# Proofs of sequential work with unique proofs

Aaron Bi Zhang
zhangaaronb@gmail.com

Advisor: Anna Lysyanskaya
Reader: Maurice Herlihy

May 2019

# 1 Introduction

## 1.1 Proofs of work and their alternatives

Proofs of work (PoW) are used to secure blockchain applications such as Bitcoin. In a typical PoW protocol, a prover receives a statement $\chi$ and a time parameter $N$, performs about $N$ steps of computation related to the statement $\chi$, and then must be able to convince a verifier that they have indeed performed about $N$ steps of computation since receiving $\chi$. The verifier should be able to check the validity of a proof while performing much less computation than the prover, typically logarithmic in $N$.

A major drawback of PoW is that they can be wasteful, because usually the work of computing a proof can be parallelized. For example, parties who mine Bitcoin by computing PoW often devote large clusters of machines to do nothing but compute these proofs. Applications dependent on PoW can incur large costs in computing resources and energy, and it is questionable whether the proofs that are generated come with any greater value to society.

These drawbacks have prompted research into alternatives such as *proofs of sequential work* (PoSW), a protocol that is similar to PoW, but requires computation that is inherently sequential. Even if many parties attempt to compute these proofs, none of them would significantly benefit from spending excessive amounts of resources and energy on parallelism.

## 1.2 Open questions

Mahmoody, Moran, and Vadhan introduce the concept of PoSW in their 2013 paper [MMV13]. They construct PoSW assuming black-box access to a cryptographic hash function modeled as a random oracle. Cohen and Pietrzak, in their 2018 paper [CP18], simplify and improve the MMV construction. We summarize these black-box constructions of PoSW in sections 4 and 5.

The main open question raised by these two results is whether it is possible to construct a PoSW with *unique* proofs. Essentially, this means that even a cheating prover cannot come up with two different values that the verifier would accept as a valid proof. The MMV and CP constructions do not have this property: it is easy to come up with many different proofs that will allow the prover to convince the verifier with high probability. There are known non-black-box constructions of unique PoSW that are described in section 2, so this paper focuses on the black-box approach. Another reason for focusing on black-box constructions is that unique PoSW can be used in cryptographic applications besides blockchains, such as fair coin tossing. Studying black-box constructions can help us better understand the question of which other cryptographic primitives are possible from only black-box sequentiality assumptions. Section 2 describes some applications of unique PoSW.

## 1.3 Contributions

In this paper, we review existing constructions of PoSW and note how known black-box constructions fail to achieve uniqueness. We do not manage to come up with a black-box construction of unique PoSW, but we do make progress on the easier problem of a black-box construction of unique PoW (as opposed to PoSW). In our construction of unique PoW, the runtime of the verifier can be much less than that of the prover, although we do not achieve a verifier runtime logarithmic in that of the prover. Our unique PoW construction might be able to substitute for unique PoSW in certain cryptographic applications, but would probably not work as well as a unique PoSW.

In section 2, we summarize some of the applications of unique PoSW and known non-black-box approaches to constructing them. In section 3, we formally define PoW and PoSW. Sections 4 and 5 describe the known black-box constructions of PoSW, namely the MMV and CP constructions. Section 6 is the original contribution of this paper, a black-box construction of unique PoW.

# 2 Related work

## 2.1 Applications of PoSW

PoSW offer an alternative to blockchains based on wasteful PoW. Chia Network [Chi] is a cryptocurrency that uses a blockchain secured by PoSW together with another type of alternative to PoW called proofs of space. Proofs of space, which require the prover to dedicate space instead of time, first appear in [DFKP13]. Although parties can buy more physical memory to attempt to mine more proofs of space, currently the cost of memory does not make this profitable. On the other hand, everyday computer users often have a lot of unused disk space that can be allocated to these proofs without significant cost.

PoSW with unique proofs are also known as verifiable delay functions (VDF). This is because there is only one valid proof that the prover can compute for a given statement, so the prover is essentially computing a function $F$ of the statement. The PoSW in Chia Network is a VDF, and the uniqueness of proofs can be desirable in blockchains and other cryptographic applications. In a blockchain, a party can add a block by computing a proof based on a statement derived from the previous block. Using a PoSW that is not a VDF could make a blockchain susceptible to a grinding attack: a party could compute many different proofs and decide which one would be most advantageous for it. For example, one of the proofs might cause the next statement in the blockchain to be a statement that the party has a better chance of solving due to precomputation that the party has performed.

VDFs are also used in contexts other than blockchains, and often in these contexts, the uniqueness of proofs becomes essential. [MMV13] provide a simple example of applying VDFs to fair coin flipping. Suppose Alice and Bob want to agree on a coin flip result. One possible solution is the following. Alice starts by sending Bob a statement $\chi$ for a VDF. Instead of solving the VDF, Bob must quickly respond with a random bit-vector $b_2$. Alice then solves the VDF (or can have already computed the proof offline) to obtain a bit-vector $b_1$, and Bob can perform the verification protocol of the VDF to check that $b_1$ is indeed a valid proof for statement $\chi$. The result of the coin flip is the dot product $b_1 \cdot b_2$. Because Bob must respond quickly with $b_2$, Bob does not have time to compute the proof and bias their random bit-vector. However, if we used a PoSW with non-unique proofs instead of the VDF, Alice can compute several different values of $b_1$ that Bob would likely accept as valid solutions to the challenge $\chi$, and upon seeing $b_2$, choose a solution that yields the desired coin flip result $b_1 \cdot b_2$.

VDFs have also been used in the more general context of randomness beacons [Rab83], which are sources of public randomness that no party can predict or manipulate. [BBBF18] describes the following natural attempt to construct a randomness beacon and why it is flawed: if there are $n$ parties, party $i$ uses a commitment scheme to commit to a random string $b_i$. After all parties have committed, they all open their commitments, and the random

string is taken to be the bitwise XOR of all the $b_i$. If there is even one honest party, then the result is truly random. The problem is that if a party is dishonest, it can wait for the other parties to open their commitments first, and then decide not to open its own commitment if the resulting random string would not be favorable for it. [LW15] construct a randomness beacon using VDFs. Any number of parties can participate in generating the randomness, and a party can do by publishing its random string $b_i$ in the clear. The output of the randomness beacon is not the bitwise XOR of the $b_i$, but rather the solution to a VDF where the statement is the concatenation of the $b_i$. The randomness beacon will allow any party to publish its string $b_i$ for a certain window of time, say 10 minutes. If the VDF takes about 20 minutes to solve, then no party will be able to predict the solution to the VDF and manipulate the result.

[BBBF18] surveys further applications of VDFs. [Vdf] describes a research effort to improve the implementation of VDFs to make them more usable in real systems including blockchains, randomness beacons, and other applications.

## 2.2 Constructing VDFs from hardness assumptions

This paper focuses on black-box constructions of PoW and PoSW, but there are two alternative approaches that have been successful. The first of these approaches uses number-theoretic hardness assumptions. [BBBF18] constructs a VDF using a family of rational functions over vector spaces that is believed to require sequential work to invert. Thus, the prover must find a preimage under one of these rational functions of the statement, while the verifier can easily check the correctness of the preimage by computing the rational function. Inverting these rational functions can be done by performing a number of greatest common divisor computations sequentially, and it is believed that excessive amounts of parallelism do not allow this computation to be done faster than the honest prover's runtime. This construction achieves an exponential gap between the runtime of the verifier and the runtime of the prover, but the drawback is that even the honest prover needs $O(N)$ parallelism (where $N$ is the time parameter) in order to finish the computation in $N$ time.

The two later constructions [Pie18] and [Wes18] eliminate this drawback and still achieve an exponential gap between the verifier's runtime and the prover's runtime. These two constructions are based on the assumption that repeated squaring is inherently sequential. That is, we suppose the prover and verifier have a group $G$ of unknown order. The class group of an imaginary quadratic number field has been proposed as one way to choose a group of unknown order. Given a statement $\chi$ interpreted as an element of $G$ and a time parameter $N$, the prover's task is to compute $\chi^{(2^N)}$. This can be done by repeatedly squaring $N$ times, namely by computing $\chi, \chi^2, \chi^4, \chi^8, \ldots$. The assumption that repeated squaring is inherently sequential is that there is no significantly faster way to compute $\chi^{(2^N)}$ without knowing the order of $G$, even with parallelism (although some parallelism may help speed

up the arithmetic in $G$).

The difficulty in using this repeated squaring protocol is that the verifier needs to be able to efficiently check that the prover's output is indeed $\chi^{(2^N)}$. If the verifier knew the order of $G$, then it could compute $\chi^{(2^N) \mod |G|}$, which equals $\chi^{(2^N)}$ due to Euler's theorem. However, if the verifier knows the order of $G$ while the prover doesn't, the verifier must have some secret information, and as a result the proof of sequential work will no longer be publicly verifiable. For example, one approach would be for the verifier to generate $G$ as an RSA group where the verifier knows the factorization of the RSA modulus. The verifier only sends the RSA modulus to the prover, so the prover doesn't know the order of $G$. The problem is that, even if the prover honestly computes and outputs $\chi^{(2^N)}$ in the resulting group $G$, an outside observer would not be convinced that the prover has done sequential work. After all, the verifier and prover may have colluded, so that the verifier secretly sent the prover the factorization of the RSA modulus.

[Pie18] and [Wes18] show how the prover can convince the verifier that it has correctly computed $\chi^{(2^N)}$ even when neither the prover nor the verifier knows the order of $G$. These two constructions require additional security assumptions besides the sequentiality of repeated squaring to prove soundness. [Pie18] uses the low-order assumption, which roughly says that, given (the description of) a group $G$, it is difficult to find a low-order element of $G$. [Wes18] uses the adaptive root assumption, which says that it is difficult for an adversary to win the following game: the adversary outputs an element $w$ of $G$ of its choice, and then the challenger chooses a random number $l$ from a large set. The adversary must output an $l$th root of $w$. The [Pie18] and [Wes18] constructions would be insecure if the low-order assumption or adaptive root assumption failed respectively.

[BBF18] surveys these two constructions in detail. The survey also compares the parameters of the two constructions and concludes that each has certain advantages over the other.

## 2.3 Constructing VDFs from succinct arguments

The other general approach to constructing VDFs, besides black-box assumptions and hardness assumptions, is succinct arguments. At a high level, succinct arguments answer the question: how can the correctness of a computation be checked faster than redoing the entire computation? This seems like exactly the question we want to answer when constructing a VDF, since the verifier wants to check that the prover has computed the solution to the VDF correctly, but using much less time than the prover. However, one problem with using certain constructions of succinct arguments in VDFs is that the succinct arguments might be efficient asymptotically but impractical in real systems.

The time parameter $N$ in a VDF can be thought of as being bounded by a polynomial in the size of the statement $\chi$, although this polynomial can be large. We can formulate

the problem of checking the correctness of the prover's output $y$ as membership in an NP language $L$, where $(\chi, y) \in L$ if $y$ is the correct solution to the VDF with statement $\chi$. We take the NP witness to be the computation history of the prover (this isn't strictly necessary; the witness can also be empty because the verifier could simulate the computation history in polynomial time). The PCP theorem shows that membership in $L$ can be checked without the verifier having to read the whole witness. Rather, the witness can be transformed into a format such that the verifier can check membership in $L$ by reading a few randomly chosen bits of the transformed witness.

