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 Test

September 2023

Concordia University
School of Graduate Studies

This is to certify that the thesis prepared

By: **Antoine Cyr**

Entitled: **Title**

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

_____ External Examiner

_____ Examiner

_____ Examiner

_____ Examiner

_____ Supervisor

Approved by _____

01 Sept 2024 _____

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 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

Contents

List of Figures

Chapter 1

Introduction

In the context of cryptocurrencies, the 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 gain universal acceptance due to the trust placed in these third parties. However, auditing a cryptocurrency marketplace presents certain challenges. Audits are not a scalable solution because the funds can be moved around more quickly than in the traditional finance world. 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 needs to trust that their data won't 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. The proof of solvency is made up of a proof of asset and a proof of liabilities, a structure that is self-explanatory.

This paper extends the work done on proof of solvencies, and places particular emphasis on the proof of liabilities. 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 along with 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 you demonstrate that for each customer, their 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 everyday, while maintaining complete privacy (i.e the Merkle Tree is kept private).

1.1 Outline

The 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, with a focus on real-world applications, is introduced.

Chapter 3 start going 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 scheme.

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. We modify the first proof of inclusion to prove the inclusion of many merkle tree at the same time, using the folding scheme. We adopt an alternate strategy for the proof of liabilities, aiming to reduce the size of generating the proof on the second day, given the original Merkle tree is already established. This involves developing a secondary circuit that constructs a Merkle tree, utilizing an initial Merkle tree alongside the changes as input. We then optimize it using the folding scheme.

The proof of liabilities circuits are evaluated and compared using the number of

contraints, the proof time and verifier time.

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 the various approaches adopted by leading exchanges 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

on its own, without the need of a middleman, typically financial institutions. Bitcoin is built on a peer-to-peer network, which means that every participant helps in safe keeping the transactions history, and propagating the new transactions. There is no single point of failure. This allows for transactions to occur in real time, in contrast to the delays encountered in the traditional finance world. Bitcoin defines two different 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 Transactions

For every participant of the network, 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. Both are one way functions, meaning you cannot derive the other way around. The public key serves as the unique identifier in the network, but it is the wallet address that typically defines a participant. The wallet address can be seen as a bank account number. When you send bitcoin to someone, you send it to their wallet address. To be able to send some bitcoin, you need to 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 make sure 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 owner of the address. The public key is then used to verify the authenticity of the signature. Sending a transaction is the easiest problem to solve. The real challenge is to keep track of who owns what, and to avoid the double spending problem. The methodology for managing this is to keep the history of every single transaction. The transactions are bundled up into blocks, and the chain of blocks creates the blockchain.

2.1.2 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, working on publishing new blocks. The nodes work collectively to establish order of transactions (sequencing). Every new transaction is broadcasted to all nodes. The nodes put the transactions into a block, and try to publish that block. In order 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 previous hash. Some nodes might accept different blocks, if multiple blocks are propagated at the same time. This would create 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 of the chains receives a new block, breaking the tie.

2.1.3 Proof-of-work

In order to submit a new block, a node has to find a hash with a specific number of leading zero bits. It is exponentially more difficult for every zero bit. This cryptographic puzzle serves as a barrier to entry, ensuring that a lot of computational power was spent on 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, you cannot change any value inside of it because the hash value would change. The immutability of the older blocks is what makes Bitcoin secure. To modify a block in the middle of the chain, you would need to redo the work of every single block made after that. The longest chain is determined by the cumulative proof-of-work invested in it. This is why we say that Bitcoin is secured as long as 51% of the nodes are honest. The chain with the majority of nodes working on it will grow up the fastest, thus will be the accepted chain. The difficulty of the new block is determined by an average, in order to generate blocks at a steady pace. There is also a bitcoin reward associated with mining (publishing) a block.

2.1.4 Merkle Tree

In the architecture of the blockchain, only the merkle root is stored in the block header. Nodes only keep the recent blocks in memory. For the older blocks, they keep only the block header in memory. 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 where the parent node is the hash of the child nodes. The tree is immutable because changing a single node would have an impact on 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 is crucial to understand that the Bitcoin obtained remains in the custody of the platform, rather than being deposited to a personal wallet address. In order to gain custody of your bitcoins in your wallet, you need to enter a transfer request. This is similar to traditional finance, where you trust the bank (marketplace), to go ahead with your transaction. Once you have bitcoins in your wallet, you can transact on the network without needing a 3rd party. Unless you are running a node, you will need to trust a 3rd party, whether it is a marketplace, or over the counter, to first acquire bitcoin.

When the bitcoin you own is in custody of the marketplace, there is no way to see your bitcoin onchain. Marketplaces have many wallets, some are made public and some are not. While this allo for the privacy of your marketplace deposits and withdrawals, it represents a fundamental contradiction. Bitcoin was designed to be transparent and public, without the need of a 3rd party to do a transaction. Not being able to track your bitcoin in the marketplace pauses a significant challenge. A user has no proof that the marketplace solvency to reimburse every client. However, this problem is being actively worked on. Marketplaces have begun to use proof of solvency (or proof of reserve) to demonstrate that they are solvent. While this is a step in the right direction, the current proof of solvency used by the marketplaces have many defaults, and they are not sufficient to prove that they are solvent.

2.3 Zero Knowledge

Zero-knowledge proofs is a cryptographic technique to prove some knowledge without divulging any informations.For instance, the classic way of proving that you know the solution to an equation, is to reveal the solution to the equation itself. However, with zero knowledge you are able to prove that you know the solution, without disclosing the solution. To construct a zero knowledge proof, you need to construct a proof that is sound and complete. You also need your proof to be zero knowledge[4].

- **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 fact that the statement is true. [21]

2.3.1 Non interactive proofs

Zero knowledge proofs were originally designed as interactive, that is, multiple rounds of interaction between the prover and the verifier [12]. leading to what are called interactive zero-knowledge proofs. This interaction allows the prover to demonstrate knowledge of the solution without revealing any additional information. An alternative model was then proposed where the verifier and prover use a reference string that is shared during a trusted setup. Once we have the reference string, a single message is needed between the prover and the verifier. The elimination of multiple rounds of interaction simplifies the verification process and reduces the computational power required. Therefore, noninteractive zero-knowledge proofs offer enhanced efficiency and scalability, which will be needed later on. [7] [11]

2.3.2 SNARKS

One of the recent advances for non-interactive proofs is what is known as SNARK (non-interactive argument of knowledge). This means a proof that is:

- **Succinct**: the size of the proof is very small compared to the size of the witness.

- **Non-interactive**: No rounds of interactions between the prover and the verifier.

- **Argument**: Secured only for provers with bounded computational resources, that is a dishonest prover with unlimited computational power could prove a wrong statement.

