ZeroSync: Introducing Validity Proofs to Bitcoin

Robin Linus and Lukas George

ZeroSync Association

Abstract

We introduce *ZeroSync*, the first-ever proof system addressing Bitcoin's scalability challenges with *Succinct Non-Interactive Argument of Knowledge* (SNARKs). ZeroSync compresses the entire Bitcoin blockchain into a compact proof of validity, enabling instant verification and unlocking various innovative applications. We discuss our prototype implementation of a chain state proof, utilizing the Cairo language, Utreexo, and recursive STARKs. Our work enables diverse applications, including quick bootstrapping of full nodes, trustless light clients, enhanced Lightning Network privacy, and secure cross-chain bridges. Chain state proofs require no consensus changes, which is crucial as forks in Bitcoin are challenging to implement and achieve consensus for. Despite the existing bottleneck of prover performance, we present a range of optimization strategies and demonstrate the practicality of generating a complete chain state proof.

Finally, we introduce *zkCoins*, a client-side validation protocol combined with zero-knowledge SNARKs, drastically improving privacy and throughput of token transactions. In combination with future Bitcoin features, such as Simplicity, zkCoins also enables private and more scalable BTC transactions.

The groundbreaking compression capabilities of SNARKs initiated a paradigm shift in cryptocurrency design, and ZeroSync is pioneering their application to Bitcoin.

1 The First Chain State Proof of Bitcoin

The security of Bitcoin [1] relies on every node verifying every transaction. Only a fully verifying node can participate in a trustless and censorship-resistant manner. Moreover, only a fully verifying node really contributes to the network's decentralization because the consensus rules are defined by what users agree to with their node. Still, there are only about 50,000 full nodes [2] while millions of Bitcoin users exist. This asymmetry is rooted in the high barrier to entry of having to download and verify about 500 GB of historical blockchain data [3] to sync a node.

Succinct Non-Interactive Argument of Knowledge (SNARKs) promise a paradigm shift in blockchain scalability and privacy. While considerable engineering effort is underway in the Ethereum community to apply this technology, no similar investment has yet been made for the Bitcoin network. Projects like the Mina protocol [4] demonstrate that recursive proofs can compress an entire blockchain into a constant-size proof. At ZeroSync, we have built the first proof of Bitcoin's chain state. We are validating the entire blockchain in a proof system. This is computationally expensive; however, the resulting proof can be verified cheaply on any device. The grand vision is millions of phones running a fully verifying node.

Bitcoin is highly decentralized, and consensus changes are intentionally hard. They require convincing a majority, which is generally a losing battle. Applying validity proofs outside of Bitcoin's consensus rules has a lot more potential since it does not require everyone to agree. It is up to every individual user if they want to sync using a chain state proof or a conventional inital block download. This approach enables great flexibility on top of Bitcoin's rigid base layer consensus.

We use a *transparent* proof system that requires no trusted setup and introduces no novel cryptographic assumptions. There is also a low risk of prover centralization because any other party can extend an existing chain proof by proving the next block. Ideas for decentralized proof generation will be explored in a later chapter.

The prototype we have implemented already verifies most of Bitcoin's consensus rules. It verifies everything except for the transaction witness data. Our benchmarks demonstrate the technical feasibility of a chain state proof. The main obstacle is the prover performance, particularly proving SHA256 hashes and ECDSA signatures. In the last chapter, we discuss various approaches to improve the prover performance significantly.

2 The ZeroSync Proof System

We are using STARK proofs[5] mostly because they are transparent and scalable to the size of bitcoin blocks. They also have the best known practical prover performance of all SNARKs. StarkWare, the company that invented STARKs, offers a domain-specific language named *Cairo*[1], which is *"a language for creating STARK-provable programs for general computation"*. We have implemented most of the Bitcoin consensus rules in the Cairo language. Our Bitcoin Cairo library is free and open-source software (FOSS).

Representing Bitcoin's entire UTXO set in a chain state proof is intractable. Therefore we are using a UTXO set commitment instead. We have implemented Utreexo [6], which is a dynamic accumulator for bitcoin state, that allows us to verify inclusion and update the commitment without having to know the complete UTXO set. We used a STARK-

[1]`https://www.cairo-lang.org/`

friendly hash function in our Utreexo implementation.
We chose the Giza[2] prover, which is based on the Winterfell[3] STARK library, as it was the most mature FOSS STARK prover available back then. Furthermore, we have contributed an implementation for recursive STARKs. In collaboration with Max Gillet, the author of Giza, we have ported the Giza verifier to Cairo to verify a proof in a proof. We modified the proof system to use a STARK-friendly hash function to accelerate proof recursion.
We have published a web verifier for chain proofs[4], which is the Giza verifier compiled to web assembly. Instantly verifying Bitcoin's chain state on a website demonstrates the potential of this technology.
Furthermore, we have published an experimental version of the ZeroSync toolkit, which allows others to import our Bitcoin Cairo library into their own projects to generate custom Bitcoin STARKs[5].