[BFLS91] shows how ideas similar to the PCP theorem can be used so that the verifier can check the correctness of a computation of length $N$ in time $O(\text{polylog}(N))$. The prover can transform the computation history into a format suitable for efficient verification in time $O(\text{poly}(N))$. However, it would be impractical to use this scheme in a VDF. Not only is a $O(\text{poly}(N))$ runtime for the prover too large because $N$ itself is very large, but also the hidden constants in the big-O are very large. [Mic00] is another PCP-type construction that offers good asymptotic guarantees but is not so feasible in practice. [BSCGT13] describes attempts to make PCPs more feasible by reducing their overhead. However, even the PCP constructions described in that paper would probably add too much overhead for VDFs.

SNARGs (succinct non-interactive arguments) are a more recent line of research introduced in [BCC$^+$17]. Like the results described above, a SNARG allows the prover to convince the verifier of the correctness of a computation by sending a succinct proof. This proof is much shorter and can be checked much more easily than the entire computation history. SNARGs and their variants are actually used in practice, including in the cryptocurrency Zcash [Zca]. However, directly using a SNARG for a VDF would likely be too expensive, because [WSH$^+$14] shows that in practice computing a SNARG proof can be thousands of times more expensive than the underlying computation.

However, as [BBBF18] notes, SNARGs can be used to implement another approach to succinct arguments called incrementally verifiable computation (IVC). IVC is introduced in [Val08]. In the setting of IVC, the prover's task is to compute $f^N(\chi)$ given input $\chi$, where $f$ is some function and we define $f^{i+1}(\chi) = f(f^i(\chi))$ and $f^0(\chi) = \chi$. For example, if $f$ is a cryptographic hash function, then it is unlikely that the prover would be able to compute $f^N(\chi)$ without actually iterating the hash function $N$ times, which requires performing $N$ sequential work. IVC allows the prover to prove the correctness of its computation incrementally. Namely, at step $i$ of the computation, the prover outputs $f^i(\chi)$, as well as a proof $\pi_i$ that $f^i(\chi)$ is correct. In a VDF where the prover has to compute $f^N(\chi)$, the verifier only cares about the correctness of the last step of the prover's computation, so it only has to check $\pi_N$ (the verification of $\pi_N$ does not require knowing the values of the previous proofs $\pi_1, \ldots, \pi_{N-1}$). That is, the VDF prover outputs $f^N(\chi)$ and $\pi_N$, and the verifier uses $\pi_N$ to check that the purported value of $f^N(\chi)$ is correct.

IVC, which can be implemented using variants of SNARGs, guarantees that the proof $\pi_N$ is

short and can be checked much faster than performing the underlying computation. Further, it is easy for the prover to modify each $\pi_i$ to obtain $\pi_{i+1}$. As a result, the multiplicative overhead of the prover in computing the proof compared to performing the underlying computation is a small constant factor.

[BBBF18] discusses the IVC approach to VDFs in more detail. In the rest of this paper, we will focus on the black-box approach to constructing PoW and PoSW.

# 3 Definitions

## 3.1 PoW example: Bitcoin

A simple PoW, which is similar to that used in Bitcoin, is as follows. The prover and verifier have access to a cryptographic hash function $H : \{0,1\}^* \to \{0,1\}^k$ that maps input strings to output strings of length $k$ (a typical value is $k = 256$). We model $H$ as a random oracle. Given a statement $\chi$ and a time parameter $N = 2^n$, the prover must find a string $y$ (called a nonce) such that $H(\chi, y)$ begins with $n$ zeroes. Here, $H(\chi, y)$ denotes calling $H$ on the concatenation of $\chi$ and $y$. The nonce $y$ is the proof, which the prover sends to the verifier.

Since we model $H$ as a random oracle, the expected number of queries to $H$ required to compute a proof is $N$, while the verifier can check the validity of the proof using a single query. Note that this protocol does not guarantee unique proofs. In other words, there will probably be many different nonces that work for any statement, due to collisions in $H$ (although such collisions are difficult to find).

## 3.2 Definition of PoW

Now we define black-box PoW. The definitions of black-box PoW and PoSW in this paper are modified from [MMV13] and [BBBF18]. The most basic form of the definition is Definition 1.

**Definition 1** (PoW)**.** *Let $f$ be a function, $f : [0,1] \to [0,1]$. A PoW with soundness $f$ is a protocol between a prover $P$ and a verifier $V$, as follows.*

***Common inputs.*** *$P$ and $V$ agree on a security parameter $k$, a statement $\chi \in \{0,1\}^k$, and a time parameter $N \leq \text{poly}(k)$. $P$ and $V$ have access to a random oracle $H : \{0,1\}^* \to \{0,1\}^k$.*

***Prover.*** *$P$ takes input $(k, \chi, N)$, runs in time $\text{poly}(k) \cdot N$, and makes $N$ queries to $H$. $P$ outputs a string $y \in \{0,1\}^{\text{poly}(k)}$, which is called the proof.*

***Verifier.*** *$V$ takes input $(k, \chi, N, y)$ and is a randomized algorithm that runs in time* $\text{poly}(k)$*. $V$ either accepts or rejects.*

***Completeness.*** *For any $N \leq \text{poly}(k)$, if $\chi$ is chosen uniformly at random from $\{0,1\}^k$, then when $V$ interacts with $P$,*

$$\Pr(V \text{ accepts}) \geq 1 - \text{negl}(k).$$

***Soundness.*** *Let $P^* = \{P^*_{k,N}\}$ be any non-uniform family of circuits indexed by $k$ and $N$, of size* $\text{poly}(k)$ *with oracle access to $H$. Let $N \leq \text{poly}(k)$, and let $\chi$ be chosen uniformly at random from $\{0,1\}^k$. If the number of queries to $H$ that $P^*_{k,N}$ makes is $\alpha N$ for $0 \leq \alpha \leq 1$, then if $V$ interacts with $P^*_{k,N}$,*

$$\Pr(V \text{ accepts}) \leq f(\alpha).$$

Although the definition only requires the verifier to run in time $\text{poly}(k)$, usually a PoW is only useful if the runtime of the verifier is small compared to the runtime of the prover. The completeness condition says that the verifier will accept in an honest execution of the protocol, while the soundness condition says that any prover that makes at most $\alpha N$ queries to $H$ can make the verifier accept with probability at most $f(\alpha)$. Hence, "work" is measured as the number of queries to $H$. We define soundness against non-uniform adversaries to account for pre-computation that an adversary may have performed.

In the Bitcoin PoW described in section 3.1, because $H$ is a random oracle, essentially the only thing a prover can do is to try different values of $y$ until finding one that works ($H(\chi, y)$ starts with $n$ zeros). The probability that a particular values of $y$ works is $1/2^n = 1/N$, so a prover that makes $\alpha N$ queries can make the verifier accept with probability

$$1 - (1 - 1/N)^{\alpha N} \approx 1 - e^{-\alpha}.$$

The fact that $\chi$ is prepended to the input of $H$ (the verifier checks $H(\chi, y)$ rather than $H(y)$) means that a prover cannot reuse any work that it has done for a different statement $\chi$. (If the verifier had checked $H(y)$ instead, then the same value of $y$ would work for all statements.) This ensures the non-uniform soundness in Definition 1. Therefore, the Bitcoin PoW satisfies Definition 1, because a prover that makes $\text{polylog}(k) \cdot N$ queries to $H$ can make the verifier accept with high probability, while no prover that performs $\alpha N$ work can make the verifier accept with probability much more than $f(\alpha)$.

The protocol in Definition 1 is non-interactive, in the sense that the prover sends a single message to the verifier and the verifier doesn't send any messages to the prover. We can modify the definition to allow the prover and verifier to interact after the prover sends $y$, in order for the prover to convince the verifier that $y$ is a valid proof.

**Definition 2** (interactive PoW)**.** *An interactive PoW with soundness $f$ is defined in the same way as Definition 1, except:*

- *After $P$ outputs $y$, $P$ and $V$ can interact. ($P$ and $V$ should still run in time* $\text{poly}(k)$*.) At the end of the interaction, $V$ accepts or rejects.*
- *We modify the soundness condition so that if the number of queries to $H$ that $P^*$ makes* ***before outputting*** *$y$ is $\alpha N$, then* $\Pr(V \text{ accepts}) \leq f(\alpha)$*.*

We could have defined an interactive PoW simply as an interactive protocol in which the verifier is unlikely to accept unless the prover has made about $N$ queries to $H$. The reason we instead define it with the modified soundness condition is that usually, the value of the proof in a PoW should be hard to predict without performing about $N$ work. This is the case, for example, in the applications to randomness beacons described in section 2. If our definition had only required that $P^*$ makes about $N$ queries in total as opposed to $N$ queries before outputting the proof $y$, then that leaves open the possibility that $y$ could just be the empty string, and the prover sends the "actual" value of the proof at the start of the interactive phase. In this case, $y$ would be very easy to predict.

In applications of PoW, the non-interactive definition in Definition 1 is usually favored over the interactive Definition 2. The non-interactive property is convenient in a decentralized blockchain, for example. However, if a PoW satisfies Definition 2 and the verifier is public-coin, then the protocol can be made non-interactive using the Fiat-Shamir heuristic [FS87].

## 3.3 Definition of PoSW

The Bitcoin PoW does not enforce sequential computation. The work of the prover is highly parallelizable, because a prover can try many different values of $y$ at once. The following definition of PoSW is similar to Definition 1, but it captures the idea that the prover must call $H$ a certain number of times in a row, not just a certain number of times overall.

**Definition 3** (PoSW)**.** *A PoSW with soundness $f$ is defined in the same way as Definition 1, except that soundness is respect to the number of sequential queries to $H$. Namely,*

***Soundness.*** *Let $P^* = \{P^*_{k,N}\}$ be any non-uniform family of circuits indexed by $k$ and $N$, of size* $\text{poly}(k)$ *with oracle access to $H$. Let $N \leq \text{poly}(k)$, and let $\chi$ be chosen uniformly at random from $\{0,1\}^k$. If the number of* ***sequential*** *queries to $H$ that $P^*_{k,N}$ makes is $\alpha N$ for $0 \leq \alpha \leq 1$, then if $V$ interacts with $P^*_{k,N}$,*

$$\Pr(V \text{ accepts}) \leq f(\alpha).$$

In terms of circuits, we say that a list of queries $q_1, q_2, \ldots$ is sequential if there is a directed

path in the circuit going through the gates for $q_1, q_2, \ldots$ in order. In the definition of soundness, we still require $P^*$ to have size $\text{poly}(k)$, because if the parallelism of $P^*$ is too large (for example, exponential in $k$), then $P^*$ might be able to break properties of the random oracle such as collision resistance.

