Daily Proof of Liabilities

Antoine Cyr

A Thesis in
The Concordia Institute for Computer Science (MCompSc)

Presented in Partial Fulfillment of the Requirements
For the Degree of
Master
(Computer Science)
at
Concordia University
Montréal, Québec, Canada

August 2025

CONCORDIA UNIVERSITY
School of Graduate Studies

This is to certify that the thesis prepared
By: **Antoine Cyr**
Entitled: **Daily Proof of Liabilities**
and submitted in partial fulfillment of the requirements for the degree of

Master (Computer Science)

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

_____ Chair
Chair's name

_____ Examiner
Examiner's name

_____ Examiner
Examiner's name

_____ Co-Supervisor
Co-Supervisor's name

_____ Co-Supervisor
Co-Supervisor's name

Approved by _____
Chair of Department or Graduate Program Director

_____ 2025 _____
Dean of Faculty

Abstract

Name: **Antoine Cyr**
Title: **Daily Proof of Liabilities**

A proof of solvency's goal is to demonstrate that a cryptocurrency exchange possesses sufficient funds to satisfy client withdrawals. In this thesis, we introduce an improvement to the prevailing way of building a proof of liabilities. We use the Nova novel way of proving that a balance is included in the proof of liabilities (i.e. proof of inclusion), and apply it to the proof of liabilities itself. We use the circuit designed to show the proof of inclusion of a Merkle tree, and modify it to prove a list of balance changes in the Merkle tree. While this is slower than producing the whole Merkle tree when you have many changes, this new circuit design enables to separate the proof into multiple smaller proofs, enabling the use of the Nova folding scheme. The folding of arithmetics circuits reduces the computation needed for a daily proof of liabilities, enabling the possibility of obtaining this proof at a higher frequency, potentially as frequently as every block.

Acknowledgments

Thank you to Sabine Bergler for supporting me all the way, even when I diverged from the original plan and went beyond her field of expertise. Thank you to Jeremy Clark, who guided me throughout the thesis and introduced me to the beautiful world of zero-knowledge proofs. Thank you to the committee members for their time and feedback.

Table of Contents

List of Figures

Chapter 1

Introduction

In the context of cryptocurrencies, trust between marketplaces and users is at an all-time low. Following the recent bankruptcy and mishandling of customer funds by marketplaces like FTX, users want and need to know that marketplaces have all the funds in their possession. Traditional financial institutions commonly rely on audits as the established method to demonstrate their solvency. The outcomes of third-party examinations are universally accepted due to the trust placed in these third parties. However, auditing a cryptocurrency marketplace presents specific challenges. Audits are not a scalable solution because funds can be moved around more quickly than in traditional finance. This would call for a high-frequency auditing approach, perhaps even daily audits. Here, automating the process becomes advantageous.

Another challenge pertains to trust in the third party, which must be mutual. The institution must trust that their data will not be compromised, and the public must trust the authenticity of the audit results. This is where zero-knowledge proofs come into play.

The proof of solvency uses zero-knowledge to demonstrate, without revealing any information, that the assets in control are greater than the liabilities. It consists of a proof of assets and a proof of liabilities, a self-explanatory structure.

This paper extends the work done on proof of liabilities only. The most prevalent way to implement a proof of liabilities is by using a Merkle Sum tree. In a Merkle tree, every parent node represents the hash of its child nodes. In the Merkle Sum tree, each node additionally holds the balance information. The leaf nodes consist of user IDs and their respective balances, while the root node contains the hash of the entire tree, summarizing the total balance.

Alongside the proof of liabilities, there is a proof of inclusion, where we demonstrate that each customer's balance is included in the tree. This is achieved by proving the Merkle path, which is sufficient for validation.

This paper explores a way to minimize the cost of doing these proofs every day while maintaining complete privacy (i.e., keeping the Merkle tree private).

1.1 Outline

Chapter 2 gives the background information needed to understand this paper. It elaborates on Bitcoin, including its transactions, the network, proof-of-work, and Merkle tree. Marketplaces are also discussed. Additionally, the chapter delves into zero-Knowledge, covering non-interactive

proofs, SNARKs, and arithmetic circuits. The concept of proof of solvency, focusing on real-world applications, is introduced.

Chapter 3 goes deeper into zero-knowledge, more specifically, the different recursion techniques, which will be used to reduce the proof size and make daily proofs easier. The techniques include aggregation, recursion schemes, accumulation and folding schemes.

Finally, Chapter 4 explores the proof and circuit construction of the thesis. We start with a simple circuit for the proof of liabilities, as well as a circuit for the proof of inclusion for a Merkle tree. Next, we modify the initial proof of inclusion to prove the inclusion of many Merkle trees simultaneously using the folding scheme. The folding scheme is a recent technique by Nova that allows to ”fold” many circuits together, reducing the proof size. They proposed using this scheme to do the proof of inclusion, which we will implement.

For the proof of liabilities, we adopt an alternate strategy to reduce the proof size on the second day, given that the original Merkle tree is already established. This is the first novel idea of the thesis. This involves developing a secondary liabilities circuit that constructs a Merkle tree, utilizing an initial Merkle tree alongside the changes as input. We then optimize it using the folding scheme.

1.2 Motivation

The novel aspect of this thesis emerged after noticing the significant boost that the Nova folding scheme provided to the proof of inclusion. This observation led me to ask: How can this be applied elsewhere? The answer is straightforward: the key component of the proof of liabilities, which is the proof of liabilities itself, can benefit from this approach.

Given how obvious this seems, why hasn't anyone else done it? There are two main reasons. First, the traditional method of conducting a proof of liabilities does not benefit from the Nova folding scheme. This thesis introduces a new circuit specifically designed to be used with the folding scheme. Without it, the circuit would not be scalable, as the proofs would grow with each change. Second, the folding scheme is relatively new, and researchers are still exploring its full potential and applications.

1.3 Results

The proof of liabilities circuits are evaluated and compared based on the number of constraints and proof time. This thesis demonstrates that the new liabilities circuit performs better when the number of changes is 0.05% of users or less. The new circuit is also better the fewer users we have, but it performs well enough with a high number of users. For instance, at 1 million users, the new circuit is 3 times smaller than the old circuit (1 000M constraints vs 3 000M constraints). At 1000 users, the results are way better; we have 300k constraints vs 3M constraints.

Additionally, we show that when combined with the Nova folding scheme, the new circuit remains superior even with 1% of changes. The folding circuit performs better the more users we have. For instance, the proof time at 256 users is similar(around 50 seconds), but at 1M users, we have around 20,000 seconds vs. 200,000 seconds.

The percentage of changes is defined by the number of root changes in the tree between a previously proven Merkle tree and the targeted Merkle tree.

Chapter 2

Background

In the dynamic landscape of cryptocurrencies, ensuring transparency, accountability, and financial stability is paramount for marketplaces. This section provides an overview of the concepts and mechanisms necessary to follow along the conception of a daily proof of liabilities. We explore leading exchanges' approaches to demonstrate their financial health through proof of reserves or solvency mechanisms. Additionally, we highlight the shortcomings and challenges associated with current solvency verification practices, mainly the lack of recurrent proof of reserves, paving the way for a daily proof in the subsequent sections.

2.1 Bitcoin

Bitcoin is recognized as the world's first successful cryptocurrency and decentralized digital currency. The goal of Bitcoin is to allow financial transactions to be settled independently without the need for a middleman, typically financial institutions. Bitcoin is built on a peer-to-peer network, which means that every participant helps to secure the transaction history and propagate new transactions. No single point of failure allows transactions to occur in real-time, in contrast to the delays encountered in the traditional finance world. [5] Bitcoin defines two concepts: Bitcoin, the cryptocurrency, and Bitcoin, the blockchain. The cryptocurrency resides on the blockchain. The Bitcoin blockchain is a decentralized ledger that records all Bitcoin (the cryptocurrency) transactions immutably and transparently. This blockchain serves as a verifiable record of all Bitcoin transactions, accessible to every participant in the network. The transparency afforded by the public blockchain engenders trust and accountability.

2.1.1 Hash function

A hash function is a mathematical function that takes an input (or message) of any size and produces a fixed-size output, called a hash or digest. A good hash function, such as SHA-256, is collision-resistant, meaning finding two inputs that produce the same output is computationally infeasible. It is also pre-image resistant, which means it is nearly impossible to determine the original input from the output. We can also say that it is hiding. The hash function is also deterministic, i.e. it will always produce the same output.

2.1.2 Transactions

For every network participant, there is a public key, a private key and a wallet address associated with the participant. The public key is derived from the private key using elliptic curve multiplication, and the wallet address is derived from the public key using a hashing function. [5] Both are one-way functions, meaning they cannot be derived the other way around. The public key serves as the network's unique identifier, but it is the wallet address that typically defines a participant. The wallet address is similar to a bank account number. When a party send Bitcoin to someone, they send it to their wallet address. To send some Bitcoin, they must create a transaction and send it to the network. When transactions are sent on the network, there is no way of knowing who propagated the transaction first. We need to ensure that a transaction originates from the sender. The way to do that is to sign your transaction. The digital signature is created from the transaction data and the private key, which is only known by the address owner. The digital signature is created using the ECDSA (Elliptic Curve Digital Signature Algorithm) over the secp256k1 curve, the specific elliptic curve used by Bitcoin. We verify a Bitcoin transaction's digital signature using the sender's public key to check that the signature matches the hashed transaction data. This is done through ECDSA, where the signature is compared to a computed value derived from the transaction hash and the public key. If they match, it confirms the transaction was signed with the corresponding private key, validating the transaction. Sending a transaction is the easiest problem to solve. The real challenge is keeping track of who owns what and avoiding the double spending problem. The methodology for managing this is to keep the history of every single transaction. The transactions are bundled into blocks, and the chain of blocks creates the blockchain.

2.1.3 Network

The challenge of the network is to have every single node achieve consensus on the transaction history. Nodes are computers connected to the network that work on publishing new blocks. The nodes work collectively to establish an order of transactions (sequencing). Every new transaction is broadcast to all nodes. The nodes put the transactions into a block and try to publish it. To publish a block, each node needs to solve a proof-of-work challenge. When a node solves the challenge, it broadcasts the block to every node. The node accepts the block if all transactions are valid. There is no formal way of approving a new block. A node shows its acceptance by starting to work on a new block using the hash of the accepted block as the previous hash. If multiple blocks are propagated at the same time, some nodes might accept different blocks, creating multiple chains. To solve this issue, the longest chain is considered to be the correct one. If two chains have the same length, nodes keep working on their respective chains until one receives a new block, breaking the tie.

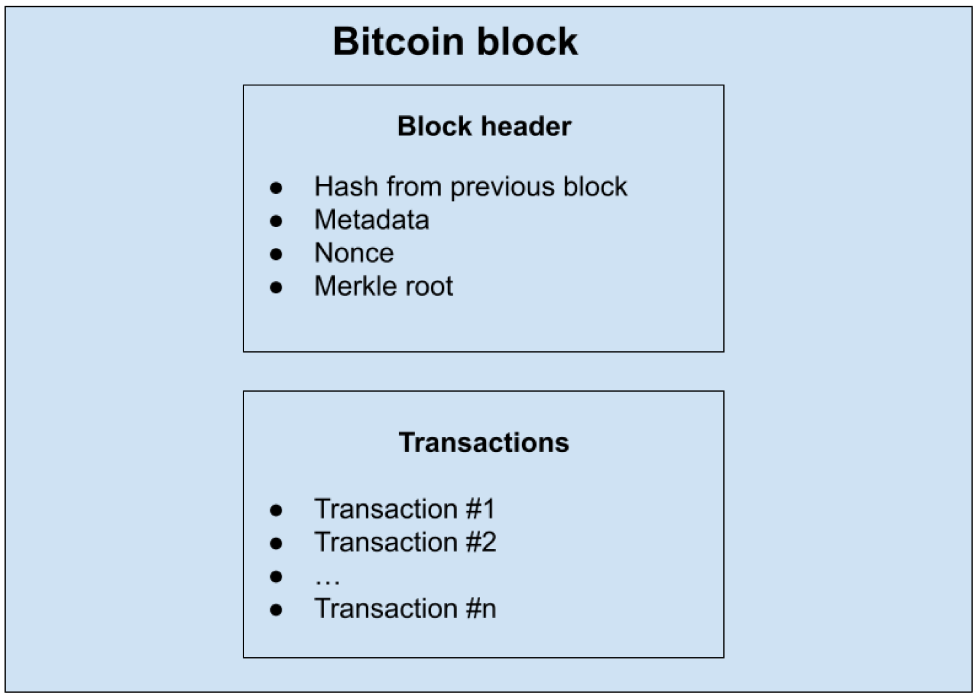


Figure 2.1: Bitcoin block

2.1.4 Proof-of-work

A node must find a hash with a specific number of leading zeros to submit a new block. The SHA-256 algorithm produces the hash, which is 256 bits long and typically represented as a 64-character hexadecimal string. The network's difficulty determines the number of leading zeros required, which sets the target value that the hash must be less than or equal to for the block to be valid. For instance, the threshold might be $0000000000000000abc123...$ This cryptographic puzzle is a barrier to entry, ensuring that much computational power is spent creating the block. The way to test different hash values is to change the block timestamp and the nonce value. The nonce value is there solely for that purpose. Once a block is published, we cannot change any value inside of it because the hash value would change. The immutability of the older blocks makes Bitcoin secure. To modify a block in the middle of the chain, we would need to redo the work of every block made after it. The longest chain is determined by the cumulative proof-of-work invested in it. This is why we say that Bitcoin is secure as long as 51% of the nodes are honest. The chain with the majority of nodes working on it will grow the fastest and thus be the accepted chain. The difficulty of the new block is determined by an average in order to generate blocks at a steady pace. A Bitcoin reward is also associated with mining (publishing) a block.