3 Types of Chain State Proofs

We aim to apply STARKs to relieve other applications from verifying the Bitcoin chain block by block by replacing the process with a verifiable proof of the latest chain state. It intends to keep security as close as possible to full node security; however, a proof cannot verify data availability. A proof also cannot verify the longest chain rule as we cannot provably connect it to a peer-to-peer network. A proof is aware of only a single chain, and all the p2p logic must be implemented on the node level. The node has to resolve conflicting claims of peers, but using proofs it can instantly verify the total work of two different chain tips.
The public inputs for every chain proof contain the current program's hash for recursive verification, as well as the best (highest) block hash, total work, and current block height of the verified chain. They include the current target, previous 11 timestamps, and current epoch start time to verify Bitcoin's retarget mechanism correctly. We aim for three different types of state proofs:

1. The header chain proof that attests to the correct validation of all light client consensus rules and refers to the chain of Bitcoin block headers,
2. the *"assumevalid"*[6] state proof which includes the verification of all consensus rules but transaction script validation and
3. the full state proof with transaction script validation.

[2]`https://github.com/ZeroSync/giza`
[3]`https://github.com/facebook/winterfell`
[4]A deployed version of the verifier is available at `https://zerosync.org/headers-chain.html`
[5]`https://pypi.org/project/ZeroSync/`
[6]`https://bitcoincore.org/en/2017/03/08/release-0.14.0/#assumed-valid-blocks`

Each type verifies a larger subset of Bitcoin's consensus rules, aiming to encapsulate all of them in a single program eventually and, therefore, some other respective public inputs. However, these are always a constant number of field elements which keeps the actual proofs a constant size; even after recursive verification. The size of the resulting chain state proof is about 800kB uncompressed.
The block data is fed into the program using *hints*[7], which are pre-initialized memory cells so it does not blow up the size of the public inputs.
The same overall procedure is used to create a state proof: First, the previous state proof is verified, and then the respective consensus rules are applied to the next set of blocks resulting in an updated chain state.

3.1 Header Chain Proof

In the case of verifying only block headers and their consensus rules, we can include multiple headers in each iteration of the recursive proving process to decrease the amount of expensive STARK proof verification. In opposite to existing approaches, e.g., zkRelay[7], we support any number of blocks in a batch per proving run. Therefore, initial proving can leverage powerful machines and large batches up until the proof is at the chain tip. From then onward, smaller batches can be proven to decrease latency. In addition to the standard public inputs already mentioned, the header proof includes a cryptographic commitment to the set of all blocks proven in the current run and the recursively verified proof. Currently, in the form of a Merkle tree, which allows verifying the inclusion of a particular block header in a proof without requiring additional information about the proving process (e.g., batch sizes used per recursive step) with a usual Merkle proof.

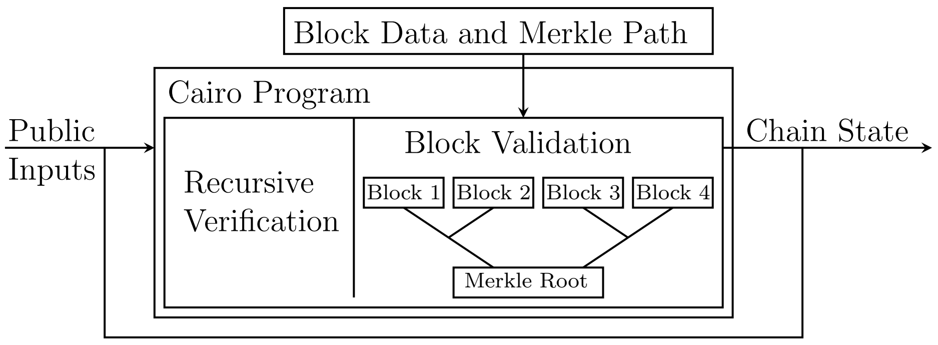


Figure 1: Header chain state proof example for a batch with four blocks.

