Proof of Necessary Work: Succinct State Verification with Fairness Guarantees

Assimakis Kattis, Joseph Bonneau

New York University

Abstract. Blockchain-based payment systems utilize an append-only log of transactions whose correctness can be verified by any observer. Classically, verification costs grow linearly in either the number of transactions or blocks in the blockchain (often both). Incrementally Verifiable Computation (IVC) can be used to enable constant-time verification, but generating the necessary proofs is expensive. We introduce the notion of Proof of Necessary Work (PoNW), in which proof generation is an integral part of the proof-of-work used in Nakamoto consensus, producing proofs using energy that would otherwise be wasted. We implement and benchmark a prototype of our system, enabling stateless clients to verify the entire blockchain history in about 40 milliseconds.

Keywords: proof-of-work · zero-knowledge proofs · consensus algorithms

1 Introduction

Balancing throughput with decentralization is a major challenge in modern cryptocurrencies. Current systems such as Bitcoin require participants to process the entire system history to verify that the current state (the most recent block in the chain) is correct. Despite strict limits on blockchain growth which cap total system throughput, verification costs are prohibitive for many clients. Joining the system requires downloading and verifying over 450 GB of blockchain history (as of 2022) which takes days on a typical laptop. In practice, most clients don't perform independent verification and rely on a trusted third party instead.

1.1 Succinct Blockchains

Succinct blockchains aim to support efficient verification of the system's entire history by any participant without trusting any third parties. Participants only need to obtain some fixed public parameters from a trusted source (e.g. the genesis block and the system's rules). Participants can then join the system at any time and receive a succinct validity proof for the most recent block in the system using minimal bandwidth and time. These proofs demonstrate both that there exists a sequence of valid transactions from the genesis state $\mathcal{S}_0$ to the state committed in the current block, and that the block's *branch* (the sequence of predecessor blocks) is of quality q according to the consensus protocol. In this

work we focus on aggregate proof-of-work (PoW) difficulty as the measure of branch quality, as used in Bitcoin consensus. Currently, systems such as Bitcoin require $O(t + h)$ work to completely verify a branch containing t transactions and h blocks. Succinct proofs enable optimal asymptotic performance of $O(1)$ verification costs for a client joining the system at any point in its history.

1.2 Incentivizing State Compression

A key challenge for succinct blockchains is incentivizing the (relatively expensive) costs of computing a validity proof for each block. Meanwhile, Bitcoin employs proof-of-work (PoW), which provides system security by verifying energy consumption. This energy, while necessary for the consensus protocol, is not used for anything else and hence is often described as 'wasted.' We propose a new approach to useful PoW in which the work aids in the verification of the system itself. We denote this as *proof of necessary work* (PoNW) and show how it can be used within a succinct blockchain architecture as a suitable PoW puzzle.

A synergistic benefit is directly incentivizing hardware acceleration of zero-knowledge proofs. This is relevant for many distributed payment systems in which proof generation time is a critical bottleneck limiting transaction throughput and/or latency [6,8,16,21]. Indeed, recent industry developments [2] based on our work have yielded interest in dedicating resources toward an industry-wide effort to maximize the performance of zero-knowledge proof systems. We believe this to be beneficial not only for distributed payments, but also for any application where high-throughput, low-latency and low-energy zero-knowledge proof generation is required.

1.3 Contributions

Building a consensus algorithm which produces validity proofs for each block as a useful byproduct requires carefully designing the PoW process to replicate the security properties of Bitcoin's non-useful puzzle. Our main technical contribution is a method to deeply embed a nonce into the proof computation process, making it suitable as a *progress-free* PoW puzzle. We formalize this intuition by introducing the notion of ϵ-amortization resistance and propose a protocol which achieves this. Our results are based on the average-case hardness of multiexponentiation in the Generic Group Model (GGM) [35].

We implement our prototype at an 80-bit security level and benchmark its performance and establish feasibility. Our system:

1. produces block headers of size < 500 bytes for any number of txs/block,
2. allows stateless clients to verify a block in < 20ms, and
3. achieves throughput of 50 tx/block.

In terms of throughput and block header size, our prototype is about an order of magnitude worse than Bitcoin. Bitcoin block headers are 80 bytes and throughput is about 1,000 transactions per block. However, our system allows

a stateless client to rapidly verify a block (and thus its complete history) in milliseconds with 500 bytes of data downloaded. In Bitcoin, full verification of a block requires many hours of computation time and downloading hundreds of gigabytes of data. The efficient block verification provided by our system also assists miners in quickly validating new blocks broadcast on the network, which may reduce the risk of block collisions and enable faster block frequency.

2 Proof of Necessary Work

To allow proof generation to serve as a PoW puzzle, we require (a) a proof π_i whose generation algorithm $\mathcal{P}$ is moderately difficult to compute and (b) a PoW puzzle $\mathsf{P}_V^{\mathcal{H},d}$ that requires the miner to fully recompute $\mathcal{P}$ to test a potential solution. The second property is necessary for the puzzle to be progress-free for fairness to miners of differing size. Indeed, if generating unique proofs π_i based on randomly sampled nonces n_i is sufficiently 'hard', then using $\mathsf{P}_V^{\mathcal{H},d}$ instead of a generic puzzle (such as computing the double SHA256 digest in Bitcoin) would allow us to not only perform PoW with the same theoretical guarantees, but also compute a valid proof π_i in the process.

We do not formally analyze any consensus properties, since our goal is not to design a new consensus protocol but to retain that used by Bitcoin (and similar systems) and inherit its properties. However, we would like the work done to be useful by producing proofs of each block's validity. We introduce the notion of performing PoW by proving the validity system state, denoted by *Proof of Necessary Work* (PoNW).

2.1 Definitions

We formalize this definition below, and provide the relevant security model. For a full specification of all terms and notations, we refer the reader to Appendix A.

Definition 1 (Proof of Necessary Work). *Given a pseudorandom function $\mathcal{H}$ and a proof $\pi_i \in \mathcal{Z}$ in some RSM with transition tuple ($\mathsf{NewState}$, $\mathsf{VerifyState}$), we define the verification puzzle $\mathsf{P}_V^{\mathcal{H},d} : \mathcal{S} \times \mathcal{S} \times \mathcal{Z} \to \{0,1\}$ with difficulty d as the solution to the following function:*

$$\mathsf{P}_V^{\mathcal{H}}(\mathcal{S}_i, \mathcal{S}_{i+1}, \pi_{i+1}) = \mathbf{1}\left[\begin{array}{c} \mathsf{VerifyState}(\mathcal{S}_i, \mathcal{S}_{i+1}, \pi_{i+1}) = 1 \\ \mathcal{H}(\pi_{i+1}) < d \end{array}\right],$$

where $\mathbf{1}[\cdot]$ is the indicator function.

By having access to a proof generating algorithm $\mathcal{P}(\mathbf{t}, \mathcal{S}_i, \mathcal{S}_{i+1}, n_i) \to \pi_{i+1}$ that generates unique (yet valid) π_{i+1} for each n_i, we can generate π_{i+1} for $\mathcal{S}_{i+1} = \mathsf{NewState}(\mathbf{t}, \mathcal{S}_i, \pi_i)$ using a uniformly randomly sampled n_i until the puzzle condition is satisfied:

$$\mathsf{P}_V^{\mathcal{H}}(\mathcal{S}_i, \mathcal{S}_{i+1}, \mathcal{P}(\mathbf{t}, \mathcal{S}_i, \mathcal{S}_{i+1}, n_i)) = 1.$$

Then π_{i+1} suffices for public verification that PoW has been performed. This is because our prover will always fail with constant probability (when $\mathcal{H}(\pi_{i+1}) \geq d$), so iteratively sampling new proofs (by sampling new n_i) until a valid one is found can be shown, under the assumption that $\mathcal{P}$ is the most efficient way to find such an n_i, to be a memoryless exponential process and hence *fair*. Note that, by construction, we also guarantee that π_{i+1} is a valid witness for the RSM. The number of transactions verified is always *fixed* (with empty transactions still 'added') as otherwise miners would be incentivized to mine puzzles with the smallest blocks.

An Initial Approach A natural thought would be to require the generation of proofs until $\mathcal{H}(\pi) < d$, as is proposed in the previous section. In the case that the proof is unique to the state and witness input, we can ensure that by adding a nonce in the input we will always get a different hash for π. However, this can lead to unfair outcomes. When computing π, the adversary can retain the parts of π that don't change between nonces and therefore substantially decrease proof generation time with respect to other provers. This means the process is not memoryless, and so the fairness of the system is compromised.

Amortization Resistance Like Nakamoto consensus, our puzzle needs the property that solutions are equally hard to test even after testing an arbitrary number of previous solutions. In other words, a miner should not be able to *amortize* costs while testing multiple potential solutions. This property is defined more formally below based on the μ-Incompressibility of [28], although we work in the bounded-size precomputation model. We model PoNW as a function $f^{\mathcal{O}}$ with limited access to some oracle $\mathcal{O}$ that performs a hard computation in an encoding of some group $\mathbb{G}$.

Definition 2 (ϵ-Amortization Resistance). *For inputs of length λ and ouputs $q \in poly(\lambda)$, function $f^{\mathcal{O}} = \{f^{\mathcal{O}}(n)\}_{n\in\mathcal{N}}$ is ϵ-amortization resistant on average with respect to a sampler S if for all adversaries $\mathcal{A} = (\mathcal{A}_1^{\mathcal{O}}, \mathcal{A}_2^{\mathcal{O}})$ with $\mathcal{A}$ performing less than $(1-\epsilon)qN$ queries to the oracle $\mathcal{O}$ on average, where N number of queries required for one evaluation of $f^{\mathcal{O}}(n)$ on average, the following is negligible in λ:*

$$\Pr\left[\forall i \in [q], \pi_i = f^{\mathcal{O}}(n_i) \middle| \begin{matrix} \{n_i\}_{i=1}^q \leftarrow n, (n, \mathsf{aux}) \leftarrow \mathsf{S}(1^\lambda) \\ \mathsf{precomp} \leftarrow \mathcal{A}_1^{\mathcal{O}}(1^\lambda, \mathsf{aux}) \\ \{\pi_i\}_{i=1}^q \leftarrow \mathcal{A}_2^{\mathcal{O}}(1^\lambda, n, \mathsf{precomp}) \end{matrix}\right].$$

This definition captures the fact that computing multiple proofs does not come with marginal gains: indeed, provers cannot use larger computational resources to batch process proofs and achieve disproportionate performance improvements. By preventing large miners from achieving algorithmic returns-to-scale, this property is crucial in ensuring fairness. With the above objectives in mind, we now look at how to adapt our implementation to realize such a system.

