Cryptographic timestamping through sequential work

Esteban Landerreche, Christian Schaffner, and Marc Stevens

CWI Amsterdam
contact: esteban@cwi.nl

Abstract. We present a definition of an ideal timestamping functionality that maintains a timestamped record of bitstrings. The functionality can be queried to certify the record and the age of each entry at the current time. An adversary can corrupt the timestamping functionality, in which case the adversary can output its own certifications of the record and age of entries under strict limitations. Most importantly, the adversary initially cannot falsify any part of the record, but the maximum age of entries the adversary can falsify grows linearly over time.
We introduce a single-prover non-interactive cryptographic timestamping protocol based on proofs of sequential work. The protocol securely implements the timestamping functionality in the random-oracle model and universal-composability framework against an adversary that can compute proofs of sequential work faster by a certain factor. Because of the computational effort required, such adversaries have the same strict limitations under which they can falsify the record as under the ideal functionality. This protocol trivially extends to a multi-prover protocol where the adversary can only generate malicious proofs when it has corrupted at least half of all provers.
As an attractive feature, we show how any party can efficiently borrow proofs by interacting with the protocol and generate its own certification of records and their ages with only a constant loss in age.
The security guarantees of our timestamping protocol only depend on how long ago the adversary corrupted parties and on how fast honest parties can compute proofs of sequential work relative to an adversary, in particular these guarantees are not affected by how many proofs of sequential work honest or adversarial parties run in parallel.

Keywords: blockchain, immutability, proof-of-sequential-work, time-lock cryptography, timestamping

1 Introduction

There are many use cases to be able to certify that a certain message was recorded at a certain time. Bayer, Haber and Stornetta considered time-stamping digital documents as well as digital signatures [13]. While timestamping evokes the idea of fixing an event in a specific point in clocktime, in practice most timestamping schemes are relative; they provide an ordering of events. Clock-based timestamping generally requires stronger trust assumptions, as it is problematic to encode actual time, so systems that provide this make use of a trusted party provide the actual timestamps. The main advantage of this second type of timestamping is the ability to compare the timestamps to events outside the protocol. A known construction for clock-based timestamping encodes events into Bitcoin transactions, effectively using the blockchain as a trusted third party. Inspired by the *immutability* of proof-of-work blockchains, we construct a non-interactive protocol for timestamping based on the use of sequential work as a way to encode time.

Interestingly, the immutability of blockchains can itself be seen as a timestamping problem. The proof-of-work based consensus of Bitcoin indirectly provides timestamping guarantees (with certain limitations) Informally, adversaries that do not have a majority of the total hashing power invested in the proof-of-work can succeed at rewriting previous blocks in the network only with success probability that is exponentially small in the depth of rewritten blocks. Moreover, even against adversaries with a fraction $\alpha > 0.5$ of the total hashing power, the adversary is computationally restricted in how deep it can rewrite blocks in a certain amount of time, *i.e.* on average after time T the adversary can only successfully rewrite blocks in the network up to $T \cdot \alpha/(1-\alpha)$ time deep. Even under total adversarial control, the blockchain cannot be arbitrarily rewritten. We are interested in replicating this *resilience to adversarial corruption* for timestamps without incurring in the costs and complications found in a PoW blockchain setting.

In our cryptographic timestamping protocol we make use of inherently sequential or non-parallelizable functions which have a long history in cryptography, where they are also known as time-lock puzzles. Such puzzles have been used for timed-release encryption [24], as way to achieve pseudonymous authentication [14] and to create non-malleable commitments [18]. Generally these functions are based on modular arithmetic but other proposals have appeared [19,20] which have a linear gap between creating the puzzle and solving it. Our use of these puzzles will be opposite to the common usage, where a puzzle is generated together with a known solution, whereas we want a puzzle that requires time to find the pseudo-random solution for but which is quick to verify. Such slow functions, which we will call proofs of sequential work, have been postulated before in order to generate publicly verifiable randomness [17,22].

1.1 Our contributions

In this work we present a definition of an ideal timestamping functionality $\mathcal{F}^{\alpha}_{\texttt{timestamp}}$ that maintains a timestamped record of bitstrings, which can be queried to generate a transferable proof of the age of each entry at the time the proof was generated. The functionality continues to perform its goal even under adversarial corruption, as the adversary will only be allowed to forge proofs under certain constraints. The adversary has a diluting factor that lower-bounds the time required for it to generate a proof with a malicious entry with a falsified age.

We present a single party timestamping protocol based on proofs of sequential work (PoSW) which takes ideas from blockchains. The prover will provide non-interactive PoSW to a verifier in order to attest the age of a bitstring. The ability to create fake proofs will depend solely on the ability to compute PoSW, which can only be achieved through a faster single processing core. We show this protocol securely implements the timestamping functionality in the random oracle model and universal composability framework against an adversary that can compute proofs of sequential work faster than the prover by a diluting factor. Our timestamping mechanism differs from other proposals as it does not simply provide an ordering but can also provide *absolute* timestamping based on assumptions of computational power. This trivially extends to a multi-party protocol where the adversary can only generate malicious proofs when it has corrupted at least half of all parties.

Furthermore, our proofs are publicly verifiable and can be used by parties who do not compute the proofs of sequential work themselves. We show how any party can efficiently borrow proofs by interacting with the protocol and generate their own timestamps without doing the computational work themselves in exchange for a loss in age. Thus our work may benefit any blockchain protocol that by interacting with the (single party or multi party) timestamping protocol can efficiently maintain a cryptographically verifiable proof of the age of its blockchain. This may seem to add a level of centralization to any blockchain. However, this is arguably less than the centralization implied by relying on a core development team, since one can easily switch to a different instantiation of the timestamping protocol without invalidating the borrowed proofs of age so far if the need or desire to do so arises. We believe this is a resource-cheap solution to costless simulation attacks on non-proof of work consensus protocols [23].

The security guarantees of our timestamping protocol only depend on how long ago the adversary corrupted parties and on how fast honest parties can compute a single proof of sequential work relative to an adversary, in particular these guarantees are not affected by how many proofs of sequential work honest or adversarial parties run in parallel.

1.2 Related Work

Proving the age of a digital document presents many more challenges than doing it in the physical world. The first paper to deal with digital timestamping is [13] which presents two different solutions that avoid having to fully trust a third party to validate timestamps. Their first solution uses a hashchain: an sequence of documents linked through a collision-resistant hash function. The second solution relies on a pseudorandom function to choose a set of validators for a timestamp, which also allows to identify misbehaving actors. Both solutions require interaction with distributed validators, as a different hashchain can be easily generated by an adversarial party.

Modifications and extensions of these protocols have appeared, most notably substituting the individual records in a hashchain with a Merkle tree (or similar) [2,5]. These timestamping systems require many parties to mantain records and their timestamps, as well as the availability to answer validation queries. Different security properties have been proved for these systems, including the fact that collision-resistance is not enough for security [6]. Other protocols do not require so much space, but they require stronger trust assumption on the set of validators [3]. In the classical literature, every timestamping service requires interaction with a group of validators and provides security guarantees only to relative timestamping.