It is also possible to define soundness with respect to the runtime of the honest prover: if $P$ runs in time $t$, then any $P^*$ that runs in time $\alpha t$ on a polynomial number of processors makes the verifier accept with probability at most $f(\alpha)$. This definition is given in [BBBF18]. We will define soundness with respect to the number of sequential queries, since reasoning about the number of queries is usually a bit easier than reasoning about exact runtime.

Like in Definition 2, we can modify Definition 3 to allow interaction:

**Definition 4** (interactive PoSW)**.** *An interactive PoSW with soundness $f$ is defined in the same way as Definition 3, except:*

- *After $P$ outputs $y$, $P$ and $V$ can interact. ($P$ and $V$ should still run in time $\text{poly}(k)$.) At the end of the interaction, $V$ accepts or rejects.*
- *We modify the soundness condition so that if the number of sequential queries to $H$ that $P^*$ makes **before outputting** $y$ is $\alpha N$, then $\Pr(V \text{ accepts}) \leq f(\alpha)$.*

Like interactive PoW, an interactive PoSW with a public-coin verifier can be made non-interactive using the Fiat-Shamir heuristic.

As mentioned in section 2, when the proof in a PoW or PoSW should be unique, the prover is essentially computing some function $F_H$ of its inputs (the output of the function depends on $H$). The prover outputs the value of $y = F_H(k, \chi, N)$, and typically we also allow the prover to output a string $\pi$ proving that $y$ was computed correctly. In some cases, $\pi$ might not be needed and can be the empty string. Only the value of $y$ needs to be unique. Thus, we define a unique PoW or PoSW as follows.

**Definition 5** (PoW (PoSW) with unique proofs)**.** *Let $f : [0,1] \to [0,1]$ and $u : \mathbb{N} \to [0,1]$. A PoW (PoSW) with soundness $f$ and uniqueness $u$ is a protocol between a prover $P$ and a verifier $V$, as follows.*

***Common inputs.*** *$P$ and $V$ agree on a security parameter $k$, a statement $\chi \in \{0,1\}^k$, and a time parameter $N \leq \text{poly}(k)$. $P$ and $V$ have access to a random oracle $H : \{0,1\}^* \to \{0,1\}^k$.*

***Prover.*** *$P$ takes input $(k, \chi, N)$, runs in time $\text{poly}(k) \cdot N$, and makes $N$ queries to $H$. $P$ outputs a string $y$, and then $P$ outputs another string $\pi$. The lengths of $y$ and $\pi$ are $\text{poly}(k)$.*

***Verifier.*** *$V$ takes input $(k, \chi, N, (y, \pi))$ and is a randomized algorithm that runs in time $\text{poly}(k)$. $V$ either accepts or rejects.*

***Completeness.*** *For any $N \leq \text{poly}(k)$, if $\chi$ is chosen uniformly at random from $\{0,1\}^k$, then when $V$ interacts with $P$,*

$$\Pr(V \text{ accepts}) \geq 1 - \text{negl}(k).$$

***Soundness.*** *Let $P^* = \{P^*_{k,N}\}$ be any non-uniform family of circuits indexed by $k$ and $N$, of size $\text{poly}(k)$ with oracle access to $H$. Let $N \leq \text{poly}(k)$, and let $\chi$ be chosen uniformly at random from $\{0,1\}^k$. If the number of (sequential) queries to $H$ that $P^*_{k,N}$ makes* ***before outputting*** *$y$ is $\alpha N$ for $0 \leq \alpha \leq 1$, then if $V$ interacts with $P^*_{k,N}$,*

$$\Pr(V \text{ accepts}) \leq f(\alpha).$$

***Uniqueness.*** *Let $(y, \pi)$ be the output of the honest prover $P$ on input $(k, \chi, N)$. For any $P^*$ as in the soundness condition, if $P^*(k, \chi, N)$ outputs $(y', \pi')$ and $y' \neq y$, then*

$$\Pr(V \text{ accepts}) \leq u(k).$$

As in Definitions 2 and 4, we define soundness with respect to the amount of work that $P^*$ performs before outputting $y$. We could also modify Definition 5 to allow interaction between $P$ and $V$ after $P$ sends $\pi$, but Definition 5 suffices for this paper.

## 3.4 PoSW: a first attempt

A first attempt at constructing a PoSW is the following. The prover calls $H$ iteratively $N$ times, starting with input $\chi$. That is, the prover must compute $F(k, \chi, N) = H^N(\chi)$, where $H^0(\chi) = \chi$ and $H^{i+1}(\chi) = H(H^i(\chi))$. Modeling $H$ as a random oracle, it is unlikely that the prover would be able to compute the correct value of $H^N(\chi)$ without actually calling $H$ $N$ times in sequence, even if the prover could make multiple queries in parallel. The prover sends the value of $H^N(\chi)$ to the verifier as $y$, and $\pi$ is empty.

This attempt at constructing a PoSW runs into problems in the verification step. While it is true (with high probability) that the prover must perform a lot of sequential work to compute the correct value of $H^N(\chi)$, it is unclear how the verifier should check that $y$ is indeed the correct value of $H^N(\chi)$. One option is for the verifier to perform the whole computation itself, but we would like verification to be easier than actually computing the proof.

Here is one attempt to reduce the verifier's runtime. Suppose that the prover, instead of sending only the value of $H^N(\chi)$, somehow gives the verifier random access to all the intermediate values $H^i(\chi)$ that it has computed, for $1 \leq i \leq N$. In other words, the prover sends the values $\phi_i = H^i(\chi)$ to the verifier, and we imagine that the verifier can access any particular value $\phi_i$ without reading all the values.

Then the verifier might operate as follows. Let $\phi_1, \ldots, \phi_N$ be the purported values of $H^1(\chi), \ldots H^N(\chi)$ that a (possibly dishonest) prover has sent. The verifier repeatedly chooses $i \in \{0, \ldots, N-1\}$ uniformly at random and checks that $\phi_{i+1} = H(\phi_i)$ (using random access to $\phi_i$ and $\phi_{i+1}$). If not, then the verifier has detected the prover cheating, and the verifier rejects. On the other hand, if these repeated tests all pass, then the verifier accepts.

The verifier might not be able to guarantee that *all* the values of $\phi_1, \ldots, \phi_N$ are *correct*. That is, depending on the number of repeated tests, the verifier might not be able to guarantee with high probability that each $\phi_i$ indeed equals $H^i(\chi)$. For example, suppose the prover uses a different statement $\chi'$ instead of $\chi$ but correctly computes $H^1(\chi'), \ldots, H^N(\chi')$ and reports those values as $\phi_1, \ldots, \phi_N$. Then, unless $i = 0$, it is true that $\phi_{i+1} = H(\phi_i)$.

What the verifier can guarantee is that *most* of the values of $\phi_i$ are *consistent* with $\phi_{i+1}$, in the sense that $\phi_{i+1} = H(\phi_i)$. For example, if the verifier makes 100 repeated tests, then the prover would probably be caught if more than 10% of the values of $\phi_i$ were inconsistent. This is because the prover will be caught if the verifier chooses any inconsistent index $i$, and in this case, the probability that the verifier never chooses an inconsistent $i$ is at most

$$(1 - \frac{1}{10})^{100} \leq e^{-(1/10)(100)} = e^{-10}.$$

However, this attempt at reducing the verifier's runtime raises the following questions:

1. Does guaranteeing that 90% of the values of $\phi_i$ are consistent actually guarantee that the prover must have performed about $N$ sequential work?

2. How can the prover give the verifier random access to the values of $\phi_i$?

In the rest of this section, we give a negative answer to the first question, which means that this first attempt at a PoSW has a serious shortcoming. In section 4, we introduce tools for addressing this shortcoming and for answering the second question.

So, suppose a cheating prover wants to pass the verification with high probability, but wants to perform significantly less than $N$ sequential work. Then one strategy it can use is the following. The cheating prover follows the "first half" of the protocol honestly, correctly computing $\phi_1, \ldots, \phi(N/2) = H^1(\chi), \ldots, H^{N/2}(\chi)$. However, then the prover cheats, and instead of computing $\phi_{N/2+1} = H^{N/2+1}(\chi) = H(H^{N/2}(\chi))$, it chooses some arbitrary value $z$. It sets $\phi_{N/2+1} = z$, and it computes the rest of the values consistently: $\phi_{i+1} = H(\phi_i)$ for $i > N/2 + 1$.

If the cheating prover deviates from the protocol in this way, then all the values of $\phi_i$ are consistent, except for $\phi_{N/2+1}$, which is chosen arbitrarily. Because the verifier chooses random points in the computation and checks for consistency, the only way the prover would be caught is if the verifier happened to choose $N/2 + 1$ as one of its checks. This is unlikely to happen, and hence the prover is likely to pass the verification.

However, note that the work of the cheating prover can be parallelized, so that it only has to perform $N/2$ steps of computation in sequence. If the cheating prover can perform two steps of computation in parallel, then it can compute the values $\phi_1, \ldots, \phi_{N/2}$ and the values $\phi_{N/2+1}, \ldots, \phi_N$ concurrently. This is because $\phi_{N/2+1}$ no longer depends on $\phi_{N/2}$, but is instead the arbitrary value $z$ that the cheating prover can choose in advance.

We conclude that the answer to the first question is negative. Even if we guarantee that 90% of the $\phi_i$ are consistent, the prover might have performed much less than $N$ sequential work. In this example, the cheating prover only sends a single inconsistent $\phi_i$ and manages to perform only $N/2$ sequential work. We can modify the cheating prover so that it can get away with even less sequential work by using a similar idea. For example, it can replace the three values $\phi_{N/4+1}, \phi_{N/2+1}, \phi_{3N/4+1}$ with three arbitrary values and then compute the rest of the $\phi_i$ consistently. Then it would only have to perform $N/4$ sequential work.

# 4 PoSW using depth-robust DAGs

In this section, we summarize the MMV construction of PoSW. We start by introducing tools that address the two questions we raised in section 3.4. The two tools described in sections 4.1 and 4.2 address the first question, and the tool described in section 4.3 addresses the second question.

## 4.1 Tool: labeling a DAG

Recall the attempt to reduce the verifier's runtime in section 3.4. Instead of the prover sending only the end result of its computation, it sends all the intermediate steps. The verifier randomly selects certain steps and checks that they are consistent (in the sense that $\phi_{i+1} = H(\phi_i)$). The first question we asked was whether, by ensuring that most of these intermediate steps are consistent, we can ensure that the prover must have performed about $N$ sequential work. Let's say that our goal is to ensure that, with high probability, the prover can only pass the verification if it has performed at least $0.8N$ sequential work.

*Remark.* We emphasize that, by checking that most of the intermediate steps are consistent, the verifier only wants to make sure that the prover must have performed about $N$ sequential work. The verifier is not checking that the result of the prover's computation is necessarily *correct* (i.e., whether $\phi_N$ actually equals $H^N(\chi)$). This distinction is important, because in general we cannot check the correctness of a computation just by checking that most steps follow from the previous step, unless the computation is presented in some special form (like the PCP-type results discussed in section 2).