2.1.5 Merkle Tree

In the blockchain's architecture, only the Merkle root is stored in the block header. Nodes keep only the recent blocks in memory, and for older blocks, they keep only the block header. This

storage ensures the integrity of the blockchain while decreasing the memory required to have the full blockchain history. Since the hash of a block is the hash of the block header, this strategy does not impact the integrity checks of the blockchain. The Merkle root is the top of the Merkle tree and is a unique identifier of the full tree. A Merkle tree is a tree in which the parent node is the hash of the child nodes. The tree is immutable because changing a single node would impact the Merkle root.

2.2 Marketplaces

The best way to buy Bitcoin for the first time is through marketplaces. Marketplaces facilitate the exchange of traditional currency for Bitcoin. However, it's important to understand that the Bitcoin you purchase initially remains in the platform's custody rather than being sent directly to your personal wallet. We need to request a transfer to our wallet to gain custody of our Bitcoin, similar to how traditional banking transactions rely on the bank to process our request. Once we have Bitcoins in our wallet, we can transact on the network without needing a third party. Unless we run a node, we must trust a third party, whether a marketplace or over-the-counter, to acquire Bitcoin first.

Since the Bitcoin ledger is public, we can use tools to view the network's transactions and to see which wallet address owns how many Bitcoins. When your Bitcoin is held in the marketplace's custody, it cannot be tracked onchain (onchain refers to everything happening in the blockchain). It blends with the plarform's holdings, because the marketplace does not have as many wallets as it has clients. Marketplaces manage many wallets, some public and some private, which enhances the privacy of deposits and withdrawals but contradicts Bitcoin's design for transparency and independence from third parties. This lack of visibility poses a challenge, as users have no proof of the marketplace's solvency to reimburse all clients. However, this issue is being addressed through the introduction of proof of solvency (or proof of reserve) mechanisms, which allows marketplaces to demonstrate their solvency. While this is a positive development, current proof of solvency methods have shortcomings and are insufficient to prove solvency.

2.3 Zero-Knowledge

Zero-knowledge proof is a cryptographic technique allowing a prover to demonstrate knowledge of a fact without divulging additional information. For instance, rather than revealing the solution to an equation, zero-knowledge proofs enable the prover to show that they know the solution without revealing it. In this context, a witness is the secret information or evidence that supports the validity of the statement being proven. The goal is to construct a proof that is both sound and complete and is zero-knowledge.

- **Completeness**: If the statement is true, an honest verifier will be convinced by an honest prover.
- **Soundness**: If the statement is false, no dishonest prover can convince the honest verifier (except with some infinitesimal probability).

- **Zero-Knowledge**: If the statement is true, a verifier learns nothing other than the statement is true. [31]

Formal definition : A proof system (P, V) for L is zero-knowledge if for every efficient verifier strategy $V*$ there exists an efficient probabilistic algorithm $S*$ (known as the simulator) such that for every $x \in L$, the following random variables are computationally indistinguishable:

- The output of $V*$ after interacting with P on input x.
- The output of $S*$ on input x.

That is, we can show the verifier does not gain anything from the interaction, because no matter what algorithm $V*$ he uses, whatever he learned as a result of interacting with the prover, he could have just as equally learned by simply running the standalone algorithm $S*$ on the same input. [7]

2.3.1 Non interactive proofs

Zero-knowledge proofs were initially designed as interactive: multiple rounds of interaction between the prover and the verifier [19]. Leading to what are called interactive zero-knowledge proofs. This interaction allows the prover to demonstrate knowledge of the solution without revealing additional information. An alternative model was proposed in which the verifier and prover share a reference string during a trusted setup. Once we have the reference string, a single message is needed between the prover and the verifier. Eliminating multiple rounds of interaction simplifies the verification process and reduces the computational power required.[11] Therefore, non-interactive zero-knowledge proofs offer enhanced efficiency and scalability, which will be needed later.

2.3.2 SNARKS

One recent advance for non-interactive proofs is SNARK (Non-Interactive Argument of Knowledge).

This means a proof that is:

- **Succinct**: The proof size is minimal compared to the witness size (the secret information).
- **Non-interactive**: There are no rounds of interactions between the prover and the verifier.
- **Argument**: It is sound only against provers with bounded computational resources. A dishonest prover with unlimited computational power could potentially prove a false statement.
- **Knowledge-sound**: A valid proof can only be generated if the prover knows the witness. [28]

Moreover, a SNARK can also be zero-knowledge, where the prover demonstrates knowledge without revealing additional information about the witness. We call such proof a zk-SNARK. There are many techniques to construct a SNARK, but all of them follow the same guideline:

- Express our problem as an arithmetic circuit

- Transform the circuit in a polynomial
- Commit the polynomial
- Interaction between the prover and the verifier to show that the polynomial solves the problem

We will show the core general idea behind a SNARK. We must transform the code we want to prove in a quadratic arithmetic program (QAP) to construct the SNARK. The best example of a SNARK using this technique is Groth16[20], one of the most widely used SNARK today. Groth16 is a Perfect Zero-Knowledge system, where the distribution of the view generated by the simulator is exactly the same as the view generated by the honest prover and verifier. There exist also Statistical and Computational Zero-Knowledge systems, both of which relies on a weaker assumption of zero-knowledge.

Arithmetic circuit

Let us say we want to prove $x^3 + x + 5 = 35$. The prover has to convince the verifier that he knows the solution without revealing it to him.

The first step in transforming a problem into the QAP form is to express it as an arithmetic circuit. An arithmetic circuit is a set of gates, each assigned a distinct set of inputs corresponding to the numbers to be processed in the operation. These gates are configured to execute arithmetic operations such as addition, subtraction, multiplication, or division. The outputs of the gate circuit represent the digits of the resulting computation.

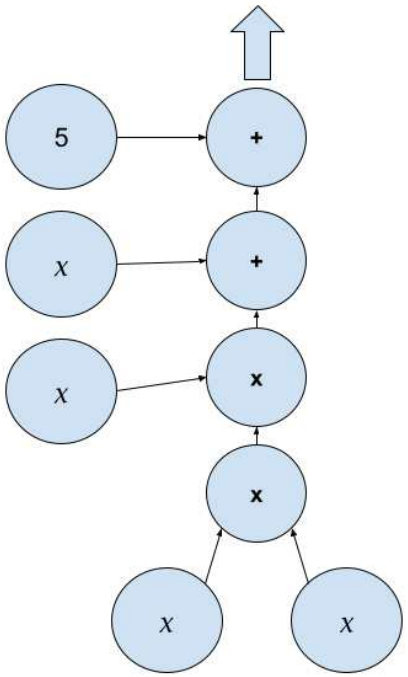


Figure 2.2: General Arithmetic circuit

R1CS

The first step is to express our arithmetic circuit as a set of constraints where each constraint contains only one arithmetic operation.

$$s_1 = x * x$$
$$s_2 = s_1 * x$$
$$s_3 = s_2 + x$$
$$out = s_3 + 5$$

We can now convert this into an R1CS. An R1CS is a sequence of groups of three vectors (a, b, c), where the solution to the R1CS is a vector s, and s must satisfy the equation $s \cdot a * s \cdot b - s \cdot c = 0$, where $\cdot$ represents the dot product. The length of each vector is the length of the total number of variables for the system. This includes a variable one at the beginning, the variable out at the end, and all intermediate variables. The standard way of representing the three vectors (a, b, c) is to create a mapping with the variables. Here is the mapping we will use:
$[one, out, x, s_1, s_2, s_3]$
This gives us the first gate:

$$a = [0, 0, 1, 0, 0, 0]$$
$$b = [0, 0, 1, 0, 0, 0]$$
$$c = [0, 0, 0, 1, 0, 0]$$

We are checking here that $x * x = s_1$. a and b both represents x, while c represents s_1.

Let us verify that our first gate satisfies the solution equation:
$s \cdot a * s \cdot b - s \cdot c = 0$
We know that the solution to the equation is $x = 3$, so given that our s vector would be:

$$s = [1, out, x, s_1, s_2, s_3]$$
$$s = [1, 35, 3, 9, 27, 30]$$

Now, if we do the dot product:

$$s \cdot a = [1, 35, 3, 9, 27, 30] \cdot [0, 0, 1, 0, 0, 0] = 0*1+0*35+1*3+0*9+0*27+0*30 = 3$$
$$s \cdot b = [1, 35, 3, 9, 27, 30] \cdot [0, 0, 1, 0, 0, 0] = 3$$
$$s \cdot c = [1, 35, 3, 9, 27, 30] \cdot [0, 0, 0, 1, 0, 0] = 9$$
$$s \cdot a * s \cdot b - s \cdot c = 3 * 3 - 9 = 0$$

This shows that our solution vector satisfies the first gate.

Following these rules, our second gate for $s_1 * x = s_2$ is:

$$a = [0, 0, 0, 1, 0, 0]$$
$$b = [0, 0, 1, 0, 0, 0]$$
$$c = [0, 0, 0, 0, 1, 0]$$

The third gate for $s_2 + x = s_3$:

$a = [0, 0, 1, 0, 1, 0]$ $(s_2 + x)$
$b = [1, 0, 0, 0, 0, 0]$ (1)
$c = [0, 0, 0, 0, 0, 1]$ (s_3)

Fourth gate for $s_3 + 5 = out$:

$a = [5, 0, 0, 0, 0, 1]$
$b = [1, 0, 0, 0, 0, 0]$
$c = [0, 1, 0, 0, 0, 0]$

The R1CS put together:

$$A = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 \\ 5 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

$$B = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

$$C = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 \end{bmatrix}$$

[29]

QAP

The next step is converting our R1CS into QAP form, which uses polynomials instead of dot products. We do this by using polynomials instead of a dot product. In order to get a polynomial, we need to do a Lagrange interpolation on a set of points. For a polynomial of degree n, we need $n + 1$ set points for the interpolation to give the polynomial. We need a polynomial of degree n, where n is the number of rows -1. (Same thing as the number of gates -1). Therefore, we need a set of 4 points to have a polynomial. We can get 18 $(3 * 6)$ polynomials by transposing our matrices. For instance, the first column of A^T is our first polynomial $[0, 0, 0, 5]$. This gives us the set of points $[(1, 0), (2, 0), (3, 0), (4, 5)]$. The polynomial interpretation of this set of points is $f(x) = 0.833x^3 - 5x^2 + 9.166x - 5$. Doing the same for every column of A, B and C, we get the matrices:

$$A_m = \begin{bmatrix} 0.833 & -5 & 9.166 & -5 \\ 0 & 0 & 0 & 0 \\ -0.66 & 5 & -11.33 & 8 \\ 0.5 & -4 & 9.5 & -6 \\ -0.5 & 3.5 & -7 & 4 \\ 0.166 & -1 & 1.833 & -1 \end{bmatrix}$$

$$B_m = \begin{bmatrix} -0.33 & 2.5 & -5.166 & 3 \\ 0 & 0 & 0 & 0 \\ 0.33 & -2.5 & 5.166 & -2 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$$C_m = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0.166 & -1 & 1.833 & -1 \\ 0 & 0 & 0 & 0 \\ -0.166 & 1.5 & -4.33 & 4 \\ 0.5 & -4 & 9.5 & -6 \\ -0.5 & 3.5 & -7 & 4 \end{bmatrix}$$

The polynomial we obtained from the first column of A appears in the first row of A_m. We obtained each row similarly.

The point of this transformation is that the dot product is now a series of additions and multiplications of polynomials, and the result will be a polynomial.

The resulting polynomial needs to equal 0 at all the x coordinates we used previously. If it does, it means that the checks pass. We do not need to evaluate our polynomial at every point to verify correctness. We can instead divide our polynomial t by another polynomial z and verify that the division leaves no remainder. We define Z as $(x-1)*(x-2)*(x-3)*...$, the simplest polynomial that is equal to zero at all points that correspond to logic gates.

In this case, computing T:

$$T(x) = S \cdot A_m * S \cdot B_m - S \cdot C_m$$
$$T(x) = -3.44x^6 + 51.5x^5 - 294.77x^4 + 805.833x^3 - 1063.77x^2 + 592.66x - 88$$

In this case, we use $Z = (x-1)(x-2)(x-3)(x-4)$.

To show that T is divided by Z, it is sufficient to verify $T(x) = 0$ at $1, 2, 3$ and 4.

More formally, we have shown that there exists a polynomial H such that $T(x) = H(x) \cdot Z(x)$ (i.e. if T is divided by Z, then it will be perfectly divisible and will leave no remainder)

Groth16 is an all-in-one zkSNARK, meaning the polynomial commitment scheme is included in the design. Other SNARK desings such as Plonk and Marlin. In a zkSNARK, the prover demonstrates that a polynomial Z exactly divides another polynomial T using Polynomial Commitments, such as KZG, Bulletproofs, or FRI. The purpose of this step is to reveal only certain evaluations of the polynomial without disclosing the entire polynomial itself.[14]

2.3.3 Polynomial Commitments

A polynomial commitment is a cryptographic technique that allows one to commit to (or "lock") a polynomial's data in such a way that the data can later be revealed (or "unlocked") while ensuring integrity and privacy.

Polynomial commitments are binding, meaning that once a commitment is made, it is computationally infeasible to alter the original polynomial without detection. The commitments are

generally of constant size. While collisions are theoretically possible due to the pigeonhole principle, finding a second polynomial that matches the same commitment is computationally infeasible.