Prover Computational Costs Before we look at designing an amortization resistant PoNW system, we summarize the computationally expensive components of proof generation in the Quadratic Arithmetic Program (QAP) Non-Interactive Proof (NIPs) of [30] compiled with [22]. For an ℓ-size statement with m internal variables and n constraints, the prover $\mathcal{P}$ needs to (1) update inputs and witnesses, and (2) perform $9m + n$ exponentiations in $\mathbb{G}$ using elements from the proving key as bases. Since updating variable assignments is orders-of-magnitude faster, amortization resistance requires $\mathcal{P}$ to recompute (almost) all exponentiations for each new nonce. We provide the formal definition of QAP instances below for completeness.

Definition 3. *A QAP Q over field $\mathbb{F}$ contains three sets of $m+1$ polynomials $\mathcal{V} = \{v_k(X)\}, \mathcal{W} = \{w_k(X)\}, \mathcal{Y} = \{y_k(X)\}$, for $k \in \{0, ..., m\}$ and a target polynomial $t(X)$ of degree n. Suppose F is a function that takes as input ℓ_1 elements and outputs ℓ_2 elements for a total of $\ell = \ell_1 + \ell_2$ elements. We say that Q computes F if: $(a_1, ..., a_\ell) \in \mathbb{F}^\ell$ is a valid assignment of F's inputs and outputs iff there exist $(a_{\ell+1}, ..., a_m)$ for which $t(X)$ divides $p(X)$ where*

$$p(X) := \left(v_0(X) + \sum_{i=1}^{m} v_i(X)\right) \cdot \left(w_0(X) + \sum_{i=1}^{m} w_i(X)\right) - \left(y_0(X) + \sum_{i=1}^{m} y_i(X)\right).$$

Amortization of Multiexponentiation Multiexponentiation is inherently amortizable [17,20] given enough memory, although space requirements scale exponentially with the number of computed elements. This is because we can precompute the exponents of specific basis elements and perform look-ups that can be used by multiple evaluations at once. We make precise the relationship between size and amortization gain to demonstrate that non-negligible amortization gains require an infeasibly large amount of space. Since we are interested in average-case guarantees, all input elements to the multiexponentiation algorithm (i.e. the enumerated exponents, or puzzle instances) are sampled uniformly randomly from some S.

We consider amortization in Shoup's Generic Group Model (GGM) [35],[1] in which the adversary can only compute products based on existing group elements (with non-negligible probability), or directly query the exponentiation of some index. The adversary has access to a multiplication oracle $\mathcal{O} : \mathbb{G} \times \mathbb{G} \to \mathbb{G}$, which returns the multiplication of the input elements over some random encoding $\sigma : \mathbb{Z}_p \to \mathbb{G}$. This oracle computes $\mathcal{O}(\sigma(i), \sigma(j)) = \sigma(i + j)$. The adversary may also use a polynomially-sized precomputation string. Since they don't have access to the exponents of the bases that are being multiplied together (so as to perform a direct look-up), computing some $\sigma(k)$ requires the generation of an addition chain ending with $\sigma(k)$.

However, this is the only assumption underlying the lower-bound results which prove the optimality of (the generalized) Pippenger's algorithm [20], as they obtain lower-bounds on the length of the minimal addition chain needed to

[1] Maurer proposed a slightly different GGM definition [27], for a comparison see [38].

compute some element. In short, our main formal contribution relies on adapting the packing lower-bound ideas of [15,31] to formalize the relationship between amortization of multiexponentiation of random indices and the amount of space available to the adversary. We do this by making explicit the average-case lower bounds for multiexponentiation, which were only stated (but not proven) in [15,31] to be a constant term away from the worst-case lower bounds.

Note that the notion of average-case hardness requires an underlying probability distribution over which the input indices are sampled. Obviously, the distribution of the sampled puzzle instances can affect the average-case bounds if, for example, the sampler provides structured output with high probability. Therefore, all results have to be taken with respect to the underlying distribution of the inputs, which is in turn specified by the choice of sampling algorithm S. Where this S is taken to be uniform (as in this work), the notion of average-case hardness defaults to the traditional average-case lower bound results referenced in the literature.

In order to make formal statements about the amortization resistance of computing multiple NIPs, we need to show that there exists some sampling algorithm S_{NIP} outputting instance-witness pairs (ϕ, w) so that, on average over its public coins, these output puzzle instances require a minimum number of oracle calls each for computation of their corresponding proof π. The first step towards this is to construct the equivalent multiexponentiation problem that the above will reduce to. In the following, we restrict ourselves to the NIP of [30], in which the valid output proof consists of 9 group elements of the form $\sum_{k=1}^{\kappa} w_k G_k^i$ for $i \in [9], w_k \in [N]$ and an additional element $\sum_{m=1}^{\mu} g(w_1, ..., w_\kappa)_m H_m$, where g an m-dimensional n-variable polynomial encoding the instance's witness and $G_i, H_m \in \mathbb{G}$.

Since the hardness of the above computation depends on the structure of w and g, it becomes apparent that we need to restrict the types of *predicates* that we are looking at. In subsequent sections, we make precise the following construction: a circuit with an efficient sampler S such that (1) accepting witness elements $w_1, ..., w_\kappa \in [N]$ are randomly distributed, (2) for each valid instance ϕ there exists only one valid w, and (3) for each valid w, there exists a unique valid g. Note that (1) and (2) are properties of the predicate, while (3) requires a stronger result on the NIP's knowledge guarantees. We will show that predicates satisfying (1) and (2) are enough to reduce the computation of a NIP from [30] (which satisfies (3)) to a multiexponentiation problem (Definition 4) whose amortization we can bound.

Definition 4. *The (κ, μ)-length MultiExp function $f : [N]^\kappa \to \mathbb{G}^\nu$ of dimension ν for bases $\{G_i^{(1)}, ..., G_i^{(\nu-1)}\}_{i=1}^{\kappa}$, $\{G_i^{(\nu)}\}_{i=1}^{\mu}$, and function $g : [N]^\kappa \to K \subseteq [N]^\mu$ is*

$$f(x_1, ..., x_\kappa) := \left(\sum_{i=1}^{\kappa} x_i G_i^{(1)}, ..., \sum_{i=1}^{\kappa} x_i G_i^{(\nu-1)}, \sum_{i=1}^{\mu} g(x)_i G_i^{(\nu)} \right),$$

where the x_i are given by sampler S, based on its random coins.

In order to provide a reduction that exactly captures the average-case hardness of the above problem, the structure of g becomes important. This requires a more technical treatment, so here we work in the case where g is a weakly collision-resistant map from the witness elements $x = (x_1, ..., x_\kappa)$ to the values $(g(x)_1, ..., g(x)_\mu) \in K \subseteq [N]^\mu$. This defines a computationally unique correspondence between witness elements and representations of μ-degree polynomials with coefficients in $[N]$. We specifically require the mapping $g : [N]^\kappa \to K \subseteq [N]^\mu$ to be collision-resistant in each of its output coordinates, or that the following probability is negligible for all PPT adversaries $\mathcal{A}$:

$$\Pr\left[\exists i \text{ s.t. } g(\mathcal{A}(z))_i = z_i; z \leftarrow g(x), x \leftarrow_R [N]^\kappa\right] \approx 0,$$

where z_i denotes the i-th coordinate of z. This is enough to provide multiexponentiation amortization bounds, which are given below for the case when $\kappa = \mu$. Note that the general case for $\mu > \kappa$ can also be calculated in the exact same way, but has been omitted for simplicity.

Theorem 1. *The (κ, κ)-length MultiExp function (c.f. Definition 4) of dimension ν over index size $\lambda := \log(N)$, group $\mathbb{G}$ with $|\mathbb{G}| = 2^\lambda$, and storage size q is ϵ-amortization resistant with respect to the uniform sampler for all collision-resistant g, and for large enough κ, λ, ν, q satisfies:*

$$\epsilon \leq \frac{\log(q) + o(1)}{\log(q) + \log(\kappa) + \log(\nu) + \log(\lambda)}.$$

We prove Theorem 1 in Appendix D. This amortization gain is unavoidable for NIPs that reduce to multiexponentiation; such as by compilation with [22].

2.2 Amortization Resistance & Efficiency

We modify the DPS predicate Π to ensure that most of the proof variables change unpredictably with modifications of the nonce or state. This gives amortization resistance in exchange for increasing the number of variables and constraints in the predicate. The performance overhead originates from the need to commit to state and 'mask' the computation, which can be expensive for large predicates.

The naive approach would be to isolate each of the different circuits in the system and show that they can be modified to change unpredictably based on some seed. The design challenge here is how to make this happen while conserving the proof's correctness guarantees. For this, we ideally want to leverage a property specific to our predicate in order to 'mask' the computations and treat the proving system as a black box. We leverage the Pedersen hash function to transform our predicate Π to an amortization resistant version in Section 4.3.

Committing to State Given some nonce n, the prover might only change a part of the input in order to (re)check difficulty. This is an issue if the same nonce can be used with many inputs (in our case, transactions), as an adversarial prover

would compute a proof and then only switch out a single transaction (or bit!), rechecking difficulty with no expensive recomputation. Define $\rho := \mathsf{PRF}_n(\mathsf{state})$ that commits to state where PRF a pseudorandom function family. We need to commit to all block transactions, ensuring that changing one transaction changes ρ. This can be expensive if we exploit no information about the underlying predicate, since PRF would have to commit to every single original variable.