We showed in section 3.4 that if the protocol calls for the prover to compute $H^N(\chi)$, then even sending all the intermediate steps doesn't let us ensure $0.8N$ sequential work. So, if we want to use the paradigm of the verifier checking the consistency of randomly chosen intermediate steps, the protocol must require the prover to perform some kind of computation other than just computing $H^N(\chi)$. In the MMV construction (and also in the CP construction), the prover must instead label a directed acyclic graph (DAG), which can be seen as a generalization of the task of computing $H^N(\chi)$.

**Definition 6** (labeling a DAG)**.** *Let $G$ be a DAG with $N$ vertices numbered $1, \ldots, N$. Given the statement $\chi$, the prover must compute a label $l_i$ for each vertex $i$ as follows. The prover computes the labels for the vertices in topological order. Suppose vertex $i$ has $d$ parents (vertices that point to vertex $i$), which are numbered $p_1, \ldots, p_d$, and the prover has already computed their labels $l_{p_1}, \ldots l_{p_d}$. Then it computes the label of $i$ as $l_i = H(\chi, i, l_{p_1}, \ldots, l_{p_d})$. (Recall that this notation means that the input to $H$ is the concatenation of $\chi, i, l_{p_1}, \ldots, l_{p_d}$.) For example, if vertex 5 has parent vertices 2, 3, and 4, then $l_5 = H(\chi, 5, l_2, l_3, l_4)$.*

The intermediate steps of this computation are the labels of all the vertices, which the prover sends to the verifier. (Recall that we will discuss in section 4.3 how the prover can send these labels without the verifier having to spend $N$ time to read all of them.) The problem in section 3.4 was that even if the verifier can guarantee that most of the intermediate steps are consistent, it is not guaranteed that the prover performed almost $N$ sequential work. The following example shows how the idea of labeling a DAG might address this problem. Consider the task of labeling the following DAG $G$ with 5 vertices:

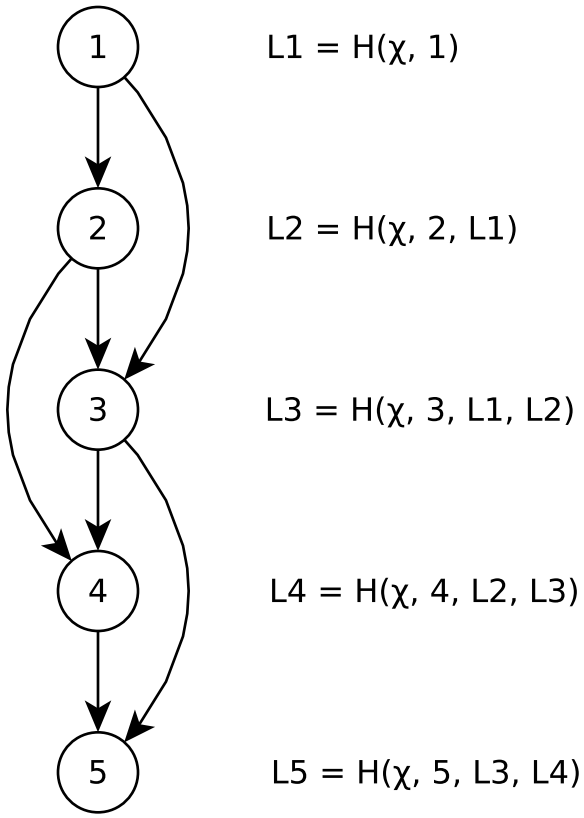


Figure 1

$G$ contains a path of length 5: vertices 1, 2, 3, 4, 5. This means that the honest prover must perform 5 steps of sequential work, because the label of 2 depends on the label of 1, the label of 3 depends on the label of 2 (in addition to the label of 1), etc.

Now, suppose the prover tries to cheat as in section 3.4. Somewhere in the middle of the computation, say at vertex 3, instead of honestly computing $l_3$ the prover uses some arbitrary value $z$ as $l_3$. The prover carries out the rest of the computation consistently. In section 3.4, we showed how the prover could reduce the required sequential computation from $N$ to $N/2$ by deviating from the protocol like this.

However, in this example, note that if we were to delete vertex 3 from $G$, then there is still a path of length 4: vertices 1, 2, 4, 5. This captures the idea that, even if the prover does not have to compute the label for vertex 3, there is still a sequence of 4 vertices whose labels depend on each other. Therefore, even if the prover fabricated a label for vertex 3, it would still have to perform 4 steps of sequential work.

We see that requiring the prover to perform a computation similar to labeling the DAG $G$

in this example can solve the problem in our first attempt. The property of $G$ that enables this is that, even if a vertex is removed, there remains a long path containing most of the vertices in $G$. Hence, even if the prover cheats on one label, there is still a long path of labels that they have to compute, and doing so requires a lot of sequential work.

Our first attempt, where the prover sends the values $\phi_i = H^i(\chi)$, is a special case of labeling a DAG. Namely, the DAG would be a path with $N$ vertices. The only difference between our first attempt and labeling a path is that, in our definition of labeling a DAG, when we compute $l_i$ we include $i$ as part of the input to $H$. We do this to ensure that the prover cannot figure out the labels of multiple vertices using a single call to $H$. Without $i$ as part of the input to $H$, all vertices with no parents would have the same label.

## 4.2 Tool: depth-robust DAGs

Because our first attempt is a special case of labeling a DAG, there has to be some property of a DAG that makes it suitable for use in a PoSW–some property that the path graph does not have. As suggested above, loosely speaking, this property is that even when some vertices are removed, a long path remains in the graph. This property is called *depth-robustness*.

**Definition 7** (depth-robust DAG)**.** *Let parameters $\alpha, \beta$ be between 0 and 1. A DAG $G$ with $N$ vertices is $(\alpha, \beta)$-depth-robust if, whenever $\alpha N$ vertices are removed from $G$, there remains a directed path of length $\beta N$.*

*For shorthand, we say that $G$ is depth-robust if it is $(0.1, 0.8)$-depth-robust. That is, if* 10% *of the vertices of $G$ are removed, there remains a path containing* 80% *of the vertices of $G$.*

For example, the graph in Figure 1 is $(0.2, 0.8)$-depth-robust, because whenever 1 of the 5 vertices is removed, there is a path through the remaining 4 vertices.

As suggested in section 4.1, the importance of depth-robust DAGs is that they can be used to construct a PoSW. Consider the following protocol. The prover has to label a depth-robust DAG $G$ with $N$ vertices, and the prover gives the verifier random access to any labels of its choice. The verifier repeatedly chooses a vertex at random and wants to check that the chosen vertex is consistent. This means that if the chosen vertex $i$ has parents $p_1, \ldots, p_d$, then the verifier checks that indeed, $l_i = H(\chi, i, l_{p_1}, \ldots, l_{p_d})$. So, when the verifier chooses vertex $i$, the prover gives it the labels of vertex $i$ and its parents $p_i, \ldots, p_d$, and the verifier checks for consistency.

The verifier repeats this check enough times so that, unless at least 90% of the vertices in $G$ are consistent, the prover is very likely to get caught. For example, if more than 10% of the prover's labels were inconsistent, then repeating the check with 100 randomly chosen vertices would allow the prover to pass with probability less than $(1 - 1/10)^{100} \leq e^{-10}$.

The verifier accepts if all its checks turn out to be consistent. If this occurs, then the verifier can be confident that at least 90% of the vertices in $G$ are consistent. But then, by the depth-robustness of $G$, there is a path of length at least $0.8N$ of consistent vertices. In other words, there is a path of length at least $0.8N$ where the label of one vertex on the path depends on the label of the previous vertex. In order for the prover to have computed such a path of consistent vertices, it almost certainly must have made at least $0.8N$ sequential queries to $H$.

In [MMV13], the authors show how to construct an explicit family of depth-robust DAGs where the vertices have degree polylog($N$). Thus, if the verifier wants to check that a vertex is consistent, it only has to check the labels of polylog($N$) vertices. We haven't yet specified exactly how this protocol fits the definition of a PoSW–for example, we haven't specified what $y$ and $\pi$ are. However, this analysis suggests that, provided an answer to the second question in section 3.4, depth-robust DAGs give us a way to construct a PoSW.

## 4.3 Tool: Merkle trees

Recall that the second question in section 3.4, in the setting of labeling a DAG, is: how can the prover send the labels to the verifier so that the verifier has random access to labels of its choice? Essentially, the prover wants to commit to the labels it has computed, and when the verifier wants to know the label of a vertex, the prover opens the commitment to that label. One scheme that accomplishes this is a Merkle tree.

A Merkle tree can be constructed from a collision-free hash function that maps strings of length $2k$ to strings of length $k$. The random oracle $H$ satisfies the collision-free property, which means that it is difficult to find two different inputs to $H$ that map to the same output. (Any polynomial-size adversary can produce such a pair of inputs with negligible probability.)

Suppose the prover wants to create a Merkle tree commitment of its $N$ labels. Assume that $N$ is a power of 2: $N = 2^n$. To construct the Merkle tree, the prover places the $N$ labels at the $N$ leaves of a binary tree of depth $n$. (So the label for vertex 1 would be the leftmost leaf, the label for vertex 2 would be the next leaf, etc.) Then, it computes labels for the remaining nodes in the binary tree as follows. If the two children of a node have labels $l_1$ and $l_2$, then the node has label $H(l_1, l_2)$. This is essentially the same idea as labeling a DAG using the hash function $H$. The binary tree with its nodes labeled in this way is a Merkle tree commitment of the labels of the $N$ vertices of $G$.

Once the prover has computed labels for the whole binary tree, let $\phi$ be the label of the root. This value $\phi$ will be the value of $y$ in the PoSW, and the prover only has to send $\phi$ to the verifier, instead of the labels of all the vertices in $G$. When the verifier wants to access the label of a vertex $i$, the prover finds the path from the root of the Merkle tree to the leaf

corresponding to vertex $i$. Then, in order to open the Merkle tree commitment, the prover sends the labels of all nodes in the Merkle tree along this path, along with the labels of all siblings of nodes along this path. (Thus, the prover sends $2n = O(\log N)$ labels in order to open the label of one vertex.)

For example, in Figure 2 below, the verifier has requested the label of the marked leaf. The prover would send the labels of all the filled-in nodes. (The prover doesn't have to send the value of the root node $\phi$ when opening the commitment, because it has already sent $\phi$ when making the commitment.)