Additionally, polynomial commitments can be hiding, which means the verifier is not exposed to any information about the polynomial. For example, we can create a commitment by hashing a polynomial with a collision-resistant hash function like SHA-256, and only the hash value is shared. To make it hiding, a random factor could be added to the polynomial data before hashing.

Instead of committing to an entire dataset (a set of values or a table), we can commit to a polynomial representing or encoding this data. This is useful because polynomials can be compact representations of structured data.

The naïve approach to committing to a polynomial is simply revealing its coefficients directly. This approach fails because the information is not hidden, among other things.

A well-designed polynomial commitment scheme allows flexibility: it enables someone to either open the commitment to reveal the entire polynomial or to open it at a specific evaluation point. This is a necessary parameter of the scheme, as it allows verification that someone knows a polynomial that evaluates at a certain point without revealing the polynomial itself.

For the proof to be complete, the polynomial must be evaluated at $d + 1$ unique points, where d is the degree of the polynomial. However, to be succinct, we want to verify the least number of points possible.

How many points do we need to verify to have a high confidence in the polynomial? In a cryptographic setting, q is a large prime. If q is 256-bits and the polynomial is degree 1000, then the probability of giving the correct value at random is $1000/2^{256}$ (As we recall, a polynomial of degree 1000 has 1000 roots). Therefore, after checking just one random point, it is statistically probable that the polynomials will be the same if the point matches. [35]

Many different polynomial schemes exist, such as KZG, Bulletproofs and FRI. In these schemes, the setup processes and the commitments vary significantly.

KZG commitments rely on a trusted setup to generate elliptic curve parameters[23]. The KZG setup is universal. Once it is done, it can be useable by anyone. Many different versions are available to the public; the Ethereum Foundation has implemented one of these versions.

Bulletproofs avoids using a trusted setup by using logarithm-based techniques[15]. However, while the KZG proofs are constant size, Bulletproofs are logarithmic.

FRI evaluates polynomials through proximity checks.[8] FRI is post-quantum since it relies only on hash function security. Other schemes rely on cryptographic assumptions, such as the hardness of the discrete logarithm problem, which quantum computers can efficiently solve.

KZG

KZG commitments work by evaluating the polynomial at a single secret point τ. As we just saw, evaluating a single point is statistically sufficient. This approach ensures that the commitment remains succinct.

To achieve this, a trusted setup generates a structured random string (SRS) containing powers of τ of a specific polynomial degree.

$$\langle g^{\tau^0}, g^{\tau^1}, g^{\tau^2}, g^{\tau^3}, \ldots, g^{\tau^d} \rangle \equiv \text{SRS}$$

where g is a generator. The prover uses these precomputed values to commit to $f(x)$ without

knowing τ or $P(\tau)$.

$$\begin{aligned} K_P(\tau) &= \text{Commit}(P(\tau)) \\ &= g^{c_0}(g^{\tau})^{c_1}(g^{\tau^2})^{c_2}(g^{\tau^3})^{c_3}\dots \\ &= g^{c_0+c_1\tau+c_2\tau^2+c_3\tau^3+\dots} \\ &= g^{P(\tau)} \end{aligned}$$

The commitment is succinct because the polynomial degree does not impact the size.

The prover can send the coefficients to open the polynomial, and the verifier will recompute $K_P(\tau)$. This is of little use since the prover could have hash the coefficient to get a similar result. KZG offers additional features that work specifically with polynomials.

Additive homomorphism in KZG allows commitments of polynomials to be combined in a way that reflects polynomial addition. If two polynomials f and g have a sum $f + g = h$, then their commitments C_f and C_g have the sum $C_f + C_g = C_h$.

Multiplication is not exactly homomorphic in KZG, but we can get something close to it using bilinear pairing. The idea is to assert the value for $K_{P_1(\tau)} \cdot K_{P_2(\tau)}$ and convince it is correct given $K_{P_1(\tau)}$ and $K_{P_2(\tau)}$.

$$\begin{aligned} e(K_{P_1(\tau)}, K_{P_2(\tau)}) &\stackrel{?}{=} e(K_{P_1(\tau)+P_2(\tau)}, g) \\ e(g^{P_1(\tau)}, g^{P_2(\tau)}) &= e(g^{P_1(\tau)\cdot P_2(\tau)}, g) \\ &= e(g,g)^{P_1(\tau)\cdot P_2(\tau)} \end{aligned}$$

The most beneficial property of KZG and any polynomial commitment scheme is the ability to open the commitment at specific points.

KZG commitments allow proving that a polynomial $P(x)$ has a root at r by demonstrating that $(x-r)$ divides $P(x)$ without remainder. As we recall, we can write any polynomial in a factorized format:

$P(x) = (x - r_0)(x - r_1)(x - r_2)\dots$

If r is a root, $P(x)$ is divisible by $(x - r)$, so we define:

$Q(x) = \frac{P(x)}{x-r}$

The prover commits to $K_{Q(\tau)}$ and provides it to the verifier.

The verifier can compute $K_{V(\tau)}$ by treating $x - r$ as a polynomial:

$V(x) = x - r = -r + 1 * x$ and using KZG to produce it.

$$\begin{aligned} e(K_{P(\tau)}, g) &\stackrel{?}{=} e(K_{Q(\tau)}, K_{V(\tau)}) \\ e(g^{P(\tau)}, g) &= e(g^{Q(\tau)}, g^{V(\tau)}) \\ e(g,g)^{P(\tau)} &= e(g,g)^{Q(\tau)\cdot V(\tau)} \end{aligned}$$

This proof is independent of the polynomial degree of Q and P, making it efficient even for large polynomials.

The security of the commitments relies on properly generating the trusted setup. If the secret τ is leaked or manipulated, the security of the scheme could be compromised. To mitigate this, multi-party computation ceremonies ensure that at least one participant securely destroys their contribution to the SRS. Zero-knowledge proofs can validate that the setup was generated correctly without revealing τ.

KZG is the standard in terms of commitments for SNARKS. It is efficient and forms the basis of many cryptographic protocols. [34]

2.4 Proof of solvency

A proof of solvency is a system where we prove that an entity, in most cases an exchange, holds enough assets to cover the balance of every customer. In traditional finance, this would be done through an audit. While an audit can be helpful, it has many limitations. It requires the trust of another third party, but it also poses a time constraint. Thus, it is impractical, even impossible, to hold an audit daily, and things move fast in the Bitcoin world. Big money transfers does not need approval and can be sent within minutes. It is essential to be able to fill a proof of solvency every single day. Therefore, implementing a mechanism to produce daily proof of solvency is of the utmost importance.

A proof of solvency is composed of a proof of assets, where we verify what assets the marketplace has control over, and the proof of liabilities, where we confirm that the total amount of user deposits is smaller than the sum of the marketplace's assets.

The first paper about a proof of solvency focuses only on the proof of liabilities. In his paper, Gregory Maxwell addresses the issue of verifying the solvency of Bitcoin exchanges. [25] Maxwell's system ensures user privacy by maintaining the confidentiality of individual account balances while only revealing aggregate information in the proof of liabilities. This is achieved through the application of Merkle trees.

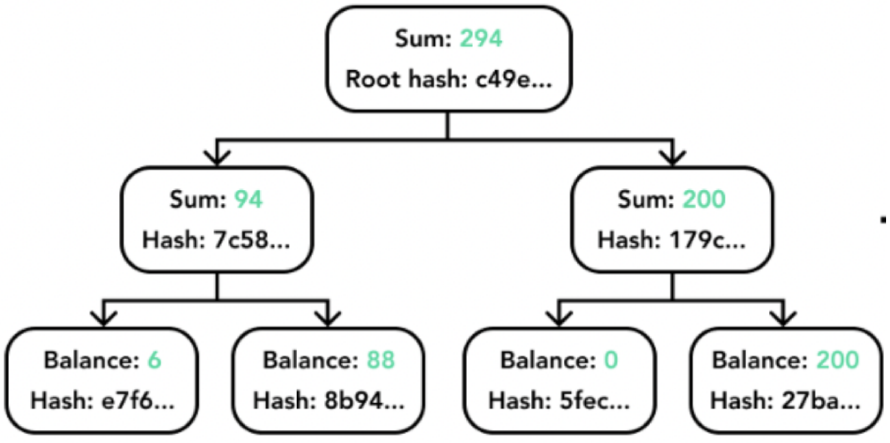


Figure 2.3: Merkle tree for proof of liabilities

In the Merkle tree, every node contains a user's balance and a hash-based commitment incor-

porating the customer ID and a nonce. The root of the tree is the sum of the balances. Users receive a subset of the hash tree from the exchange to verify their inclusion in the total liabilities. This subset includes the user's nonce and the sibling nodes along the unique path from the user's leaf node to the root. Users can confirm the inclusion of their balance by comparing the received information to the exchange's broadcasted root node. While elegant, the protocol does have privacy implications. The exact value of the exchange's total liabilities, published in the root node, may be sensitive data. Additionally, the proof of inclusion reveals the neighbor's balance and the subtree's balance along the Merkle path.

The proof of liabilities is only part of the proof of solvency. A complete proof of solvency needs a proof of asset as well. Provision describes the first preserving privacy proof of asset [17].

In Provisions, the focus shifts toward preserving privacy while still proving ownership of assets. Instead of publicly demonstrating control over specific addresses, Provisions enables exchanges to prove ownership of an anonymous subset of addresses sourced from the blockchain.

Since these 2 papers, a lot of work has been done to evolve the proof of solvency. However, marketplaces still do not implement a proof of solvency or a flawed and limited version of it.

2.4.1 Real world proof of solvency

In recent years, several major cryptocurrency exchanges, including Binance, Crypto.com, and Kraken, have taken steps to enhance transparency by providing various proof of reserves. Binance has implemented a proof of reserves system where they publish a monthly Merkle tree as their proof of liabilities and disclose a list of their assets (every cryptocurrency under their control) [9]. Crypto.com published a one-time audit [26]. Kraken also publishes proof of liabilities every few months without proof of assets. [9].

Although these proof of reserves may seem promising initially, they are primarily superficial. The proofs have many shortcomings; they are insufficient to prove that the marketplaces are solvent. The first concern is the lack of proof of assets, or in Binance's case, the lack of evidence demonstrating control over the wallets. Without reliable proof of assets, the proof of liabilities is worthless because it has nothing to compare against.

Moreover, the frequency of reporting is another area of concern. Given the dynamic nature of cryptocurrencies, a monthly report is not sufficient. Binance is the only marketplace describing its proof of solvency, and we can see that it is not built with recurrence in mind. The proof is created from scratch every time. They would need 150 servers to produce a daily proof [10].

Chapter 3

Recursion proofs

This chapter serves as an intermediary background exploration, diving into more advanced concepts beyond the last chapter's concepts. Zero-knowledge is dynamic and continuously evolving, marked by ongoing advancements and active research endeavors. Recursion proofs are among the most important and active subjects of these advancements. Recursion proofs are created to accelerate the generation of multiple zero-knowledge proofs. Recursion proofs differ in their design. They can vary significantly and serve different use cases. In this section, we will examine these variations and distinguish between them. We aim to identify the most suitable method for our daily proof of liabilities and proof of inclusion. [22]

The prover's process in SNARKs is to create a proof using the setup parameters, a private witness, and public input. As discussed in the previous section, the proof begins by transforming the code into an arithmetic circuit, which is then represented as a polynomial. The polynomial encodes a trace of the circuit (every wire value from inputs to intermediary values to output). This polynomial is created using the witness (private input).

The objective is to demonstrate that the polynomial satisfies the solution without revealing the polynomial itself. This is where the polynomial commitment scheme becomes essential. The setup phase generates public parameters that assist in creating and verifying the proof without disclosing the polynomial.

Table 3.1: Definitions of Symbols in SNARK Circuit

Symbol	**Definition**
$S(C)$	Setup phase that generates the public parameters (pp and vp) for the prover and verifier.
$P(\text{pp}, x, w)$	Prover function that generates the proof π using the public parameters, public input x, and private witness w.
$V(\text{vp}, x, \pi)$	Verifier function that uses the verification parameters vp, public input x, and proof π to decide whether to accept or reject.
$C(x, w)$	Arithmetic circuit to generate the polynomial, taking public input x and private witness w.
w	Private witness, representing the prover's secret input.
x	Public input, accessible to the prover and verifier.

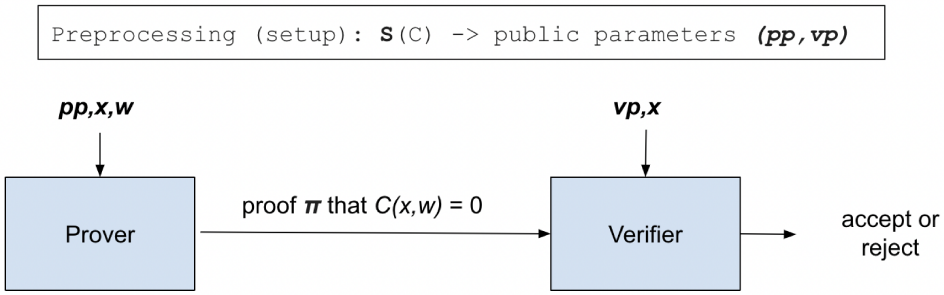


Figure 3.1: SNARK Circuit [2]

3.1 Aggregation

Aggregation is the simplest form of recursion for zk proofs. Its purpose is to reduce the number of proofs that must be verified. It takes a list of proofs as input and outputs a single proof. The process involves two phases.

In the first phase, we create our initial proofs. In the second phase, these proofs are combined into "`proof of proofs`". The second phase can be considered a tree, with the initial proofs being the leaves. Each parent node proves the proof of the child nodes. The tree can have any number of degrees. We can go from a tree of 2 levels to a binary tree with as many levels as needed.