Haber and Stornetta's hashchain protocol served as a fundamental building block ifor the Bitcoin blockchain [21]. Nakamoto's blockchain consists of a hashchain where new elements of the chain, or blocks, are added if and only if they solve a cryptographic puzzle known as a proof-of-work. The work required to create a new block ensures that the blockchain cannot be easily rewritten, especially for block long in the past [11]. The protocol is tuned in order to create a block every ten minutes in average, meaning that one can broadly associate each block to a ten minute slot. This can allow for timestamping in the absolute sense and not only relatively (although with certain limitations as this is not its primary purpose[1]). Current constructions that utilize the blockchain to provide timestamping [12,26] simply treat it as a trusted party and do not formally prove security. The use of computational work to encode time is interesting for timestamping purposes, but constructing a system based on Bitcoin is problematic as it requires a significant investment in computational resources which come with cost and sustainability concerns [1]. In essence, the Bitcoin blockchain is simply treated as a trusted third party in this construction so no formal security guarantees are provided.

Proofs of work can be computed in parallel, making it very problematic to find a link between work done and clock time, making it impossible without additional assumptions (like in Bitcoin). However, if the work can only be performed sequentially it could be a source for timestamping. In order to connect computational work with time, we must ensure that the work can only be carried out sequentially. In [19], the authors present the concept of a *proof of sequential work* (PoSW) in order to verify that a number of computation steps have happened since something existed. It is inspired in time-released cryptography [24], which attempts to encrypt messages in such a way that they can be decrypted by anyone after a certain amount of time has passed. Further work has characterized and optimized these proofs [20,9], which are based on directed acyclic graphs. An example of a proof of sequential work not based in graphs can be found in which is based on randomized encodings [4]. In a different context, [17] also presents a function that may be used as a proof of sequential work based on the difference in the time needed to compute a modular square root and a simple squaring.

1.3 Proofs of Sequential Work (PoSW)

Informally, proofs of sequential work are proofs that some long and inherently sequential computation was performed, whereas any verifier can quickly verify the correctness of the proof. More formally, we consider a (non-interactive) proof of sequential work to be a pair of algorithms $(\mathsf{PoSW.gen}, \mathsf{PoSW.verify})$ with security parameter μ and parameters $g, v \in \mathbb{N}$ as defined below.

$\mathsf{PoSW.gen}(x, s)$ is a slow cryptographic algorithm that for an input $x \in \{0,1\}^*$ and strength $s \in \mathbb{N}$ computes an output $(p, s) \in \{0,1\}^\mu \times \mathbb{N}$ in $s \cdot g$ parallel time steps.

$\mathsf{PoSW.verify}(x, p, s)$ is a fast cryptographic algorithm that for inputs $x \in \{0,1\}^*$, $p \in \{0,1\}^\mu$, and $s \in \mathbb{N}$ outputs accept if $(p, s) = \mathsf{PoSW.gen}(x, s)$, and reject otherwise, in at most $s \cdot v$ time steps.

We will use a canonical unambiguous encoding of integers $s \in \mathbb{N}$ as bitstrings, so (p, s) has a natural description as a bitstring $p||s \in \{0,1\}^*$

We require perfect **correctness**: $\mathsf{PoSW.verify}(x, \mathsf{PoSW.gen}(x, s)) = \mathsf{accept}$ for all $x \in \{0,1\}^*$ and $s \in \mathbb{N}$. The PoSW is called **secure** if no efficient adversary given an input x with sufficient min-entropy can compute values (s, p) in less than $s \cdot g$ parallel time steps for which $\mathsf{PoSW.verify}(x, p, s) = \mathsf{accept}$ with non-negligible probability. The **usability** of the PoSW is the factor g/v by which verification is faster than generation of the proof.

We will also use an extended notion of proof of sequential work to make it variable time, where one does not have to choose the strength in advance, but whose strength continuously increases with time spent computing it. A variable-time non-interactive proof of sequential work is a proof of sequential work defined as above with an additional algorithm $\mathsf{PoSW.extend}$:

[1] `http://culubas.blogspot.nl/2011/05/timejacking-bitcoin_802.html`

$\mathsf{PoSW.extend}$ is a slow cryptographic algorithm that for inputs $x \in \{0,1\}^*$, $(p, s) = \mathsf{PoSW.gen}(x, s)$ and $s^* \in \mathbb{N}$ returns the output $(p^*, s + s^*)$, where $(p^*, s + s^*) = \mathsf{PoSW.gen}(x, s + s^*)$, in $s^* \cdot g$ parallel time steps.

A candidate construction that may satisfy this notion is the sloth construction by Lenstra and Wesolowski [17] that iterates modular square root and (keyed) binary permutation functions.

Our definition for proofs of sequential works differs from other constructions [20,9] in two main ways: non-interactivity and extensibility. Proofs of sequential work are generally presented through a protocol where a verifier issues a challenge to a prover who will first commit to and subsequently proved that they executed the necessary work. Our choice of function avoids the need for two phases, as the only the input and output of the function are needed for verification. While other PoSW schemes can be made non-interactive by the Fiat-Shamir transform, sloth is non-interactive by definition which allows for succint proofs. We additionally present PoSW that can be arbitrarily extended. This property is relevant for timestamping purposes as we want to encode all the time since something was stamped, which means that our proofs need to be able to grow arbitrarily. In fact, we will achieve timestamping by concatenating proofs, but extensibility will allow flexibility when applied to blockchain protocols.

We are aware of the shortcomings of the sloth function as a proof of sequential work, as presented in [9]. While the computation-verification gap is only logarithmic, the proof is unique, publicly verifiable and consists of a single string in $\{0,1\}^\mu$. These properties allow for efficient *proof borrowing* where only the input and the proof (including the strength s) are necessary for anyone to verify that the computation happened. Even if sloth is computed for an arbitrary number of steps, the proof will always be of the same size (checkpoints may be added for efficiency but are not necessary to verify), which is not the case for other functions The fact that a puzzle and a solution cannot be sampled simultaneously makes sloth unfit for time-release cryptography while simultaneously allowing it to be secure over our definition (if there is no challenge a prover could simply sample a puzzle-solution pair and send the solution to the verifier).

In contrast with other presentations of proofs of sequential work, we will deal with real-world time and not only with computational steps. For this, we will assume that parties have access to a PoSW-rate γ, which encodes the number of time steps needed to compute a PoSW of strength 1. In practice, the assumption that party can make computations at a certain clock-speed and not faster requires additional conditions. However, we are comfortable with this construction due to the current state of processor technology. Current advances in hardware are based on speeding up computers through parallelization, which provides no advantages for PoSWs. This allows us to estimate the fastest possible realistic rate. In Section 5 we present a setting where we believe our assumptions hold true by having an entity which has access to enough sequential computational power to provide a public *immutability beacon*.

2 Model and Definitions

Interactive Turing Machines We will work in the Interactive Turing Machine (ITM) model presented in [7] where two PPT algorithms $\mathcal{Z}$ and $\mathcal{A}$ interact with parties executing a protocol. We will assume a hybrid model where parties have access to random oracles, an unforgeable signature functionality and the $\mathcal{F}_\gamma^{\mathsf{PoSW}}$ functionality that will represent our proofs of sequential work.

Time and network. We consider a setting where time is essentially continuous, but it may be divided into intervals of time of a certain length which will be context-dependent. For instance, when a party computes a certain slow function at a rate of γ, then a time-step for this process will be $1/\gamma$ long, but for rounds of a (network) protocol this may be a pre-agreed length of time. Parties are equipped with synchronized clocks with at most an insignificant difference in time with respect to rounds of network protocols. We assume that timestamps can be described in bitstrings of length θ at a sufficient granularity.

Where applicable, we assume that the network graph of honest parties is well connected and that the length of the time window for each round of a network protocol is sufficient to guarantee that any message transmitted by an honest party to any other honest party at the beginning of the round will be received by the end of the round.