The Merkle root is created from all blocks in the batch and the previous proof's Merkle root. A *zero* node is added to the tree in case it would not be complete such that it

[7]`https://www.cairo-lang.org/docs/how_cairo_works/hints.html`

can be appended using only the Merkle path for the respective *zero* node and without recomputing the entire tree in the subsequent proving run.

3.2 Full Chain State Proof

Proving entire blocks and transactions is substantially more computationally expensive than proving block headers. For easier development, we left the batching of blocks and the Merkle tree out of our prototype. Another issue is the sheer size of the unspent transaction outputs (UTXO set) that we must update with every block. The solution is our Utreexo implementation that serves data and inclusion proof for each UTXO spent in the to-be-currently validated block. Since Utreexo is a forest data structure, we include a fixed number of roots in the public inputs. Updating these roots is part of the Cairo program. In other words, their correctness is proven by the STARK.
Our *assumevalid* state proof is almost a complete chain state proof as it verifies all consensus rules except for the transaction scripts, which it assumes to be valid. So far, we have implemented the header chain proof and the assumevalid state proof as prototypes. The former is feasible to prove, while the latter still requires performance improvements to prove reasonable-sized blocks. It does not include the *SegWit* logic yet, a consensus update that activated in August 2017, which means that our current implementation can prove the chain only up to this point. The *Taproot* update has not been implemented either.

4 Applications of Chain State Proofs

4.1 Accelerated Initial Sync

The canonical application of a chain state proof is to accelerate the initial block download of Bitcoin Core. Naively, users can sync in three simple steps: Verify the current chain state using a proof, then download the corresponding UTXO set (≈ 5 GB of data), copy it into the *"chainstate"* folder, and run Bitcoin Core as usual. This procedure allows users to bootstrap a (pruned) full node without having to download and verify 500 GB of historical blockchain data. It reduces the initial sync time from many hours (or even days) to minutes.
A chain state proof combined with Utreexo blocks allows to sync even faster. While the chain state proof allows us to skip the initial block download, with Utreexo blocks, we do not even have to download the UTXO set. After verifying the chain state proof, Utreexo nodes can immediately listen for new blocks. The downside is that Utreexo requires to "augment" blocks with Merkle inclusion proofs for all UTXOs spent in a block. That

leads to an overhead of about 2 megabytes per block. However, no bridge node is required. Utreexo nodes can compute inclusion proofs for their own UTXOs from incoming blocks. Approaches like these are great for bandwidth-constrained settings, which lead us to partner with Blockstream to optimize the sync times via Blockstream Satellite.

Another intriguing application is trustless light clients. Many users, particularly those on mobile devices, rely on trusted servers to serve them the correct transaction history. Clients relying on "simplified payment verification" (SPV), as described in the Bitcoin white paper, solve this issue only partially because servers could still withhold relevant transactions. Servers do not require trust when they prove their responses with state proofs. However, it requires a different UTXO set commitment, which provides a key-value mapping, like a Merkle-Patricia tree, that can prove all UTXOs of an address. Our current setup with Utreexo has yet to support that. We want to add to our chain proof a more sophisticated accumulator to serve light clients efficiently.

4.2 Further Applications of Validity Proofs to Bitcoin

We have identified numerous further applications of succinct zero-knowledge proofs to Bitcoin. For example, the privacy of the Lightning Network can be improved substantially. Routing nodes have to publicly announce their UTXOs in the p2p gossip protocol, which might even link them to their IP address. A chain state proof can be extended to prove the validity of a payment channel without sacrificing privacy.

Another interesting application is attestations. Custodians, such as exchanges, can prove their solvency to their customers.

A chain state proof turns a simple blockchain into an authenticated data structure, which can efficiently answer complex queries with verifiable responses. A chain-processing proof can generally act as an adaptor to create indices over the blockchain data, and apply arbitrary filters or data transforms.

An open engineering challenge is how to design a chain state proof to be cheaply customizable for many different use cases. For example, a decentralized domain name system, which is based on the general observation that chain proofs can compute Merkle-like accumulators over any kind of blockchain inscriptions. We will present our ideas in a future work.

5 Performance Limitations and Optimizations

In our performance benchmarks[8], proving a full block, containing about 3,000 transactions, took about 3.5 hours and required about 100GB of RAM. However, this was not

[8]An incomplete table of benchmarks is available at `https://github.com/ZeroSync/ZeroSync/blob/main/docs/roadmap.md#milestone-2-measure-and-optimise`

a full block proof as it did not verify witness data. Such a signature verification costs about 250,000 instructions, which, we estimate, will increase the total proving time to five to seven hours in the unoptimized setup.
To keep an existing chain proof in sync, we must prove a block every 10 minutes. Therefore, we want to increase proving performance by a factor of more than 40. In the following, we discuss various approaches to achieve that.