- **Knowledge-sound**: If the statement is true, a verifier learns nothing other than the fact that the statement is true. [20]

Moreover, a SNARK can also be zero-knowledge, where the prover demonstrates knowledge without revealing any additional information about the witness. We call such proof a zk-SNARK.

2.3.3 Arithmetic circuit

Arithmetic circuits are a core component of SNARKS. 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. This structure allows SNARKs to efficiently verify complex mathematical computations while preserving succinctness and scalability.

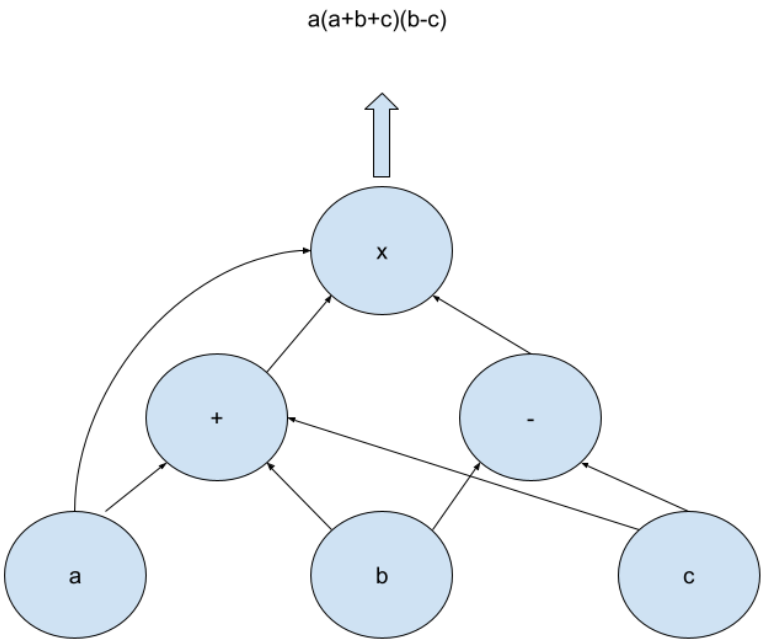


Figure 2.1: General Arithmetic circuit

The prover's process in SNARKs is to create a proof using the setup parameter, a private witness, and public input. The proof shows that the arithmetic circuit is equal to 0. Using the same setup parameter and the public input, the verifier confirms the accuracy of the proof by making sure it aligns with the parameters.

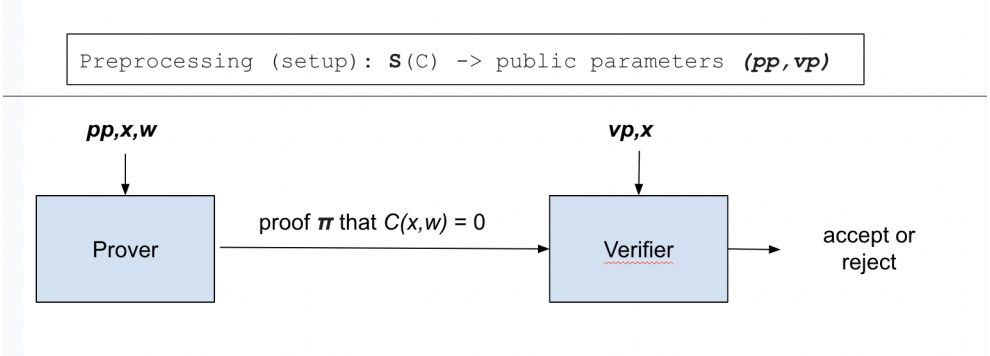


Figure 2.2: Arithmetic circuit for SNARK [1]

- **S(C)**: Public parameters (pp,vp) for prover and verifier
- **P(pp,x,w)**: Proof π
- **V(vp,x,π)**: Accept or reject
- **C(x,w)**: Arithmetic circuit
- **w**: Private witness
- **x**: Public input

2.4 Proof of solvency

A proof of solvency is a system where we prove that an entity, in most cases an exchange, hold enough assets to cover the balance of every customer. In traditional finance, this would be done through an audit. While an audit can be useful, it has many limitations. Obviously it requires the trust of another 3rd party, but it also poses a time constraint. Thus, it is impractical, even impossible to hold an audit

every single day, and in the bitcoin world things move fast. It is essential to be able to fill a proof of solvency every single day. The implementation of a mechanism to produce daily proofs of solvency is therefore of the utmost importance.

A proof of solvency is composed by a proof of assets, where we verify what assets the marketplace as control over, and the proof of liabilities, where we confirm that the total amount of user deposits is smaller than the marketplaces 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. [17] Maxwell's system ensures user privacy by maintaining confidentiality of individual account balances, while only revealing aggregate information in the proof of liabilities. This is achieved through the application of Merkle trees.

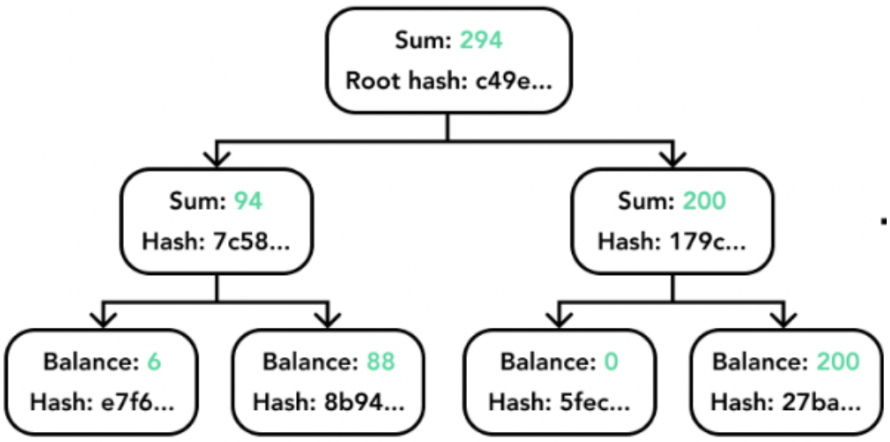


Figure 2.3: Merkle tree for proof of liabilities

In the Merkle tree, every node contains a user's balance along with a hash-based commitment incorporating the customer ID and a nonce. The root of the tree is the

sum of the balances. To verify their inclusion in the total liabilities, users receive a subset of the hash tree from the exchange. This subset includes the user's nonce and the sibling nodes along the unique path from the user's leaf node to the root. By comparing the received information to the exchange's broadcasted root node, users can confirm the inclusion of their balance. 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 [10].