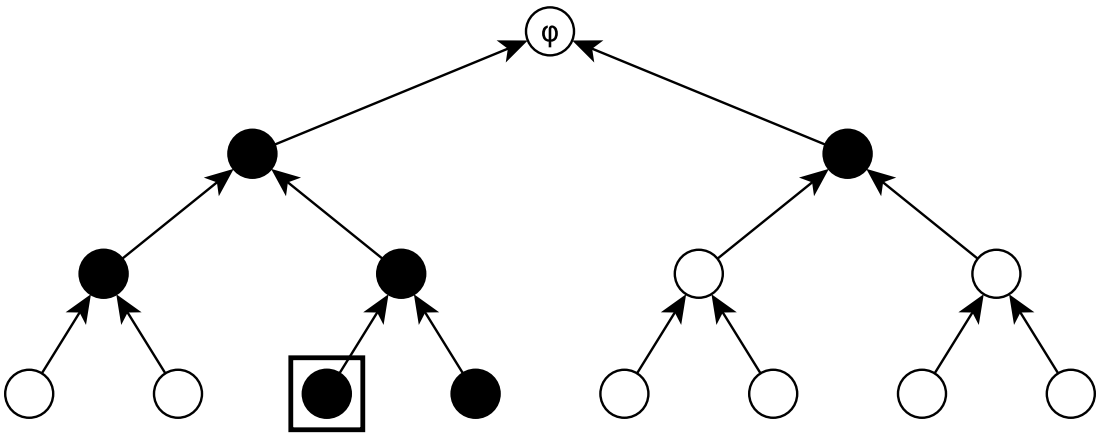


Figure 2

The verifier checks the consistency of each non-leaf node in the path from the leaf to the root (including the root itself). This means that, using the values sent by the prover, the verifier checks that the label of each node is the hash of the labels of its two children. This is why the prover sends both the labels of the nodes on the path and the labels of siblings of the nodes on the path: so that the verifier has all the labels necessary to make this check. If all the nodes are consistent, the verifier accepts the opening of the commitment to the label of the leaf; otherwise, the verifier has caught the prover cheating.

It is important that the prover cannot open the label of a leaf to two different values. If the prover could open a commitment to an arbitrary value (after $\phi$ has already been sent), then it would be able to label the DAG inconsistently but open the commitments to consistent values to fool the verifier. To see why it is unlikely that the prover would be able to open a leaf to two different values, we use the collision-free property of $H$. Suppose that the prover could successfully open a leaf in two different ways; let $O_1$ and $O_2$ represent these two openings. (So $O_1$ and $O_2$ are each a list of $2n$ labels that the prover sends to the verifier to make the opening.) Then we claim that the prover must have found a collision in the hash function. This is because $O_1$ and $O_2$ agree at the top level of the tree, where the two nodes

at depth 1 hash to $\phi$, but $O_1$ and $O_2$ differ at the bottom level of the tree, where a leaf is opened to two different values. Thus, the highest level of the tree where $O_1$ and $O_2$ differ yield a collision. For example, in Figure 3, the numbers are the labels of nodes that are involved in the openings of the marked leaf. The prover is trying to open the marked leaf to two different values, 3 in $O_1$ and 2 in $O_2$. In order to be able to make these two different openings here, the prover has also found a collision in the hash function. We see that both $(10, 5)$ and $(1, 6)$ hash to the value 9.

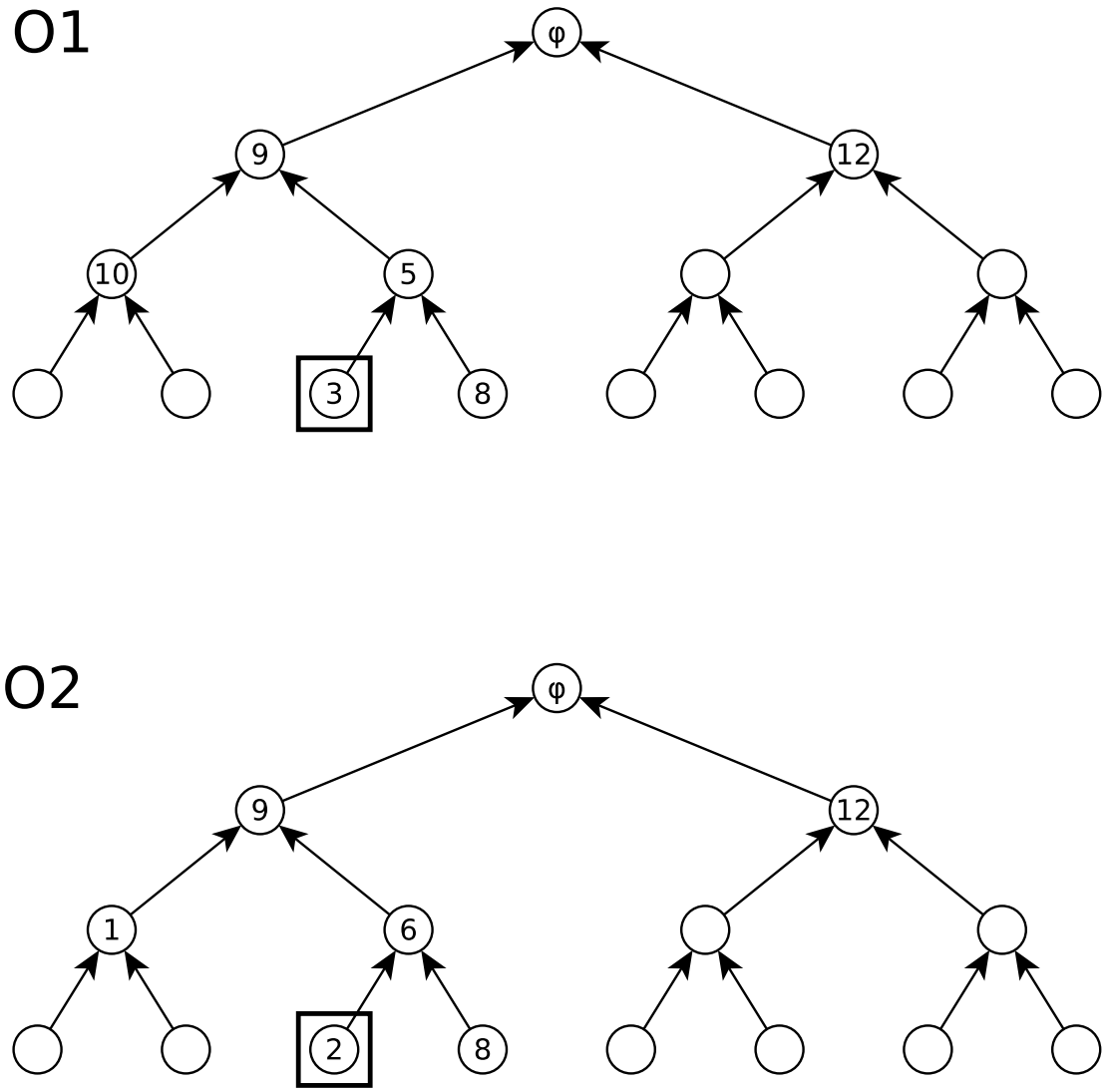


Figure 3

To summarize, Merkle trees answer the second question raised in section 3.4. They allow the

prover to commit to the $N$ labels it has computed by sending just the root label $\phi$. Then, the prover can reveal the label of any leaf as requested by the verifier, and unless the prover finds a collision in $H$, it can only open a given leaf in one way.

## 4.4 The MMV construction of PoSW

Using the tools we have described in the previous sections, we present the MMV construction of PoSW. The prover and verifier agree on a depth-robust DAG $G$ with $N$ vertices and degree polylog($N$), as in section 4.2. The prover starts by labeling $G$ as described in section 4.1. Then, the prover commits to this labeling by associating the vertices of the depth-robust DAG with the leaves of a Merkle tree, as in section 4.3. If $\phi$ denotes the label of the root of the Merkle tree, then the prover sends $y = \phi$ to the verifier, and $\pi$ is the empty string.

As suggested in section 4.2, the verifier can check that the prover has performed about $N$ sequential work by choosing random vertices of $G$ and checking that those vertices are labeled consistently. That is, the verifier chooses certain vertices of the $G$, and the prover provides Merkle tree openings for the labels of those vertices and of their parents in $G$. The verifier checks that the labels of the chosen vertices are computed correctly based on the labels of their parents, and that the Merkle tree openings are correct.

The reason this protocol requires the prover to perform almost $N$ sequential work has mostly been described in the previous sections. Consider the following two cases:

- The prover computes at least 90% of the labels in the DAG consistently. Then, by the depth-robustness of the DAG, there is a path of length $0.8N$ of consistent vertices, meaning that the prover must have performed at least $0.8N$ sequential work.

- The prover computes less than 90% of the labels in the DAG consistently. Then, the prover is very likely to be caught if the verifier makes enough queries. For example, if the verifier makes 100 queries, then the probability that the prover does not get caught is at most $(1 - 1/10)^{100} \leq e^{-10}$.

Therefore, this protocol is a PoSW. The verifier is very efficient, because checking that a vertex is consistent requires time polylog($N$). The verifier only has to check the consistency of polylog($k$) vertices to ensure that a cheating prover is caught with all but negligible probability. Also, as mentioned in section 3, because this verifier is public-coin, the protocol can be made non-interactive using the Fiat-Shamir heuristic.

The MMV construction is the first black-box construction of PoSW, but it also has drawbacks. First, labeling a depth-robust DAG is not space-efficient, and [ABP17] shows that labeling a depth-robust DAG in $O(N)$ time also requires $O(N)$ space. Second, as noted in section 1.2, this construction does not achieve unique PoSW. That is, it is easy for the

prover to generate many different values of $y = \phi$ that are likely to pass the verification. For example, suppose the prover follows the protocol honestly to compute $\phi$. Then the prover chooses a single vertex of the DAG and modifies its label to an arbitrary value. The prover recomputes the Merkle tree, which only requires recomputing the labels of the nodes of the Merkle tree along the path from the modified vertex to the root. In this way, the prover obtains a new value $\phi'$ as the root label. If the prover uses this value $\phi'$ as $y$, then it is very unlikely that the verifier would catch the prover cheating. This is because the only way the verifier would detect what the prover has done is if the verifier queries the specific vertex that the prover modified, or queries one of its children. But the probability that this happens is very small if the runtime of the verifier is much less than that of the prover. Thus, a cheating prover can arbitrarily change the label of a single vertex to obtain many different values of $y$ that are all likely to be accepted by a verifier.

# 5 Simple proofs of sequential work

[CP18] improve on the construction in [MMV13]. The CP construction also uses the DAG-labeling paradigm, but it shows that depth-robust DAGs are not necessary in order to construct PoSW. By eliminating the need for depth-robustness, the CP construction reduces the space complexity of the prover from $O(N)$ to $O(\log N)$, and also eliminates the need for somewhat complicated explicit constructions of depth-robust DAGs.