5.1 Prover Performance Optimizations

Numerous optimizations are possible to increase the prover performance. Switching to the currently developed Rust-based cairo-rs[9] runner to generate the program trace (a necessary step before proving the execution which takes about half of the proving process's time) will yield an improvement over the present Python implementation. Another relatively simple optimization is to use a STARK-friendly hash function to accelerate the verification of recursive STARKs significantly, as they consist mainly of Merkle paths. We have already implemented a prototype using Pedersen hashes, which enabled us to verify all Merkle paths of Utreexo and recursive STARKs in a reasonable time. Further optimization is possible by switching to faster hash functions such as Rescue[8], Poseidon[9] or Poseidon2[10].

Family	Name	$\mathbb{F}$	t / invocation	w	d	c / 10^5 invocations
Standard	$\mathsf{SHA2}_{128}$	64-bit prime	1000	20	2	5.6×10^{10}
		$GF(2^{64})$	3762	56	11	7.2×10^{11}
	$\mathsf{SHA3}_{128}$	$GF(2^{64})$	1536	25	2	1.1×10^{11}
AES-DM	AES_{64}	$GF(2^{64})$	48	62	8	7.5×10^{9}
	$\mathsf{Rijndael}_{80}$	$GF(2^{64})$	58	68	8	9.9×10^{9}
Algebraic Sponge	$\mathsf{Pedersen}_{128}$	256-bit prime	128	16	2	8.7×10^{10}
	MiMC_{126}	253-bit prime	320	2	3	6.4×10^{10}
	GMiMC_{122}	61-bit prime	101	1	3	9.4×10^{8}
	$\mathsf{Starkad}_{126}$	$GF(2^{63})$	10	14	3	3.4×10^{8}
	$\mathsf{Poseidon}_{122}$	61-bit prime	8	17	3	3.1×10^{8}
	Rescue_{122}	61-bit prime	10	12	3	3×10^{8}
	Vision_{126}	$GF(2^{63})$	20	12	6	7.5×10^{8}
			40	12	4	1.4×10^{9}

Unmodified excerpt from *STARK Friendly Hash – Survey and Recommendation* [11] licensed under CC-BY-4.0

Figure 2: An overview of hash functions and their cost when computing them in a STARK. Proof-friendly hashing is crucial for Utreexo inclusion proofs and for recursive proofs because STARK proof verification requires to check many Merkle paths.

An essential feature of Cairo is builtins[10]. They are *"predefined optimized low-level execution units which are added to the Cairo CPU board to perform predefined computa-*

[9]`https://github.com/lambdaclass/cairo-rs`
[10]`https://www.cairo-lang.org/docs/how_cairo_works/builtins.html`

tions which are expensive to perform in vanilla Cairo (e.g., range-checks, Pedersen hash, ECDSA,...)". We plan to implement a builtin for sha256, as it currently comprises about 50-80% of the total proving time. Adding the ECDSA signature verification, to complete the chain proof, requires further improvements. In joint work with Andrew Milson, the author of the *Sandstorm* prover[11], we are researching Starkware's schemes [12] [13] on accelerating arithmetic operations over other fields than Cairo's base field. It is possible to compute STARKs directly over the base field of secp256k1, so verification of curve point operations becomes orders of magnitude cheaper. With this approach, signature verification could be out-sourced to a second proof over the base field of secp256k1.
A different optimization is to reduce the field size by switching to the Goldilocks field, which is around 4 to 10 times faster because it is smaller and optimized for 64-bit architectures. It also requires only about a quarter of the memory, because in practice, most variables in our program are much smaller than the virtual machine's register size, given by Cairo's standard ≈ 256 bit field. However, the highly optimized implementations of Cairo's builtins are tightly entangled with its standard field. That means it is required to reimplement builtins for the Goldilocks field (e.g., rangecheck, bitwise, sha256).
It is possible to parallelize block proof generation. Independent provers can prove individual transactions, including the Utreexo state, before and after the transaction was executed. Those transaction proofs get aggregated into a block proof. Provers can batch multiple transactions into a single proof to minimize the overhead of proof recursions. Proving small batches might be feasible on consumer hardware. Furthermore, batching allows decentralizing proving, such that a network of zk-nodes would not have to rely on a single prover with a specialized setup.
However, specialized hardware for proof generation, based on FPGAs or ASICs, might be needed to initially prove the almost 800,000 existing Bitcoin blocks to catch up with the current chain state. Generating this first chain state proof is very expensive, but when the current chain is proven once, extending an existing state proof with the next block every ten minutes will require much fewer resources.