Public-key signatures. We assume a public-key infrastructure for digital signatures with security parameter κ which we model as a global ideal signing functionality Σ. It generates private-public key pairs for each party. Parties can interact with Σ in the following way:

- Query public key pk_j of party $\mathcal{P}_j$.
- On query $\Sigma.\mathsf{sign}(pk_j, msg)$ (or $\Sigma.\mathsf{sign}(j, msg)$):
  If this was not input by $\mathcal{P}_j$ (or the adversary if $\mathcal{P}_j$ is corrupted), ignore. Otherwise, generate $sig \in \{0,1\}^\kappa$. If $(j, msg, sig, \mathsf{reject})$ was recorded, **halt**. Otherwise, output sig and save $(j, msg, sig, \mathsf{accept})$.
- On query $\Sigma.\mathsf{verify}(pk_j, msg, sig)$ (or $\Sigma.\mathsf{verify}(j, msg, sig)$):
  If Σ has recorded some (j, msg, sig, x), output x.
  Otherwise, record $(j, msg, sig, \mathsf{reject})$ and output reject.

The adversary can query the functionality to corrupt parties, in which case the adversary can also query signatures on behalf of the corrupted parties from then on. We will assume that Σ is existentially unforgeable prior corruption. In order to realize this we will add a restriction that whenever the adversary attempts to forge a signature sig for a message msg for a party $\mathcal{P}_j$, *e.g.* whenever it sets the signature field of a data structure to some bitstring, the adversary must query $\Sigma.\mathsf{verify}(j, msg, sig)$ at that time.

Cryptographic hash function. Let $\mathsf{H} : \{0,1\}^* \to \{0,1\}^\lambda$ be a collision-resistant cryptographic hash function, which we model as a random oracle.

Merkle Trees. Merkle Trees are balanced binary trees, where the ordered leaf nodes are each labeled with a bitstring, and where each non-leaf node has two child nodes and is labeled by the hash of its children's labels. The root hash of a Merkle Tree equals the label of the root node. Merkle Trees allow for short set membership proofs of length $O(log(N))$ for a set of size N. For convenience we define some interface functions that deal with Merkle Trees in some canonical deterministic way.

$\mathsf{MT.root}(T)$ computes the root hash h of the Merkle Tree for some ordered finite sequence $T \in (\{0,1\}^*)^*$ of bit strings and outputs $h \in \{0,1\}^\lambda$.

$\mathsf{MT.path}(T, v)$ outputs the Merkle path described as a sequence of strings $(x_0, \ldots, x_l)$ where $x_0 = v$, $x_l = \mathsf{MT.root}(T)$, $x_i \in \{0,1\}^\lambda$ and either $x_{i+1} = \mathsf{H}(x_i || \mathsf{H}(x_{i-1}))$ or $x_{i+1} = \mathsf{H}(\mathsf{H}(x_{i-1}) || x_i)$ for all $i > 0$.

$\mathsf{MT.verify}(P)$ given an input sequence $P = (x_0, \ldots, x_l)$ outputs accept if P is a valid Merkle path. It outputs reject otherwise.

With a slight abuse of notation we also use $\mathsf{MT.root}(T)$ recursively, *i.e.*, if one of the elements S of T is not a bitstring but a set or sequence, we use $\mathsf{MT.root}(S)$ as the bitstring representing S. E.g., if $T = (a, b, S)$ with bitstrings $a, b \in \{0,1\}^*$ and a set of bitstrings $S = \{c, d, e\}$, then $\mathsf{MT.root}(T) = \mathsf{MT.root}((a, b, \mathsf{MT.root}(S)))$. This similarly extends to $\mathsf{MT.path}(T, v)$, *e.g.*, where $v \in S$ in the previous example.

For convenience, sometimes we will use $\mathsf{MT}(T)$ as a short-hand for $\mathsf{MT.root}(T)$.

Proof of sequential work functionality. In this work we will assume that every party and the adversary have access to certain computational resources (a CPU running at some clock speed) or some specific optimizations which implies that they each can compute proofs of sequential work at certain (potentially distinct) rates γ. So for every party we model their capability to compute PoSW as a slow oracle $\mathcal{F}_\gamma^{\mathsf{PoSW}}$ interfacing with a global random oracle as defined below.

Oracle $\mathcal{F}_\gamma^{\mathsf{PoSW}}$

The functionality is parametrized by a computation rate $\gamma > 0$. Let $\mathsf{PoSW} : \{0,1\}^* \times \mathbb{N} \to \{0,1\}^\mu$ be a global random oracle each oracle instance has access to. The oracle also has access to a global clock clock (to exactly measure time elapsed computing the proof of sequential work). Let $Q := \varnothing$ be the (initially empty) query log.

- Upon receiving (start, x) at time t, update $Q \leftarrow Q \cup \{(x, t)\}$.
- Upon receiving (output, x) at time $t_o = \mathsf{clock}()$:
  Let t_i be the earliest time such that $(x, t_i) \in Q$, return $\perp$ if there is no such t_i;

Let $s := \lceil (t_o - t_i) \cdot \gamma \rceil$ be the strength of the resulting proof and $p := \mathsf{PoSW}(x, s)$;
Return (p, s) at time $t_i + s/\gamma = t_o + \epsilon$, with $\epsilon < 1/\gamma$.

– Upon receiving $(\mathsf{verify}, x, p, s)$, return accept if $\mathsf{PoSW}(x, s) = p$ and reject otherwise.

The main security property of this functionality is unforgeability. That is, no party can create a PoSW p that the functionality will accept after a $(\mathsf{verify}, x, p, s)$ query, as parties cannot query PoSW directly. This means that given a PoSW (p, s) with input x, there must have been a (start, x) call to instance of $\mathcal{F}_\gamma^{\mathsf{PoSW}}$ at least ${}^{s}/_{\gamma}$ time steps ago.

2.1 Random Oracle Sequences

In this work we will analyze recursive calls to the random oracle H and to the random oracle PoSW underlying all oracles $\mathcal{F}_\gamma^{\mathsf{PoSW}}$ and analyze the cumulative strength of the proofs of sequential work. We have adapted the following lemmas from [9] to this setting.

Lemma 2.1 (Random Oracles are Collision-Resistant). *Consider any adversary $\mathcal{A}^h$ given access to a random function $h : \{0,1\}^* \to \{0,1\}^n$. If $\mathcal{A}$ makes at most q queries, the probability it will make two colliding queries $h(x) = h(y)$ with $x \neq y$ is at most $q^2/2^{n+1}$.*

The above lemma applies independently both to H and PoSW, since we assume that their output spaces are disjoint as $\lambda \neq \mu$.

Definition 2.2 (H2-sequence). *Given functions $\mathsf{H} : \{0,1\}^* \to \{0,1\}^\lambda$ and $\mathsf{PoSW} : \{0,1\}^* \times \mathbb{N} \to \{0,1\}^\mu$, an H2-sequence of length l is defined as a sequence $S = ((x_0, s_0), \ldots, (x_l, s_l))$, where $(x_i, s_i) \in \{0,1\}^* \times (\mathbb{N} \cup \{\perp\})$ and the following holds for each $0 \leq i < l$: if $s_i = \perp$ then $\mathsf{H}(x_i)$ is contained in x_{i+1} as continuous substring; otherwise $s_i \in \mathbb{N}$ and $\mathsf{PoSW}(x_i, s_i)$ is contained in x_{i+1} as continuous substring. We call $\sum_{i=0,\ldots,l-1|s_i \neq \perp} s_i$ the strength of the $H2$-sequence. By $|S|$ we denote the total bitlength $\sum_{i=0}^{l} |x_i|$ of all bitstrings in S.*

it is simple to see that any[2] Merkle path $\mathsf{MT.path}(T, v) = (x_0, \dot{;} x_l)$ induces an $H2$-sequence of the form $\big((x_0, \perp), (x_1 || \mathsf{H}(x_0), \perp), \ldots, (x_{l-1} || x_{l-2}, \perp), (x_l, \perp)\big)$ of length $l + 1$. With an abuse of notation, we will refer to Merkle path $\mathsf{MT.path}(T, v)$ as an $H2$-sequence.