In Provisions, the focus shifts towards 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 was done to make the proof of solvency evolve. 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 provinding 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 [5]. Crypto.com published a one-time audit [18]. Kraken is also publishing a proof of liabilities every few months, without a proof of assets. [16].

Although these proof of reserves may seem promising initially, they are primarily superficial. The proofs have many shortcomings, they are not sufficient 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 a 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 at all. Binance is the only marketplace describing their 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 [6].

Chapter 3

Recursion proofs

This chapter serves as an intermediary background exploration, diving into more advanced concepts beyond last chapters concepts. The realm of zero knowledge is dynamic and continuously evolving, marked by ongoing advancements and active research endeavors. Among these advancements, recursion proofs is one of the most important and active subjects. Recursion proofs are created to accelerate the generation of multiple zero-knowledge proofs. Not all recursion proofs are created equal, they can vary significantly and serve different use cases. In this section, we will examine these variations, distinguishing between them. We aim to identify the most suitable method for our daily proof of liabilities and proof of inclusion.

3.1 Aggregation

Aggregation is the first and simplest type of recursion for zk proofs. It comprises 2 different phases. Initially, you create a standard individual proof for multiple blocks. In the second phase, a proof of proof is created. You could either merge all proofs into a single one, or do a tree-like structure where each parent node proves its child nodes.

This is a quite simple process. Each block as its own proof, and then you prove that the other proofs are valid, giving you only one prove to verify at the end. Since there are 2 different phases, 2 different circuits need to be constructed. On the surface, you can parallelize the initial proofs, which decrease the proving time, and you only have one proof to verify, which decreases the verifying time. However, there are still some issues with aggregation. The most important issue is that the proof time of the second circuit grows linearly with the number of blocks to verify, making it less scalable. [14]

3.2 Recursion scheme

The next technique is the recursion scheme, or Incrementally Verifiable Computation (IVC). Like the aggregation, the proof is separated into blocks. However, in this case each block serves two purposes. Provings that the block itself is valid, and that the previous proof is valid. The number of constraints will always stay the same because it is only proving 2 things of fixed sizes. This creates a chain structure,

where verifying the latest block is the equivalent of verifying every block in the chain. It is an improvement from the previous technique, because this allows to maintain a constant number of constraints in the circuit. This implies that the latest does not take longer to verify than the second block. However, each block still has to verify the previous block, which takes additional. [14]

3.3 Accumulation

To mitigate the exponential growth of recursion proofs, the new scheme created by nova differs every computationally heavy task to the end. All the different parts are accumulated at a later point. Unlike the aggregation scheme, the last part of accumulation does not grow with every new step. Instead of doing n times heavy work, we are doing it only once. The heavy part of the verification step in most SNARKS is found when opening the polynomial commitment.

3.3.1 Polynomial Commitments

A polynomial commitment is a cryptographic technique that allows to commit (lock) data, and reveal (unlock) it later.

The commitments are binding, meaning there is no way to alter data once it is committed. They are also significantly shorter than the data committed. Collisions are inevitable given the vastness of potential data sets mapped onto compact commitments (pigeonhole principle). However, it is computationally infeasible to find a

second dataset matching the same commitment.

The commitments can also be hidden. This means that no information is shared to the receiving party. An example of a commitment scheme would be to hash a dataset with a collision-resistant hash function such as SHA256, and the hash value is shared. To add the hiding property, a random string could be added at the beginning of the dataset.

Instead of commiting to a dataset, we could be committing to a polynomial. While a commitment to the coefficient would work, we would be revealing the polynomial when opening the commitment. We would like to be able to open the commitment only to a certain point of the polynomial. We are also interested in keeping the polynomial secret. An interesting property of polynomials is that it is evaluable at specific points. A polynomial commitment is proving to another party that we have a commitment to a polynomial that evaluates to a certain value at a certain point.

Polynomials support linear operations like addition or multiplication. A commitment can be additively homomorphic if the sum of the commitment values is equal to the sum of the polynomials.

Polynomial commitments are essentials to SNARKs. Verifying opening claims of polynomial commitments constitutes the computationally intensive task we mentioned earlier. [23]

3.3.2 Halo accumulation

The concept of deferring the polynomial commitment opening checks, and consolidating them into a single operation, was introduced by Bowe, Grigg, and Hopwood Halo.[9] The task of checking an opening claim for a polynomial commitment is defined in two steps. The first part is fast, where you just output a pair of a polynomial and its commitment. The second part is expensive, where we verify the pair (f_1, c_1) of the polynomial f_1 and its commitment c_1. The second part can be accumulated if the commitment scheme is additively homomorphic. Instead of verifying each pair (c_1 is a commitment for f_1, c_2 is a commitment for f_2, etc.), we can verify the linear combination of every pairs ($c_1 + c_2 + ...$ is a commitment for $f_1 + f_2 + ...$). [23]

3.4 Folding scheme

Nova takes accumulation a step further. Instead of accumulating the hard part at the end, the folding scheme accumulates everything up until the end. The setup is similar to the recursion scheme, but instead of computing the proof of the previous block, the R1CS are folded together at every block. Instead of having n set of R1CS, we are left with a single set. The new R1CS are called relaxed R1CS, and are used to compute a single proof at the end of the folding. [14] [15]

3.4.1 R1CS

Going back to the previous section, we saw the definition of an arithmetic circuit. R1CS is simply a representation of an arithmetic circuit. A R1CS constraint has the form $a*b = y$, where a,b and y are combinations of variables. Lets define our previous circuit as a R1CS:

$$w_1 = x_1 + x_2 + 1$$

$$w_2 = x_2 - 1$$

$$x = x_1 * w_1 * w_2$$

Now the 3rd constraint does not follow the rules, because there are 3 variables being multiplied. We need to change it using an intermediary value:

$$w_1 = x_1 + x_2 + 1$$

$$w_2 = x_2 - 1$$

$$w_3 = w_1 * w_2$$

$$x = x_1 * w_3$$

[**?**] If we define $z = (x, w)$, we can express the arithmetic circuit as $Az \circ Bz = Dz$ where $\circ$ can be define as: $(x_1, x_2) \circ y_1, y_2 = (x_1 y_1, x_2 y_2)$ [**?**]

3.4.2 Relaxed R1CS

The goal is to combine 2 R1CS and obtain another R1CS. If we are able to do that, we can fold every R1CS together and be left with only one.