This is a simple process. Each node has its proof, and then we prove that the other proofs are valid, giving us only one proof to verify. Since there are two different phases, two different circuits need to be constructed. On the surface, we can parallelize the initial proofs, which decreases the proving time, and we only have one proof to verify, which reduces the verifying time.

However, there are still some aggregation issues. The main issue is that a verification circuit is more complex than a proof of transaction. Verifying a proof involves constructing a circuit to represent the verification process itself, which is inherently complex. This circuit must perform cryptographic operations, including curve arithmetic and bilinear pairings, which must be implemented directly at the circuit level. As a result, the verification circuit is essentially the exponent of the cryptographic process. This becomes a scaling issue since the proof time of the second circuit grows linearly with the number of proofs to verify. [22]

3.2 Recursion scheme

The following technique is the recursion scheme, or Incrementally Verifiable Computation (IVC). Like the aggregation, the proof is separated into nodes. However, in this case, each node serves two purposes: to prove that the node itself is valid and that the previous proof is valid. The number of constraints will always stay the same because it only proves two things of fixed sizes. This creates a chain structure, where verifying the latest node is the equivalent of verifying every node in the chain. It is an improvement from the previous technique for two reasons. First, this allows the maintenance of a constant number of constraints in the circuit. Second, we do not have to redo a

trusted setup once the proof size is defined. This means the trusted setup remains the same; we do not have to redo it. This implies that the latest node does not take longer to verify than the second node. However, each node still has to verify the previous node, which takes additional time. [22]

3.3 Accumulation

Accumulation schemes offer a different approach to aggregating proofs than IVC. Instead of verifying proofs at each step, accumulation defers all heavy computational tasks to the final step. This allows proofs to be efficiently added to an accumulator while minimalizing the intermediate verification workload. Each step adds a new proof to the accumulator while maintaining a small and constant-sized state. Unlike recursion, where each proof is verified within the circuit immediately, accumulation schemes delay all verification until the final stage, where all accumulated proofs are checked simultaneously.

There are different types of accumulation schemes, each handling proof accumulation in a distinct way. Some, like polynomial accumulation, track commitments to polynomial evaluations over multiple steps. Others, such as accumulation based on cryptographic accumulators (e.g., RSA or Merkle accumulators), focus on aggregating statements into a compact structure. Regardless of the method, the core idea remains the same: verification is deferred to the final proof, preventing the verification cost from increasing with each step.

Accumulation schemes relate to Incrementally Verifiable Computation (IVC) because both aim to make verifying long computations more efficient. However, the key difference is that IVC requires each step to verify the previous one while keeping proof sizes constant, while accumulation postpones all verification until the final step. This makes accumulation more suitable for cases where verifying everything at the end is preferable to verifying at every step.

The advantage is clear: instead of paying the cost of verification n times throughout the process, we pay it once at the end. This is especially useful when multiple independent proofs need to be efficiently combined without introducing the recursive overhead of IVC. However, this benefit comes with a tradeoff. Since all verifications are deferred, the final step can be computationally expensive.[16]

3.3.1 Halo accumulation

The concept of deferring the polynomial commitment opening and consolidating them into a single operation was introduced by Bowe, Grigg, and Hopwood Halo.[13] This process involves two parts: The first part is fast, where we output a polynomial and its commitment. The second part is expensive. In it, we verify the pair (f_1, c_1) of the polynomial f_1 and its commitment c_1. We open the polynomial only at a specific point. The second part can be accumulated if the commitment scheme is additively homomorphic. Instead of individually verifying each pair, we accumulate the pairs and verify their linear combinations ($c_1 + c_2 + ...$ is a commitment for $f_1 + f_2 + ...$). [35]

3.4 Folding scheme

In the Nova scheme, the folding technique accumulates the R1CS progressively as the computation proceeds rather than accumulating only the complex portions of the R1CS. This means that at each step in the folding process, instead of storing the separate R1CS sets for each node and combining them later, we incrementally combine them into a single R1CS set. [22] This set, called "relaxed R1CS," simplifies the final proof generation. As a result, we avoid the need to handle multiple sets of R1CS in the final step, leading to a more efficient recursive proof process. The relaxed R1CS is then used to compute a single proof at the end of the folding process.

3.4.1 Relaxed R1CS

Returning to the previous section, we previously defined what an R1CS is. The goal is to combine 2 R1CS and obtain another R1CS. If we succeed, we can fold every R1CS together and be left with only one.

If we **define our R1CS**:

fix an R1CS program $A, B, D \in \mathbb{F}_p^{u \times v}$
We define x as the public input and w as the witness.
Instance 1: $x_1 \in \mathbb{F}_p^n$, $z_1 = (x_1, w_1) \in \mathbb{F}_p^v$
Instance 2: $x_2 \in \mathbb{F}_p^n$, $z_2 = (x_2, w_2) \in \mathbb{F}_p^v$
Where u is the number of constraints in the R1CS, v is the number of variables in the constraint system, p is the private field, and n is the number of public variables.
We know $Az_i \circ Bz_i = Dz_i$ for $i = 1, 2$

Recall: Our R1CS must be valid for any field element. To prove this, the verifier chooses a random point r at which we will evaluate our constraints system.

The naïve approach to folding an R1CS is to sum the two instances together.

First attempt:

Let us define r as a random variable:
$r \leftarrow \mathbb{F}_p$
Then we set
$x \leftarrow x_1 + rx_2$
$z \leftarrow z_1 + rz_2 = (x_1 + rx_2, w_1 + rw_2)$

Then:

$Az \circ Bz = A(z_1 + rz_2) \circ B(z_1 + rz_2)$
$= (Az_1) \circ (Bz_1) + r^2(Az_2) \circ (Bz_2) + (r(Az_2) \circ (Bz_1) + r(Az_1) \circ (Bz_2))$
$= Dz_1 + r^2 Dz_2 + E$
Where E is a combination of the remaining values.

Simply summing the constraints does not preserve the R1CS structure we just defined: $Az_i \circ Bz_i = Dz_i$. Since the direct addition of two R1CS instances does not produce another valid R1CS, we need a different approach.

We need to modify the R1CS so that it can be folded. To solve this, we introduce a relaxed R1CS, which includes a new error term E. The goal is to have the new terms produced by the folding comprised in the term E. We will try the new form: $Az \circ Bz = c(Dz) + E$.

Let's **define a relaxed R1CS**:

$A, B, D \in \mathbb{F}_p^{u \times v}, (x \in \mathbb{F}_p^n, c \in \mathbb{F}_p, E \in \mathbb{F}_p^u)$
Witness: $z = (x, w) \in \mathbb{F}_p^v$
s.t. $(Az) \circ (Bz) = c(Dz) + E$

Lets **fix the R1CS** program once again:

$A, B, D \in \mathbb{F}_p^{u \times v}$
Instance 1: public (x_1, c_1, E_1), witness $z_1 = (x_1, w_1) \in \mathbb{F}_p^v$
Instance 2: public (x_2, c_2, E_2), witness $z_2 = (x_2, w_2) \in \mathbb{F}_p^v$
We know $(Az_i) \circ (Bz_i) = c_i(Dz_i) + E_i$ for $i = 1, 2$

Second attempt:

$T \leftarrow (Az_2) \circ (Bz_1) + (Az_1) \circ (Bz_2) - c_1(Dz_2) - c_2(Dz_1)$
$x \leftarrow x_1 + rx_2, c \leftarrow c_1 + rc_2,$
$E \leftarrow E_1 + rT + r^2E_2$
$z \leftarrow z_1 + rz_2 = (x_1 + rx_2, w_1 + rw_2)$
$Az \circ Bz =$
$= A(z_1) \circ rB(z_1) + r^2(Az_2) \circ (Bz_2) + r(Az_2) \circ (Bz_1) + r(Az_1) \circ (Bz_2)$
$= c_1(Dz_1) + E_1 + r^2c_2(Dz_2) + r^2E_2 + r((Az_2) \circ (Bz_1) + (Az_1) \circ (Bz_2))$
$= (c_1 + rc_2)(Dz_1 + rDz_2) + E_1 + r^2E_2 + rT$
$= c(Dz) + E$

We have a valid relaxed R1CS.

Validity While the error term E allows folding, it questions the validity of the proof. As long as E is bounded, the proof remains valid. [22] The relaxed formulation preserves constraint structure, enables incremental folding, and prevents uncontrolled error propagation. If needed, the system can return to a strict R1CS by proving $E = 0$ in the final verification, ensuring flexibility and correctness.

Chapter 4

Proof construction

In this chapter, we will construct the proof of liabilities and the proof of inclusion. We will go through different circuit designs to find the optimal proof of inclusion and proof of liabilities. We will start by describing a standard proof of liabilities and inclusion, which we will then optimize and propose novel solutions using recursion schemes. We will propose a proof of inclusion derived from existing literature and a novel proof of liabilities.

4.1 Constants

Before constructing the circuits, we need to define the constants that will be shared between the different circuits.

4.1.1 Circom

Circom is a domain-specific language designed to create arithmetic circuits specifically utilized in zk-SNARKs. These proofs verify calculations without revealing private data, like proving we have funds without showing our account. Within Circom, circuit code is written to define the desired constraints. One notable distinction from other languages is the utilization of signals and templates. Templates can be conceptualized as functions that operate as circuits. The primary template receives signals as inputs and produces other signals as outputs. Signals can be classified as either public or private. Assigning a value to a signal contributes to the constraint system and the witness calculation process. Input and outputs are signals, and we can have additional intermediate signals. In Figure 4.1, the template can be used to calculate the square of an input. The template creates a constraint $in * in = out$.

```
template Square() {
signal input in;
signal output out;
out <== in * in;
}
```

Figure 4.1: Template and Signal Example

During circuit compilation, constraints are generated in r1cs format. Additionally, compiling a circuit produces a witness file containing the data essential for verifying the constraints and demonstrating the circuit's correct behavior.

4.1.2 MiMCSponge

MiMCSonge is a standard hashing algorithm often used in zero-knowledge protocols. It was chosen because it is secure and well-integrated with Circom. The hash function is $F_i(x) = (x+k+c_i)^3$, where i is the number of rounds, k is a fixed constant, and c_i is a constant specific for a round. In our implementation, we use $i = 220$ and $k = 1$, both standard values. We use four inputs: the two sums and two hashes of the children. The hash function is a sponge construction. This is a modern way to do hash functions where it is easier to do security assertions. It does the 220 rounds on the first input, then adds the second input and starts over again.

4.1.3 Constraints

A zero-knowledge circuit generally enforces two types of constraints: range checks and arithmetic constraints. The simplest arithmetic case is when we define a value in Circom, such as $a = b + c \bmod q$, where q is a parameter of the elliptic curve Circom uses. Circom compiles the assignment to create an arithmetic constraint that verifies the equality holds within the finite field q.

A range check verifies that a value lies within specified bounds. This is necessary because operations occur in a finite field. This means an overflow would cause the value to return to 0. We limit the range of the values to prevent overflow. This is critical because an overflow would change the balance sum we are trying to prove. Range checks are critical, especially for multiplications, as the product of two numbers can grow exponentially, leading to overflows much faster than additions. We limit the range of values to ensure no overflow occurs, preserving the correctness of computations.

In Circom, the range check constraint is transformed into an arithmetic constraint. For instance, to verify $x < n$, we can use a library function like `lessThan`. This function decomposes the numbers into their binary representations and performs a series of bit equality checks to determine the result. Circom compiles these bit equalities into arithmetic constraints. A more straightforward way to prove a small number is to prove that the leading bits are 0. Proving that an n-bit number has i leading zeros demonstrates it is less than 2^{n-i}. We can do the sum of the leading bits and prove it equals 0. For instance, if we add n balances where each balance is k bits, then the sum can never exceed $n \cdot (2^k - 1)$

As mentioned before, in Circom, we use a specific variable type called a signal. When we assign a value to a signal, a constraint is usually created. There is a way to assign value without creating a constraint, but it adds complexity, and we have never used it in this paper. Therefore, it is safe to assume that assigning a value to a signal creates a constraint. Every variable in this chapter will be of the type signal. This means that the values will be constrained automatically as we define them.

4.1.4 SnarkJS

SnarkJS is a JavaScript library that provides tools for working with zk-SNARKs, including compiling circuits defined in Circom. After writing our circuit using Circom, SnarkJS generates the r1cs and witness file for it. It then uses those files to create the zk-SNARK proof. SnarkJS also provides utilities for verifying the proofs. The library provides a verified way to generate a valid SNARK.

4.1.5 Benchmark

We use Groth16 as a SNARK design choice because it is simple and fast. We will benchmark the circuits using the prover time and verifier time. The prover time does not include the time to generate the setup. The setup needs to be generated for every circuit, but it only needs to be generated once. This is why we exclude it from our benchmarks. The prover time is the time it takes to generate the proof, including generating and folding the constraints when we use the folding scheme. The verifier time is the time to verify the proof.

As mentioned in the KZG section, a trusted setup must precompute values. These values need to remain a secret or, even better, destroyed. This introduces a potential vulnerability, as you need to think about who will do the setup. Typically, multi-party computation ceremonies are used. In practice, someone could use a universal setup like PLONK, which makes key generation easier because we only need one for every circuit. Groth 16 requires one secret per circuit. Someone could also use a transparent setup (STARKS), which does not require any secret.

4.1.6 Tree construction

The proof is based on constructing a Merkle tree, where the leaves are the users' balances and hashes. The tree's root is the root hash and root sum, where the root sum is the sum of all balances or total liabilities.