Definition 2.3 (linking H2-sequences). *We define* linking *$H2$-sequence $S_2 = \{(x_2, s_2), \widehat{\ldots}\}$ to $H2$-sequence $S_1 = \{\widetilde{\ldots}(x_0, s_0), (x_1, s_1)\}$ where x_1 is a continuous substring of x_2 to result in the $H2$-sequence $S_1 \bowtie S_2 = \{\widetilde{\ldots}, (x_0, s_0), (x_2, s_2), \widehat{\ldots}\}$.*

Note that the result of the query (x_0, s_0) is a continuous substring of x_1, and thus also a continuous substring of x_2, it follows that $\{\widetilde{\ldots}(x_0, s_0), (x_2, s_2)\}$ is a valid $H2$-sequence and by concatenating the rest of S_2 it follows that S is a valid $H2$-sequence.

Lemma 2.4 (Random Oracles are sequential). *Consider any adversary $\mathcal{A}^{(\mathsf{H},\mathsf{PoSW})}$ which is given a bitstring x_0 of sufficient min-entropy and access to two random functions $\mathsf{H} : \{0,1\}^* \to \{0,1\}^\lambda$ and $\mathsf{PoSW} : \{0,1\}^* \times \mathbb{N} \to \{0,1\}^\mu$ that it can query. If $\mathcal{A}$ makes at most q_1 queries of total length Q_1 bits to H and at most q_2 queries of total length Q_2 to PoSW, then the probability that it outputs an $H2$-sequence $(x_0, s_0), \ldots, (x_l, s_l)$ without making the queries $(x_0, s_0), \ldots, (x_{l-1}, s_{l-1})$ to respectively H and PoSW sequentially is at most*

$$2 \cdot \left(q_1 \cdot 2^{-\lambda} + q_2 \cdot 2^{-\mu}\right) \cdot \left(Q_1 + Q_2 + \sum_{i=0}^{l} |x_i|\right).$$

Proof. Constructing an $H2$-sequence without making the queries (x_0, s_0), …, (x_{l-1}, s_{l-1}) sequentially can happen when there is a cycle such that the adversary can repeat previous queries without making them again. I.e., there exist $0 \leq i \leq j < l$ such that $\mathsf{H}(x_j)$ (when $s_j = \perp$) or $\mathsf{PoSW}(x_j, s_j)$ (otherwise) is contained in x_i. This event can only arise when the output of a query is a substring of the input of a previous query. Using that outputs of H and PoSW are uniform randomly selected, the probability of a cycle is upper bounded by $q_1(Q_1 + Q_2)2^{-\lambda} + q_2(Q_1 + Q_2)2^{-\mu}$.

[2] w.l.o.g. we assume that every lement of the sequence starting with x_2 is of the form $x_i = x_{i-1} || x_{i-2}$

If there are no cycles then at least one query did not happen or did not happen after its dependent query. This event can only arise when an output y of a query to H or PoSW would be a continuous substring of some bitstring (one of the queried inputs or one of the x_j), whether or not the adversary actually made the query. As outputs of H and PoSW are uniform randomly selected, the probability of this event is upper bounded by $q_1(Q_1 + Q_2 + \sum_{i=0}^{l} |x_i|)2^{-\lambda} + q_2(Q_1 + Q_2 + \sum_{i=0}^{l} |x_i|)2^{-\mu}$.

The claimed bound follows from a union bound over these two events. □

Thus when an adversary outputs an $H2$-sequence of strength L where $2 \cdot (q_1 2^{-\lambda} + q_2 2^{-\mu}) \cdot (Q_1 + Q_2 + \sum_{i=0}^{l} |x_i|)$ is negligible, we can assume that it made all queries sequentially. (In practice this will certainly be the case for output lengths λ and μ of 256 bits and larger.) In particular, if the adversary can query $\mathsf{PoSW}(x,s)$ only through $\mathcal{F}_\gamma^{\mathsf{PoSW}}$ with a rate of γ then each query $\mathsf{PoSW}(x,s)$ will take time s/γ time. It follows that the adversary used at least L/γ time to construct the $H2$-sequence.

Note that our construction differs from the one in [9] as we aggregate all the calls to PoSW into one element of the sequence. We do this in order to distinguish the calls to different random oracles and more directly show the numbers of executions of PoSW. We will effectively treat calls to H as "free" with regards to time but we are still interested in having them has a part of our $H2$-sequences in order to include them into our proofs of age in the case of proof borrowing.

We will go a step further and add timestamps to our $H2$-sequences making them proofs of age. We will embed unchangeable timestamps into $H2$-sequences which we will then call $H2T$-sequences. These timestamps will be enforced by each PoSW in the sense that altering the timestamp will require to redo the PoSW.

Definition 2.5 **($H2T$-sequence).** *Let $S = ((x_0, s_0), \dots, (x_l, s_l))$ be an $H2$-sequence of length l. Let $I = \{0 \le i < l \mid s_i \in \mathbb{N}\} = \{i_0, \dots, i_k\}$ be the index set of the PoSWs in S. We call S an* ***$H2T$-sequence*** *if the following two properties hold:*

1. *For $i \in I$: $x_i = t_i || r_i$ where $t_i \in \{0,1\}^\theta$ is a timestamp.*
2. *For $i, j \in I$: if $i < j$ then $t_i < t_j$.*

If $I \neq \varnothing$, we call the first timestamp t_{i_0} the root time *of S.*

3 LIPWIG – Leaning on Impossible-to-Parallelize Work for Immutability Guarantees

We will now present our timestamping construction. Our primary goal will be to show that proofs of sequential work give similar immutability guarantees as a trusted third-party. Additionally, the proofs are still reliable even when the prover is adversarial up to a linear factor over the adversarial prover's *hidden* rate. In essence, our construction enhances the hashchain from [13] in order to provide a notion of absolute timestamping to it. We will speak of blockchains instead of hashchains as we believe that our construction can be applied to this domain, in particular to solve long-range attacks. This construction can be enhanced in a decentralized setting, where many individual instances of the protocol are being executed in parallel and interact with each other. In this case, verification cost can be reduced while reliability increases.

3.1 Ideal timestamping functionality

Now we will present our definition of the ideal timestamping functionality $\mathcal{F}^\alpha_{\texttt{timestamp}}$. The environment $\mathcal{Z}$ will interact with the functionality through dummy parties $\mathcal{P}$ and verifier $\mathcal{V}$, which can be seen as interfaces of the functionality. We will also define the extent of an adversary's ability to influence the functionality in order to make it output false statements.