Fortunately, for our predicate the input to PRF is small: we use $\rho = \mathsf{PRF}_n(rt)$ where rt the root of the new state and n the given nonce. Since this input will anyways be computed as part of the protocol, we don't actually suffer any overhead apart from having to verify the above computation. Note that this is actually *constant* in predicate size. In the GGM, we can replace the PRF by a collision resistant hash function CRT instead, since the randomness of the group encoding is sufficient for the witness elements to look random to an adversary.

Masking the Computation We can force unique changes to the Merkle path updating the account if we require n to be part of the leaf: since a change in the block (or nonce) would lead to a new n, all update paths need to be recomputed if any transaction is changed. However, we also need to enforce change to the *old* Merkle path checking account existence. This technique is thus not ideal, since these paths do not depend on the current nonce (or state) at all, meaning that around half our variables will remain the same, giving $\epsilon \approx 1/2$.

To get around this, we opt for a different approach. We 'mask' the input variables to $\mathcal{H}$ by interaction with ρ (which also commits to n) and transform the constraints of the hash function subcircuit $C_{\mathcal{H}}$ into a new circuit that retains the original Proof of Knowledge (PoK) guarantees by verifying the same underlying computation. By the unpredictability of ρ and randomness of n, we hope to achieve upper bounds for amortization resistance based on the security of the CRT. In this case, the sampler would need to provide valid witnesses for $C_{\mathcal{H}}$ of the form $w = (w_1, ..., w_m)$ whose encodings are indistinguishable from random, given n sampled uniformly randomly and access to a multiplication oracle $\mathcal{O}$ for a randomized encoding of some $\mathbb{G}$.

3 Implications for Nakamoto Consensus

PoNW introduces two novel effects on the consensus protocol due to the fact that checking a nonce (on the order of seconds to minutes) can now take a significant fraction of the average block frequency (ten minutes in the case of Bitcoin), whereas it was negligible for traditional PoW puzzles. We can evaluate these effects assuming a single puzzle solution takes time τ to check (with the mean block arrival time normalized to 1).

3.1 Quantization Effects

When τ becomes a significant fraction of the average block generation time ($\tau \sim 1$), miners face a loss of efficiency as they will often be forced to discard

a partially-checked puzzle solution when a block is broadcast while checking previous solutions. We prove the scale of this efficiency loss in a short theorem:

Theorem 2. *A miner in a PoW protocol with puzzle checking time τ will discard a fraction $1 - \frac{\tau}{e^{\tau}-1}$ of their work due to newly broadcast solutions.*

Note that as $\tau \to 0$ (fast puzzle checking time relative to block interval), the fraction of wasted work drops to 0. This is why this effect has never been considered in prior work. In the reverse direction, as $\tau \to \infty$ the fraction of wasted work approaches 1. For $\tau = 1$ (solutions take as long to check as the mean block interval), the fraction of wasted work is $\frac{e-2}{e-1} \approx 0.42$, suggesting that we should aim to keep the time (even for slow miners) to get a solution significantly shorter than the mean block time.

3.2 Stubborn Mining and Collisions

Slow puzzle checking time also introduces a concern that miners might refuse to stop working on a partially-checked solution (and hence discard partial work) even if a valid solution is found and broadcast. These *stubborn* miners might cause collisions in the blockchain (two blocks being found at the same height in the chain). We can analyse a worst-case scenario in which all miners are synchronized with identical proving time, in effect making all miners stubborn and maximizing the probability of simultaneous solutions. If miners aren't synchronized, they may opt to finish their current effort after a block is found, but even if all miners do so this reduces to the above case where all miners finish checking a solution simultaneously. We call each synchronized period in which all miners check a solution a *round*.

Theorem 3. *The expected number of solutions in a synchronized mining round is defined by a Poisson distribution with $\lambda = \tau$. The proportion of rounds with multiple solutions (of rounds with any solution) is upper bounded by $\tau/2$.*

By Theorem 3, our prototype unoptimized 100 second proving time (and 10 minute block time) would lead to collisions for fewer than $\frac{1}{12}$ of blocks in the worst case.

4 Design & Instantiation

4.1 Proof System & Predicate

Since we'll be broadcasting each proof π_i to the network, we would like them to be quite small (ideally $<$ 1kB). We also require that the size of π_i does not increase with i, ideally staying the same size after every state transition. With these design choices in mind, we prototype our system using libsnark[34], a C++ library implementing the IVC system in [4] using the construction from [30]. This is done using Succinct Non-Interactive Arguments of Knowledge (SNARKs) [3], non-interactive proofs of knowledge with the additional property of *succinctness*:

producing constant-sized proofs that can be instantly verified. We can equivalently consider Π_S as an arithmetic circuit C_Π, evaluating to 1 on some input B_i if and only if B_i is a valid commitment to the output of UpdateState given some transaction set $\mathbf{t}$ and $\mathcal{S}_{i-1}$. In our implementation, C_Π is a QAP.

The circuit is encoded over elliptic curve elements through vectors in $\mathbb{F}_p$, where the number of gates increases with the size of π_i and the time required to generate it. By manually designing a circuit C_Π, we minimize the number of gates used and provide a deployable implementation. Note that the system need also allow for *recursive proof composition*, or the capability of new proofs to check the validity of previous proofs efficiently. Since this construction depends on SNARKs over pairs of elliptic curves that form *IVC-friendly cycles*, we use the same pair of non-supersingular curves of prime order as [4] with 80 bits of security and field size $\log p \approx 298$.

4.2 Circuit Requirements

A tree depth of 32 for our implementation allows for 4.2 billion accounts. We compare this to 32 million unique used wallets on the Bitcoin blockchain after 10 years of operation. This requires $32 \cdot 4 = 128$ hash checks for each transaction. We use the circuits in libsnark to verify such proofs of inclusion and modification.

Pedersen Hashes Since it is desirable for $\mathcal{H}$ to be efficiently represented with a low gate count, we opt for using Pedersen hashes [14]. We modify the Pedersen hash to compute $\prod_{i=1}^{D} G_i^{1-2x_i}$ where $\{x_i\}_{i=1}^{D}$ is the bit representation of the input x and $\{G_i\}_{i=1}^{D}$ is a set of primitive roots for an elliptic curve group $E(\mathbb{F}_p)$. We encode each root as two field elements and, based on the sign of each input x_i, perform multiplication of an intermediate field variable c by each G_i to arrive at the digest if the corresponding $x_i = 1$. We use the same underlying elliptic curve for the SNARK with $|p| = 2^{298}$, which reduces in security to the elliptic curve discrete-logarithm problem (ECDLP) at a security of 80 bits.

Signature Scheme We use Schnorr signatures [33] over an elliptic curve (EC), based on the hardness of DLP. This choice is motivated by our desire to minimize the size of the verifying circuit, since this has to be built inside C_Π. The Schnorr verification circuit only requires two exponentiations, a hash computation, and a comparison between scalars. The same curve from the IVC construction is also used here, offering a security of 80 bits. Schnorr signatures use elliptic curve elements as public keys, resulting in key sizes of 596 bits, or $298 + 1 = 299$ bits with point compression. Secret keys are sampled as random 298-bit strings.

4.3 Randomizing the Pedersen Hash

In addition to some input x of length n bits, our evaluation requires a pseudorandom seed $\rho \in \{0,1\}^n$. Consider the following modification, which can be

thought of as masking the underlying evaluation by using two sets of input variables: $\mathcal{H}_G(\rho)^2 \cdot \mathcal{H}_H(\rho)$ and x_i for $i \in [n]$, where $\mathcal{H}_G(\cdot)$ the evaluation of the Pedersen function $\mathcal{H}_G(x) = \prod_{i=1}^{n} G_i^{1-2x_i}$.

The variable $h_0 = \mathcal{H}_G(\rho)^2 \cdot \mathcal{H}_H(\rho)$ forms the 'starting point' of the evaluation. In the beginning, the prover will have access to generator constants $\{H_i, H_i^{-1}, G_i^{-2}H_i^{-1}, G_i^2 H_i\}$ for the specific instance of the problem. It would then perform a 2-bit lookup based on x_i and ρ_i, multiplying the intermediate variable c_i by one of the above. By carefully choosing these q_i, we can design the circuit in such a way that unpredictability based on the seed is retained by all intermediate variables except the output y, which we ensure equals $\mathcal{H}_G(x)$.

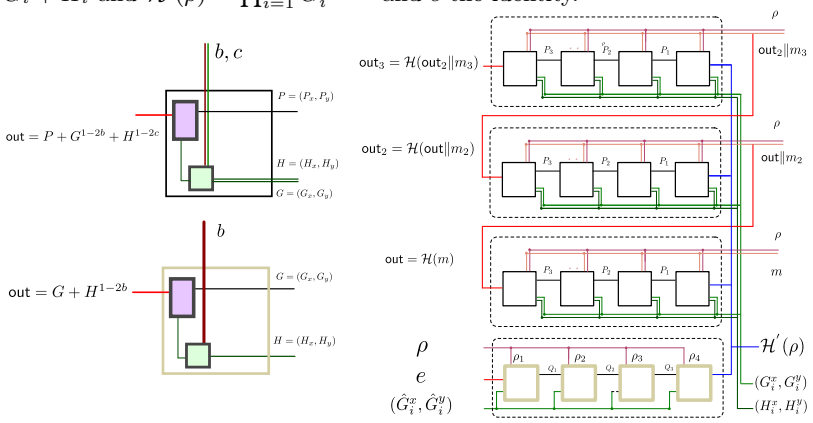
Algorithm 1 MaskedPedersen

Require: $x, \rho \in \{0,1\}^n, G, H \in \mathbb{G}^n$
Ensure: $y \in \mathbb{G}$

procedure CacheGenerators(ρ, G, H)
  Parse $\{\rho_i\}_{i=1}^n \leftarrow \rho$
  Compute $h \leftarrow \mathcal{H}(\rho, G)$, $h_2 \leftarrow \mathcal{H}(\rho, H)$, $h_0 = h^2 \cdot h_2$
  return h_0, h
end procedure
procedure MaskedHash(x, ρ, h_0, h)
  Parse $\{x_i\}_{i=1}^n \leftarrow x$, $\{\rho_i\}_{i=1}^n \leftarrow \rho$
  Define $q = \{q_i\}_{i=1}^n$, $c = \{c_i\}_{i=0}^n$ and set $c_0 = h_0$
  for $i \leq n$ **do**
    if $\rho_i = 0, x_i = 0$ **then** $q_i = H_i^{-1}$
    else if $\rho_i = 0, x_i = 1$ **then** $q_i = G_i^{-2} \cdot H_i^{-1}$
    else if $\rho_i = 1, x_i = 0$ **then** $q_i = G_i^2 \cdot H_i$
    else if $\rho_i = 1, x_i = 1$ **then** $q_i = H_i$
    end if
    $c_i = c_{i-1} \cdot q_i$
  end for
  $y = c_n \cdot h^{-1}$
  return y
end procedure