The size of the tree is dynamic with the number of users. For instance, if we have $2^5 - 10$ users, we will use a tree of 5 levels. This means we will have 10 empty nodes. We will use an empty hash and zero as the empty nodes' values. Those values are valid inputs for the MiMCSponge hash construction and will give a valid hash as an output. An empty node has no impact on the state of the tree.

The size of the tree decides the number of constraints. We need a separate setup for every tree size, but those setups are precomputed. For instance, a power of Tau of 8 (8 refers to the number of degrees of the QAP polynomial) can handle up to 256 constraints, 9 for 512 constraints, and so on.

4.2 Proof of liabilities

The proof of liabilities operates on a list of balances and a list of email hashes as private inputs kept private to preserve privacy. The proof of liabilities aims to generate a Merkle root without revealing additional information about the users and their balances. The first purpose of the circuit is to validate that all values are non-negative and that all balances fall within a specified range.

Since operations occur within a finite field, these verifications are crucial to prevent overflow or underflow issues. An insolvent exchange can deceive a proof of solvency by exploiting overflows in finite field arithmetic. It can encode negative balances as large positive values that will reduce the size of the liabilities. For instance, assume the total liabilities of an exchange is 1000. It can encode a negative value -400 as a large positive value $q - 400$. The exchange would then add this false balance to the tree, and the new total liabilities would be 600. The range-check constraint prevents this type of attack. A balance is assured to contribute positively to the sum, or at least not negatively (if the balance is 0). Therefore, there is no incentive to encode false balances for the exchange.

Subsequently, the proof of liabilities constructs a Merkle tree and outputs the total balance sum and the root hash of the Merkle tree. The pseudocode for the circuit is found in Appendix A.1.

Inputs

- List of balance (private): List of user balances, kept secret to protect individual amounts.
- List of email hash (private): User emails, hidden to protect identities and used in the Merkle tree. It is included to prevent clash attacks.

Outputs

- Balance Sum: Total of all balances, showing overall liabilities without revealing individual amounts.
- Root hash: Merkle tree's top value, used to check the tree's integrity.
- All small range: True if balances are within a safe range.

This proof of liabilities operates as intended because it returns the sum of the liabilities and the root hash, ensuring we cannot alter any values inside the Merkle tree. The Merkle tree is hidden, so we do not give any information about users and their balances. The root hash will be used to verify the inclusion of the balances.

The balance sum would be a private output in a complete proof of reserves. We would have another circuit proving that the sum of liabilities is smaller than the sum of assets without revealing the balance.

The circuit follows the zero-knowledge properties we highlighted previously.

Properties

- Completeness: A Merkle sum tree will be produced alongside the proof if the input balances are non-negative and within the range. The arithmetic constraints and range checks enforce this.
- Soundness: A valid proof cannot be produced if any input balance is negative or outside the range. Moreover, the Merkle sum tree cannot produce an inaccurate root hash and sum if the inputs are valid due to constraints ensuring sum accuracy and Merkle path integrity.
- Zero-Knowledge: The verifier learns only the public outputs and does not gain any information about the individual inputs. The hash provides not useful information.

4.3 Proof of inclusion

The proof of inclusion aims to prove that a user's balance is included in the Merkle tree created in the proof of liabilities. To do so, the exchange reveals the leaf associated with the user, which is the hash of the user's email and balance. This data will be the public input to the circuit alongside the root data. The exchange uses private inputs to show that the leaf is part of the tree. The circuit combines the user's data with the neighbors' data to show that the hash of the user's leaf is part of the hash of the Merkle root, showing at the same time that the user's balance is included in the total liabilities. The neighbors' data refers to the data of the sibling nodes along the unique path from the user's leaf to the root. It includes their sum, hash, and binary. The neighbors' binary variable indicates whether the neighbor is on the left or the right, which is needed to calculate the proper hashing order.

In short, we use the generated hash root in the proof of liabilities to prove that the balance is included in the Merkle tree. It checks if a path through the tree leads to the correct top value, proving the balance is included without revealing other data. To prove that a user balance is included, it is sufficient to show that you know the Merkle path of that balance.

If the exchange wanted to prove a leaf not included in the Merkle tree, it would not be able to do so.

For the following figure, the blue nodes represent the value of the Merkle path. We would have $neighborsSum = [29, 61]$, $neigborsHash = [Hash(User1), Hash(L2, R2)]$ and $neighborsBinary = [0, 1]$.

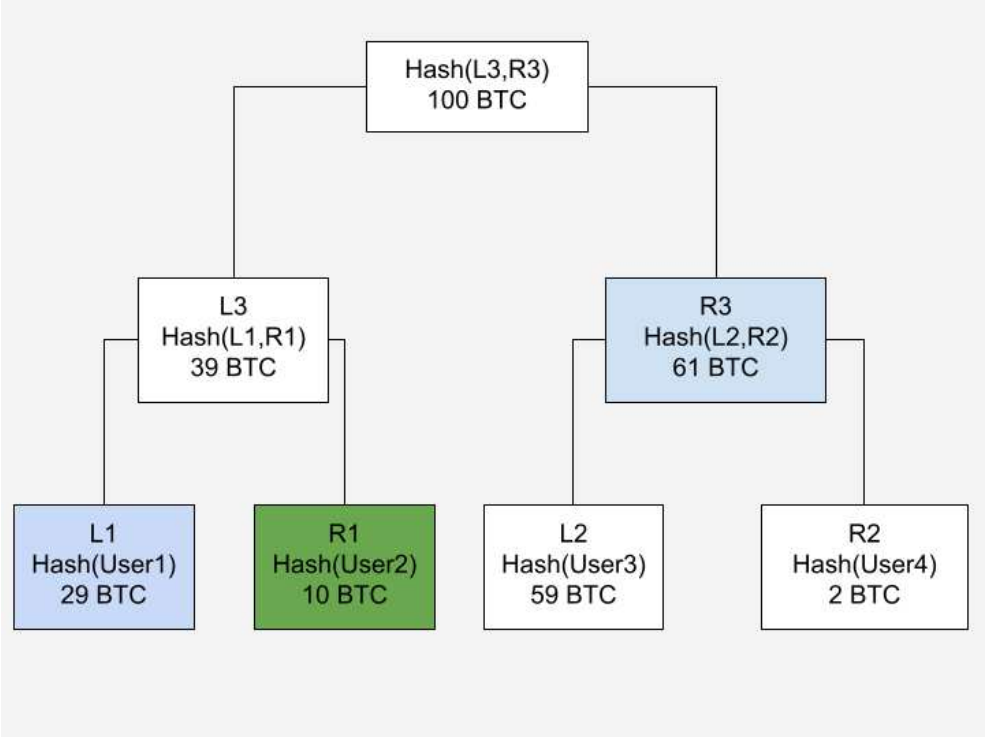


Figure 4.2: Merkle Path

Inputs

- List of neighbors sum (private): Sums of nearby tree nodes, kept secret to hide the tree's structure.
- List of neighbors hash (private): The hash value of nearby nodes, hidden for privacy.
- List of neighbors binary (private): Value showing if a neighbor is left (0) or right (1).
- Root hash (public): Merkle tree's top value, shared to identify the tree.
- Root sum (public): Total balance sum, shared for verification.
- User balance (public): User's balance, shared to prove it is included.
- User email hash (public): User email hash shared to identify the user.

Outputs

- balanceIncluded: True if the balance is in the Merkle tree, confirming it is part of the data.

The constraints in the circuit ensure that the path from the leaves to the root is correct. In the circuit, we verify that combining the user balance, sum, and Merkle path gives the correct root hash and root sum. The circuit iterates over the path, combining values to recompute the root without revealing private data. No additional verifications are required since they have already been done for this root hash in the proof of liabilities. Constraints enforce the hash and sum calculations and path directions. Each new value in Circom is defined in a way that adds a constraint. For instance, the Root sum is constrained to be equal to its two children; the children are constrained to be equal to their two children, and so on.

Once again, the circuit follows the zero-knowledge properties.

Properties

- Completeness: An honest prover can provide a valid Merkle path if a user's balance and email hash are included in the Merkle tree. This is verified by recomputing the public root hash and sum with constraints.
- Soundness: If the user's balance or email hash is not in the Merkle tree, or if the Merkle path is incorrect, no dishonest prover can produce a valid proof. This is because the hash function is collision-resistant.
- Zero-Knowledge: The verifier only learns whether the user's balance is included in the Merkle tree and nothing about the private inputs. All of the tree's internal structure is preserved with the SNARK privacy properties.

4.4 Vulnerabilities

In this section, we discuss the potential vulnerabilities of our proofs.

4.4.1 Clash attack

A clash attack can happen when two users have the same balance on the exchange, and the node of the tree is not linked to the user. For instance, if Alice and Bob both have 5.5 BTC on the exchange, the exchange could tell them that their balance is in leaf 157. It would then create a single proof of inclusion for leaf 157 and show it to Alice and Bob. Both would be happy because they would know that their balance is included in the total liabilities.

However, if we change leaf 157 for leaf x, where x is the hash of Alice's email, then the exchange cannot use the same leaf for Bob.

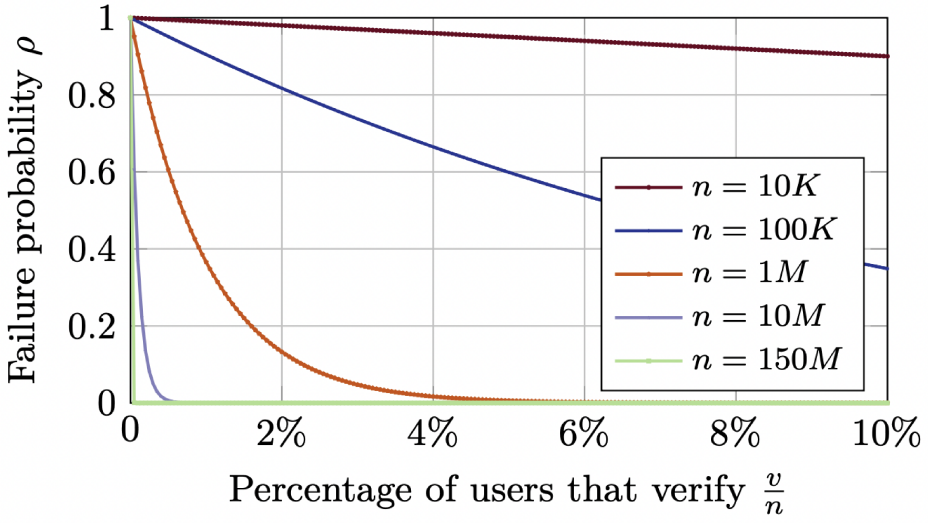
4.4.2 Collusion

Collaboration attacks occur when two exchanges collude to exchange funds to show that they are more solvent than they actually are. This attack affects the proof of assets, indirectly affecting the proof of liabilities. Since we do not want exchanges to move funds around in between proofs, we need to generate the proof of assets at a higher frequency and, therefore, the proof of liabilities at a higher frequency.

4.4.3 User verification

For the proof of liabilities to be valid, we must verify that the user's balance is included in the Merkle tree. The users must verify their proof of inclusion to validate the proof of liabilities. However, how many user verifications do we need to trust the proof of liabilities?

In this chart, we can observe the failure probability, which represents the likelihood that a dishonest prover gets away with misbehaving.



(c) $\tau = 0, c = 0.01\% \cdot n$ and varying n

Figure 4.3: Failure Probability [21]

If we take the orange line, for example, where an exchange has 1 million users, the failure probability approaches 0 when over 4% of users perform their verifications. This might not seem like a reasonable threshold. However, this is only for one proof. We would need 4% of users to verify every single proof. If we do a daily proof to avoid collusion, we are unlikely to reach the threshold every single day.

The last two attacks are the main vulnerability of the proof of inclusion and liabilities. To prevent these attacks, we need to generate the proof of liabilities more frequently and make it easier for the user to verify its proof of inclusion.

4.5 Daily proof of liabilities

In order to generate a daily proof of liabilities, we need to reduce the computational effort. We aim to enhance our current circuit by minimizing the work needed in subsequent rounds. The first thing to explore is recursion proofs, specifically created to reduce the total computational effort across rounds.

4.5.1 Recursion scheme

The main advantage of the recursion scheme is that it streamlines the verification process. For instance, in the second round, we can prove the integrity of the current and all previous rounds. However, this benefit comes with trade-offs. The first drawback is the increase in proof size, as it necessitates proving the current circuit and verifying the previous ones.

When considering the frequency of verification, having a fixed number of nodes verify the proof daily would be illogical, as it would be illogical to have the nodes verify the previous rounds every single day. On the other hand, if the verification is not consistent or new nodes need to be able to quickly verify every proof, then aggregation becomes more appealing.

In our case, the priority is to produce daily succinct proofs. We need to ensure the integrity of every round while keeping the proof size to a minimum. Therefore, having our nodes verify the circuit at every round without the computational overhead of the aggregated proof is sufficient.

4.5.2 Other recursion proofs

The recursion scheme is the only recursion proof that is proper in specific scenarios. The other types of recursion proofs, namely Aggregation, Accumulation and Folding schemes, all have one thing in common. They are all designed to verify multiple rounds concurrently. This approach is not aligned with our objectives. We are interested in working on rounds independently and reducing the individual workload.

4.5.3 Change circuit

If we cannot use any recursion schemes, we need an alternative approach to reduce the complexity of subsequent rounds. Our solution is to reutilize the same Merkle tree as the previous rounds and modify and adapt it to include the changes. The key challenge is that the Merkle tree was built inside the circuit and is, therefore, inaccessible.