If we **define our R1CS**:

fix an R1CS program $A, B, D \in \mathbb{F}_p^{u \times v}$

instance 1: public $x_1 \in \mathbb{F}_p^n$, witness $z_1 = (x_1, u_1) \in \mathbb{F}_p^v$

instance 2: public $x_2 \in \mathbb{F}_p^n$, witness $z_2 = (x_2, u_2) \in \mathbb{F}_p^v$

We know $Az_i \circ Bz_i = Dz_i$ for $i = 1, 2$

First attempt:

Lets define r as a random variable:

$r \leftarrow \mathbb{F}_p, x \leftarrow x_1 + rx_2$

$z \leftarrow z_1 + rz_2 = (x_1 + rx_2, w_1 + rw_2)$

Then:

$Az \circ Bz = A(z_1 + rz_2) \circ B(z_1 + rz_2)$

$= (Az_1) \circ (Bz1) + r^2(Az_2) \circ (Bz2) + (r(Az_2) \circ (Bz_1) + r(Az_1) \circ (Bz_2))$

$= Dz_1 + r^2 Dz_2 + E$

Where E is a combination of the remaining values.

This is not quite an R1CS, because it is not of the form $Az_i \circ Bz_i = Dz_i$. We need to modify the R1CS so that it can be folded.

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

$A, B, D \in \mathbb{F}_p^{u \times v}, (x_1 \mathbb{F}_p^n, c \in \mathbb{F}_p, E \in \mathbb{F}_p^u)$

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:

$T \leftarrow (Az_2) \circ (Bz_1) + (Az_1) \circ (Bz_2) - c_1(Dz_2) - c_2(Dz_1)$

$x \leftarrow x_1 + rx_2, c \leftarrow c_1 + rc_2, E \leftarrow E_1 + rT + r^2 E_2$

$z \leftarrow z_1 + rz_2 = (x_1 + rx_2, w_1 + rw_2)$

$Az \circ Bz =$

$= A(z_1) \circ rB(z_1) + r^2(Az_2) \circ (Bz_2) + r(Az_2) \circ (Bz_1) + r(Az_1) \circ (Bz_2)$

$= c_1(Dz_1) + E_1 + r^2 c_2(Dz_2) + r^2 E_2 + r((Az_2) \circ (Bz_1) + (Az_1) \circ (Bz_2))$

$= (c_+ rc_2)(Dz_1 + rDz_2) + E_1 + r^2 E_2 + rT$

$= c(Dz) + E$

We have a valid relaxed R1CS.[**?**] [2]

Chapter 4

Proof construction

We will be using Circom to generate an arithmetic circuit, and SnarkJS for the proof generation.

Circom is a domain-specific language designed for creating arithmetic circuits, specifically utilized in zk-SNARKs. 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 you can have additionnal intermediate signals. 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 correct behavior of the circuit.

SnarkJS is a JavaScript library that provides tools for working with zk-SNARKs, including circuit compilation. SnarkJS facilitates generating zk-SNARK proofs for specific instances of the circuit using the r1cs and witness file. SnarkJS also provides utilities for verifying the proofs.

4.1 Proof of liabilities

The proof of liabilities operates on a list of balances and a list of email hashes as private inputs. The first purpose of the circuit is to validate that all values are non-negative and that all balances fall within a specified range. These verifications are crucial to prevent overflow or underflow issues, given that the operations occur within a finite field.

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 here.

Inputs

- List of balance (private)
- List of email hash (private)

Outputs

- Balance Sum
- Root hash
- No negative values - boolean
- All small range - boolean

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

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

4.2 Proof of inclusion

The proof of inclusion aims to prove that the balance of a user is included in the Merkle Tree created in the proof of liabilities. To prove that a balance is included, it is sufficient to show that you know the Merkle path of a user balance, which we define using the list of neighbors sum, hash and binary. The neighbors binary variable indicates whether the neighbor is on the left or the right. The root hash, root sum,

user balance and user email hash are public because it needs to be shown which user is in which tree.

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

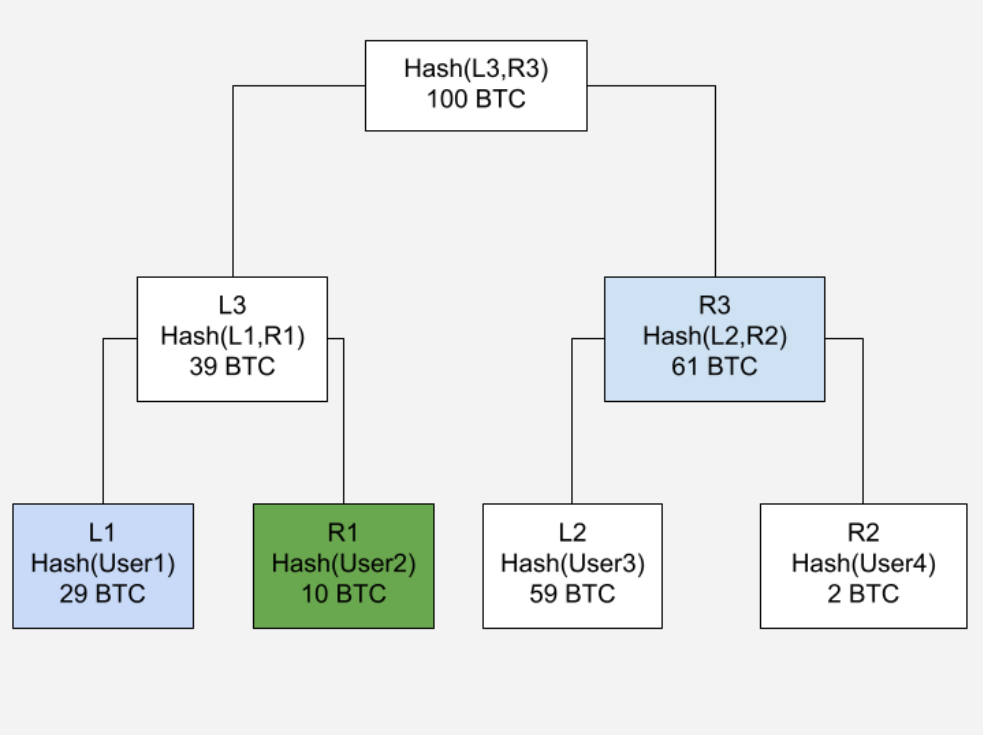


Figure 4.1: Merkle Path

Inputs

- List of neighbors sum (private)
- List of neighbors hash (private)
- List of neighbors binary (private)

- Root hash (public)
- Root sum (public)
- User balance (public)
- User email hash (public)

Outputs

- Balance included - boolean

In the circuit, we verify that the combination of the user balance, sum and merkle path gives the right root hash and root sum. There is no additional verifications since they are already done for this root hash, in the proof of liabilities.

4.3 Daily proof of liabilities

We aim to enhance our current circuit by minimizing the work needed in subsequent rounds. The first thing to explore is recursion proofs, as they were specifically created to reduce the total computational effort across rounds.