Informally, the $\mathcal{P}$ interface is used to record the messages to be timepestamped by the functionality, while queries regarding the age of all recorded messages are input to $\mathcal{V}$. When $\mathcal{A}$ 'corrupts' the functionality, it is allowed to choose which messages to output after $\mathcal{V}$ is queried as well as being able to record any message. The adversary can also change the ages of certain messages, bounded by the amount of time that the functionality has been under $\mathcal{A}$'s control based on a parameter α. If the functionality has been under adversarial control for T_{corr} time steps, then the adversary can only forge timestamps to look younger than $\alpha \cdot T_{corr}$ time steps ago. Essentially, either every record older than some age $\alpha \cdot T_{corr}$ in the honest record should be contained in the falsified record or none (modelling that the adversary deletes the old record and started from scratch). Moreover, $\mathcal{A}$ can decide to include each entry younger than age $\alpha \cdot T_{corr}$ or not, but each included entry can have age at most α times its honest age and at most age $\alpha \cdot T_{corr}$.

More formally, we define $\mathcal{F}^{\alpha}_{\texttt{timestamp}}$ as the ideal timestamping functionality as follows:

Ideal timestamping functionality $\mathcal{F}^{\alpha}_{\texttt{timestamp}}$

The functionality is parametrized with an adversarial diluting factor $\alpha \geq 1$. It defines the allowed actions of a player $\mathcal{P}$, the verifier $\mathcal{V}$ and the adversary $\mathcal{A}$. The internal variables are initialized to $k := 0$, $corr := 0$, $T_{corr} := 0$. It maintains an internal list for (record,timestamp)-tuples.

– **Record:** Upon receiving ($\texttt{record}, c \in \{0,1\}^*$) at time t from $\mathcal{P}$, record entry $(c_k, t_k) \leftarrow (c, t)$ and set $k \leftarrow k+1$. As records are public knowledge, (c, t) is also returned to $\mathcal{A}$ (irrespective of $corr$).
– **Corrupt:** Upon receiving `corrupt` at time t from $\mathcal{A}$, set $corr \leftarrow 1$ and $T_{corr} \leftarrow t$.
– **Prove:** Upon receiving `prove` at time t from $\mathcal{V}$:
1. Let $\pi_t := \{(c_i, a_i) \mid 0 \leq i < k, a_i = t - t_i\}$ be the set of all records with corresponding ages.
2. If $corr = 0$, then **return** π_t to $\mathcal{V}$.
3. Otherwise, send π_t to $\mathcal{A}$ and receive π'_t back from $\mathcal{A}$.
4. Parse $\{(c_i, a'_i) \mid i \in I \subseteq \{0, \ldots, k-1\}\} \leftarrow \pi'_t$. If parsing fails (e.g., because a c_i value is different from originally recorded), return reject to $\mathcal{V}$.
5. Let $A := (t - T_{corr}) \cdot \alpha$ be the (diluted) time since corruption and $\pi^A := \{(c_i, a_i) \in \pi_t \mid a_i > A\}$ the subset of records which are older than A.
6. Among the records in π_t older than A, either none or all of them have to be present in π'_t. Formally, if $\pi'_t \cap \pi^A \notin \{\varnothing, \pi^A\}$, then return reject to $\mathcal{V}$.
7. If for any $i \in I$, a'_i has been modified by more than α (i.e., $a'_i > \alpha \cdot a_i$) or a'_i has been modified beyond A (i.e., $a_i \neq a'_i$ and $a'_i > A$), then return reject to $\mathcal{V}$.
8. Otherwise, **return** π'_t to $\mathcal{V}$.

Corruption of $\mathcal{F}^{\alpha}_{\texttt{timestamp}}$ allows $\mathcal{A}$ to have certain control over the output of the functionality after a `prove` query. It also allows the adversary to see the entire list of records maintained by the functionality and the ability to add new ones. However, $\mathcal{A}$ is not allowed to erase any records or to (directly) modify the timestamp of any message (even its own). The adversary is only allowed to affect the timestamps of a message in the output of a `prove` query and is limited in how much it can stretch the timestamps by α. $\mathcal{A}$ is also forced to record every content that it wishes to timestamp, otherwise a proposed proof will fail in step 4.

3.2 Single Party PoSW Blockchain

In this section we will define our cryptographic timestamping protocol that implements the ideal timestamping functionality. To this end we will combine timestamped proofs-of-sequential-work together with a blockchain structure and digital signatures. The blockchain structure will imply that if one block is changed then all subsequent blocks must also be changed, and that will allow us to extract $H2T$-sequences. Whereas we use digital signatures solely to prevent the adversary from maintaining its own valid chain before it has corrupted the party.

Definition 3.1 (SP-PoSW-block). *We define a* SP-PoSW-block *for a party $\mathcal{P}$ with public key $pk \in \{0,1\}^*$ as a tuple $B = (data, sig)$, where $data = (pk, rnd, prev, posw, content)$ and*

1. *$rnd \in \mathbb{N}$ is the sequence number of the block;*
2. *$prev \in \{0,1\}^*$ is the root hash $\mathsf{MT.root}(B_{rnd-1})$ of the previous block B_{rnd-1}, or $prev = \mathsf{H}(pk)$ when $rnd = 0$;*
3. *$posw = ((t||prev), (p||s))$ represents the proof of sequential work, where $(p, s) = \mathsf{PoSW}(t||prev, s)$ and $t \in \{0,1\}^\theta$ is a timestamp the PoSW was started,* i.e. *when the previous block was finished;*
4. *$content \in \{0,1\}^*$ represents some content;*
5. *$sig \in \{0,1\}^*$ is a digital signature on $\mathsf{MT.root}(data)$,* i.e., *$\Sigma.\mathsf{verify}(pk, data, sig) = \mathsf{accept}$.*

For convenience we will use the notation $B.data$, $B.sig$ as well as $B.pk$, $B.rnd$, $B.prev$, $B.posw$, $B.t$, $B.s$, $B.p$ and $B.content$ to refer to these elements in block B.

Definition 3.2 (SP-PoSW-chain). *We define a* SP-PoSW-chain *for a party $\mathcal{P}$ with public key $pk \in \{0,1\}^*$ as a sequence of SP-PoSW-blocks $C = (B_0, \dots, B_k)$ where for all $0 \leq i \leq k$:*

1. $B_i.pk = pk$;
2. $B_i.rnd = i$;
3. $B_0.prev = \mathsf{H}(B_0.pk)$ *and* $B_i.prev = \mathsf{MT.root}(B_{i-1})$ *for* $i > 0$;
4. $B_i.posw = ((B_i.t || B_i.prev), (p_i || s_i))$ *and* $(p_i, s_i) = \mathsf{PoSW}(B_i.t || B_i.prev, s_i)$;
5. $B_i.t < B_j.t$ *for all* $i < j \leq k$;
6. $\Sigma.\mathsf{verify}(pk, \mathsf{MT.root}(B_i.data), B_i.sig) = \mathsf{accept}$;

Let $\mathsf{len}(C) = k$ *be the* ***length*** *of C. We define the notations $C[i] = B_i$ for block indexing, $C[i, r) = (B_i, \dots, B_{r-1})$ for subchains, and $\mathsf{last}(C) = B_k$ for the last block of C.*

Note that our definition allows us to extract an *H2T*-sequence $S = ((x_0, s_0), \dots, (x_l, s_l))$ starting from any $\mathsf{MT.root}(B_i)$ as follows:
Initialize empty sequence $S = ((\mathsf{MT.root}(B_j), \perp))$ and for $j = i, \dots, k-1$ do:

- Replace $(\mathsf{MT.root}(B_j), \perp)$ at the end of S with $(B_{j+1}.t || \mathsf{MT.root}(B_j), B_{j+1}.s)$ followed by $(B_{j+1}.p || B_{j+1}.s, \perp)$;
- Link the Merkle path $\mathsf{MT.path}(B_{j+1}, B_{j+1}.p || B_{j+1}.s) = \{B_{j+1}.p || B_{j+1}.s, \dots, \mathsf{MT.root}(B_{j+1})\}$ to S;

This allows a party to attest a certain age to any content included in block B_i by prepending the Merkle path from that content to $\mathsf{MT.root}(B_i)$. In between blocks, the party appends its running PoSW over $\mathsf{MT.root}(B_i)$ up to that point, which at the same time is also continued until the creation of the next block since we use a variable-time PoSW.