In our modified circuit, we will adjust the Merkle tree inside the circuit. We send the corresponding Merkle path for every change, which the circuit will verify. The circuit will then compute a new Root Hash for each change and output the final Merkle Hash. Internally, the circuit iterates over each change, verifies the old path, updates the leaf with new values, recomputes the hash, and sums up the tree.

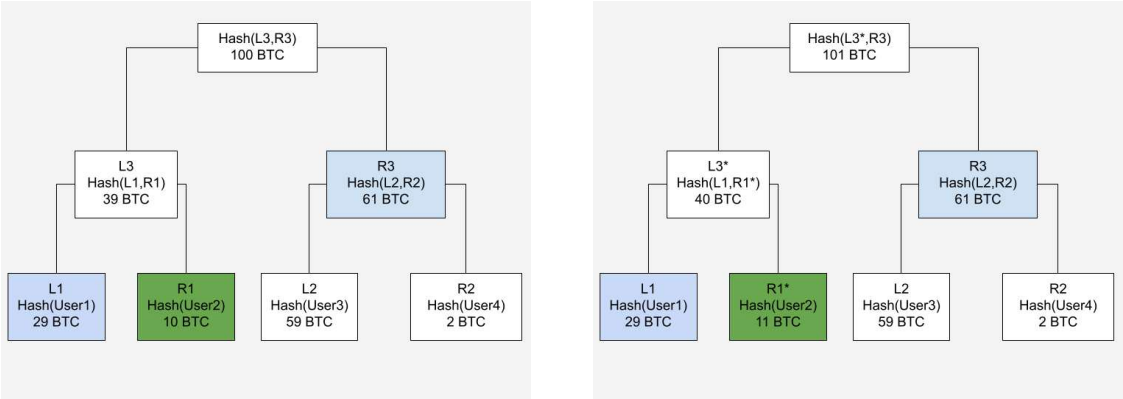
The standard verification will be applied to the new values. The constraints ensure correct hash and sum updates, valid path directions and valid ranges.

Inputs

- List of old email hash (private) - 1 per change: Hashed email addresses from the previous Merkle tree, kept secret to protect user identities.
- List of old values (private) - 1 per change: Previous balances, hidden to maintain user privacy.
- List of new email hash (private) - 1 per change: Updated hashed email addresses, kept secret to ensure user confidentiality.
- List of new values (private) - 1 per change: Updated balances, hidden to safeguard individual amounts.
- List of temporary root hash (private) - 1 per change: Intermediate Merkle tree top values, kept secret to conceal tree updates.
- List of temporary root sum (private) - 1 per change: Intermediate total balance sums, hidden to preserve privacy.
- Old root hash (public): Previous Merkle tree's top value, shared to verify the starting tree.
- Old root sum (public): Previous total balance sum, shared to confirm the initial sum.
- List of neighbors sum (private) - 1 list per change: The sum of nearby nodes in Merkle paths is kept secret to hide tree structure.
- List of neighbors hash (private) - 1 list per change: Hashed nearby nodes in Merkle paths, hidden to ensure privacy.
- List of neighbors binary (private) - 1 list per change: Left/right indicators (0 or 1) for Merkle paths, kept secret to protect path details.

Outputs

- Valid hash: True if the new Merkle hash is correct, confirming tree integrity.
- Valid sum: True if the new sum is correct, ensuring accurate totals.
- All small range: True if new balances are within safe limits to avoid overflow.



(a) Pre change

(b) Post change

Figure 4.4: Merkle tree change

- New root hash: Final Merkle tree top value, used for future verifications.
- New root sum: Final total balance sum, reflecting updated liabilities.

Here is an example of a change. In this case, the change is a change in balance. First, we prove that the 10 BTC of user one is included in the Merkle tree. Then, we prove that the 11 BTC of user one is included in the new Merkle tree. After the last change, we are left with the new root balance and hash. Note that for every change, we only need 1 Merkle path.

For every change in the Merkle tree, we have the Merkle path with the old values, the new values, the temporary root hash, and the temporary root sum. The circuit is iterating over the changes and gives a final root hash and final root sum. The circuit verifies each old path, updates the leaf with new values, recomputes the hash, and sums up the tree. The constraints enforce hash and sum calculations, valid path directions, and range checks.

The change circuit also follows the zero-knowledge properties.

Properties

- Completeness: If the changes are valid, with correct old and new Merkle paths, an honest prover can generate a proof that convinces an honest verifier of the statement. Constraints on the paths and updates enforce this.
- Soundness: If any change is invalid or the old state is manipulated, no dishonest prover can produce valid proof because of the constraints.
- Zero-Knowledge: The verifier learns only the public outputs and nothing about the private inputs.

Minimizing the number of changes

To minimize the number of constraints, we want to minimize the number of changes. We define a change as a hash change at the leaf level. Any new user, balance change, or removal of the old user

is considered a change. The naive change calculation would be $changes = balanceChanges + newUsers + deprecateUser$. However, a new user can take the node of a deprecated user. So we would have the equation $changes = balanceChanges + max(newUsers, deprecateUser)$. Constraints enforce this calculation, reducing the circuit's complexity by limiting updates.

Constraint analysis

The new circuit makes sense theoretically. It should be faster than the original one, so let us analyze it. We want to compare the number of constraints in the original circuit with those in the changed circuit. The first assumption is to assume the number of changes in a day. A completely arbitrary value would be 1%.

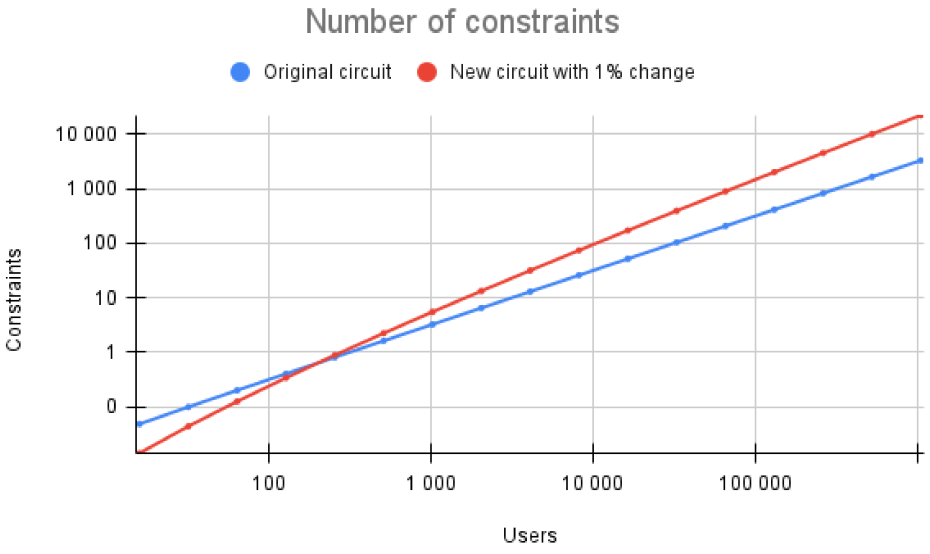


Figure 4.5: Number of constraints original circuit vs new circuit with 1% change

Once you reach more than 128 users with 1% change, it is not worth it to use the new circuit. This is not practical because any marketplace will have more than 128 users. However, we can see that the number of constraints of the new circuit grows linearly with the number of changes. This means that two circuits with 0.5% changes would have a similar number of constraints as one circuit with 1% changes. We can take this a step further and look at daily proof instead of hourly proof. With the assumption that 1% changes daily, we will assume that 0.05% of the balances change hourly. Let us look at the number of constraints if 0.05% of users change every hour.

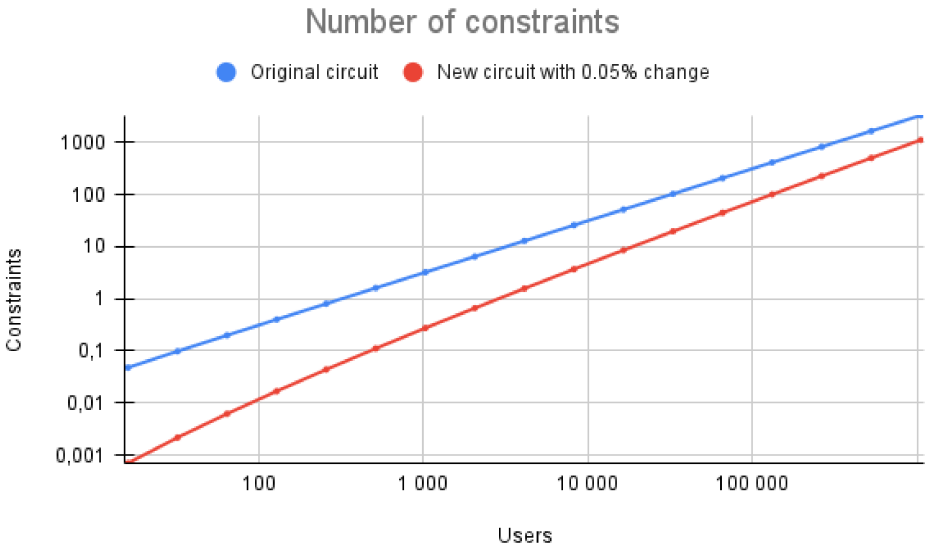


Figure 4.6: Number of constraints (millions) original circuit vs new circuit with 0.05% change

Even with more than 1 million users, our new circuit has three times fewer constraints. So, if we want to do an hourly proof, the new circuit is less expensive than the old circuit. We can take this further and provide proof at every new block. We could have a Merkle tree up to date for the liabilities side at every block.

While the new circuit improves the hourly and block proofs, a daily proof may be sufficient for most use cases. It is still less work to do the daily proof with the original circuit than 24 hourly proofs with the new circuit.

4.6 Daily proof of inclusion

Because a marketplace can have millions of users, it is impractical to build a proof of inclusion for every user daily. The user is responsible for requesting proof that their balance is included in the published Merkle tree.

Ideally, a Merkle tree would be published at least daily. However, it is once again impractical to require every user to verify their inclusion daily. Nevertheless, each user verifying their inclusion in the tree increases the chance that the proof of liabilities is valid and includes every balance. This is why it is primordial to find a way to make it easier to verify the inclusion. This is where Nova folding schemes come in. Nova suggested using the folding scheme to verify the proof of inclusion from multiple rounds simultaneously[12]. This is used in a zero-knowledge proof of reserves tool called Summa [30]. They implemented a similar proof of inclusion to what we are working on. They have similar Merkle tree inputs, outputs and constraints. However, they use Halo2, and their circuit can handle multiple cryptocurrencies. Halo2 is an established tool for generating SNARKS, using Plonk instead of Groth16, like us.

The novel way to generate proof of inclusion is to generate a proof of inclusion that is valid from the day the account is created to the day the user requests it. Normally, you would need to cre-

ate 500 different proofs for 500 days. However, as we saw in the previous section, the Nova folding scheme enables combining 500 proofs into just one. The folding scheme circuit compresses multiple proofs by combining their constraints. Verifying this one proof is the equivalent of verifying the 500 or any other number of days required. This drastically simplifies the verification process.

Previously, when requesting a proof of inclusion, the user would only receive proof that their balance is included in the latest published Merkle tree. Now, the proof verifies that the balance has been included in every previously published Merkle tree. This ensures that any malicious entity would be unable to alter ownership of balances across multiple days.

It might seem like a small detail, but it is the detail that makes all the difference. Let us evaluate why. In the initial round, with no changes, the failure probability for 20k users remains at 13%.

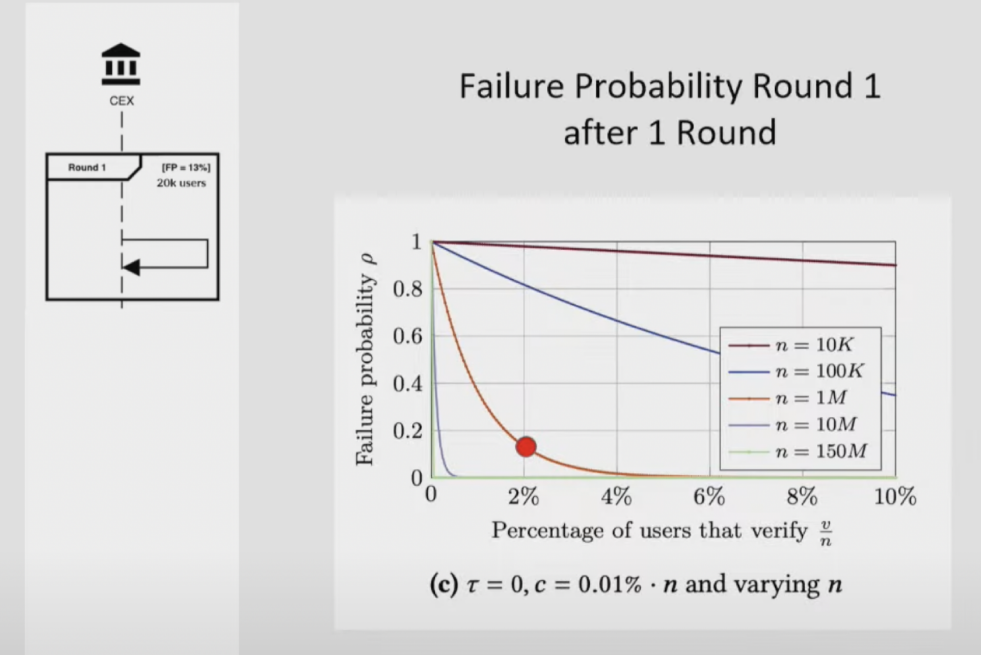


Figure 4.7: Failure Probability Round 1 [12]

However, if we have 20k distinct users in round 2, the failure probability of round 1 decreases to 1.6%.