6 zkCoins: Improving Scalability and Privacy

6.1 Blockchain Design Principles

In 2017, Andrew Poelstra formulated a powerful idea: *"Use the chain for what the chain is good for, which is an immutable ordering of commitments to prevent double-spending"*[14]. He originally mentioned this in the context of Scriptless Scripts, which are off-chain smart contracts. We generalized his idea, applied it to cryptocurrency scalability resarch, and two key principles became apparent:

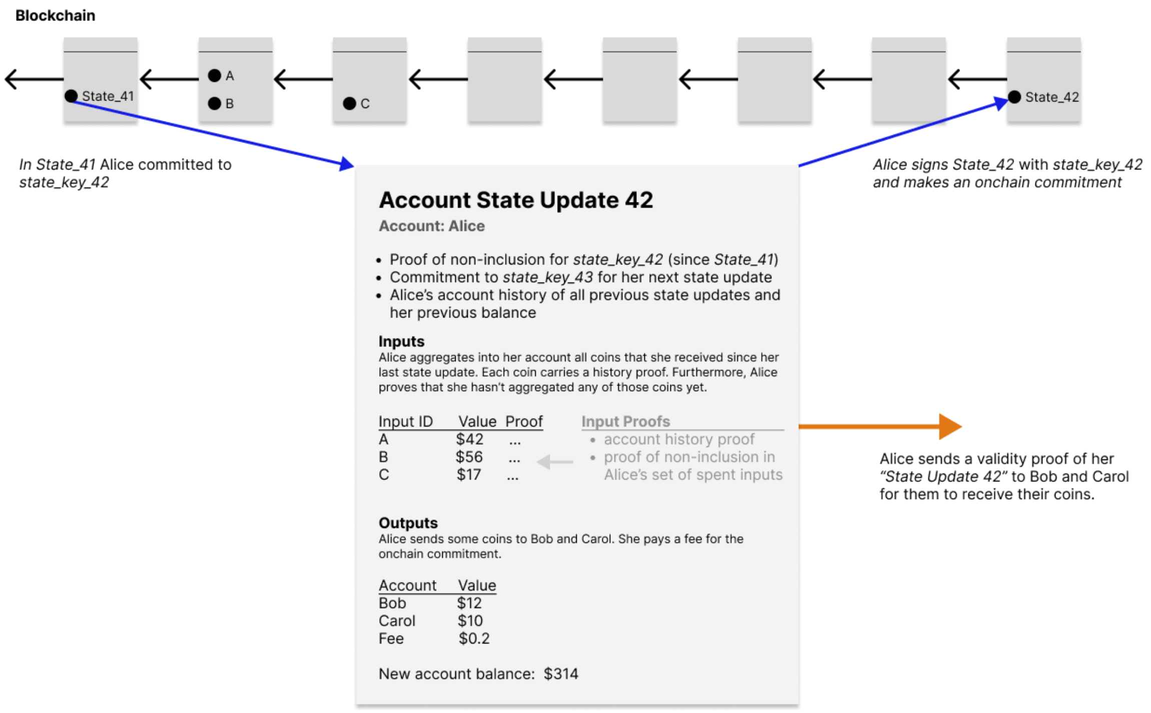
[11]`https://github.com/andrewmilson/sandstorm`

1. Never write into the blockchain what can be communicated off-chain.
2. Avoid validation on the global layer that can be performed on the client side.

Following this paradigm, we created *zkCoins*, a novel payment system combining a client-side validation protocol with zero-knowledge validity proofs to achieve high throughput and strong privacy. In the next sections, we define zkCoins in more detail, particularly an implementation as a second layer on Bitcoin.