3.3 SingleLipwig protocol

We define the following algorithm SingleLipwig which allows player $\mathcal{P}$ to maintain a SP-PoSW-chain.

Algorithm SingleLipwig

Assume that party $\mathcal{P}$ with public key pk has access to H, $\mathcal{F}_\gamma^{\mathsf{PoSW}}$ for PoSW-rate $\gamma > 0$. $\mathcal{P}$ initializes $i := 0, prev := \mathsf{H}(pk)$, $t_0 := \mathsf{clock}()$, sends $(\mathsf{start}, t_0 || prev)$ to $\mathcal{F}_\gamma^{\mathsf{PoSW}}$.

```
– Upon receiving (record, $content$) from the environment $\mathcal{Z}$, $\mathcal{P}$ does the following:
    1. retrieve $(p_i, s_i)$ by sending $(\mathsf{output}, t_i || prev)$ to $\mathcal{F}_\gamma^{\mathsf{PoSW}}$
    2. query $sig_i \leftarrow \Sigma.\mathsf{sign}(pk, \mathsf{MT.root}(data_i))$,
       where $data_i = (pk, i, prev, ((t_i || prev), (p_i || s_i)), content)$
    3. set $B_i \leftarrow (data_i, sig_i)$, $prev \leftarrow \mathsf{MT.root}(B_i)$, and $t_{i+1} \leftarrow \mathsf{clock}()$
    4. send $(\mathsf{start}, t_{i+1} || prev)$ to $\mathcal{F}_\gamma^{\mathsf{PoSW}}$
    5. set $i \leftarrow i + 1$
– Upon receiving prove, $\mathcal{P}$ does the following:
    1. retrieve $(p'_i, s'_i)$ by sending $(\mathsf{output}, t_i || prev)$ to $\mathcal{F}_\gamma^{\mathsf{PoSW}}$
    2. query $sig'_i \leftarrow \Sigma.\mathsf{sign}(pk, \mathsf{MT.root}(data'_i))$,
       where $data'_i = (pk, i, prev, ((t_i || prev), (p'_i || s'_i)), \varnothing)$
    3. set $\widehat{B_i} \leftarrow (data'_i, sig'_i)$ and $t'_{i+1} \leftarrow \mathsf{clock}()$
    4. send to $\mathcal{V}$ the SP-PoSW-chain $C_i = (B_0, \dots, B_{i-1}, \widehat{B_i})$ together with time $t'_{i+1}$
```

Note that there is some time between steps 1 and 4 in which the proof of sequential work is not being executed. Therefore, this time is not encoded by our PoSW. We will refer to this "wasted" time as the *PoSW-interrupt time* of $\mathcal{P}$ and denote it as ϵ. In this case, the effect of ϵ is negligible but whenever there are multiple parties this may not be the case, as ϵ can be larger whenever parties need to reach consensus or, as seen in the following chapter, they have different block schedules.

In order to realize the $\mathcal{F}_{\mathtt{timestamp}}^{\alpha}$ functionality, we will construct a $\Pi_{\mathsf{SingleLipwig}}^{\gamma,\varepsilon}$ *protocol*, in which a player $\mathcal{P}$ interacts with an uncorruptible verifier $\mathcal{V}$. The verifier will ensure that the blockchain maintained by $\mathcal{P}$ is well-constructed and that the proofs-of-sequential-work actually encode the time represented in the timestamps.

(γ, ϵ)-SP-PoSW-chain-verifier $\mathcal{V}$

The verifier $\mathcal{V}$ interacts with party $\mathcal{P}$ with public key pk. The verifier's only role is the following:

1. $\mathcal{V}$ sends `prove` to $\mathcal{P}$ and receives back a SP-PoSW-block sequence $C = (B_0, \dots, B_k)$ together with its output time T.
2. $\mathcal{V}$ checks all conditions from Definition 3.2 and returns reject if C is not a valid SP-PoSW-chain for public key pk.
3. For $0 \leq i \leq k$, let $t_i := B_i.t$ and $t_{k+1} := T$.
4. For every block B_i, if the rate implied by the time stamps is too small ($B_i.s/(t_{i+1} - t_i - \epsilon) < \gamma$), $\mathcal{V}$ returns reject.
5. Otherwise, $\mathcal{V}$ returns $\{(B_i.content, T - t_{i+1}) \mid 0 \leq i < k\}$.

$\Pi^{\gamma,\varepsilon}_{\mathsf{SingleLipwig}}$ Protocol

The environment $\mathcal{Z}$ interacts with two parties: a (γ, ϵ)-SP-PoSW-chain-verifier $\mathcal{V}$ and a prover $\mathcal{P}$ running $\mathsf{SingleLipwig}$ with access to Σ, $\mathcal{F}^{\mathsf{PoSW}}_{\gamma}$ and PoSW-interrupt time ϵ. The adversary $\mathcal{A}$ has access to $\mathcal{F}^{\mathsf{PoSW}}_{\gamma_\mathcal{A}}$ (with $\gamma_\mathcal{A} \geq \gamma$), can communicate with $\mathcal{Z}$ and corrupt $\mathcal{P}$ at any moment by inputting `corrupt`. After corruption, $\mathcal{A}$ gains total control of $\mathcal{P}$. The environment $\mathcal{Z}$ can interact with the parties in the following ways:

– Input (`record`, *content*) to $\mathcal{P}$
– Input `prove` to $\mathcal{V}$

In order to prove that the protocol realizes the timestamping functionality we will first define our simulator $\mathcal{S}$.

$\mathcal{F}^{\alpha}_{\mathtt{timestamp}}$ simulator $\mathcal{S}$ for $\Pi^{\gamma,\varepsilon}_{\mathsf{SingleLipwig}}$

Given a $\Pi^{\gamma,\varepsilon}_{\mathsf{SingleLipwig}}$ adversary $\mathcal{A}$ and an instance of $\mathcal{F}^{\alpha}_{\mathtt{timestamp}}$, the simulator spawns an instance of $\Pi^{\gamma,\varepsilon}_{\mathsf{SingleLipwig}}$ with a party $\mathcal{P}$ with public key pk who constructs a blockchain and a dummy verifier $\mathcal{V}$ who simply forwards messages.. The adversary is given access to $\mathcal{F}^{\mathsf{PoSW}}_{\gamma_\mathcal{A}}$ with $\gamma_\mathcal{A} = \alpha/\gamma$. The adversary is also allowed a PoSW-interrupt time of ϵ/α.

