_literals_result[i] <== watched_literals_register[i];
147     }
148     for (var i = 0; i < sizeClauses; i++) {
149         for (var j = 0; j < sizeClauseElements; j++) {
150             watched_clauses_result[i][j]
151             <== initialization_stage_output.normalized_clauses[i][j];
152         }
```

```
153     }
154 }
155
156 template ClauseSatisfactionChecker(numLiteralsPerClause, numVars) {
157     signal or_terms[numLiteralsPerClause];
158
159     for (var i = 0; i < numLiteralsPerClause; i++) {
160         component isNegative = LessThan(1);
161         isNegative.in[0] <== clause[i];
162         isNegative.in[1] <== 0;
163
164         signal literalIdx;
165         literalIdx <== clause[i] >= 0 ? clause[i] - 1 : (-clause[i]) - 1;
166
167         signal actualValue;
168         actualValue <== assignment[literalIdx];
169
170         signal satisfiedPositive;
171         satisfiedPositive <== clause[i] > 0 ? actualValue == 1 : 0;
172
173         signal satisfiedNegative;
174         satisfiedNegative <== clause[i] < 0 ? actualValue == -1 : 0;
175
176         or_terms[i] <== satisfiedPositive + satisfiedNegative;
177     }
178     signal sum;
179     sum <== 0;
180     for (var i = 0; i < numLiteralsPerClause; i++) {
181         sum <== sum + or_terms[i];
182     }
183
184     isSatisfied <== sum > 0 ? 1 : 0;
185 }
```

```
186
187
188 template ChaffSolverCore(sizeLiterals, sizeClauses, sizeClauseElements) {
189     signal input raw_literals[sizeLiterals];
190     signal input raw_clauses[sizeClauses][sizeClauseElements];
191
192     signal output Chaff_normalized_literals[sizeLiterals];
193     signal output Chaff_normalized_clauses[sizeClauses]
194     [sizeClauseElements];
195
196     component pre_watch_literals_output = ChaffNormalizeLiterals(
            sizeLiterals,
197     sizeClauses,
198     sizeClauseElements);
199     for (var i = 0; i < sizeLiterals; i++) {
200         pre_watch_literals_output.input_literals_initial[i] <==
                raw_literals[i];
201     }
202     for (var i = 0; i < sizeClauses; i++) {
203         for (var j = 0; j < sizeClauseElements; j++){
204             pre_watch_literals_output.input_clauses_initial[i][j] <==
                    raw_clauses[i][j];
205         }
206     }
207
208     component watched_literals_component =
209     ApplyWatchedLiteralScheme(sizeLiterals,
210     sizeClauses,
211     sizeClauseElements);
212     for (var i = 0; i < sizeLiterals; i++) {
213         watched_literals_component.SAT2_literals[i]
214         <== pre_watch_literals_output.normalized_literals[i];
215     }
```

```
    for (var i = 0; i < sizeClauses; i++) {
        for (var j = 0; j < sizeClauseElements; j++) {
            watched_literals_component.SAT2_clauses[i][j]
            <== pre_watch_literals_output.normalized_clauses[i][j];
        }
    }

    signal conflict_learned_clauses[sizeClauses][sizeClauseElements];
    for (var i = 0; i < sizeClauses; i++) {
        for (var j = 0; j < sizeClauseElements; j++) {
            conflict_learned_clauses[i][j] <== 0;
        }
    }

    for (var i = 0; i < sizeLiterals; i++) {
        Chaff_normalized_literals[i]
        <== watched_literals_component.watched_literals_result[i];
    }
    for (var i = 0; i < sizeClauses; i++) {
        for (var j = 0; j < sizeClauseElements; j++) {
            Chaff_normalized_clauses[i][j]
            <== watched_literals_component.
            watched_clauses_result[i][j];
        }
    }
}

template MAIN_ChaffSolverEntry(sizeLiterals, sizeClauses,
    sizeClauseElements) {
    signal input input_literals_initial[sizeLiterals];
    signal input input_clauses_initial[sizeClauses][sizeClauseElements];

    signal output publicOutput;
```

```
248
249     component chaff_core = ChaffSolverCore(sizeLiterals, sizeClauses,
            sizeClauseElements);
250     for (var i = 0; i < sizeLiterals; i++) {
251         chaff_core.raw_literals[i] <== input_literals_initial[i];
252     }
253     for (var i = 0; i < sizeClauses; i++) {
254         for (var j = 0; j < sizeClauseElements; j++) {
255             chaff_core.raw_clauses[i][j] <== input_clauses_initial[i][j];
256         }
257     }
258
259     signal final_literal_sum_check[sizeLiterals + 1];
260     final_literal_sum_check[0] <== 0;
261     for (var i = 0; i < sizeLiterals; i++) {
262         final_literal_sum_check[i + 1] <== final_literal_sum_check[i] +
                chaff_core.Chaff_normalized_literals[i];
263     }
264
265     publicOutput <== final_literal_sum_check[sizeLiterals];
266 }
267
268 component main = MAIN_ChaffSolverEntry(1000, 2000, 3);
```