6.2 Combining Client-Side Validation with zkSNARKs

Recently, client-side validation (CSV)[15] protocols, such as RGB[16] and Taro[17], have become popular as they enable tokenized assets on Bitcoin, which require no consensus changes, introduce a novel form of smart contracts, and have a low on-chain footprint. In CSV protocols, only the recipient verifies a token's history. The transaction data and the token history is communicated off-chain, directly from sender to the recipient, which minimizes the load on the global layer. Only a compact *commitment* to the transaction data is written into the chain, establishing a global ordering, which prevents double-spending. Historically, the fundamental problem of CSV protocols has been that a token's history grows quasi-exponentially. Quickly, most transactions become related to each other, such that over time, a token's history converges against the size of the chain. However, the recent advancements in the field of recursive validity proofs now allow compressing token histories to negligible sizes. To transfer a coin in such a *"zkCSV"* protocol, the sender extends the coin's history proof to prove the transaction's validity to its recipient. We named our scheme *zkCoins*, because a coin is literally its proof of validity. The sender gives a zkSNARK to the recipient.
In contrast to similar concepts like zk-rollups or Mina, there is no data availability problem, and no global proof aggregation (no "sequencer") is required because the proofs are communicated off-chain.
Additionally, zkCoins provide best-in-class privacy, as they obfuscate transaction amounts and graphs using *zero-knowledge* SNARKs (zkSNARKs). This provides strong unlinkablility and censorship resistance, because eavesdropper cannot correlate commitments or distinguish transactions in any way. Furthermore, zkCoins can greatly improve throughput because all transaction data is communicated off-chain, and arbitrarily many inputs and outputs can be contained in a small, constant-sized commitment in the chain.
Most interestingly, the global state can be reduced to a constant size. The sender proves to the recipient that no double spending happened, by proving that their commitment has never occurred in the chain before. In the following, we introduce a simple accumulator scheme offering compact proofs of non-inclusion.


Blockchain
State_41
A
B
C
State_42
In State_41 Alice committed to *state_key_42*
Alice signs State_42 with state_key_42 and makes an onchain commitment
Alice sends a validity proof of her *"State Update 42"* to Bob and Carol for them to receive their coins.

Account State Update 42

Account: Alice

- Proof of non-inclusion for *state_key_42* (since *State_41*)
- Commitment to *state_key_43* for her next state update
- Alice's account history of all previous state updates and her previous balance

Inputs

Alice aggregates into her account all coins that she received since her last state update. Each coin carries a history proof. Furthermore, Alice proves that she hasn't aggregated any of those coins yet.

Input ID	Value	Proof
A	$42	...
B	$56	...
C	$17	...

Input Proofs
- account history proof
- proof of non-inclusion in Alice's set of spent inputs

Outputs

Alice sends some coins to Bob and Carol. She pays a fee for the onchain commitment.

Account	Value
Bob	$12
Carol	$10
Fee	$0.2

New account balance: $314

Figure 3: Alice sends zkCoins to Bob and Carol, and also aggregates into her account the incoming coins that she received since her last update.

6.3 Timechain Accumulator

In every on-chain commitment, a sender commits to their next commitment key for their next transaction. To ensure each key is used only once, they must prove non-inclusion since the previous on-chain commitment, which committed to that key.

Once a month, all users sort all keys that occurred in the chain during that month. Then they compute a Merkle tree over that sorted set of keys. The resulting Merkle paths are non-inclusion proofs for all other keys because a path can prove that a particular key is not at the position where it would be if it was included in the sorted set.

At the end of every month, each user keeps only a single proof of non-inclusion for their latest state key and discards the rest of the tree. They aggregate that proof of non-inclusion into their account state proof. Then they start over with the next month. The required storage capacity is the maximum number of keys in the chain per month, which is some large constant depending only on the block size, block time, and the size of commitments, and not on the number of users.