4.3.1 Aggregation proofs

The main advantage of the aggregation proof is in streamlining the verification process. For instance, in the second round, we can prove the integrity of the current round and all previous rounds. However, this benefit comes with trade-offs. The first

drawback is the increase in proof size, as it necessitates to prove the current circuit and verify the previous ones. When considering the frequency of verification, having a fixed number of nodes verify the proof daily, it would be illogical to make the nodes verify the previous rounds every single day. On the other hand, if the verification is not consistent, or there is a need for new nodes to be able to come in and verify every proof quickly, 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.

4.3.2 Other recursion proofs

The aggregation proof stands out as the only recursion proof potentially useful in certain scenarios. If we examine the other types of recursion proofs, namely Recursion scheme, accumulation and Folding scheme, they 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.

4.3.3 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, modify and adapt it to include the changes. The key challenge is that the Merkle Tree was built inside the circuit, and is therefore not accessible.

What we will be doing, in our modified circuit, is adjust the Merkle Tree inside the circuit. For every change, we send the corresponding Merkle Path which will be verified by the circuit. The circuit will then compute a new Root Hash for each change and output the final Markle Hash.

The standard verification will be applied to the new values (e.g., non-negative values, limited range).

Inputs

- List of old email hash (private) - 1 per change
- List of old values (private) - 1 per change
- List of new email hash (private) - 1 per change
- List of new values (private) - 1 per change
- List of temporary root hash (private) - 1 per change
- List of temporary root sum (private) - 1 per change
- Old root hash (public)
- Old root sum (public)
- List of neighbors sum (private) - 1 list per change

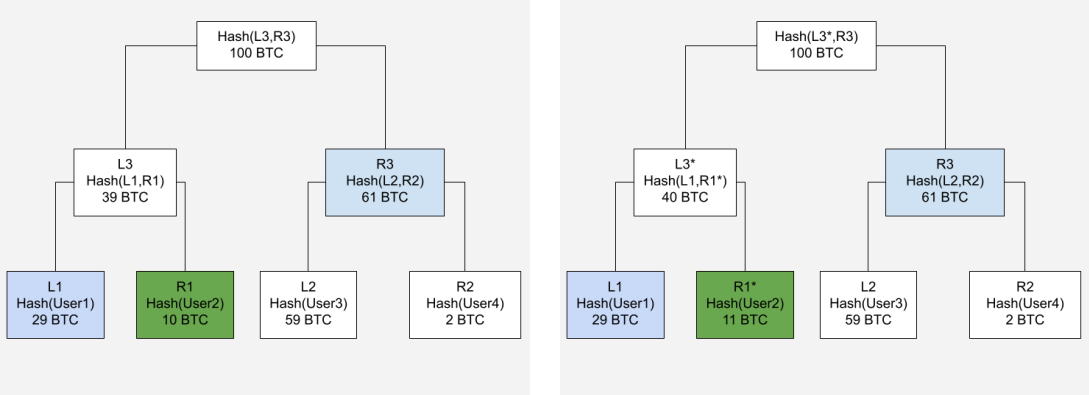
- List of neighbors hash (private) - 1 list per change
- List of neighbors binary (private) - 1 list per change

Outputs

- Valid hash
- Valid sum
- No negative values
- All small range
- New root hash
- New root sum

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 1 are included in the merkle tree. Then we prove that the 11 BTC of user 1 are included in the new merkle tree. After the last change we are left with the new root balance and the new root hash. Not 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.



(a) Pre change (b) Post change

Figure 4.2: Merkle Tree change

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 old user is considered as a change. The naive change calculation would be $changes = balanceChanges + newUsers + deprecateUser$. However, a new user can take the node of a deprecater user. So we would have the equation $changes = balanceChanges + max(newUsers + deprecateUser)$.

Constraint analysis

Theorically, the new circuit makes sense. It should be quite faster than the original one, so let's analyze it. We want to compare the number of constaints for the original circuit with the number of constraints with the change circuit. The first assumption we are gonna need to do, is to assume the number of changes in a day. A completely arbitrary value would be 1%.

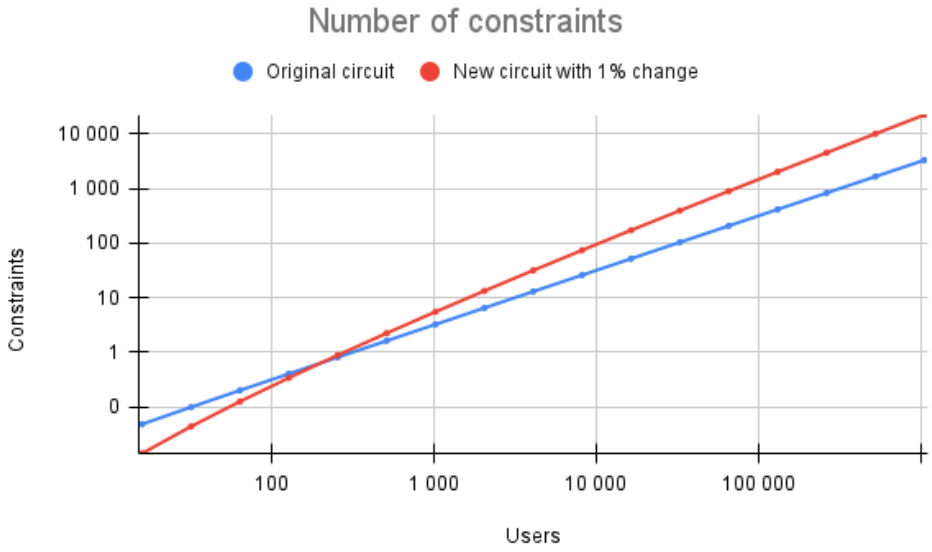


Figure 4.3: Number of constraints original circuit vs new circuit with 1% change

Once you reach more than 128 users with 1% change, it is not worth it to use the new circuit. This is not really 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 for the number of change. This means that two circuits of 0.5% change would have a similar number of constraints as the one circuit with 1% change. We can take this a step further, and instead of looking at daily proof look at hourly proof. Going with the assumption that 1% change daily, we will assume that 0.05% of the balances changes hourly. Lets look at the number of constraints if 0.05% of users change every hour.