Correctness follows from the following observation: at step 0, the variable $c_0 = \mathcal{H}_H(\rho) \cdot \mathcal{H}_G(\rho)^2 = \mathcal{H}_G(\rho) \cdot \prod_{i=1}^{n} G_i^{1-2\rho_i} \cdot H_i^{1-2\rho_i}$ is initialized as the hash of the seed. For all intermediate steps $j < n$, we have that $c_j = \mathcal{H}_G(\rho) \cdot \left(\prod_{i=1}^{j} G_i^{1-2x_i}\right) \cdot \left(\prod_{i=j+1}^{n} G_i^{1-2\rho_i} H_i^{1-2\rho_i}\right)$. Finally, after the n-th bit has been processed the final intermediate variable c_n is equal to the Pedersen hash of the original input x multiplied by (the unpredictable) $\mathcal{H}_G(\rho)$. By multiplying with $\mathcal{H}_G(\rho)^{-1}$, we get $\mathcal{H}_G(x)$. This follows easily from the fact that at every step we are performing the following operation: $c_i = c_{i-1} \cdot (H_i \cdot \mathbf{1}[\rho_i, x_i = 1] + H_i^{-1} \cdot \mathbf{1}[\rho_i, x_i = 0] + G_i^{-2}H_i^{-1} \cdot \mathbf{1}[\rho_i = 0, x_i = 1] + G_i^2 H_i \cdot \mathbf{1}[\rho_i = 1, x_i = 0])$. It can be quickly checked that this computation ensures the previous recursive property when initialized with $c_0 = \mathcal{H}_H(\rho) \cdot \mathcal{H}_G(\rho)^2$. By induction, this implies

Fig. 1. *Left*: The TwoBitGroupAddition and SymmetricGroupAddition circuits from top to bottom respectively. *Right*: Layout of a single Merkle authentication path circuit, with $M = 3$ evaluations of $\mathcal{H}$ on an $n = 4$-bit Pedersen hash. $\hat{G}_i = (\hat{G}_i^x, \hat{G}_i^y) = G_i + G_i + H_i$ and $\mathcal{H}'(\rho) = \prod_{i=1}^{4} \hat{G}_i^{1-2\rho_i}$ and e the identity.



that after the n-th bit, only $\mathcal{H}_G(\rho)$ and the exponentiations due to the bits of x remain in the output variable i.e. $c_n = \mathcal{H}_G(\rho) \cdot \prod_{i=1}^{n} G_i^{1-2x_i}$.

We observe that in all cases where we know that the variable a_i has small support (when, for example, it is boolean $a_i \in \{0, 1\}$), the prover can always precompute once and use the same answers without performing exponentiations. However, this is not a problem since all miners would know what the precomputed answers are from the very beginning and can incorporate them with a small memory cost.

The problem with creating variables that become more and more 'deterministic' is that at some point their support becomes so small that an adversary will be able to precompute some oracle queries. However, since the end value of the sequence of variables $\{c_i\}_{i=1}^{n}$ is $h \cdot \mathcal{H}_G(x)$ which is also unpredictable due to h, it is not feasible to predict any index $i \in [n]$ without violating the security of the operation $\mathcal{H}_G(\rho) = h$ even if $\mathcal{H}_G(x)$ is previously known. Note that h can be 'offset' by a random element I as $h_i' = h + I_i$ for each path $i \in [N]$. This provides independence between authentication paths using the same nonce.

4.4 Security

Unique Witness Extraction We must restrict the proof systems used because certain constructions are inherently insecure: Groth16 [18] can easily be re-randomized, for example, with only a few additional group multiplications. We thus need a notion akin to non-malleability, ensuring that we cannot construct proofs given access to previous valid proofs. To achieve this, we show that Pinocchio [30] satisfies *unique witness extractability*. This property requires the proof system to output proofs with unique encodings for each distinct statement-witness pair, and hence rules out malleability.

Definition 5. *Let* $\mathsf{NIP} := (\mathsf{Setup}, \mathsf{Prove}, \mathsf{Verify}, \mathsf{Simulate})$ *denote a NIP for relation* $\mathcal{R}$*. Define the PPT algorithm* $\mathcal{A}$ *with extractor* $\chi_{\mathcal{A}}$*,* $\mathbf{Adv}^{uwe}_{\mathcal{BG},R,\mathcal{A},\chi_{\mathcal{A}}}(\lambda) = \Pr[\mathcal{G}^{uwe}_{\mathcal{BG},R,\mathcal{A},\chi_{\mathcal{A}}}(\lambda)]$*, and* $\mathcal{G}^{uwe}_{\mathcal{BG},R,\mathcal{A},\chi_{\mathcal{A}}}(\lambda)$ *as:*

$\mathsf{Main}\ \mathcal{G}^{uwe}_{\mathcal{BG},R,\mathcal{A},\chi_{\mathcal{A}}}(\lambda)$

$(p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e, g) \leftarrow \mathcal{BG}(1^\lambda)$
$(\mathbf{crs}, \tau) \leftarrow \mathsf{Setup}(\mathcal{R})$
$(\phi, \pi_1, \pi_2) \leftarrow \mathcal{A}^{\mathcal{O}}(\mathbf{crs})$
$(w_1, w_2) \leftarrow \chi_{\mathcal{A}}(\mathsf{tr}_{\mathcal{A}})$
$b_1 \leftarrow (w_1 = w_2) \cup (\mathcal{R}(\phi, w_1) \neq 1) \cup (\mathcal{R}(\phi, w_2) \neq 1)$
$b_2 \leftarrow \mathsf{Verify}(\mathbf{crs}, \phi, \pi_1) \cap \mathsf{Verify}(\mathbf{crs}, \phi, \pi_2) \cap ((\phi, \pi_1) \notin Q) \cap ((\phi, \pi_1) \notin Q) \cap (\pi_1 \neq \pi_2)$
$\mathsf{Return}\ b_1 \cap b_2$

$\mathcal{O}(\phi)$

$\pi \leftarrow \mathsf{Simulate}(\mathbf{crs}, \tau, \phi)$
$Q = (\phi, \pi) \cup Q$
$\mathsf{Return}\ \pi$

NIP *is unique witness extractable if* $\forall\ \mathcal{A}\ \exists \chi_{\mathcal{A}}$ *s.t.* $\mathbf{Adv}^{uwe}_{\mathcal{BG},R,\mathcal{A},\chi_{\mathcal{A}}}(\lambda) \in negl(\lambda)$.

Theorem 4. *Assume the q-PDH, 2q-SDH and d-PKE assumptions hold for* $q \geq \max(2d-1, d+2)$*. [30] satisfies unique witness extractability.*

Single Witness Hardness The ability to resample witnesses for a provided statement-witness pair can also be advantageous to an adversary, since an 'easy' witness could be found by repeated sampling. We follow the definition of 2-hard instances in [13] and define *single witness hard* languages, for which it is hard to find a new witness given an existing one.

Definition 6. *Let* R_L *be a relation, and* $\mathcal{L} = \{\phi | \exists w \text{ s.t. } R_L(\phi, w) = 1\}$ *an NP language.* $\mathcal{L}$ *is a hard single-witness language if:*

1. ***Efficient Sampling:*** *There exists a PPT sampler* $\mathsf{S}(1^\lambda)$ *outputting a statement-witness pair* $\langle \mathsf{S}^x, \mathsf{S}^w \rangle$ *with* $\mathsf{S}^x \in \{0,1\}^\lambda$ *and* $(\mathsf{S}^x, \mathsf{S}^w) \in R_L$*.*
2. ***Witness Intractability:*** *For every PPT* $\mathcal{A}$ *there exists a negligible function* $\mu(\cdot)$ *such that:*

$$\Pr\left[\left(\mathsf{S}^x(1^\lambda), \mathcal{A}(\mathsf{S}(1^\lambda), 1^\lambda)\right) \in R_{\mathcal{L}}, \mathcal{A}(\mathsf{S}(1^\lambda), 1^\lambda) \neq \mathsf{S}^w(1^\lambda)\right] \leq \mu(\lambda).$$

A relation whose statements are outputs of a CRT hash function $\mathcal{H}$ defines a hard single-witness language. We show this for $\mathcal{L}(\mathcal{H}_P) = \{\phi : \exists w \text{ s.t. } \mathcal{H}^G_{P,|w|}(w) = \phi\}$ where $\mathcal{H}^G_{P,n}$ a weakly collision-resistant hash function.

We show that computing a [30] proof for the evaluation of $\mathsf{MaskedHash}$ (and our DPS predicate) will take on average a similar number of queries as a suitably parametrized MultiExp instance. We restrict to the case of outputs from a sampler S which samples a ρ randomly and generates valid witnesses. Since we are working in the GGM, the witness variables of the $\mathsf{MaskedHash}$ instance have an encoding that is indistinguishable from random. Therefore, the amortization bounds of Theorem 1 apply.