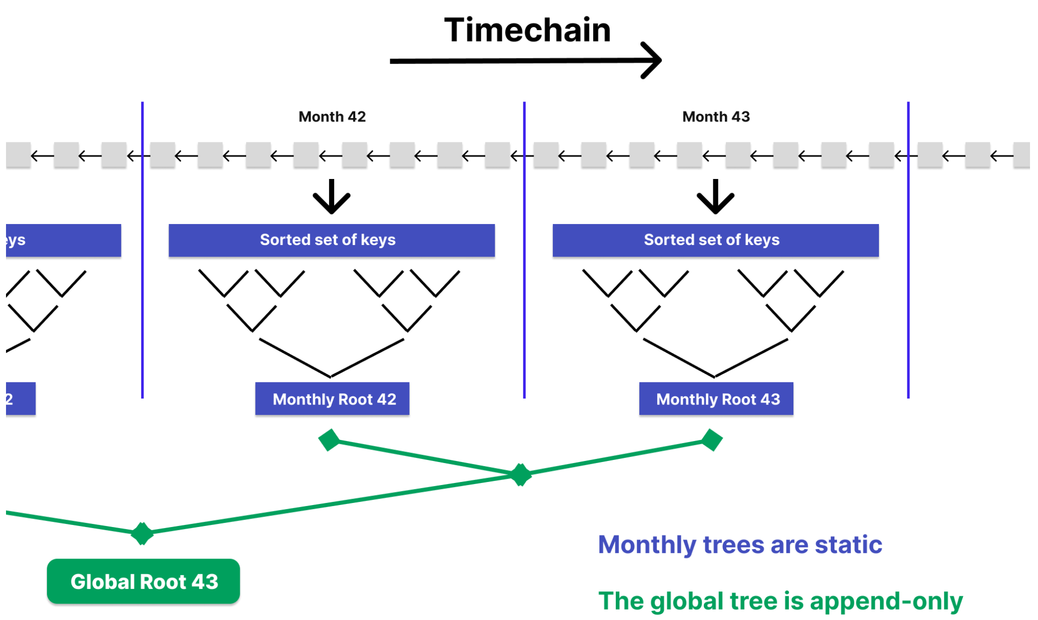


Figure 4: We can reduce the timechain accumulator to a constant size. Users periodically aggregate their monthly non-inclusion proofs into their account state proofs and discard all other data.

6.4 Improving Bitcoin with zkCoins

Existing CSV protocols, such as RGB and Taro, modulate tokens onto Bitcoin UTXOs. Our *zkCSV* protocol scales substantially better as it is fully decoupled from Bitcoin's UTXO set. Large batches of compact transaction commitments can be inscribed into a Bitcoin block similar to inscriptions in the Ordinals protocol[18]. Every user can become an aggregator and inscribe other users' commitments into the blockchain. In most other token protocols, users have to pay transaction fees denominated in the blockchain's native currency. Users cannot transfer any tokens, if they own no BTC. For zkCoin transactions, users pay fees to aggregators denominated in tokens, and *aggregators* pay the BTC fees to miners for them to include an inscription in a Bitcoin block. We estimate that the Bitcoin network can process about 100 zkCoin transactions per second – while *decreasing* the amount of verification required on the main layer, in comparison to a block full of Bitcoin transactions.

6.5 Limitations of zkCoins

Major limitations of zkCoins are:

- Transactions are interactive, which implies that the recipient has to be online. However, there's only one-way interaction required, from sender to the recipient.

This is a fundamental improvement over RGB because the sender can finalize the transaction on-chain without having to interact with the recipient first. The sender could store redundant copies of the encrypted transaction data with multiple trust-minimized middlemen, for the recipient to download when they come back online.

- The sender needs sufficient computational resources to prove their account updates. Is that feasible on a phone? Also backups are crucial. Losing an account's history proof leads to loss of funds.
- Bridging BTC coins to zkCoins requires some kind of SNARK verifier on Bitcoin's main layer. However, an initial prototype could use a workaround, such as a federated peg, Tether, or an eternal one-way peg (minting a zkToken by burning BTC coins).
- There is no global state, which implies fundamental data availability problems when trying to use zkCSV for a smart contract platform.

7 Conclusion and Outlook

We have proposed the first concept to generate a succinct validity proof of Bitcoin's chain state. We implemented a sophisticated prototype and our benchmarks indicate that a full chain state proof is computationally feasible indeed; however, open-source proof systems have yet to mature. Optimizing our tool stack to generate production-ready chain state proofs will take more engineering effort. We outlined a set of possible optimizations that will likely increase the prover's performance, such that we can prove all the existing blocks to catch up with the latest chain state and then extend a chain proof with the next block within the 10-minute block time. In addition, we have outlined an approach to decentralize proof generation.

Furthermore, we have introduced *zkCoins*, a client-side validation protocol combined with validity proofs, which enables tokens on top of Bitcoin with superior scalability properties and best-in-class privacy. We are working on publishing a concrete design and a prototype implementation.

With great excitement, we are following the development of the Simplicity[19] language, which would allow us to implement a SNARK verifier on Bitcoin's base layer. That enables various novel features such as zk-rollups and trustless two-way pegs. The latter allows us to peg BTC to a zkCoin, increasing the throughput of Bitcoin and improving its privacy. Simplicity will soon be activated on the Liquid sidechain, becoming an excellent testing field to experiment with on-chain proof systems and, eventually, the integration of a SNARK verifier into the base layer of Bitcoin.