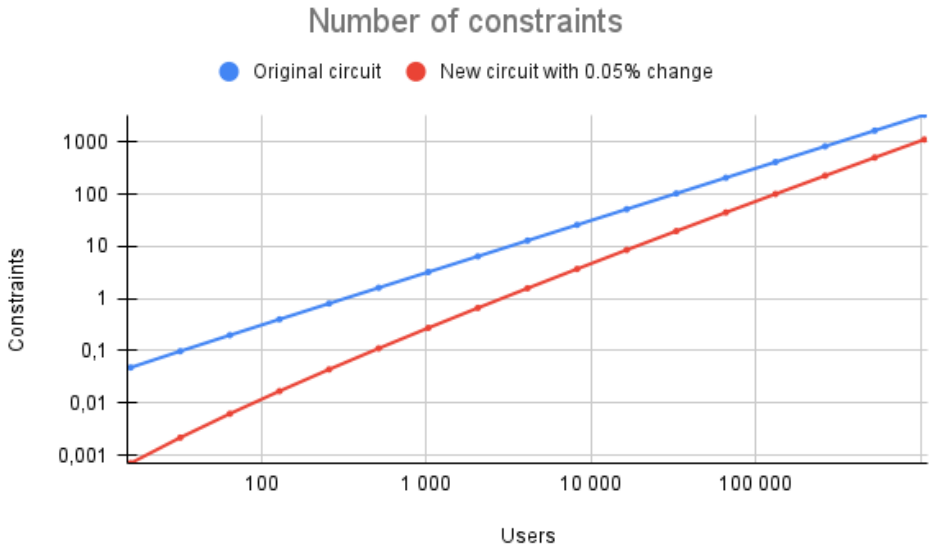


Figure 4.4: Number of constraints original circuit vs new circuit with 0.05% change

Even with more than 1 million users, there is still 3 times less constraints with our new circuit. So if we want to do an hourly proof, it is less expensive with the new circuit than the old circuit. We can take this a step further, and do a proof at every new block. We could have a merkle tree up to date for the liabilities side at every single block.

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

4.4 Daily proof of inclusion

Because a marketplace can have millions of users, it is impractical to build a proof of inclusion for every single user on a daily basis. It is the user's responsibility to request a proof that their balance is included in the published Merkle tree.

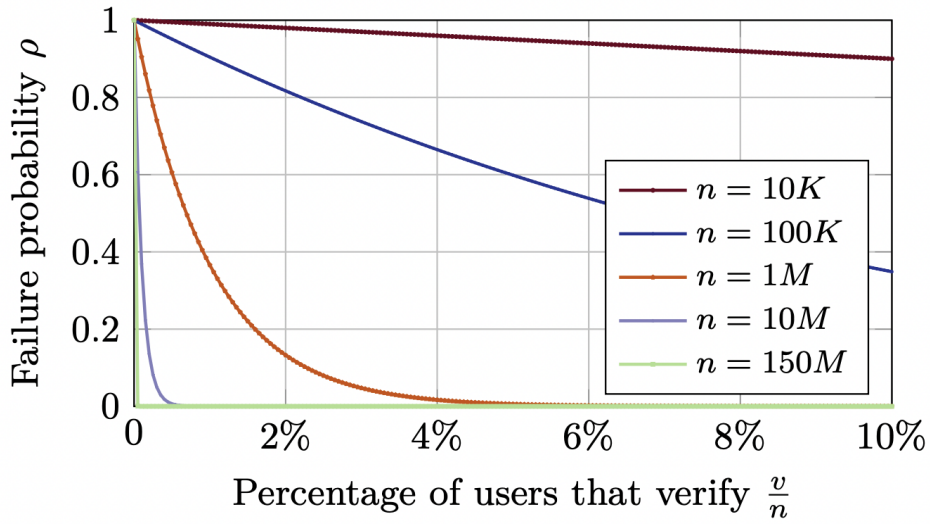
In an ideal world, a Merkle tree would be published at least daily. However, it is once again impractical to require every user to verify their inclusion everyday. Nevertheless, each user verifying its inclusion in the tree increases the chance that the proof of liabilities is valid and includes every single 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.

The novel way to do proof of inclusion is to generate a proof of inclusion that is valid from the day of creation of the account, to the requested day by the user. Normally you would need to create 500 different proofs for 500 days. However, we saw in the previous section that the nova folding scheme enables the 500 proofs into just one. 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, and here is why.

In this chart, we can observe the failure probability, which represents the likelihood that a dishonest prover gets away with misbehaving. If we take for example the orange line, where an exchange has 1 million users, we notice that the failure probability approaches 0 when over 4% of users perform their verifications. However, without utilizing folding, this principle doesn't extend to every round. It implies that for each round, we require a minimum of 4% of users to have a proof with significant confidence.



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

Figure 4.5: Failure Probability [13]

Now, lets evaluate the failure probability if we are using folding. In the initial round, with no changes, the failure probability for 20k users remains at 13%.

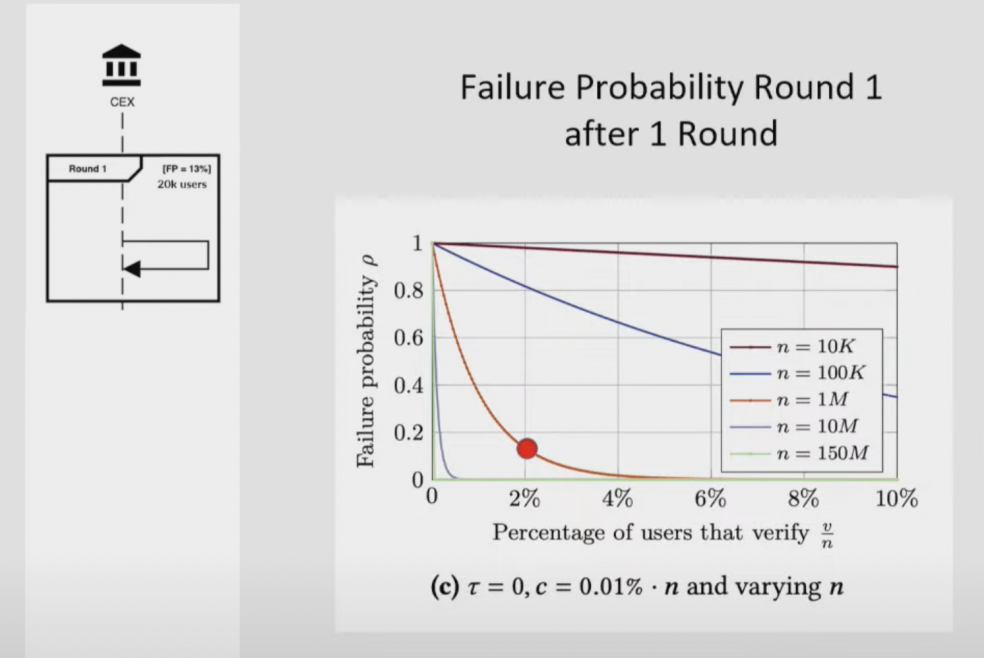


Figure 4.6: Failure Probability Round 1 [8]