Theorem 5. *There exists a sampler* S *and QAP* R *evaluating* N *parallel instances of* k*-bit inputs of* MaskedHash *for which the [30] prover and the* $(4N(k+1), 8N(k+1)+2k)$*-length MultiExp problem of dimension* 10 *are equivalent up to constant terms with respect to multiplicative hardness.*

The vast majority of the constraints and variables in the predicate of the designed system are hash evaluations, so Theorem 5 can be used to show that there exists a proof system verifying state transitions for the DPS with bounded amortization-resistance guarantees. This is because the DPS predicate spends the vast majority of its time computing a proof whose hardness can be bounded by Theorem 5, since it is a sequence of iterated Pedersen hashes over a unique simulation extractable NIP.

Corollary 1. *The DPS in Section B.3 with block size* T*, state tree depth* d*, and index size* λ *admits a Proof of Necessary Work that is* ϵ*-amortization resistant w.r.t. a multiplication oracle and for which:*

$$\epsilon \lessapprox \frac{\log(q)}{\log(q) + \log(dT\lambda) + \log(\lambda)},$$

where q *is memory size measured in proof elements.*

4.5 Performance

We construct the DPS based on the above specifications and investigate its running time and memory consumption. Results are displayed in Table 1. Our benchmark machine was an Amazon Web Services (AWS) c5.24xlarge instance, with 96 vCPUs and 192GiB of RAM. The security properties of the DPS are based on the guarantee of Π-compliance provided by IVC. It is apparent that setup and proving times dominate both the running time and memory consumption in the protocol. Setup takes place once by a trusted third-party and hence is less critical for day-to-day system performance.

The prover is run by the miners, or full nodes. These generate PoW solutions repeatedly and would compute proof instances for many input nonces. Thus, larger storage requirements ($\sim$ 5.42GB key sizes) could be easily met by these nodes, as could the need for more parallelism and better computing power to bring down the proving rate.

We normalize the block time to achieve $\tau = 1/3$ in the sense of Theorem 2 for a proof including 30 transactions. This gives us that a miner will discard in expectation 15.59% of their work for an efficiency of $\sim 84\%$ if all miners operated based on the above benchmarks. Theorem 3 then gives an upper bound on the block orphan rate (or likelihood of block collisions) of 16.65%. Since we are keeping block times constant at 10 minutes, we note that any improvements in SNARK proof generation times will correspondingly decrease the amount of wasted work and orphan rate. Moreover, this does not depend on the way that the proofs are generated: distributed techniques among many participants (such as [37]) would also benefit efficiency through the corresponding decrease of average proof time.

Txs	**Constraints**	**Generator**	**Prover**	**Verifier**	**Size**		
#	#	Avg (s)	Avg (s)	Avg (ms)	pk (GB)	vk (kB)	π (B)
3	3658281	53.99	24.57	16.0	0.74	0.76	373
10	10071527	161.24	88.14		1.96		
20	19233307	268.93	185.10		3.74		
30	28395087	354.83	198.61		5.61		
40	37556867	485.52	286.50		7.15		
50	46718647	570.09	358.95		9.01		

Table 1. Prototype Times and Key Sizes for Predicates verifying different numbers of transactions: Average running times for setup $\mathcal{G}$, prover $\mathcal{P}$ and verifier $\mathcal{V}$ over 10 iterations are shown alongside proving/verification key and proof sizes.

5 Related Work

Several proposals have aimed to reduce verification costs for light clients; Chatzigiannis et al. provide a survey [10]. Most relevant to our work are Vault [26] and MimbleWimble [32] which speed up verifying transaction history and NIPoPoW [23] and FlyClient [9] which speed up verifying consensus. We summarize these results in Table 2. None of these proposals achieve constant-time verification, though they require significantly less work from provers.

Succinct blockchains, which provide optimal $O(1)$ bandwidth and computation costs to verify both history and consensus, were proposed in 2020, simultaneously by this work and the Mina project [5] (formerly Coda). Mina takes a similar high-level approach, encoding state transitions in a recursive proof system for asymptotically optimal verification time. The two proposals vary in a number of technical details, but the the main conceptual differences lie in our choice of consensus protocol. Mina implements proof-of-stake consensus, specifically a variant of Ouroboros[25] designed for succinct proofs, but does not incentivize efficient proof generation. By contrast, we implement a PoW variant specifically designed to incentivize proving efficiency.

Subsequent work has provided novel and efficient constructions for succinct blockchains, though not focused directly on prover incentivization. Chen et al. [11] propose a general framework for succinct blockchains over arbitrary transition functions, alongside benchmarks using the Marlin [12] proof system. Hegde et al. [19] tackle a related but critical problem: that of minimizing the total memory requirements of *full nodes*. Vesely et al. [36] propose Plumo, which leverages offline signature aggregation to design a cost and latency optimized light client for the Celo [36] blockchain. Abusalah et al. [1] propose SNACKS, a formal framework that adds knowledge extraction guarantees to Proofs of Sequential Work. We note that these contributions are orthogonal to our main focus of incentivizing efficient proving, and all could be incorporated in a practical PoNW implementation.

References

1. Abusalah, H., Fuchsbauer, G., Gaži, P., Klein, K.: SNACKs: Leveraging Proofs of Sequential Work for Blockchain Light Clients. Cryptology ePrint Archive, Paper 2022/240 (2022)
2. Announcing the ZPrize Competition. `https://www.aleo.org/post/announcing-the-zprize-competition` (2022), accessed: 2022-08-09
3. Ben-Sasson, E., Chiesa, A., Tromer, E., Virza, M.: Succinct non-interactive zero knowledge for a von Neumann architecture. In: USENIX Security (2014)
4. Ben-Sasson, E., Chiesa, A., Tromer, E., Virza, M.: Scalable zero knowledge via cycles of elliptic curves. Algorithmica **79**(4), 1102–1160 (2017)
5. Bonneau, J., Meckler, I., Rao, V., Shapiro, E.: Mina: Decentralized cryptocurrency at scale. `https://docs.minaprotocol.com/static/pdf/technicalWhitepaper.pdf` (2020), accessed: 2022-08-09
6. Bowe, S., Chiesa, A., Green, M., Miers, I., Mishra, P., Wu, H.: Zexe: Enabling decentralized private computation. Cryptology ePrint Archive, Report 2018/962 (2018)
7. Buterin, V.: Ethereum: A next-generation smart contract and decentralized application platform (2014), `https://github.com/ethereum/wiki/wiki/White-Paper`, accessed: 2016-08-22
8. Bünz, B., Agrawal, S., Zamani, M., Boneh, D.: Zether: Towards privacy in a smart contract world. Cryptology ePrint Archive, Report 2019/191 (2019)
9. Bünz, B., Kiffer, L., Luu, L., Zamani, M.: Flyclient: Super-Light Clients for Cryptocurrencies. Cryptology ePrint Archive, Report 2019/226 (2019)
10. Chatzigiannis, P., Baldimtsi, F., Chalkias, K.: SoK: Blockchain Light Clients. Cryptology ePrint Archive, Paper 2021/1657 (2021)
11. Chen, W., Chiesa, A., Dauterman, E., Ward, N.P.: Reducing participation costs via incremental verification for ledger systems. Cryptology ePrint Archive, Paper 2020/1522 (2020)
12. Chiesa, A., Hu, Y., Maller, M., Mishra, P., Vesely, P., Ward, N.: Marlin: Preprocessing zksnarks with universal and updatable srs. Cryptology ePrint Archive, Paper 2019/1047 (2019), `https://eprint.iacr.org/2019/1047`, `https://eprint.iacr.org/2019/1047`
13. Dahari, H., Lindell, Y.: Deterministic-prover zero-knowledge proofs. Cryptology ePrint Archive, Paper 2020/141 (2020)
14. Damgård, I.B., Pedersen, T.P., Pfitzmann, B.: On the existence of statistically hiding bit commitment schemes and fail-stop signatures. In: CRYPTO (1993)
15. Erdös, P.: Remarks on number theory III. On addition chains. Acta Arithmetica **6** (1960)
16. Fisch, B., Bonneau, J., Greco, N., Benet, J.: Scaling proof-of-replication for Filecoin mining. Tech. rep., Stanford University (2018)
17. Gordon, D.M.: A survey of fast exponentiation methods. Journal of Algorithms **27**(1) (1998)
18. Groth, J.: On the size of pairing-based non-interactive arguments. In: Eurocrypt (2016)
19. Hegde, P., Streit, R., Georghiades, Y., Ganesh, C., Vishwanath, S.: Achieving almost all blockchain functionalities with polylogarithmic storage. arXiv preprint arXiv:2207.05869 (2022)
20. Henry, R.: Pippenger's multiproduct and multiexponentiation algorithms. Tech. rep., University of Waterloo (2010)