The CP construction defines the family of graphs $\{G_n\}_{n\in\mathbb{N}}$. $G_n$ is obtained by starting with the binary tree of depth $n$ and then adding some more edges. The binary tree is directed so that a child node points to its parent node. Then, for each node that is a left child, we add edges from that left child to all leaves in the subtree of the corresponding right child. The following example shows $G_3$:

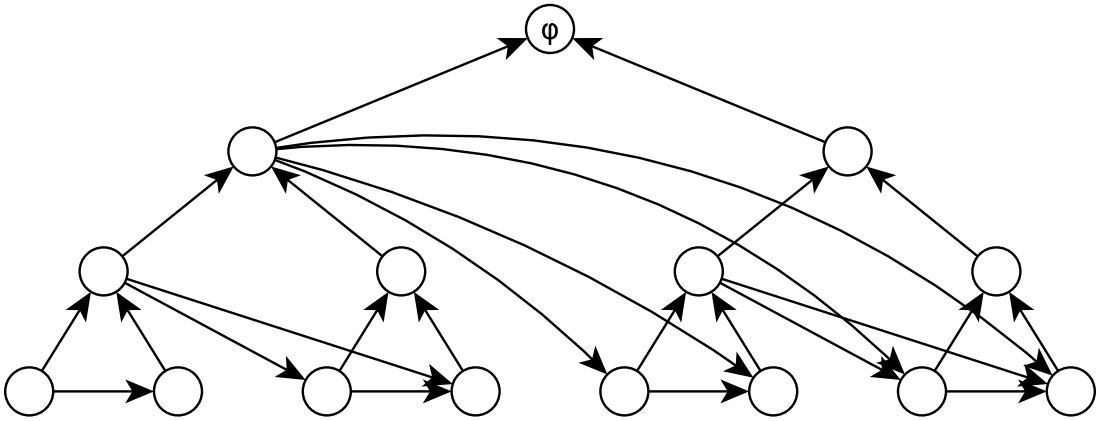


Figure 4

Another way to view this construction is that $G_n$ is obtained by taking two copies of $G_{n-1}$, and then adding edges from the root of the first copy to all leaves of the second copy. Intuitively, this helps enforce sequentiality, because it means that the prover cannot do anything with the second copy until it finishes working on the first copy. Also, we can think of this construction as labeling a Merkle tree with some additional edges (the edges from a left child to all leaves in the right child's subtree). Thus, the ideas used in the MMV construction involving Merkle tree commitments also apply to this construction.

The overall protocol is as follows. Given $N = 2^n$ and a statement $\chi$, the prover labels $G_n$ using $H$ and $\chi$. As in MMV, the prover obtains the root $\phi$ of $G_n$ and sends it to the verifier as the value of $y$ in the PoSW, and $\pi$ is the empty string. Also as in MMV, the verifier chooses several random leaves of $G_n$ and checks that they're consistent: the prover must provide Merkle tree openings of those leaves. (In this case, all parents of a leaf are already included in the Merkle tree opening of the leaf.) The verifier accepts if all the chosen vertices are consistent and the Merkle tree openings are correct. As in MMV, here the verifier is public-coin, and the protocol can be made non-interactive using Fiat-Shamir.

To analyze the sequentiality guarantees of this protocol, first note that $G_n$ is not depth-robust. For example, removing just the root of the left subtree of $G_n$ will cut off any paths between the left subtree and the right subtree. However, also note that, if the prover were to cheat on the root of the left subtree, then the prover would be caught if the verifier queries any leaf in the left subtree. This is the idea behind the proof of sequentiality.

Elaborating on this idea, suppose a prover cheats on certain nodes in $G_n$, so those nodes are inconsistent. The prover will be caught if the verifier queries any of the leaves in the subtrees rooted at these inconsistent nodes. [CP18] show by induction on $n$ that there is always a path of consistent nodes that contains all the leaves that do not belong to the

subtrees rooted at inconsistent nodes. Thus, we can consider two cases:

- At least 90% of the leaves computed by the prover do not belong to subtrees rooted at inconsistent nodes. Then there is a path containing all of these leaves, and so the prover must have performed at least $0.9N$ sequential work.

- More than 10% of the leaves computed by the prover belong to subtrees rooted at inconsistent nodes. Then the prover cannot open the Merkle tree commitment to any of these leaves and will be caught if the verifier queries any of them. As in the analysis of the MMV construction, this means that the verifier can efficiently catch the prover cheating in this case.

The CP construction improves on MMV by eliminating the need for depth-robust graphs, hence significantly reducing the space complexity of the prover and also making the protocol simpler. [CP18] show by induction on $n$ that the prover only needs $O(\log N)$ space, because after labeling a left subtree, the prover only needs to remember the root of the left subtree in order to label the right subtree. However, this construction does not achieve unique proofs, for a similar reason as in MMV. Suppose a prover follows the protocol honestly to compute the root value $\phi$. Then, the prover modifies the label of a single leaf to an arbitrary value and computes the remaining nodes honestly with the modified label. Then the prover will only be caught if the verifier queries the modified leaf. Also, note that if the prover chooses to change the rightmost leaf, then the only labels that the prover will have to recompute are the $n$ nodes along the path from the rightmost leaf to the root. This is because the only edges we add to the binary tree in order to construct $G_n$ are edges that start from a left child. Thus, once the prover honestly computes $y = \phi$, they can easily generate many more values of $y$ that are likely to be accepted. (Even if the verifier decides to always check the consistency of the rightmost leaf, the prover could still change other leaves that require relatively little recomputation.)

# 6 A weaker notion of unique PoW

In this section, we consider an easier version of the problem of a black-box construction of unique PoSW. We present a black-box construction of PoW (not sequential) where proofs are likely to be unique. Recall that a PoW like that in Bitcoin does not have unique proofs, because there will likely be many different nonces that work, due to collisions in $H$. Our unique PoW construction is not ideal. In particular, the verifier's runtime is not polylogarithmic in $N$, but more like $O(N^{4/5}) = O(2^{(4/5)n})$, where $N = 2^n$. But for large $N$, which is typical of a PoW, the verifier still performs much less computation than the prover.

It remains an open question whether unique PoSW can be constructed from black-box assumptions. Perhaps an idea similar to our unique PoW can be useful, although natural

attempts to extend this construction to a unique PoSW seem to run into difficulties.

## 6.1 Intuition

In our protocol, the prover starts by computing $H(\chi, 1), H(\chi, 2), \ldots, H(\chi, N)$. Like in the MMV and CP constructions, the prover builds a Merkle tree with $N$ leaves, where the label of the $i$th leaf is $H(\chi, i)$. These steps can be seen as a superficial way of labeling a DAG, where the DAG consists entirely of isolated vertices. We saw in MMV and CP that the root of the Merkle tree would not constitute a unique PoW. Recall that this is because changing the label of one leaf and computing everything else consistently would change the label of the root, but the prover wouldn't get caught unless the verifier examined that leaf.

The idea of our unique PoW is that, although changing one leaf and computing everything else consistently will change the label of the root, changing one leaf and computing everything else consistently doesn't change the labels of other leaves. Note that this is not true in a PoSW as in CP, because there are edges from certain nodes in the Merkle tree that point to other leaves. Therefore, in our PoW, if we can guarantee that *most* of the leaves are computed correctly, then it is unlikely that even a cheating prover could change the value of some robust statistic on the set of all leaves.

For example, let the bit $\sigma$ be 0 if the labels of most of the leaves start with the bit 0, and otherwise $\sigma = 1$. If a cheating prover could only change the labels of a small fraction of the leaves without being caught, the resulting value of $\sigma$ that the cheating prover obtains would likely be the same as if the prover computed all the leaves correctly. Hence, $\sigma$ can be thought of as a unique proof of work. Its value is difficult to guess with probability better than $1/2$ without computing a significant fraction of the leaves (we analyze this in section 6.4), and even a cheating prover is unlikely to be able to change it without getting caught. This intuition will be developed in the following sections.

The above discussion can be generalized in several ways. First, the fact that changing one leaf doesn't change other leaves can be generalized. At any level of the Merkle tree, say at depth $n/2$ where there are $N^{1/2}$ nodes, if the prover computes some of the nodes dishonestly, then the remaining nodes are still the same as what the honest prover would compute. Second, we can generalize the construction so that the proof of work is more than one bit long.

## 6.2 Construction

Following the ideas in section 6.1, Construction 1 is an (interactive) PoW where the proofs are likely to be unique.

**Construction 1.** *The prover and verifier take input as in Definition 5.*

1. *The prover builds a Merkle tree using the values $H(\chi, 1), \ldots, H(\chi, N)$, so the label of leaf $i$ is $H(\chi, i)$. This Merkle tree has $n$ levels, where $N = 2^n$.*

2. *The prover uses the $N^{1/2}$ nodes of the tree at depth $n/2$ to compute $\sigma$: if the majority of the nodes at depth $n/2$ start with the bit 0, then $\sigma = 0$. Otherwise, $\sigma = 1$.*

3. *The prover sends $(y, \pi)$ to the verifier. $y = \sigma$, and $\pi$ consists of the labels of the $N^{1/2}$ nodes at depth $n/2$.*

4. *Upon receiving $(y, \pi)$, the verifier checks that $y = \sigma$ agrees with the $N^{1/2}$ labels in $\pi$. Namely, if a majority of those labels start with 0, then $\sigma$ should be 0; otherwise, $\sigma$ should be 1. If this is false, the verifier rejects.*

5. *The verifier chooses $cN^{3/8}$ of the nodes at depth $n/2$ uniformly and independently at random, for some constant $c$.*

6. *For each of these chosen nodes, the verifier manually checks that it was computed correctly, by computing the subtree of the Merkle tree rooted at the chosen node. That is, the verifier computes the $N^{1/2}$ values of $H(\chi, i)$ for the appropriate leaves $i$ and checks that their Merkle tree hash is the corresponding label in $\pi$. If any of the chosen nodes was not computed correctly, the verifier rejects; otherwise, it accepts.*

The prover actually doesn't have to compute the entire Merkle tree, but rather only the nodes between depth $n/2$ and $n$. The runtime of the verifier is $O(N^{7/8})$. This is because the verifier reads the $N^{1/2}$ labels in $\pi$, and then the verifier chooses $O(N^{3/8})$ labels and manually performs $O(N^{1/2})$ work to check that each chosen label is correct. Thus, the step of manual verification requires $O(N^{7/8})$ work and dominates the runtime of the verifier. We can replace $n/2$ with some other depth to improve this runtime, as discussed in section 6.5. In the following sections, we analyze Construction 1 to show that it satisfies Definition 5.

## 6.3 Analysis of uniqueness

Suppose the verifier accepts in step 6 of the protocol in Construction 1. Then it is very likely that $\pi$ contains no more than $N^{1/8}$ incorrect labels, out of the $N^{1/2}$ nodes at depth $n/2$. Otherwise, the probability that all $cN^{3/8}$ of the verifier's checks would pass is bounded by:

$$\left(1 - \frac{N^{1/8}}{N^{1/2}}\right)^{cN^{3/8}} = \left(1 - N^{-3/8}\right)^{cN^{3/8}} \leq e^{(-N^{-3/8})cN^{3/8}} = e^{-c}.$$

By choosing an appropriate value of $c$, the probability of this event is very small. Hence, if the verifier accepts, we are essentially guaranteed that at most $N^{1/8}$ of the $N^{1/2}$ labels at

depth $n/2$ differ from what the honest prover would have computed.

Let $z$ be the number of labels at depth $n/2$ that would start with 0 if all labels were computed correctly; let $z'$ be the number of labels in $\pi$ that start with 0. Since we have established that $\pi$ can differ from the correct labels in at most $N^{1/8}$ places, this means that $z$ and $z'$ differ by at most $N^{1/8}$. Hence, even a cheating prover cannot report an incorrect value of $\sigma$ (without being caught in step 4) unless $z$ is within $N^{1/8}$ of $N^{1/2}/2$.