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

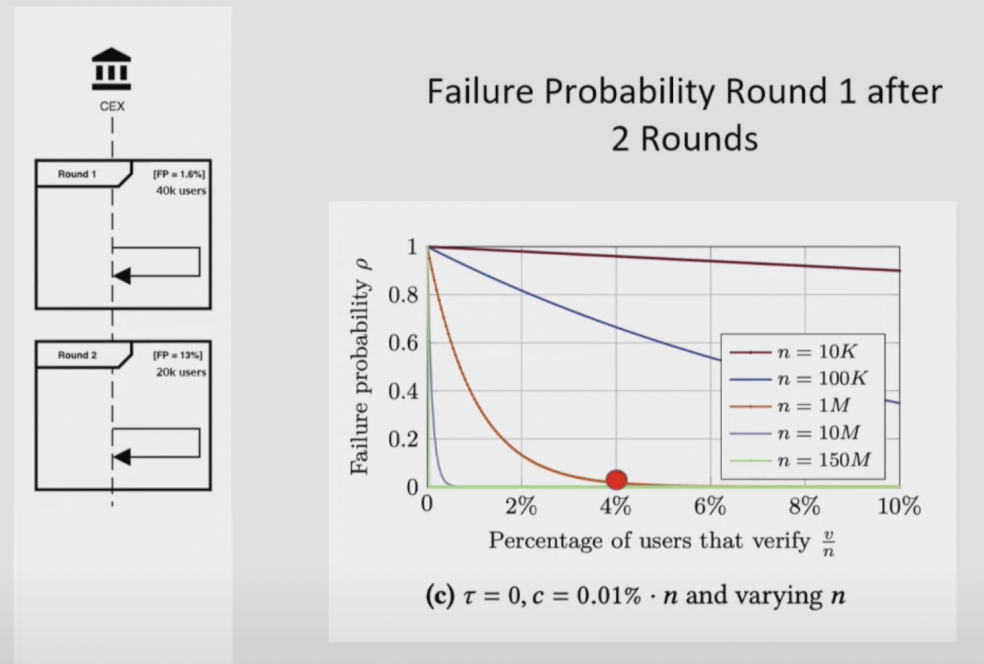


Figure 4.7: Failure Probability Round 1 After 2 Rounds[8]

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

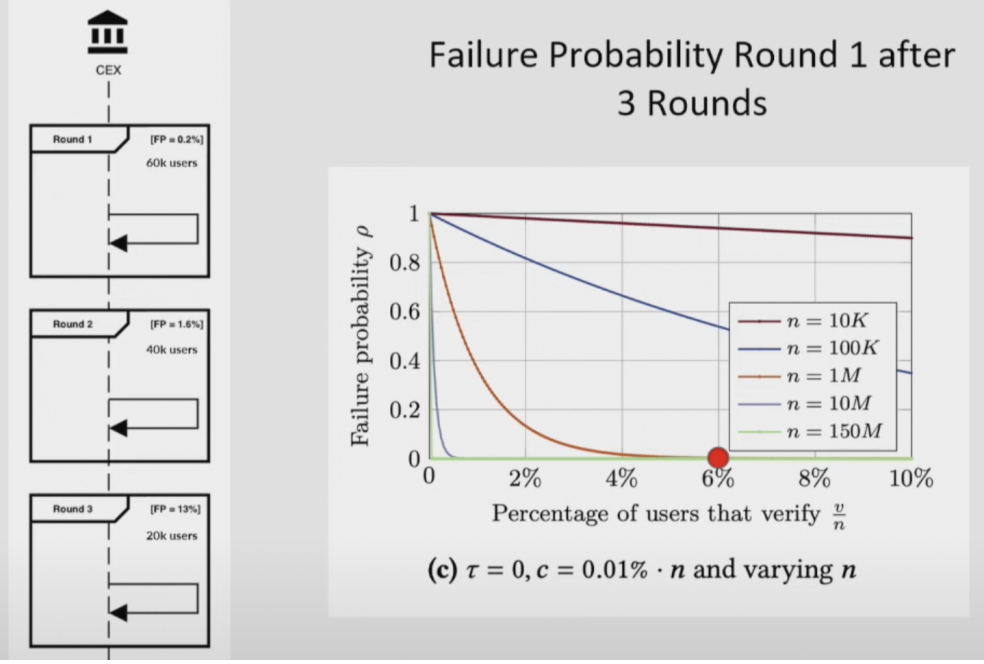


Figure 4.8: Failure Probability Round 1 after 3 Rounds [8]

The key takeaway here is that without folding, you require 4% of users to verify every round. However, with folding you only need 4% of users to participate in at least 1 round. This significantly reduces the burden on the user to verify, while increasing the confidence in the proof.

4.4.1 Circuit Design

The private inputs vary for each instance, whereas the public inputs are carried over from round to round. In order to implement folding, we need to slightly adjust our circuit. 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.

Inputs

- List of neighbors sum (private)
- List of neighbors hash (private)
- List of neighbors binary (private)
- Root hash (private)
- Root sum (private)
- User balance (private)
- User email hash (private)
- Steps in (public)

Outputs

- Steps out (public)

Steps

- Balance included
- Root sum
- Root hash
- User balance

- User email hash

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

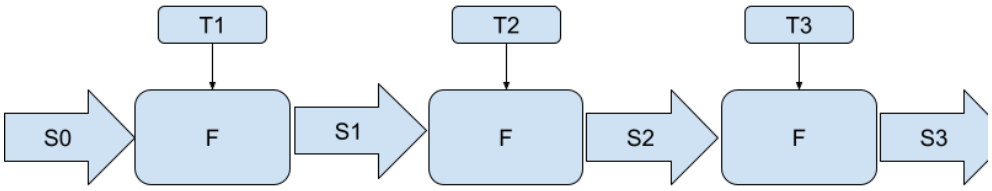


Figure 4.9: Folding Circuit

S0 is the step in of the first change, S1 is the step out of first change and step in of second change. F is the folded circuit and T is the number of time 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 as a means 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. At the end you only have to verify a single proof. You also need to compare the circuit output with the proof of liabilities outputs for every round.

4.5 Folded daily proof of liabilities

With the original circuit, we saw how it was not usefull to use folding or any other recursion scheme. However, with the new circuit we saw that we were able to separate a big proof into multiple smaller ones. This aligns prefectly wth the recursions ideas. We can separate our new circuit of n changes into m circuits, where $m <= n$, and fold the circuits together.