– Upon receiving (`record`, *content*) from the environment $\mathcal{Z}$, $\mathcal{S}$ does the following:
  1. Input (`record`, *content*) to $\mathcal{F}^{\alpha}_{\mathtt{timestamp}}$ and to $\mathcal{P}$
– Upon receiving `corrupt` from the adversary $\mathcal{A}$, $\mathcal{S}$ does the following:
  1. Gives total control of $\mathcal{P}$ to $\mathcal{A}$ and input `corrupt` to $\mathcal{F}^{\alpha}_{\mathtt{timestamp}}$
  2. Whenever $\mathcal{A}$ queries `start`(x) $\mathcal{F}^{\mathsf{PoSW}}_{\gamma}$ or $\mathcal{F}^{\mathsf{PoSW}}_{\gamma_\mathcal{A}}$, if x contains the response of a $\Sigma.\mathsf{sign}$ query, input (`record`, x) to $\mathcal{F}^{\alpha}_{\mathtt{timestamp}}$
– Upon receiving `prove` from the environment $\mathcal{Z}$, $\mathcal{S}$ does the following:
  1. Set $T \leftarrow \mathsf{clock}()$ and input `prove` to $\mathcal{V}$ and to $\mathcal{F}^{\alpha}_{\mathtt{timestamp}}$
  2. If $\mathcal{A}$ has input `corrupt`, take the input C of $\mathcal{P}$ to $\mathcal{V}$ and verify whether it is a valid chain according to $\Pi^{\gamma,\varepsilon}_{\mathsf{SingleLipwig}}$
  3. **If** C is valid: parse it as $\{(B_i.content, T - t_{i+1}) \mid 0 \leq i < k\}$, then input it to $\mathcal{F}^{\alpha}_{\mathtt{timestamp}}$ as the adversarial proof-sequence (step 3) **else** input any invalid string to $\mathcal{F}^{\alpha}_{\mathtt{timestamp}}$
  4. Forward the output of $\mathcal{F}^{\alpha}_{\mathtt{timestamp}}$ to $\mathcal{Z}$ through $\mathcal{V}$.

We are now ready to state our main result in this paper that the cryptographic timestamping protocol $\mathsf{SingleLipwig}$ securely implements the ideal timestamping functionality.

Theorem 3.3. *Consider party $\mathcal{P}$ with PoSW-rate $\gamma > 0$ running the* $\mathsf{SingleLipwig}$ *algorithm and an adversary $\mathcal{A}$ with PoSW-rate $\gamma_\mathcal{A} \geq \gamma$, where $\mathcal{P}$ spends $\epsilon \geq 0$ time in between PoSWs (Steps 2 and 3 of the* `record` *routine of the algorithm) and $\mathcal{A}$ spends ϵ/α time in between PoSWs. Let $\mathcal{V}$ be the (uncorruptible) (γ, ϵ)-SP-PoSW-chain-verifier. Then the protocol $(\mathcal{P}, \mathcal{V})$ securely implements the timestamping functionality $\mathcal{F}^{\alpha}_{\mathtt{timestamp}}$ with $\alpha = \gamma_\mathcal{A}/\gamma$ with failure probability at most*

$$2 \cdot (q_1 \cdot 2^{-\lambda} + q_2 \cdot 2^{-\mu}) \cdot (Q_1 + Q_2 + |C|),$$

where C is the total bitlength of the last SP-PoSW-chain and the adversary $\mathcal{A}$ made q_1 queries with total bitlength Q_1 to H and q_2 queries with total bitlength Q_2 to $\mathcal{F}^{\mathsf{PoSW}}_{\gamma_\mathcal{A}}$ and $\mathcal{F}^{\mathsf{PoSW}}_{\gamma}$. This failure probability is negligible in λ and μ when $\mathcal{A}$ is PPT and sufficiently large security parameters.

Proof. Here is a sketch of the proof.

We first prove *correctness*: consider $\mathcal{P}$ running algorithm $\mathsf{SingleLipwig}$ uncorrupted. We want to show that any environment $\mathcal{Z}$ which interacts with $(\mathcal{P}, \mathcal{V})$ cannot see a difference to interacting with $\mathcal{F}^{\alpha}_{\mathtt{timestamp}}$ instead. Whenever $(\mathtt{record}, c_i)$ is sent to $\mathcal{P}$ at time T_i, $\mathcal{P}$ creates a block B_i with *content* $= c_i$ and inputs the timestamp $t_{i+1} = T_i + \epsilon$ with $\mathsf{MT.root}(B_i)$ to the PoSW. Whenever $\mathtt{prove}$ is sent to $\mathcal{V}$ at time T, the verifier $\mathcal{V}$ asks $\mathcal{P}$ for a proof, so $\mathcal{P}$ also creates a block with *content* $= \varnothing$ and returns a SP-PoSW-chain C and time $T+\epsilon$ to $\mathcal{V}$. The verifier will correctly output (c_i, a_i) with $a_i = T + \epsilon - (T_i + \epsilon) = T - T_i$ for all i, as C is correctly constructed and the strength of each PoSW will be such that $B_i.s/(t_{i+1} - t_i - \epsilon) = \gamma$ by definition of ϵ.

To show *security against any* $\mathcal{A}$, we show that our simulator $\mathcal{S}$ from item 3.3 can securely simulate an execution of $\Pi^{\gamma,\varepsilon}_{\mathsf{SingleLipwig}}$ such that neither the environment $\mathcal{Z}$ or the adversary $\mathcal{A}$ can distinguish. For this, we will show that the probability of distinguishability is at most the claimed failure probability in the theorem, which is negligible in λ and μ for any PPT $\mathcal{A}$.

What remains is to show that each time $\mathcal{S}$ sends π'_t to $\mathcal{F}^{\alpha}_{\mathtt{timestamp}}$, it will give the same output π'_t to the environment except with a certain probability.

First, whenever $\mathcal{A}$ sends a sequence C for which $\mathcal{V}$ returns reject then $\mathcal{F}^{\alpha}_{\mathtt{timestamp}}$ will also output reject. The simulator parses the chain as if it were $\mathcal{V}$ and if it is correctly constructed it sends the content in a way that $\mathcal{F}^{\alpha}_{\mathtt{timestamp}}$ can undesrstand. Note that the construction of $\mathcal{S}$ ensures that if $\mathcal{A}$ constructed a valid blockchain but outputs an invalid one, it will still be rejected.

Suppose now that $\mathcal{A}$ did send a valid SP-PoSW-chain $C = (B_0, \ldots, B_k)$. If $\mathcal{A}$ did not create C sequentially, through queries to $\mathcal{F}^{\mathsf{PoSW}}_{\gamma_\mathcal{A}}$, then $\mathcal{S}$ will not be able to input the correct $\mathtt{record}$ queries to $\mathcal{F}^{\alpha}_{\mathtt{timestamp}}$. This means that in the case of $\Pi^{\gamma,\varepsilon}_{\mathsf{SingleLipwig}}$ the verifier would accept while $\mathcal{F}^{\alpha}_{\mathtt{timestamp}}$ rejects. By Lemma 2.4 this event can happen only with probability at most

$$2 \cdot (q_1 \cdot 2^{-\lambda} + q_2 \cdot 2^{-\mu}) \cdot (Q_1 + Q_2 + |C|),$$

where the adversary made q_1 queries with total bitlength Q_1 to H and q_2 queries with total bitlength Q_2 to $\mathcal{F}^{\mathsf{PoSW}}_{\gamma_\mathcal{A}}$ and $\mathcal{F}^{\mathsf{PoSW}}_{\gamma}$.