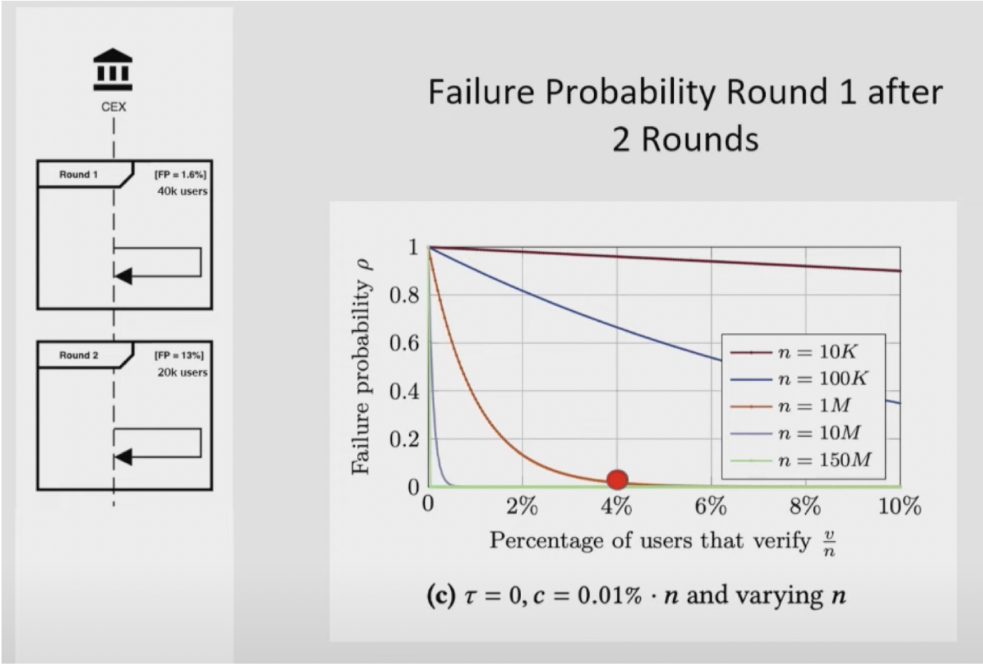


Figure 4.8: Failure Probability Round 1 After 2 Rounds [12]

If we have 20k different users in round 3, the failure probability of round 1 further decreases to 0.2%.

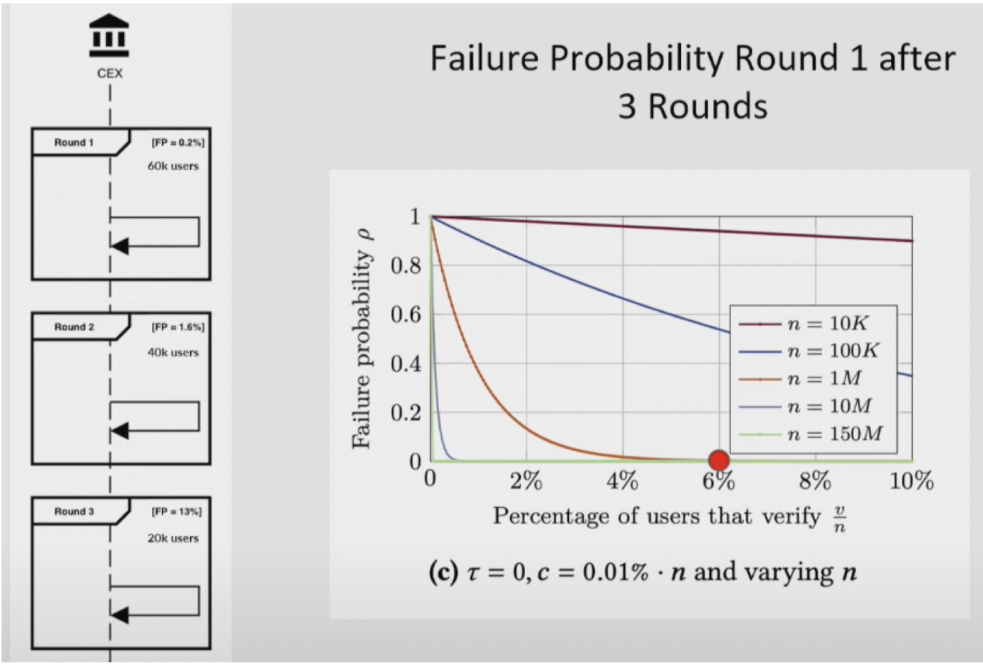


Figure 4.9: Failure Probability Round 1 after 3 Rounds [12]

The key takeaway is that we require 4% of users to verify every round without folding. However, with folding, we only need 4% of users to participate in at least 1 round. This significantly

reduces the burden on the user to verify while increasing confidence in the proof.

4.6.1 Circuit Design

In order to implement folding, we need to adjust our circuit slightly. Everything except the way we handle inputs and outputs stays the same. The private inputs vary for every instance, while the public inputs are carried over from round to round. The circuit verifies the Merkle path for a user's balance, recomputing the root hash and sum via hashing.

Inputs

- List of neighbors sum (private): Sums of nearby Merkle tree nodes, kept secret to hide the tree structure.
- List of neighbors hash (private): Hashed values of nearby nodes, hidden to protect tree privacy.
- List of neighbors binary (private): Left/right indicators (0 or 1) for the Merkle path, kept secret to conceal path details.
- Root hash (private): Merkle tree's top value, hidden to maintain proof privacy.
- Root sum (private): Total balance sum, kept secret to protect aggregate data.
- User balance (private): User's balance, hidden to preserve individual privacy.
- User email hash (private): Hashed user email, kept secret to protect identity.
- Steps in (public): Public values from the previous circuit's output, shared to link folded rounds.

Outputs

- Steps out (public): Public values passed to the next circuit, enabling folding across rounds.

Steps

- Balance included: Confirms the user's balance is in the Merkle tree.
- Root sum: Total balance sum for the tree.
- Root hash: Merkle tree's top value.
- User balance: User's balance value.
- User email hash: Hashed user email.

All the data passed around between the circuits is public and is called step in and step out, while the regular inputs are private. The step in of a circuit is the step out of the previous one.

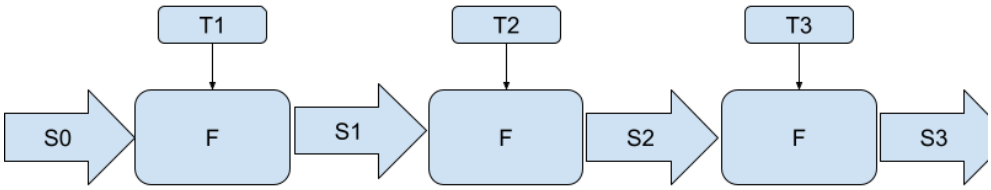


Figure 4.10: Folding Circuit

S0 is the step in of the first change, S1 is the step out of the first change and the step in of the second change. F is the folded circuit and T is the number of times it is folded.

These steps encompass all the public values we initially had in the circuit. None of the variables of the steps are used in the circuit itself. They are used to make a value public.

The initial circuit takes meaningless values as inputs, while the following circuits take the public values of the previous circuit. In the end, we only have to verify a single proof. We must also compare the circuit output with the proof of liabilities outputs for every round.

4.7 Folded daily proof of liabilities

With the original circuit, we saw that folding or any other recursion scheme was ineffective. However, with the new circuit, we were able to separate a big proof into multiple smaller ones. This aligns perfectly with the recursion ideas. We can separate our new circuit of n changes into m circuits, where $m <= n$, and fold the circuits together. The folding process combines constraints from each circuit.

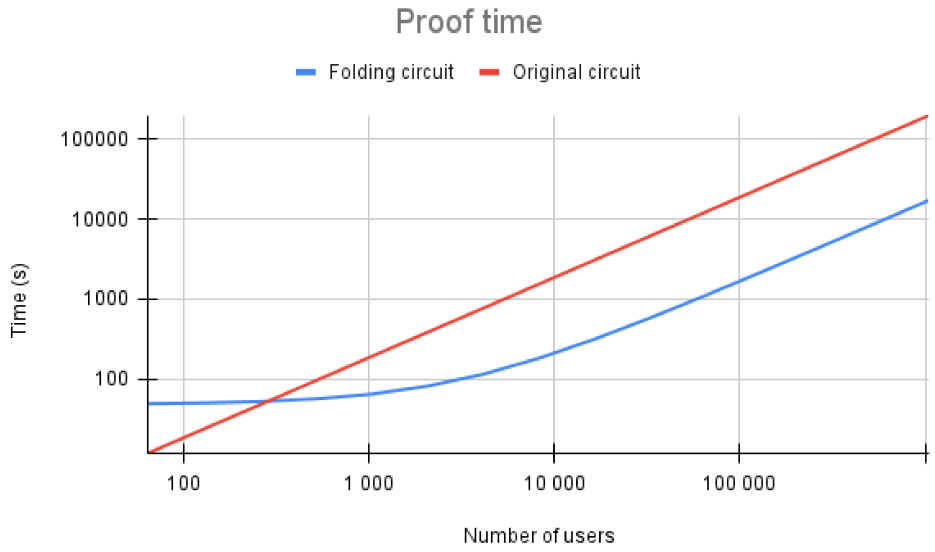


Figure 4.11: Proof time original circuit vs folded circuit with 1% change, 1 change by fold

After 10,000 users, both circuits grow linearly, which is in line with Groth16's big O notation of O(n). The folded circuit has a lower constant factor, which is why it is faster. However it has an initial cost to pay, which is why it lags behind at the beginning. With the folded circuit doing much better. For instance, for 1M users, the old circuit takes 196,608 seconds, and the new circuit take 17 188 seconds. This is more than 10 times better. We should take the number of seconds with a grain of salt since these measurements were done on my Mac, and there are much more performant computers. The graphs were generated by interpolating the data from the tests we did. The data we should focus on is that the folded circuit took 10x less time, and it was not even optimized. We were doing one change by circuit, so at 1M users, we folded 10k circuits.

Now that we know our circuit is better, we can optimize it.

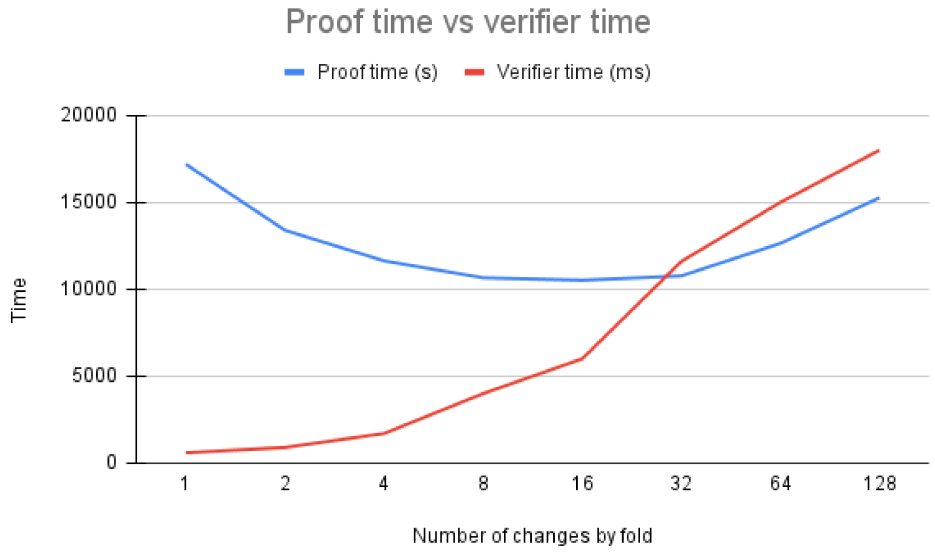


Figure 4.12: Optimization of the proof time and verifier time of the folded circuit with 1% change and 1M users

As we can see, the optimal time for the prover is between 8 and 32 changes by fold, while the verifier time seems to be growing at a monotonic pace. We can decide the number of changes we should have for every fold depending on the use case.

4.7.1 Circuit design

The folded change circuit is the same as the regular change circuit, but the inputs work differently because some data is passed around the circuits. There are two types of step variables: those needed in the following circuit and those just there to be public. Starting from the inputs of the regular change circuit, the old hash and old sum root are moved to the step in, while all the outputs are moved to step out. The circuit processes each change by verifying old paths, updating leaves, and recomputing hashes and sums.

Inputs

- List of old email hash (private) - 1 per change: Hashed user email from the previous tree, kept secret to protect identities.
- List of old values (private) - 1 per change: Previous balances, hidden to preserve privacy.
- List of new email hash (private) - 1 per change: Updated hashed email, kept secret for user confidentiality.
- List of new values (private) - 1 per change: Updated balances, hidden to protect amounts.
- List of temporary root hash (private) - 1 per change: Intermediate tree top values, kept secret to hide updates.

- List of temporary root sum (private) - 1 per change: Intermediate balance sums, hidden for privacy.
- List of neighbors sum (private) - 1 list per change: Sums of nearby nodes in Merkle paths, kept secret to conceal tree structure.
- List of neighbors hash (private) - 1 list per change: Hashed nearby nodes in Merkle paths, hidden for privacy.
- List of neighbors binary (private) - 1 list per change: Left/right indicators (0 or 1) for Merkle paths, kept secret to protect path details.
- Step in (public): Public values from the previous circuit, shared to link folded circuits.

Outputs

- Steps out: Public values passed to the next circuit, enabling folding.