21. Kamvar, S., Olszewski, M., Reinsberg, R.: Celo: A multi-asset cryptographic protocol for decentralized social payments. https://celo.org/papers/whitepaper (2019)
22. Kate, A., Zaverucha, G.M., Goldberg, I.: Constant-size commitments to polynomials and their applications. In: Asiacrypt (2010)
23. Kiayias, A., Lamprou, N., Stouka, A.P.: Proofs of proofs of work with sublinear complexity. In: Financial Crypto (2016)
24. Kiayias, A., Miller, A., Zindros, D.: Non-interactive proofs of proof-of-work. IACR Cryptology ePrint Archive **2017**, 963 (2017)
25. Kiayias, A., Russell, A., David, B., Oliynykov, R.: Ouroboros: A provably secure proof-of-stake blockchain protocol. In: CRYPTO (2017)
26. Leung, D., Suhl, A., Gilad, Y., Zeldovich, N.: Vault: Fast bootstrapping for the algorand cryptocurrency. NDSS (2018)
27. Maurer, U.: Abstract models of computation in cryptography. In: IMA International Conference on Cryptography and Coding (2005)
28. Miller, A., Kosba, A., Katz, J., Shi, E.: Nonoutsourceable scratch-off puzzles to discourage bitcoin mining coalitions. In: ACM CCS (2015)
29. Nakamoto, S.: Bitcoin: A peer-to-peer electronic cash system. http://bitcoin.org/bitcoin.pdf (2008)
30. Parno, B., Gentry, C., Howell, J., Raykova, M.: Pinocchio: Nearly Practical Verifiable Computation. Cryptology ePrint Archive, Report 2013/279 (2013)
31. Pippenger, N.: On the evaluation of powers and monomials. SIAM Journal on Computing **9**(2) (1980)
32. Poelstra, A.: Mimblewimble. https://download.wpsoftware.net/bitcoin/wizardry/mimblewimble.pdf (2016), accessed: 2022-08-09
33. Schnorr, C.P.: Efficient identification and signatures for smart cards. In: Eurocrypt (1989)
34. SCIPRLab: libsnark: a c++ library for zksnark proofs. https://github.com/scipr-lab/libsnark (2017)
35. Shoup, V.: Lower bounds for discrete logarithms and related problems. In: Eurocrypt (1997)
36. Vesely, P., Gurkan, K., Straka, M., Gabizon, A., Jovanovic, P., Konstantopoulos, G., Oines, A., Olszewski, M., Tromer, E.: Plumo: An ultralight blockchain client. Cryptology ePrint Archive, Paper 2021/1361 (2021)
37. Wu, H., Zheng, W., Chiesa, A., Popa, R.A., Stoica, I.: DIZK: A Distributed Zero Knowledge Proof System. In: USENIX Security (2018)
38. Zhandry, M.: To label, or not to label (in generic groups). Cryptology ePrint Archive, Paper 2022/226 (2022)

A Model Definitions

We model the processing of payments using a state machine. A state machine is defined by an initial state, a set of possible states, and a state transition function which governs the transition from one state to another given some information as input. Moreover, we work under the assumption that this is a *replicated state machine* (RSM), with local copies of the state machine in each node so as to achieve fault tolerance.

We define our payment system state machine as follows: we have a set of participants who share a broadcast communication channel, and who may join

or leave the system at will. There are two types of nodes we concern ourselves with here: miners and light clients.

Miners: A mining (or full) node has access to the current state $\mathcal{S}_i \in \mathcal{S}$ at timestep i, performing any consensus-specific computation and verifying state transitions.

Light Clients: Light clients (or end-users) can issue transactions $t \in \mathcal{T}$ and verify their inclusion, but do not need to keep mutable state.

We investigate how the system transitions from $\mathcal{S}_i$ to $\mathcal{S}_{i+1}$ while retaining consensus over state. Transitions between states happen through the processing of transactions by a model-specific transition function $\mathsf{NewState}$. We also require a transition validation function $\mathsf{VerifyState}$ that ensures the state update was done correctly. By defining the notion of *validity* between state transitions, we differentiate between legitimate and illegitimate transactions and only permit processing of the former. Moreover, we require that such tuples are also internally consistent, namely that all new states are correctly validated. For example, the Bitcoin and Ethereum protocols both define their own transition functions between blocks (states) and each one is based on its own notion of transaction validity.

Definition 7. *A tuple of efficiently computable algorithms* $(\mathsf{VerifyState}, \mathsf{NewState})$ *is considered a transition tuple if the following conditions hold:*

- $\mathsf{VerifyState} : 2^{\mathcal{T}} \times \mathcal{S} \times \mathcal{S} \times \{0,1\}^* \to \mathit{Yes/No}$
- $\mathsf{NewState} : 2^{\mathcal{T}} \times \mathcal{S} \times \{0,1\}^* \to \mathcal{S}$

and moreover we consider such a tuple consistent if $\forall \mathcal{S}_i, \mathcal{S}_{i+1} \in \mathcal{S}, \boldsymbol{t} \in 2^{\mathcal{T}}$ *:*

$$\exists z_i \ \textit{s.t.}\ \mathsf{VerifyState}(\boldsymbol{t}, \mathcal{S}_i, \mathcal{S}_{i+1}, z_i) = \mathit{Yes} \iff \mathsf{NewState}(\boldsymbol{t}, \mathcal{S}_i) = \mathcal{S}_{i+1}.$$

$\Sigma^n = (\mathcal{S}_i, \mathbf{t}_i, z_i)_{i=1}^n$ *is valid with respect to* $(\mathsf{VerifyState}, \mathsf{NewState})$ *if* $\forall i \in [n]$ $\mathsf{VerifyState}(\boldsymbol{t}_i, \mathcal{S}_i, \mathcal{S}_{i+1}, z_{i+1}) = \mathsf{Yes}$*, or equivalently* $\mathsf{NewState}(\boldsymbol{t}_i, \mathcal{S}_i) = \mathcal{S}_{i+1}$*.*

The above notion can be used to define the minimal semantics for a model DPS, where $2^{\mathcal{T}}$ refers to the power set of $\mathcal{T}$. Note that the above does not imply deterministic state transitions. In addition, we associate the monetary value $c \in \mathbb{N}$ of each account with user address values $z \in \mathcal{Z}$, of which there can be multiple in a given state. This provides us with all the ingredients needed to define the fundamental system.

We require a theoretical model for a distributed payment system (DPS), defined as a tuple of algorithms necessary for minimal payment functionality. Many subsequent and concurrent works have focused on developing various DPS architectures, each depending on a different set of trade-offs and desirable protocol properties. For us, the structure of the DPS is not the main goal, so we opt for working with a minimal design. We restrict ourselves to a simple construction

based on a standard approach,. In terms of security, the system needs to provide both completeness and correctness guarantees. This requires that the protocol should guarantee that state transitions considered correct by VerifyState will not be rejected by compliant nodes. Similarly, satisfying correctness requires that transactions and state transitions that are invalid should not be accepted by compliant nodes. These definitions are constructed in the usual way in the auxiliary supportive materials.

Our model can easily be adapted to describe existing blockchain-based payment systems. We illustrate this informally for Bitcoin (in its original form) to provide intuition for what the essential components of a distributed payment system are.

Bitcoin: The Bitcoin protocol is a UTXO-based payment clearing system, for which a valid block update includes a set of valid ordered transactions and specific block header information. The components of the RSM are illustrated below:

- **State:** The list of all UTXOs.
- **Witness:** Not required; validation happens by inspection of the ledger.
- **NewState:** Generation of a new block.
- **VerifyState:** Validity of a block transition requires:
  - Verifying all UTXOs exist in state.
  - Verifying that the header is well formed.
  - Checking the nonce satisfies PoW.
  - Ensuring all transactions are valid.

A similar treatment would allow us to characterize Ethereum using the same basic components. This paradigm also makes obvious that, in order to verify the state of the *whole* system without any external information, we would need to iteratively validate each state transition. We use the witness z_i to provide 'hints' to the validation function, which we will demonstrate later allows us to construct protocols tailored for much more efficient state verification.

B Succinct Verification

We are interested in working with RSMs that facilitate state verification, so we will also define the notion of "succinct verifiability". This restricts RSMs to be succinctly verifiable if the computational and memory resources they require to perform verification of the RSM's current state are small. Since in practice the size of the state of some RSM is extremely large (and grows with the number of processed transactions and blocks), any sufficiently efficient verification algorithm will need to take as input a "succinct representation" of the current state transition, while still being able to verify it. Otherwise, verification would require parsing $\mathcal{S}_i, \mathcal{S}_{i+1}$, which is prohibitively expensive. This is modelled as a function $\psi : 2^{\mathcal{T}} \times \mathcal{S} \times \mathcal{S} \to \mathcal{C}$, which provides a commitment $c \in \mathcal{C}$ that suffices for verification.

Definition 8. *An RSM Σ^n with n state transitions is a tuple $\Sigma^n = (\mathcal{S}_i, \mathbf{t}_i, z_i)_{i=1}^n$ of states $\mathcal{S}_i \in \mathcal{S}$, sets of transactions $\mathbf{t}_i \in 2^{\mathcal{T}}$ (where $\mathcal{T}$ is the set of possible transactions), and witnesses $z_i \in \{0,1\}^*$. We denote $\mathcal{S}_n$ as the current state of Σ^n and $\mathcal{S}_0$ as its genesis state. Moreover, a valid RSM $\Sigma^n = (\mathcal{S}_i, \mathbf{t}_i, z_i)_{i=1}^n$ with respect to a consistent transition tuple ($\mathsf{VerifyState}$, $\mathsf{NewState}$) is considered succinctly verifiable if there exist $\psi : 2^{\mathcal{T}} \times \mathcal{S} \times \mathcal{S} \to \mathcal{C}$ and $\mathsf{SuccinctVerify} : \mathcal{C} \times \{0,1\}^* \to Yes/No$ such that $\mathsf{SuccinctVerify}$ has $O(1)$ time and size complexity over $n, |\mathcal{S}_i|, |z_i|$ and:*

$$\Pr\left(\mathsf{SuccinctVerify}(\psi(\mathbf{t}, \mathcal{S}_i, \mathcal{S}_{i+1}), z_i) \neq \mathsf{VerifyState}(\mathbf{t}, \mathcal{S}_i, \mathcal{S}_{i+1}, z_i)\right) \approx 0,$$

over the random coins of $\mathsf{SuccinctVerify}$ and ψ.

Here we demonstrate a specific instantiation of a DPS for which we define a transition function tailored to fast state verification by stateless clients. To achieve this, we leverage the capabilities of IVC systems and construct a succinct proof of state validity to represent each state transition. Since we will be basing our implementation of the proofs on SNARKs, we design the transition function so as to minimize SNARK proof sizes. This is critical for efficiency and feasibility.

Following the longest chain quality update rule, our system updates the quality q of solving a PoW puzzle according to the depth of the chain. We are thus required to include (and commit to) q_i and n_i with every proof, where q_i is the quality of state $\mathcal{S}_i$ and n_i the associated nonce. This is because these quantities are needed by miners in order to follow the longest chain and achieve consensus.