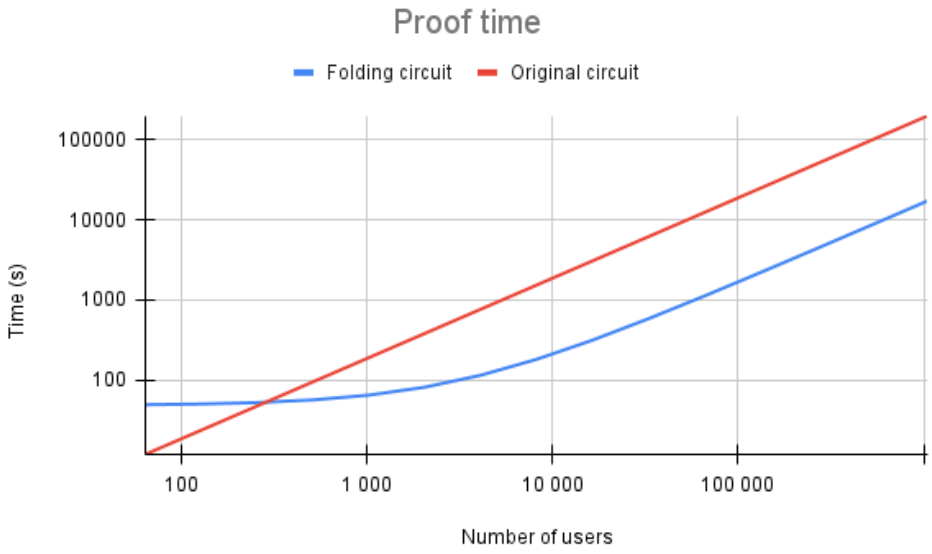


Figure 4.10: Proof time original circuit vs folded circuit with 1% change, 1 change by fold

As we can see, after 10 000 users, both circuits seems to be groing linearly, with the folded circuit doing much better. For instance, at 1M users, we have the old circuit taking 196608 seconds, and the new circuits taking 17188 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 out there. The data we should focus on is that it took 10x less time for the folded circuit, and it was not event optimized. We were doing 1 change by circuit, so at 1M users we folded 10k circuits.

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

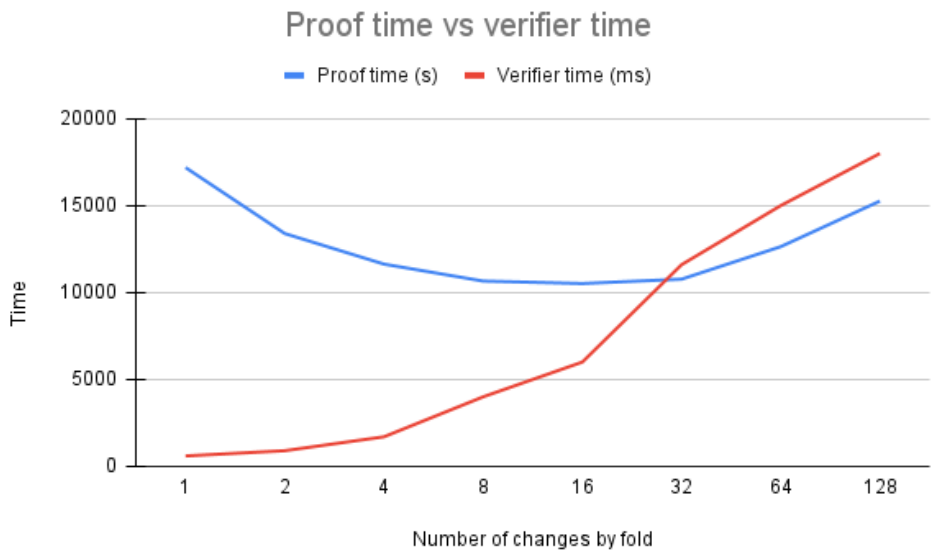


Figure 4.11: Optimization of the proof time and verifier time of the folded circuit with 1% change and 1M users

As we can see, the optimize time for the proofer, is between 8 and 32 changes by fold, while the verifier time seems to be groing at a monotonic pace. Depending on

the use case, you can decide the numbe of changes you should have for every fold.

4.5.1 Circuit design

The folded change circuit is exactly the same as the regular change circuit, but the inputs works differently because some data is passed around the circuits. There is 2 types of step variables; the variables needed in the following circuit, and the variables that are 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.

Inputs

- List of old email hash (private) - 1 per change
- List of old values (private) - 1 per change
- List of new email hash (private) - 1 per change
- List of new values (private) - 1 per change
- List of temporary root hash (private) - 1 per change
- List of temporary root sum (private) - 1 per change
- List of neighbors sum (private) - 1 list per change
- List of neighbors hash (private) - 1 list per change
- List of neighbors binary (private) - 1 list per change

- Step in (public)

Outputs

- Steps out

Steps

- Valid hash and valid sum
- No negative value and all small range
- Root hash
- Root sum

Chapter 5

Conclusion

This thesis introduces a novel approach to generating a more compact proof of liabilities. Leveraging a fresh circuit design, along with leveraging prior proofs and the folding scheme, we achieved notable reductions in proof size. This smaller proof position itself well to be generated daily, and even at higher frequencies. Our analysis demonstrates the superior performance of the folding scheme combined with the updated circuit, compared to the existing model. The significance of a more concise proof of liabilities cannot be overstated, serving as a cornerstone for broader adoption of robust proof of solvency.

To complete the proof of solvency for marketplace integration, a proof of assets is imperative. One challenge in this endeavor lies in conducting proofs for multiple currencies. However, by employing the folding scheme, we can segment the proof of assets into currency-specific proofs before conducting the folded proof, mitigating this complexity.

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
    not_negative = False
```

```
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
temp_not_negative = 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 and negative check
    rangecheck = RangeCheck(maxBits, balances[i])
    tempNotBig += rangecheck
    negativecheck = NegativeCheck(newValues[i])
```

```
        tempNotNegative += negativecheck

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

    # Check if all balances are non-negative
    if temp_not_negative == inputs:
        not_negative = 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, not_negative, 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
    tempNotNegative = 0
    maxBits = 100

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

```
    # Perform range check and negative check
    rangecheck = RangeCheck(maxBits, newValues[i])
    tempNotBig += rangecheck
    negativecheck = NegativeCheck(newValues[i])
    tempNotNegative += negativecheck

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

# Check if all new values are not negative
notNegative = IsEqual([changes, tempNotNegative])

# 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 old 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 (notNegative, allSmallRange, validHash,
```

validSum , newRootHash , newSum)

Bibliography

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

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

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

[4] B. Barak. Lecture 14: Zero knowledge proofs. *Boaz Barak's Blog*.

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

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

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

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

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

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

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

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

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

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

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

[16] Kraken. Proof of reserves.

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

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

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

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

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

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

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

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

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