References

[1] S. Nakamoto. Bitcoin: A Peer-to-Peer Electronic Cash System. `https://bitcoin.org/bitcoin.pdf`, 2008. Accessed: 2023-05-07.

[2] Global Bitcoin Nodes - Bitnodes. `https://bitnodes.io/nodes/all/#global-bitcoin-nodes`. Accessed: 2023-05-07.

[3] Statista. Size of the Bitcoin blockchain from January 2009 to July 11, 2022 . `https://www.statista.com/statistics/647523/worldwide-bitcoin-blockchain-size/`, 2021. Accessed: 2023-05-07.

[4] Mina Protocol — The World's First ZK Blockchain. `https://minaprotocol.com/`. Accessed: 2023-05-07.

[5] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Scalable, transparent, and post-quantum secure computational integrity. *Cryptology ePrint Archive*, 2018.

[6] Thaddeus Dryja. Utreexo: A dynamic hash-based accumulator optimized for the Bitcoin UTXO set. Cryptology ePrint Archive, Paper 2019/611, 2019. `https://eprint.iacr.org/2019/611`.

[7] Martin Westerkamp and Jacob Eberhardt. zkRelay: Facilitating Sidechains using zkSNARK-based Chain-Relays. Cryptology ePrint Archive, Paper 2020/433, 2020. `https://eprint.iacr.org/2020/433`.

[8] Alan Szepieniec, Tomer Ashur, and Siemen Dhooghe. Rescue-prime: a standard specification (sok). *Cryptology ePrint Archive*, 2020.

[9] Lorenzo Grassi, Dmitry Khovratovich, Christian Rechberger, Arnab Roy, and Markus Schofnegger. Poseidon: A New Hash Function for Zero-Knowledge Proof Systems. In *USENIX Security Symposium*, volume 2021, 2021.

[10] Lorenzo Grassi, Dmitry Khovratovich, and Markus Schofnegger. Poseidon2: A Faster Version of the Poseidon Hash Function. *Cryptology ePrint Archive*, 2023.

[11] Eli Ben-Sasson, Lior Goldberg, and David Levit. STARK Friendly Hash – Survey and Recommendation. Cryptology ePrint Archive, Paper 2020/948, 2020. `https://eprint.iacr.org/2020/948`.

[12] Eli Ben-Sasson, Dan Carmon, Swastik Kopparty, and David Levit. Elliptic curve fast fourier transform (ECFFT) Part I: fast polynomial algorithms over all finite fields. *arXiv preprint arXiv:2107.08473*, 2021.

[13] Eli Ben-Sasson, Dan Carmon, Swastik Kopparty, and David Levit. Elliptic Curve Fast Fourier Transform (ECFFT) Part II: Scalable and Transparent Proofs over All Large Fields. 2022.

[14] Andrew Poelstra. Using the Chain for what Chains are Good For . `https://www.youtube.com/watch?v=3pd6xHjLbhs&t=5755s`, 2017. Accessed: 2023-05-07.

[15] Peter Todd. Progress on Scaling via Client-Side Validation . `https://www.youtube.com/watch?v=uO-1rQbdZuk&t=6201s`, 2016. Accessed: 2023-05-07.

[16] The RGB Project. What is RGB? . `https://www.rgbfaq.com/faq/what-is-rgb`, 2021. Accessed: 2023-05-07.

[17] Lightning Labs. Taro . `https://docs.lightning.engineering/the-lightning-network/taro`, 2022. Accessed: 2023-05-07.

[18] Casey Rodarmor. Ordinals Inscriptions . `https://docs.ordinals.com/inscriptions.html`, 2023. Accessed: 2023-05-07.

[19] Russell O'Connor. Simplicity: A new language for blockchains. In *Proceedings of the 2017 Workshop on Programming Languages and Analysis for Security*, pages 107–120, 2017.

Acknowledgement

We would like to acknowledge the support and contributions of various individuals and organizations in the development of the ZeroSync project. We are grateful to Geometry for providing the initial research grant and guidance that facilitated the commencement of this project. We also appreciate StarkWare's financial support through an additional grant, which was essential for the project's progress.
We extend our gratitude to Ruben Somsen, Louis Guthmann, Max Gillett, Tino Steffens, Andrew Milson, Giacomo Zucco, and Blockstream for their valuable mentoring and support during the course of our research and prototype development. Their expertise and guidance have been crucial in shaping the project's direction and outcomes.
We thank everyone who has contributed to the ZeroSync project and look forward to further collaborations and advancements in this field.