Each participant in our system has a public and secret key that they generate when they first join the network. The participants use these keys to digitally sign transactions and verify other participants' signatures. The state $\mathcal{S}_i$ contains the distribution of money between the participants (stored as a tree), state quality and a nonce corresponding to the most recent PoW. We also distinguish between the i-th block, which in our case will be represented by a proof π_i that the i-th state transition is valid along with the set of transactions $\mathbf{t}_i$ corresponding to the transition, and *commitments* to state, which we denote by B_i and use for client verification. We require an account-based system (like Ethereum but not Bitcoin) and keep track of state with an 'Account Tree' of all account-value pairs. These building blocks are:

Account Tree: We use a Merkle tree construction with a compressible Collision Resistant (CRT) hash function $\mathcal{H} : \{0,1\}^{2\lambda} \to \{0,1\}^{\lambda}$. We assume a fixed size tree T with height h throughout.

State: We denote S_i the state after the i-th update:

- Account tree T^i with leaves the lexicographically-ordered (by address) accounts in state.
- The block number i, quality q_i, and nonce n_i.

State Commitment: Set B_i as the commitment to S_i:

- The root rt_i of the Account tree T^i in $\mathcal{S}_i$.

– The block number i, quality q_i, and nonce n_i.

Protocol Initialization: Initially all accounts in the Account tree are set to null. In every transition, the tree allows the following modifications:

– *Account Initialization:* Set the public key to a non-null value and initialize the balance and the nonce. An account with a non-null public key is considered initialized. An account can be initialized only once. Uninitialized accounts have null public key.
– *Balance Update:* Modify account balance *bal*, ensuring money conservation.
– *Nonce Update:* Modify account nonce n to that of the current block.

We denote the initial state of the system (or "genesis state") by $\mathcal{S}_0$; this is agreed to by an out-of-band process. For example, a system might start with all addresses having a balance of zero or it might pre-populate some accounts with non-zero balance (colloquially known as "pre-mining"). Note that in the initial state, the Account tree is a full tree and contains one leaf/account for every address that can exist in the state. The genesis state can contain initialized and uninitialized accounts. All preliminary data structures have been included in the auxiliary supportive material.

B.1 State Transition Semantics

Below we define our semantics used for transaction and state transition validity.
Verifying Transactions: $\textsc{VerifyTx}(t, T^i) \rightarrow$Yes/No takes as input a transaction t and an Account tree T^i, outputting Yes/No (1 or 0). A transaction is considered valid if:

1. Sender and receiver are legitimate accounts in T^i.
2. Amount transferred is not more than sender's balance.
3. Signature authenticates over the sender's public key.
4. Sender and receiver accounts in the Account tree are updated correctly.
5. Recipient and Account public keys match, or the address is uninitialized.

Updating System State: $\textsc{UpdateState}(\mathcal{S}_i, \mathbf{t}, n) \rightarrow \mathcal{S}_{i+1}$ is a procedure that takes as input a state $\mathcal{S}_i$, am ordered set of transactions $\mathbf{t}$ with $|\mathbf{t}| = N$ and a nonce n. It outputs the next state $\mathcal{S}_{i+1}$ and a witness w of objects proving the update was done correctly. A transition is valid if:

1. All transactions in $\mathcal{T}$ are valid.
2. The previous state has performed PoW.
3. Only last transaction t_N is of coinbase type.
4. Each transaction builds on top of the previous one; the first builds on the previous root.

B.2 State Transition as an NP statement

In order to instantiate a DPS that is capable of verifying a given state transition function, we encode the transition function ValidState as a compliance predicate Π_S. With every state transition, we include a proof that the transition was Π_S compliant. This is done by verifying the transition from the previous state and producing an attesting witness w in the process. In this context, we are interested in verifying the transition between two states of the Account tree by processing transactions between them into the system. This is achieved by tracking changes to the root rt_i of the Account tree after the input of each transaction.

We capture all requirements for transaction, PoW and state validity in an NP language that onl y accepts commitments of the form $B_i = (rt^i, i, q_i, n_i)$ that build 'correctly' on top of a previous state. At a high level, the elements of this language are state commitments that, given some previous state's root, have only processed valid transactions.

Compliance Predicate Given input $B_{i+1} = (rt_{i+1}, i+1, q_{i+1}, n_{i+1})$, the compliance predicate $\Pi_{\mathcal{S}}$ evaluates to 1 if and only if all of the following are satisfied:

1. Exists state $\mathcal{S}_i$ satisfying PoW with nonce n_i and quality q_i.
2. Exists a tuple of ordered transactions $\mathbf{t}$ with $|\mathbf{t}| = N$. These transactions need to be sequentially valid with respect to $\mathcal{S}_i$.
3. $\textsc{UpdateState}(\mathcal{S}_i, \mathcal{T}, n_i) = \mathcal{S}_{i+1}$.

We use the compliance predicate $\Pi_{\mathcal{S}}$ to design an IVC system consisting of algorithms $(\mathcal{G}, \mathcal{P}, \mathcal{V})$, where each message z_i is commitment B_i.

B.3 DPS Specification

Here we define how the system transitions from $\mathcal{S}_i \to \mathcal{S}_{i+1}$. Algorithm 1 generates a new state and associated proof of compliance, along with a nonce certifying that the system performed PoW. When validating, we check that the new state $\mathcal{S}_{i+1}$ is a valid next state for the system by being (a) $\Pi_{\mathcal{S}}$ compliant and (b) providing PoW. Note that the validation only requires the *root* of the Account tree corresponding to $\mathcal{S}_i$, thus making it efficient enough for light clients. A detailed specification alongside security definitions and proofs can be found in the attached auxiliary supportive material.

When updating the state of the system, each participating miner receives π_{i+1} and $\mathbf{t}_{i+1}$. This allows them to update their own state to $\mathcal{S}_{i+1}$ and begin mining again. In Table 2, we provide a comparison of the asymptotic time and memory requirements of existing SPV protocols implementing transaction and/or PoW verification. Given that transaction volume and chain length both grow linearly over time, we can ideally provide verification that is constant with respect to both.

Algorithm 2 NewState

Require: $pp, \mathcal{T}, \mathcal{S}_i, \pi_i$
Ensure: $\mathcal{S}_{i+1}, \pi_{i+1}$
procedure NewState($pp, \mathcal{T}, \mathcal{S}_i, \pi_i$)
  if $\mathcal{V}(vk, \mathcal{S}_i, \pi_i) = 0$ **then return** 0
  end if
  while $\mathcal{H}(\pi_{i+1}) > d$ **do**
    Pick n_{i+1} uniformly at random
    $(\mathcal{S}_{i+1}, w) \leftarrow$ UpdateState($\mathcal{S}_i, \mathcal{T}, n_{i+1}$)
    $\pi_{i+1} \leftarrow \mathcal{P}(pk, \mathcal{S}_{i+1}, \mathcal{T}, \mathcal{S}_i, \pi_i, w)$
  end while
  return $(\mathcal{S}_{i+1}, \pi_{i+1})$
end procedure

C Proof of Theorems 2 and 3

Proof. Assume that a blocks are found in a Poisson process with a mean of $\lambda = 1$ and an individual miner can check one puzzle solution in time τ. Consider the expected number of blocks this individual miner is able to check before the network broadcasts a solution. A block will be found by the network in less than time τ with probability:

$$\int_0^\tau e^{-x} dx = 1 - e^\tau.$$

In this case, the miner will not even finish checking a single block. If the network does not broadcast a block within time τ, the miner will check at least one block. The Poisson process then repeats, since it is memoryless. So the expected number of blocks checked is:

$$\begin{aligned} \mathrm{E}_{blocks} &= (1 - e^\tau) \cdot 0 + e^{-\tau} \cdot (1 + \mathrm{E}_{blocks}) \\ e^\tau \cdot \mathrm{E}_{blocks} &= 1 + \mathrm{E}_{blocks}\mathrm{E}_{blocks} \qquad\qquad = \frac{1}{e^\tau - 1}. \end{aligned}$$

If no partially-checked solutions were wasted, the miner would always expect to check $\frac{1}{\tau}$ solutions. Thus, the fraction of wasted work is:

$$1 - \frac{\frac{1}{e^\tau - 1}}{\frac{1}{\tau}} = 1 - \frac{\tau}{e^\tau - 1}.$$

Proof. Since solutions are Poisson random variables:

$$\Pr[\mathsf{collision}] = [1 - \mathsf{Po}(1, \tau)/(1 - \mathsf{Po}(0, \tau))] \leq \tau/2.$$

D Proofs of Amortization Resistance

We borrow notation from [31] and parametrize with q input indices, p outputs and maximum index size 2^λ. Where not specified, $H = pq\lambda$. Let $L(\mathbf{y})$ be the minimum number of multiplications to compute $\mathbf{y} = (y_1, ..., y_p)$ with $y_i \in [2^\lambda]^q$ and $[2^\lambda] = \{1, ..., 2^\lambda - 1\}$ from the inputs and unit vectors and $L(p, q, 2^\lambda)$ be the maximum over all of them.

Lemma 1. *For any value of $c \leq L(p, q, N)$, there are at most:*

$$\left(\frac{H^2}{c}\right)^c 2^{q+1} e^c (q+1) 2^{O(1)},$$

addition chains of length at most c.

Lemma 2. *Define $H := \kappa q \nu \lambda$, $\phi(q, \kappa, \nu, \lambda) :=$*

$$q\kappa\nu \log(q\kappa\nu) + \kappa \log(H) + q + \log(q+1) + 1,$$

and fix $\mu := \delta H$, corresponding to:

$$c_\delta := \frac{(1-\delta)H - \phi(q, \kappa, \nu, \lambda)}{\log(H) - \log(e) + \log(\mu) + \log(1/\delta)}.$$

For the (κ, κ)-length MultiExp function of dimension ν for collision resistant g:

$$\Pr_{\mathbf{x} \in_R [2^\lambda]^{\kappa \times q}, \mathsf{G} \in_R \mathbb{G}^{\kappa \times \nu}} [L(f(x_1), ..., f(x_q)) \leq c_\delta] \leq \left(\frac{1}{2}\right)^\mu.$$

Proof. Write $G_k^{(j)} = r_{jk} G$. As the $x_i \in [2^\lambda]^\kappa$ and $r_{jk} \in [2^\lambda]$ are sampled randomly, the values $x_{ik} G_k^{(j)} = x_{ik} r_{jk} G$ for $i \in [q], j \in [\nu - 1], k \in [\kappa]$ will be distinct w.h.p. The $\kappa \cdot q$ values $g(x_i)_k \cdot r_{\nu k} G$ will also be distinct w.h.p. as g is collision resistant in each of its κ output coordinates.