How likely is it that $z$ is so close to $N^{1/2}/2$? Using the random oracle assumption, each label at depth $n/2$ (if correctly computed) starts with 0 or 1 independently with probability 1/2. Let $M = N^{1/2}$. If we flip a fair coin $M$ times, then it's reasonable to expect that the number of heads differs from $M/2$ by about a standard deviation, which is $\sqrt{M}/2$. Substituting $M = N^{1/2}$, we find that a "typical" deviation of $z$ from $N^{1/2}/2$ is $N^{1/4}$. Hence, it should be quite unlikely that a deviation as small as $N^{1/8}$ would occur, and only such a small deviation would allow a prover to convince the verifier to accept an incorrect value of $\sigma$.

More formally, we can bound the probability that $z$ is within $N^{1/8} = M^{1/4}$ of $M/2$ by bounding the sum of the middle $2M^{1/4}$ values of the binomial distribution with parameters $(M, 1/2)$. For $k = 0, \ldots, M$, the binomial distribution assigns to $k$ the probability $(1/2)^M \binom{M}{k}$. Note that the largest value in this distribution is the middle value, $(1/2)^M \binom{M}{M/2}$. Therefore, the sum of the middle $2M^{1/4}$ values is at most $2M^{1/4} \cdot (1/2)^M \binom{M}{M/2} = O(M^{-1/4})$. The last equality follows from the fact that $\binom{M}{M/2} = O(2^M \cdot M^{-1/2})$, which can be shown using Stirling's formula, for example.

We conclude that the probability that a cheating prover can break the uniqueness condition in Definition 5 is $O(M^{-1/4}) = O(N^{-1/8})$, which approaches 0 for large $N$.

## 6.4 Analysis of soundness

Consider an adversary $P^*$ as in the soundness condition in Definition 5. If $P^*$ performs much less than $N$ work, then it cannot make the verifier accept, for the same reason as in the analysis of MMV and CP. Namely, if $P^*$ computes only 90% of the labels at depth $n/2$ correctly, then it will be caught if the verifier checks any of the incorrect labels. However, recall that the soundness must hold for any $P^*$ that performs little work before outputting $y$, even if $P^*$ performs more work after outputting $y$. This means that $P^*$ cannot predict the correct value of $\sigma$ with too high a probability unless it performs a lot of work.

Since $\sigma$ is the majority value of the first bit of the labels of the nodes at depth $n/2$, and these labels are computed by using a random oracle for a Merkle tree hash, essentially the only reasonable strategy an adversary $P^*$ can try is as follows. Say that the adversary performs $\alpha N$ work for $\alpha < 1$ and wants to predict the correct value of $\sigma$. That means the adversary

can only determine the correct labels of an $\alpha$ fraction of the $M$ nodes at depth $n/2$. The adversary can compute the labels of these $\alpha M$ nodes, take the majority of the first bit of the labels it computed, and predict that $\sigma$ will equal that majority value.

How likely is the adversary to succeed in guessing $\sigma$, in terms of $\alpha$? When we simulate this adversarial strategy with $M = 10^4$ and running $10^5$ trials (so in each trial, we randomly generate the labels of the $M$ nodes), the success probabilities are shown in Figure 5. $\alpha$ ranges from 0 to 1 along the x-axis, while the simulated probability of guessing $\sigma$ ranges from 0.5 to 1 on the y-axis (because the adversary cannot do worse than just flipping a coin).

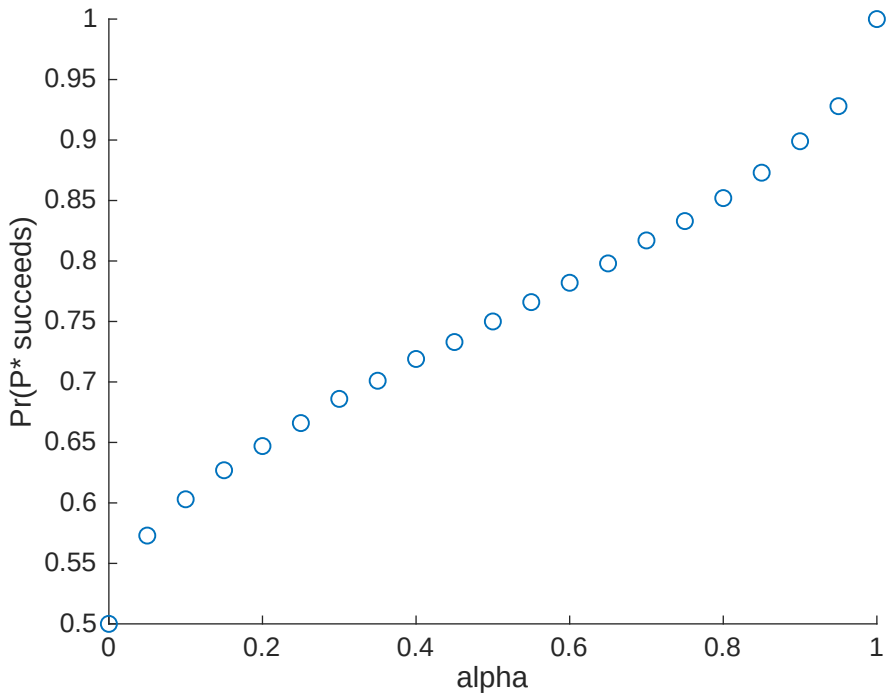


Figure 5

The shape of this chart is reminiscent of the arcsine distribution that appears in the context of random walks, which we now describe.

### 6.4.1 Arcsine laws

Suppose Alice and Bob play a game in which they flip a fair coin $m$ times. For each flip, if the coin comes up heads, then Alice gives Bob a dollar; otherwise, Bob gives Alice a dollar. At a given point in this sequence of flips, we say that Alice is in the lead if Alice has won

more times than Bob so far, so Alice has made a profit so far. Consider the question: over this sequence of $m$ coin flips, how many times will Alice be in the lead? The following figure from Grinstead and Snell's *Introduction to Probability* [GS12] shows the pdf (probability density function) of the number of times that Alice is in the lead when $m = 40$:

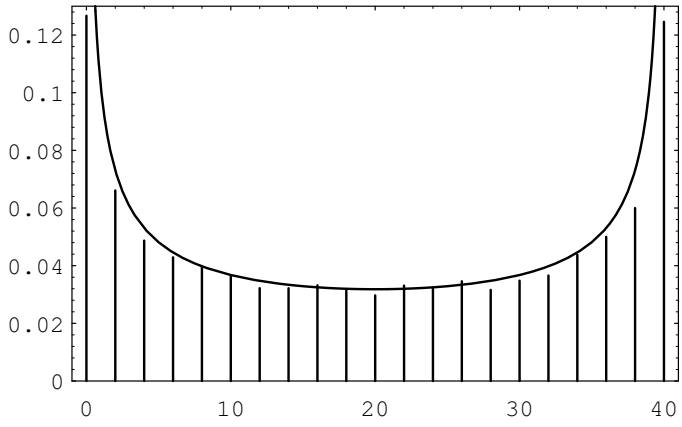


Figure 6

This pdf shows that (perhaps counterintuitively) the most likely outcomes are that Alice is always in the lead or that Alice is never in the lead. [GS12] proceeds to determine the cdf (cumulative distribution function) of the number of times that Alice is in the lead. If $L$ is the number of times that Alice is in the lead, then for $\alpha \leq 1$,

$$\Pr(L < \alpha m) \approx \frac{2}{\pi} \arcsin(\sqrt{\alpha}).$$

This is the arcsine distribution. The game between Alice and Bob can be thought of as a random walk $Z_0, \ldots, Z_m$, where $Z_i$ is the profit that Alice has made after $i$ coin tosses. The equation above shows that the fraction of times this random walk is positive follows the arcsine distribution. In fact, the arcsine distribution provides answers to a larger class of problems related to random walks. Another such problem is the following. Again, consider a random walk $Z_0, \ldots, Z_m$, where $Z_0 = 0$, and $Z_{i+1}$ equals $Z_i + 1$ with probability $1/2$ and $Z_i - 1$ with probability $1/2$. Let $T$ be the *last* time such that $Z_T = 0$. How is $T$ distributed? This problem is known as the problem of the last equalization in a random walk. As shown in [GS12], it turns out that the cdf of $T$ follows the same arcsine distribution, namely

$$\Pr(T < \alpha m) \approx \frac{2}{\pi} \arcsin(\sqrt{\alpha}).$$

### 6.4.2 Analysis of soundness (continued)

The range of the arcsine distribution $\frac{2}{\pi}\arcsin(\sqrt{\alpha})$ is 0 to 1. The range of $\Pr(P^* \text{ succeeds})$ in the simulation is 0.5 to 1. We can try to fit the arcsine distribution to the data from the simulation by interpolating it between 0.5 and 1, namely,

$$f(\alpha) = \frac{1}{2} + \frac{1}{\pi}\arcsin(\sqrt{\alpha}).$$

When we plot $f(\alpha)$ on Figure 5, it fits the simulated results almost perfectly. We will prove:

**Theorem 1.** *For the adversary $P^*$ described above,* $\Pr(P^* \text{ succeeds}) = f(\alpha)$. *We say that $P^*$ succeeds if it outputs the correct value of $\sigma$.*

We prove this claim by showing how the success of the adversary relates to the last equalization of a random walk. Recall that $P^*$ computes $\alpha M$ of the $M$ labels at depth $n/2$ outputs the majority value of the first bit of the labels it has computed. Without loss of generality, $P^*$ computes the first $\alpha M$ labels at depth $n/2$. For $i = 1, \ldots, M$, let $X_i$ be 1 if the $i$th label at depth $n/2$ starts with a 1, and $-1$ if the $i$th label starts with a 0. Define the random walk $Z_1, \ldots, Z_m$, where $Z_i = \sum_{j=1}^{i} X_j$. Thus, $Z_i$ is the difference between the number of 1s and the number of 0s among the first bits of the first $i$ labels, and $Z_i$ equals $Z_{i-1} + 1$ with probability 1/2 and $Z_{i-1} - 1$ with probability 1/2, which is the same random walk we considered in section 6.4.1.

As in section 6.4.1, let $T$ be the time of last equalization of $Z_1, \ldots, Z_M$. Consider what happens when $T < \alpha M$. If, among the first $\alpha M$ labels, there are more 1s than 0s, then $Z_{\alpha M} > 0$ and $P^*$ guesses $\sigma$ to be 1. But in this case, the fact that $T < \alpha M$ means that all the remaining values of the random walk $Z_{\alpha M+1}, \ldots, Z_M$ are also positive, because there are no more equalizations. Therefore, $P^*$'s guess for $\sigma$ will be correct. A similar thing happens if there are more 0s than 1s among the first $\alpha M$ labels. We conclude that if $T < \alpha M$, then $P^*$ always guesses correctly.