Steps

- Valid hash and valid sum: Confirms the new hash and sum are correct.
- All small range: Ensures balances are non-negative and within safe limits.
- Root hash: Final Merkle tree top value.
- Root sum: Final total balance sum.

Chapter 5

Conclusion

By combining a new circuit design, prior proofs, and the Nova folding scheme, we achieve significant reductions in proof size, enabling daily or even hourly generation. While the relative cost of each proof decreases with higher frequency, the absolute cost rises compared to current monthly proofs. This tradeoff is worthwhile, as monthly proofs are insufficient. Smaller exchanges may adopt this approach more easily, given their lower rate of changes. Our analysis confirms the superior performance of the folding scheme with the updated circuit, establishing concise liability proofs as a cornerstone for robust solvency systems.

A proof of assets remains necessary to complete proof of solvency. Handling multiple currencies is a key challenge, but the folding scheme allows asset proofs to be segmented per currency before aggregation, reducing complexity.

Security considerations persist. Reliance on users to verify inclusion proofs is insufficient to guarantee a valid tree, and each proof assumes state validity, forcing verifiers to check all proofs individually. Recursive proofs offer a solution: each proof verifies the previous one, allowing a verifier to validate only the latest proof.

Performance can also be improved. The change circuit can be parallelized, unlike the folding circuit, which behaves like a linked list. This makes the change circuit central to the long-term goal of live proving for example, parallelizing 100 change proofs in a single block before aggregation.

Further work includes exploring alternative SNARK designs, testing different hash functions, and optimizing constraint counts. While Groth16 provides benchmarks, other SNARKs may mitigate trusted setup or post-quantum risks. Likewise, Poseidon or Keccak may offer better tradeoffs in speed or security. The current implementation uses a secure but naive constraint construction, leaving room for merging constraints to reduce overhead.

Finally, rigorous auditing is essential before deployment. While the Merkle tree design prevents false balances, correctness depends on comprehensive and accurate constraint implementation.

By addressing these challenges, the proposed system can deliver scalable, secure, and practical solvency proofs for blockchain marketplaces, and represents the right step towards live proving: a proof every block.

Appendix A

Sample Code

This section contains the pseudocode for the circuits used in both the proof of liabilities, and proof of inclusion.

A.1 Proof of liabilities

Listing A.1: Liabilities circuit pseudocode

```
# Define a function for the Merkle tree sum circuit
def sum_merkle_tree(levels, inputs, balances[inputs],
                            email_hashes[inputs]):
    # Outputs variables for sum, root hash, and range checks
    root_sum = 0
    root_hash = 0
    all_small_range = False

    # Define signals lists for sum and hash nodes at each level
    sum_nodes = [[0] * inputs for _ in range(levels+1)]
    hash_nodes = [[0] * inputs for _ in range(levels+1)]

    # Initialize variables
    level_size = inputs
    max_bits = 100
    temp_not_big = 0

    # Loop through each input
    for i in range(inputs):
        # Assign input values to hash and sum nodes
        hash_nodes[0][i] = email_hashes[i]
        sum_nodes[0][i] = balances[i]

        # Perform range check
        rangecheck = RangeCheck(maxBits, balances[i])
```

```
        tempNotBig += rangecheck

    # Check if all balances are within a small range
    if temp_not_big == inputs:
        all_small_range = True


    # Loop through each level
    for i in range(levels):
        # Loop through each pair of nodes at the current level
        for j in range(0, level_size, 2):
            # Store sum and root hash for the next level
            sum_nodes[i+1][nextLevelSize] =
            sum_nodes[i][j] + sum_nodes[i][j+1]
            hash_nodes[i+1][nextLevelSize] =
            hash_nodes[i][j] + hash_nodes[i][j+1]

        # Update the size of the current level
        levelSize = nextLevelSize;
        nextLevelSize = 0;

    # Assign final sum and root hash values
    root_sum = sum_nodes[levels][0]
    root_hash = hash_nodes[levels][0]

    return root_sum, root_hash, all_small_range
```

A.2 Proof of inclusion

Listing A.2: Inclusion circuit pseudocode

```
# Define a function for the inclusion proof circuit
def inclusion(levels, neighborsSum, neighborsHash, neighborsBinary,
              step_in, sum, rootHash, userBalance, userEmailHash):
    # Initialize sum and hash nodes
    sumNodes = [0] * (levels+1)
    hashNodes = [0] * (levels+1)
    sumNodes[0] = userBalance
    hashNodes[0] = userEmailHash

    # Iterate through each level
    for i in range(levels):
        # Update hash node
        if neighborsBinary[i]:
```

```
            hashNodes[i+1]=getHash(neighborsHash[i],
            neighborsSum[i],hashNodes[i],sumNode[i])
        else:
            hashNodes[i+1]=getHash(hashNodes[i],sumNode[i],
            neighborsHash[i],neighborsSum[i])
        # Update sum node
        sumNodes[i + 1] = neighborsSum[i] + sumNodes[i]

    # Check validity of root hash
    validHash = IsEqual([hashNodes[levels], rootHash])

    # Check validity of sum
    validSum = IsEqual([sumNodes[levels], sum])

    return validSum, validHash
```

A.3 Change circuit

Listing A.3: Liabilities change circuit pseudocode

```
# Define a function for the liabilities change proof circuit
def liabilities(levels, changes,oldEmailHash[changes],
        oldValues[changes],newEmailHash[changes],newValues[changes],
        tempHash[changes+1],tempSum[changes+1],
        neighborsSum[changes][levels],neighborsHash[changes][levels],
        neighborsBinary[changes][levels]):
    # Calculate newRootHash and newSum
    newRootHash = tempHash[changes+1]
    newSum = tempSum[changes+1]
    oldSum = tempSum[0]
    oldRootHash = tempHash[0]
    currentSum = oldSum

    # Part 1: Check validity of new values
    tempNotBig = 0
    maxBits = 100

    # Iterate through each change
    for i in range(changes):
        # Calculate currentSum
        currentSum += newValues[i] - oldValues[i]

        # Perform range check
        rangecheck = RangeCheck(maxBits,newValues[i])
```

```
        tempNotBig += rangecheck

    # Check if all new values are within range
    allSmallRange = IsEqual([changes, tempNotBig])

    # Check if newSum equals currentSum
    equalSum = IsEqual([newSum, currentSum])

    # Part 2: Check validity of old and new paths
    # Ensure that old root + change = temp root
    tempValidHash = [0] * (changes + 1)
    tempValidSum = [0] * (changes + 1)
    tempOldHashEqual = [0] * (changes + 1)
    tempOldSumEqual = [0] * (changes + 1)
    tempValidHash[0] = 1
    tempValidSum[0] = 1

#For each level, we are veryfing the inclusion of the value changing
#and the new value.
#hashNode[0] and sumNodes[0] is for the inclusion of the value changing
#hashNode[1] and sumNodes[1] is for the inclusion of the new value

    for j in range(changes):
        for i in range(levels):
        if neighborsBinary[j][i]:
            hashNodes[0][j][i+1]=getHash(neighborsHash[j][i],
            neighborsSum[j][i],hashNodes[0][j][i],sumNode[0][j][i])
            hashNodes[1][j][i+1]=getHash(neighborsHash[j][i],
            neighborsSum[j][i],hashNodes[1][j][i],sumNode[1][j][i])
        else:
            hashNodes[0][j][i+1]=getHash(hashNodes[0][j][i],
            sumNode[0][j][i],neighborsHash[j][i],neighborsSum[j][i])
            hashNodes[1][j][i+1]=getHash(hashNodes[1][j][i],
            sumNode[1][j][i],neighborsHash[j][i],neighborsSum[j][i])

        sumNodes[0][j][i+1] = sumNode[0][j][i] + neighborsSum[j][i]
        sumNodes[1][j][i+1] = sumNode[0][j][i] + neighborsSum[j][i]


        # Old calculated hash is in tempHash
        hashEqual = IsEqual(hashNodes[0][j][levels], tempHash[j])
        tempOldHashEqual[j+1] = tempOldHashEqual[j] * hashEqual

        # New temp hash is valid
        hashEqual = IsEqual(hashNodes[1][j][levels], tempHash[j+1])
```

```
        tempValidHash[j+1] = tempValidHash[j] * hashEqual

        # Old calculated sum is in tempSum
        sumEqual = IsEqual(sumNodes[0][j][levels], tempSum[j])
        tempOldSumEqual[j+1] = tempOldSumEqual[j] * sumEqual

        # New temp sum is valid
        sumEqual = IsEqual(sumNodes[1][j][levels], tempSum[j+1])
        tempValidSum[j+1] = tempValidSum[j] * sumEqual

validHash = tempValidHash[changes] * tempOldHashEqual[changes]
validSum = tempValidSum[changes] * tempOldSumEqual[changes]

return (allSmallRange, validHash,
          validSum, newRootHash, newSum)
```

Bibliography

[1] Zero knowledge proofs, lecture 10. MOOC, 2023. Dan Boneh, Shafi Goldwasser, Dawn Song, Justin Thaler, Yupeng Zhang.

[2] Zero knowledge proofs, lecture 2. MOOC, 2023. Dan Boneh, Shafi Goldwasser, Dawn Song, Justin Thaler, Yupeng Zhang.

[3] Zero knowledge proofs, lecture 3. MOOC, 2023. Dan Boneh, Shafi Goldwasser, Dawn Song, Justin Thaler, Yupeng Zhang.

[4] L. A. An incomplete guide to folding: Nova, sangria, supernova, hypernova, protostar. https://taiko.mirror.xyz/tk8LoE-rC2w0MJ4wCWwaJwbq8-Ih8DXnLUf7aJX1FbU, August 2023.

[5] A. M. Antonopoulos. Mastering bitcoin: Unlocking digital cryptocurrencies. O'Reilly Media, 2017. 2nd edition.

[6] Bake. Merkle tree proof of reserves and liabilities: Raising the bar for enhanced transparency, Dec 2022.

[7] B. Barak. Lecture 14: Zero knowledge proofs. *CS127: Cryptography*, Spring 2016.

[8] E. Ben-Sasson, I. Bentov, Y. Horesh, and M. Riabzev. Fast reed-solomon interactive oracle proofs of proximity. 2023. Technion, Haifa, Israel and Cornell University, Ithaca, NY, USA.

[9] Binance. Proof of reserves. *Binance*.

[10] Binance. Proof of solvency. *Binance*.

[11] M. Blum, P. Feldman, and S. Micali. Non-interactive zero-knowledge and its applications. page 103–112, 1988.

[12] E. Bottazzi. Zk10: Incremental proof of solvency with nova. Presentation at ZK10 Summit, September 30 2023. Video: https://www.youtube.com/watch?v=sRAA1RYYHEs.

[13] S. Bowe, J. Grigg, and D. Hopwood. Recursive proof composition without a trusted setup. *Electric Coin Company*, 2023.

[14] V. Buterin. Quadratic arithmetic programs: From zero to hero. *Medium*, December 2016.

[15] B. Bünz, J. Bootle, D. Boneh, A. Poelstra, P. Wuille, and G. Maxwell. Bulletproofs: Short proofs for confidential transactions and more. pages 319–334, 2018. Accessed: 2024-09-08.

[16] B. Bünz, A. Chiesa, P. Mishra, and N. Spooner. Proof-carrying data from accumulation schemes. *Cryptology ePrint Archive*, (2020/499), 2020. Accessed: 2025-01-29.

[17] G. G. Dagher, B. Bunz, J. Bonneau, J. Clark, and D. Boneh. Provisions: Privacy-preserving proofs of solvency for bitcoin exchanges. 2015.

[18] O. Goldreich, S. Micali, and A. Wigderson. Interactive and noninteractive zero knowledge are equivalent in the help model? page 113–131, 1991.

[19] S. Goldwasser, S. Micali, and C. Rackoff. The knowledge complexity of interactive proof systems. *SIAM Journal on Computing*, 18(1):186–208, 1989.

[20] J. Groth. On the size of pairing-based non-interactive zero-knowledge proofs. In *Proceedings of the 2016 ACM Conference on Computer and Communications Security (CCS)*, pages 480–491. ACM, 2016.

[21] Y. Ji and K. Chalkias. Generalized proof of liabilities, 2021.

[22] z. L. Jim.ZK. Nova studies i: Exploring aggregation, recursion, and folding. *zkLinkBlog*, Nov 2023.

[23] A. Kate and G. M. Zaverucha. Polynomial commitments. December 2010. Accessed: 2024-09-08.

[24] A. Kothapalli, S. Setty, and I. Tzialla. Nova: Recursive zero-knowledge arguments from folding schemes. 2021.

[25] D. Malkhi. Exploring proof of solvency and liability verification systems. 2023.

[26] Mazars. Proof of reserve report. 2022.

[27] S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system. *Bitcoin.org*, 2008. Technical report.

[28] A. Nitulescu. zk-snarks: A gentle introduction. 2020.

[29] RisenCrypto. R1cs and qap, 2023.

[30] Summa. Merkle sum tree inclusion, 2024.

[31] L. Team. Introduction to zero-knowledge proofs. *LCX*, November 2023.

[32] Trace. The proof supply chain. *Figment Capital*, January 2024.

[33] E. van Oorschot, Y. Deng, and J. Clark. Folding (sangria). GitHub Pages, 2024.

[34] E. van Oorschot, Y. Deng, and J. Clark. Kzg commitments. GitHub Pages, 2024.

[35] Veridise. Halo and accumulation. *Veridise*, August 2023.

[36] Veridise. Nova and folding (1/2). *Veridise*, September 2023.

[37] Veridise. Recursive snarks and incrementally verifiable computation (ivc) part i: Recursive snarks and incrementally verifiable computation. *Veridise*, July 2023.