Let M be the $q \times (\kappa\nu)$ sized matrix with these values as entries. As each entry is an element in $[2^\lambda]$, the number of matrices M with $q\kappa\nu$ distinct elements is:

$$\binom{2^\lambda}{q\kappa\nu} \geq \frac{2^{\lambda q \kappa \nu}}{(q\kappa\nu)^{q\kappa\nu}},$$

and to each M there corresponds a unique matrix $F = (f(x_1), ..., f(x_q))$ with dimension $q \times \nu$, where the κ products over random bases for each x_i have been computed. Note that $L(F) = L(M) + \kappa - 1$.

We can thus upper bound the minimal addition chain size $L(F)$ using $L(M)$ and the number of matrices M:

$$\Pr_{\mathbf{x} \in_R [2^\lambda]^{\kappa \times q}, \mathsf{G} \in_R \mathbb{G}^{\kappa \times \nu}} [L(F) \leq c] \leq \frac{|\{\mathbf{z} : L(\mathbf{z}) \leq c\}|}{2^{H - q\kappa\nu \log(q\kappa\nu)}}.$$

The numerator is upper bounded by Lemma 1 and the fact that a single chain corresponds to at most H^κ matrices, giving:

$$\Pr_{\mathbf{x}\in_R[2^\lambda]^{\kappa\times q},\mathsf{G}\in_R\mathbb{G}^{\kappa\times\nu}}[L(F)\leq c]\leq\left(\frac{1}{2}\right)^{H-\psi(c)},$$

where $\psi(c):=c(2\log H+\log e)+\phi(q,\kappa,\nu,\lambda)-c\log(c)$.

Suffices to show that for $c\leq c_\delta$, $\psi(c)\leq(1-\delta)H$. Since $\psi(c)$ is increasing for $c\leq L(\kappa,\nu q,2^\lambda)$, required to show that $\rho\geq c_\delta$ for $\psi(\rho)=(1-\delta)H$:

$$\rho(2\log H+\log e)+\phi(q,\kappa,\nu,\lambda)\geq(1-\delta)\cdot H,$$

$$\log\rho\geq\log\left((1-\delta)\cdot H-\phi(q,\kappa,\nu,\lambda)\right)-\log\left(2\log H-\log(e)\right),$$

$$\therefore\rho\geq\frac{(1-\delta)H-\phi(q,\kappa,\nu,\lambda)}{\log H-\log(e)+\log(\mu)+\log(1/\delta)},$$

since $\mu=\delta H$.

Corollary 2. *Fix $\delta>0$ and let $\psi(\rho_\delta)-(1-\delta)\cdot H=0$.*

$$\mathbb{E}[L(f(x_1),...,f(x_q))]\geq\rho_\delta\cdot(1-2^{-\delta H}).$$

Proof. By Markov's inequality:

$$\Pr[L(f(x_1),...,f(x_q))\geq\rho_\delta]\cdot\rho_\delta\leq\mathbb{E}[L(\mathbf{x})],$$

$$(1-\Pr[L(f(x_1),...,f(x_q))<\rho_\delta])\cdot\rho_\delta\leq\mathbb{E}[L(f(x_1),...,f(x_q))].$$

Proof (Proof of Theorem 1). Required to compute q iterations of the MultiExp function. Each iteration includes ν multiproducts over random bases, with the indices also sampled from $[2^\lambda]$.

Using c oracle queries to do this corresponds to knowledge of an addition chain of length c containing all of $F=(f(x_1),...,f(x_q))$ with $x_i\in[2^\lambda]^\kappa$. Therefore, the probability that we compute F for $\mathbf{x}\in_R[2^\lambda]^{\kappa\times q}$ with less than c queries is upper bounded by the probability that $L(F)\leq c$.

Fix $\delta>0$. Lemma 2 states that $\exists c_\delta$ s.t. this probability is negligible in $\mu:=\delta\kappa\nu q\lambda$ for $c\leq c_\delta$. One function computation of dimension ν with κ inputs has an upper bound on the expected number of multiplications of:

$$\min(\kappa,\nu)\cdot\lambda+\frac{\kappa\nu\lambda}{\log(\kappa\nu\lambda)}\cdot(1+o(1)).$$

Corollary 2 implies that:

$$\epsilon\leq1-q^{-1}\cdot\left(\min(\kappa,\nu)\cdot\lambda+\frac{\kappa\nu\lambda}{\log(\kappa\nu\lambda)}\cdot(1+o(1))\right)^{-1}\cdot(1-2^{-\delta\kappa\nu q\lambda})\cdot c_\delta$$

$$\leq\frac{\log(q)+\delta\log(\kappa\nu\lambda)+o(1)}{\log(\kappa\nu q\lambda)}\leq\frac{\log(q)+o(1)}{\log(\kappa\nu q\lambda)},$$

where we have taken $\delta\leq1/\log(\kappa\nu\lambda)$.

Proof (Proof of Theorem 4). We know that the NIP has a PKE extractor from its security proof and so $\mathcal{A}$ can extract two witnesses almost surely using extractor $\chi_{\mathcal{A}}^{PKE}$. If the polynomials are distinct, so are their witnesses. This follows directly from the fact that, since $\pi_1 \neq \pi_2$, either (1) one of $u_i(X), v_i(X), w_i(X)$ differs in one of the proofs, or (2) the extracted witnesses differ. Since the predicate is the same, it follows that the witnesses must differ.

Lemma 3. *Let $\mathcal{H}_P = \{\mathcal{H}_{P,\lambda}^G\}_{\lambda \in \mathbb{N}_+}$ be a family of efficiently computable functions for which each $\mathcal{H}_{P,\lambda}^G : \{0,1\}^\lambda \to \mathbb{G}$ is weakly collision-resistant. $\mathcal{L}(\mathcal{H}_P)$ is hard single-witness.*

Proof (Proof of Lemma 3). Define S in the natural way: fix $\lambda \in \mathbb{N}_+$ and define S to randomly sample an element $x \in \{0,1\}^\lambda$, outputting $(\mathcal{H}_{P,\lambda}(x), x)$. The sampler is efficient by the efficiency of $\mathcal{H}_{P,\lambda}(x)$, and $(\mathcal{H}_{P,\lambda}(x), x) \in R_{\mathcal{L}(\mathcal{H}_{P,\lambda})}\}$ by definition. Witness intractability (WI) follows from the collision resistance of $\mathcal{H}_{P,\lambda}$ on constant-size inputs. If some $\mathcal{A}$ exists that violates WI, then running S on 1^λ and then $\mathcal{A}$ on $\mathsf{S}(1^\lambda)$ and 1^λ, we non-negligibly find a collision in $\mathcal{H}_{P,\lambda}^G$.

Proof (Proof of Theorem 5). The MaskedHash QAP has $4N(k+1)$ intermediate witness variables (and $8N(k+1)+2k$ constraints) which admits witnesses from a sampler S where the seed ρ is uniformly random and so all witness variables (with full support) also look random by the randomness of the group encoding. This is as the intermediate values are distinct powers of a group element that is random due to ρ and the independence of the I_j index elements. By unique witness extractability and single witness hardness of CRT functions, all valid witnesses have a unique encoding and hence a unique witness polynomial h.

We start with ℓ instances of N k-bit hash evaluations from S, and require ℓ valid proofs. We reduce to the $4N(k+1)$-length MultiExp problem for ℓ instances and g equal to the function evaluating the representation of h given the witness elements. We provide ℓ of the $4N(k+1)$ intermediate witness variables and the corresponding 9 sets of bases to the MultiExp function. The representation of h will be unique w.r.t. the witness (since the instance is single witness hard) and thus look random due to the inputs. Note that $\mu = 8N(k+1)+2k$. We finally perform a linear in ℓ number of multiplications to add any witness variables that were not included (i.e. not randomly distributed). Since the MultiExp index distributions are also random, a proof verifies iff the MultiExp solution is valid.

Conversely, given ℓ $(4N(k+1), 8N(k+1)+2k)$-length MultiExp instances of dimension 10 with inputs and bases sampled from the QAP's sampler and proving key respectively, we reduce to computing ℓ proofs for N k-bit hash evaluations. This is because the unassigned witness variables can be discerned from the auxiliary input to g, which comes from the QAP sampler. By the uniqueness of the proof's encoding, the set of ℓ valid proofs will have to equal the MultiExp instances after a linear in ℓ number of operations to 'undo' products by any of the additional variables.

E Comparison to light client solutions

Table 2 provides a comparison between the asymptotic efficiency of this work (succinct blockchains) and lightweight proposals.

Technique	**Tx Verif.**	**PoW Verif.**	**Memory**
Bitcoin [29]/Ethereum [7]	$\Theta(t)$	$\Theta(h)$	$\Theta(h+t)$
Mimblewimble [32]	$\Theta(u) = O(t)$	$\Theta(h)$	$O(\log^4(h))$
NIPoPoW [24]	$\Theta(t)$	$\text{polylog}\,(h)$	$\log h \cdot (\log t + \log\log h)$
FlyClient [9]	$\Theta(t)$	$O(\log^2 h)$	$O(\log^2 h)$
Succinct Blockchains	$O(1)$	$O(1)$	$O(1)$

Table 2. Previous Work on Light-Client Verification: Asymptotic state and PoW verification times for clients verifying t transactions in h Blocks