Now, consider $T \geq \alpha M$. We claim that in this case, $P^*$'s guess is correct with probability 1/2. We can think of the random choice of the $M$ labels as being done in order: we first randomly choose a label for the first node at depth $n/2$, then for the second node, and so on, up to node $M$. $P^*$ outputs its guess after labels have been chosen for the first $\alpha M$ nodes. The fact that $T \geq \alpha M$ means that, sometime in the future after $P^*$ outputs its guess, there will be an equal number of 0s and 1s. In other words, for some $i \in \{\alpha M + 1, \ldots, M\}$, we have $Z_i = 0$. Among the remaining labels $i+1$ through $M$, it is equally likely that a majority of them start with 0 or a majority of them start with 1. Since there are an equal number of 0s and 1s among the first $i$ labels, the correct value of $\sigma$ is the majority value of labels $i + 1$ through $M$. This majority value is 0 or 1 with probability 1/2, independent of the labels of any previous nodes, and in particular independent of the guess made by $P^*$. Therefore, when $T \geq \alpha M$, $P^*$ succeeds with probability 1/2.

Finally, using the arcsine law of the time of last equalization, we conclude that

$$\begin{aligned}\Pr(P^* \text{ succeeds}) &= \Pr(T < \alpha M) \cdot 1 + (1 - \Pr(T < \alpha M)) \cdot \frac{1}{2} \\ &= \frac{2}{\pi} \arcsin(\sqrt{\alpha}) \cdot 1 + (1 - \frac{2}{\pi} \arcsin(\sqrt{\alpha})) \cdot \frac{1}{2} \\ &= \frac{1}{2} + \frac{1}{\pi} \arcsin(\sqrt{\alpha}). \quad \square\end{aligned}$$

## 6.5 Modifications

Here we present several modifications to improve Construction 1. First, as noted in section 6.2, the runtime of the verifier is $O(N^{7/8})$. We can improve the runtime of the verifier by noting that the construction can use some other level of the Merkle tree besides $n/2$. In other words, we can modify the protocol so that, for some $0 < \beta < 1$, the prover sends the $N^\beta$ labels at depth $\beta n$ of the Merkle tree. Then, the verifier manually checks the correctness of $O(N^{(3/4)\beta})$ of these labels. The analysis in section 6.3 implies that the probability that a prover can come up with two different valid proofs would be $O(N^{-(1/4)\beta})$.

To optimize the verifier's runtime, we can choose $\beta = 4/5$. Then the verifier would take $O(N^{4/5})$ time to read the labels in $\pi$. The verifier would manually check $O(N^{3/5})$ nodes, each of which takes time $O(N^{1/5})$, because the height of the subtree below a node at depth $(4/5)n$ is $(1/5)n$. In this case, the total runtime of the verifier would be $O(N^{4/5})$. Further, the probability that a prover can compute two different valid proofs is bounded by $O(N^{-1/5})$, which improves on the original $O(N^{-1/8})$ bound. (Similarly, changing the number of nodes that the verifier manually checks can change how likely the proof is to be unique.)

We can also extend Construction 1 by making $y$ longer than one bit. Construction 1 takes the majority value of the first bit of each label at depth $n/2$ as the value of $y = \sigma$ (or at some other depth, if using the above modification). To make $y$ longer, we can set each bit of $y$ to be the majority value of that bit over the labels at depth $n/2$. In order to preserve uniqueness, we want the value of each bit of $y$ to be unique. For example, if we want a 256-bit proof and the probability that a particular bit violates uniqueness is $\epsilon$, then the probability that any bit violates uniqueness is at most $256\epsilon$ by the union bound.

Extending the construction to multiple bits also allows a further modification to make $y$ harder to guess for an adversary that performs less than $N$ work. As in the single-bit case, an adversary might try to compute only some of the labels at depth $n/2$ (or some other depth), and take the majority value of each bit over the computed labels in order to predict the majority value of each bit over all labels. When $y$ is longer, it is harder for the adversary to guess all the bits correctly by using this strategy, although it does give the adversary more opportunities to guess at least one bit correctly. If we want to make it difficult to guess even one bit correctly, one solution is the following. If we want $y$ to be 256 bits, let $\sigma \in \{0,1\}^{256}$

be the result of taking the majority value of each of the 256 bits over the labels at depth $n/2$. Instead of setting $y = \sigma$, we can set $y$ to be some other value computed from $\sigma$, such as $H(\chi, \sigma)$. This way, if an adversary performs less than $N$ work and computes only some of the labels at depth $n/2$, even if it can predict with some confidence that the majority value of a particular bit is a 0, that doesn't directly reveal anything about any of the bits in $y$. This is because the value of $y$ is the result of the hash applied to $\sigma$, which looks random unless the adversary can significantly narrow down the number of likely values of $\sigma$.

In fact, even suppose that an adversary could predict the majority value of many of the bits while performing much less than $N$ work (which is very unlikely to begin with). For example, suppose the adversary computes only some of the labels at depth $n/2$ and, by chance, is fairly confident about the value of all but 10 of the bits of $\sigma$. Then, if the adversary wants to guess the first bit of $y = H(\chi, \sigma)$, it might apply the hash to all $2^{10}$ values of $\sigma$ that it considers to be likely, and check whether the first bit of the hashed value is usually a 0 or usually a 1.

We could make a further modification to the construction to make this strategy much more difficult, if it is a concern at all. Instead of setting $y = H(\chi, \sigma)$, we can set $y = g(\chi, \sigma)$, where $g$ is some function that is harder to compute. $g$ might be a proof of work of size $N^{4/5}$: for example, $y = g(\chi, \sigma)$ equals the root label of a Merkle tree commitment of the values $H(\chi, \sigma, 1), H(\chi, \sigma, 2), \ldots, H(\chi, \sigma, N^{4/5})$. ($\pi$ is still the set of labels at depth $n/2$ or some other depth.) The idea here is that the verifier already has to perform $O(N^{4/5})$ work (if we use the nodes at depth $(4/5)n$), so the verifier could manually check the correctness of $g$ without too much overhead. In other words, the verifier uses the labels in $\pi$ to compute $\sigma$, and then checks that $g(\chi, \sigma) = y$. However, if an adversary wants to use the above strategy to try to predict the value of a bit of $y$, it would have to compute $g$ for each of the $2^{10}$ candidate values of $\sigma$. This would take about $2^{10}N^{4/5}$ work, which is probably at least as much as, if not more than, the work required just to follow the protocol honestly. The honest prover, on the other hand, would only have to perform $N^{4/5}$ work on top of the $N$ work that it already performs when computing $\sigma$.

# References


[ABP17] Joël Alwen, Jeremiah Blocki, and Krzysztof Pietrzak. Depth-robust graphs and their cumulative memory complexity. In Jean-Sébastien Coron and Jesper Buus Nielsen, editors, *Advances in Cryptology – EUROCRYPT 2017*, pages 3–32, Cham, 2017. Springer International Publishing.

[BBBF18] Dan Boneh, Joseph Bonneau, Benedikt Bünz, and Ben Fisch. Verifiable delay functions. In Hovav Shacham and Alexandra Boldyreva, editors, *Advances in Cryptology – CRYPTO 2018*, pages 757–788, Cham, 2018. Springer International Publishing.

[BBF18] Dan Boneh, Benedikt Bünz, and Ben Fisch. A survey of two verifiable delay functions. Cryptology ePrint Archive, Report 2018/712, 2018. https://eprint.iacr.org/2018/712.

[BCC+17] Nir Bitansky, Ran Canetti, Alessandro Chiesa, Shafi Goldwasser, Huijia Lin, Aviad Rubinstein, and Eran Tromer. The hunting of the snark. *J. Cryptol.*, 30(4):989–1066, October 2017.

[BFLS91] László Babai, Lance Fortnow, Leonid A. Levin, and Mario Szegedy. Checking computations in polylogarithmic time. In *Proceedings of the Twenty-third Annual ACM Symposium on Theory of Computing*, STOC '91, pages 21–32, New York, NY, USA, 1991. ACM.

[BSCGT13] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, and Eran Tromer. On the concrete efficiency of probabilistically-checkable proofs. In *Proceedings of the Forty-fifth Annual ACM Symposium on Theory of Computing*, STOC '13, pages 585–594, New York, NY, USA, 2013. ACM.

[Chi] Chia network. https://www.chia.net/. (Accessed on 04/15/2019).

[CP18] Bram Cohen and Krzysztof Pietrzak. Simple proofs of sequential work. In Jesper Buus Nielsen and Vincent Rijmen, editors, *Advances in Cryptology – EUROCRYPT 2018*, pages 451–467, Cham, 2018. Springer International Publishing.

[DFKP13] Stefan Dziembowski, Sebastian Faust, Vladimir Kolmogorov, and Krzysztof Pietrzak. Proofs of space. Cryptology ePrint Archive, Report 2013/796, 2013. https://eprint.iacr.org/2013/796.

[FS87] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In *Proceedings on Advances in cryptology—CRYPTO '86*, pages 186–194, London, UK, UK, 1987. Springer-Verlag.

[GS12] C.M. Grinstead and J.L. Snell. *Introduction to Probability*. American Mathematical Society, 2012.

[LW15] Arjen K. Lenstra and Benjamin Wesolowski. A random zoo: sloth, unicorn, and trx. Cryptology ePrint Archive, Report 2015/366, 2015. https://eprint.iacr.org/2015/366.

[Mic00] Silvio Micali. Computationally sound proofs. *SIAM J. Comput.*, 30(4):1253–1298, October 2000.

[MMV13] Mohammad Mahmoody, Tal Moran, and Salil Vadhan. Publicly verifiable proofs

of sequential work. In *Proceedings of the 4th Conference on Innovations in Theoretical Computer Science*, ITCS '13, pages 373–388, New York, NY, USA, 2013. ACM.

[Pie18] Krzysztof Pietrzak. Simple verifiable delay functions. Cryptology ePrint Archive, Report 2018/627, 2018. `https://eprint.iacr.org/2018/627`.

[Rab83] Michael O. Rabin. Transaction protection by beacons. *Journal of Computer and System Sciences*, 27(2):256 – 267, 1983.

[Val08] Paul Valiant. Incrementally verifiable computation or proofs of knowledge imply time/space efficiency. In *Proceedings of the 5th Conference on Theory of Cryptography*, TCC'08, pages 1–18, Berlin, Heidelberg, 2008. Springer-Verlag.

[Vdf] Vdf research. `https://vdfresearch.org/`. (Accessed on 04/15/2019).

[Wes18] Benjamin Wesolowski. Efficient verifiable delay functions. Cryptology ePrint Archive, Report 2018/623, 2018. `https://eprint.iacr.org/2018/623`.

[WSH+14] Riad S. Wahby, Srinath Setty, Max Howald, Zuocheng Ren, Andrew J. Blumberg, and Michael Walfish. Efficient ram and control flow in verifiable outsourced computation. Cryptology ePrint Archive, Report 2014/674, 2014. `https://eprint.iacr.org/2014/674`.

[Zca] Zcash. `https://z.cash/`. (Accessed on 04/15/2019).