We can now assume that $\mathcal{A}$ created C sequentially. Note that $\mathcal{A}$ had access to Σ for time $T_{prove} - T_{corr}$ and could not make signature forgeries before corruption, and thus could not make any meaningful query to $\mathcal{F}^{\mathsf{PoSW}}_{\gamma_\mathcal{A}}$ that could be part of C before corruption. By meaningful, we mean that no input to $\mathcal{F}^{\mathsf{PoSW}}_{\gamma_\mathcal{A}}$ before T_{corr} can consist of an adversarially created block with a valid signature. We define $A' := (T_{prove} - T_{corr}) \cdot (\gamma_\mathcal{A}/\gamma)$ which equals A in $\mathcal{F}^{\alpha}_{\mathtt{timestamp}}$.

We can consider two cases:

No prefix of C_{valid} is a prefix of C. This means that $\mathcal{A}$ after corruption started its own new SP-PoSW chain. Since $\mathcal{A}$ has PoSW-rate $\gamma_\mathcal{A}$ and had time $T_{prove} - T_{corr}$, it can legitimately only create a SP-PoSW chain with root time at most A', which matches the case $\pi'_t \cap \pi^\mathcal{A} = \varnothing$ in $\mathcal{F}^{\alpha}_{\mathtt{timestamp}}$. For each $B_i.content$ with timestamp $B_{i+1}.t$ in C, it was input to $\mathcal{F}^{\alpha}_{\mathtt{timestamp}}$ at time t_i. As C was considered valid and $\mathcal{A}$ has PoSW-rate $\gamma_\mathcal{A}$ it follows that $T_{prove} - B_{i+1}.t \leq \alpha \cdot (T_{prove} - t_i)$. Since π'_t consists only of pairs (c_i, a'_i) for messages c_i recorded at time $T_{prove} - a_i$ where $a'_i \leq \alpha a_i$, it follows that $\mathcal{F}^{\alpha}_{\mathtt{timestamp}}$ outputs π'_t and does not reject.

A non-empty prefix of C_{valid} is a prefix of C. This means that $\mathcal{A}$ after corruption started extending the shared prefix of C_{valid} and C, the remainder of C can cover at most time A'. Since the timestamps in the shared prefix are unmodified and C is accepted by $\mathcal{V}$ it follows that for all c_i that were recorded at time $t_i < T_{prove} - A'$ that $(c_i, T_{prove} - t_i) \in \pi'_t$. For the remainder of C, by the above reasoning it follows that the remainder of π'_t consists only of pairs (c_i, a'_i) for messages c_i recorded at time $T_{prove} - a_i$ where $a'_i \leq \alpha \cdot a_i$. Hence, this case also satisfies all criteria for $\mathcal{F}^{\alpha}_{\mathtt{timestamp}}$ to output π'_t and not to reject.

□

3.4 Signature Functionality Subtleties

Our construction restricts the ability of the adversary to always query $\Sigma.\mathsf{verify}$ when attempting to forge a signature. This restriction arises from the fact that the classical UC construction for signatures in the UC model [8] can be abused in a way that is not reflected in practice. Unforgeability in the UC framework is usually achieved by a functionality which records any valid query. In the case of signatures, a valid query is one made by someone who should have access to the secret key. Any attempt to forge a signature will fail, as the functionality will reject every verification query of a signature it did not create.

In our protocol, where temporality is fundamental, this construction can be easily abused. If an adversary constructs a block before corrupting $\mathcal{P}$, it can attempt to forge a signature and then use that block as an input to $\mathcal{F}^{\mathsf{PoSW}}_{\gamma_{\mathcal{A}}}$. In the case they could succeed in the forgery, they would have a PoSW that encodes an age greater than the one expected by the protocol. This would allow $\mathcal{A}$ to distinguish the protocols, as $\mathcal{F}^{\alpha}_{\mathtt{timestamp}}$ would reject this record while it would be acceptable in $\Pi^{\gamma,\varepsilon}_{\mathsf{SingleLipwig}}$. In general, this is allowed, as the possibility of $\mathcal{A}$ corectly guessing a signature is negligible. However, the structure of ideal signing functionalities allows adversaries to always succeed in this attempt.

In order to account for the predictability of signatures, ideal signing functionalities allow the adversary to determine the signature for any message. This construction allows an adversary to retroactively create a valid blockchain (according to $\mathcal{V}$ in $\Pi^{\gamma,\varepsilon}_{\mathsf{SingleLipwig}}$) The adversary runs SingleLipwig with the desired content generating a blockchain C^*, substituting signatures ($B^*_i.\mathsf{sig}$) for a random string (of length κ) whenever the block has not been signed by $\mathcal{P}$ (which we can assume is always true if there is a block iin the chain that differs to the one computed by $\mathcal{P}$). Whenever $\mathcal{A}$ corrupts $\mathcal{P}$ it queries the signature functionality to sign all the blocks $\{B^*_i.\mathsf{data} \mid B^*_i \in C^*\}$ which $\mathcal{A}$ had previously created. The functionality will then ask $\mathcal{A}$ for the signature, who will then choose the random string it had used in place of a signature in the block ($B^*_i.\mathsf{sig}$). Every block C^* will now be valid according to $\mathcal{V}$, with timestamps that precede the time of corruption by more than the acceptable margin $\gamma_{\mathcal{A}}/\gamma$.

In practice, this attack is impossible, as an adversary cannot retroactively make signatures valid in any actual signature scheme. This attack is only made possible by the fact that a verification query for the forged signatures is only done after corruption, as $\mathcal{A}$ simply hides it from view until it corrupts a party. This is not the case generally, as an adversary can create valid signatures after corruption without resorting to this trick. Therefore, restricting the adversary to query $\Sigma.\mathsf{verify}$ for every forgery attempt, while counter-intuitive, actually reflects reality by avoiding this loophole.

3.5 Multi Party Timestamping protocol

The single party protocol SingleLipwig trivially extends to a multi-party protocol by running multiple instances, where individual proofs are joined together and where the verifier simply applies a majority vote over the accepted records:

Consider $N \in \mathbb{N}$ provers $\mathcal{P}_1 \ldots, \mathcal{P}_N$ that each run algorithm SingleLipwig individually. Any `record` queries are broadcast to all provers and the adversary. At any time $\mathcal{V}$ can query all provers for a proof of their records. Verifier $\mathcal{V}$ individually verifies these as in the single-prover case and thus obtains up to N accepted records, after which it outputs every entry (c_i, a'_i) that occurred in more than half the accepted records. It is clear that when more than half of the provers are honest then $\mathcal{V}$ outputs the honest record. It is only after the adversary corrupts more than half of the provers that it can provide enough falsified accepted records that all contain a certain falsified entry for the verifier to output it. However, even when it corrupted all provers, it is clear from the security proof of SingleLipwig that the adversary can only generate malicious proofs under the same constraints as in the single-party protocol.

4 Proof Borrowing

4.1 An Inverse Problem

We are interested in creating proofs that can be verified by anyone, regardless of whether they are aware of who is computing the proofs of sequential work. Therefore, we cannot make the assumption that the verifier knows the rate over which the PoSW were computed. This prevents the verifier from linking computational work to time. There are two approaches to this problem; either the verifier chooses a rate or they compute the average rate presented by the proof. After finding the rate implied by an $H2T$-sequence, a party can choose whether the timestamp is valid or not.

The following verifier checks correctness of $H2T$-sequences and returns the average PoSW-rate with respect to the claimed *root time* of S. It should follow that a higher average PoSW-rate implies a higher assurance of the real age.

Definition 4.1 (Time-lock verifier). *We define a time-lock verifier $\mathcal{V}$ with access to oracles $\mathcal{F}_{\gamma}^{\mathsf{PoSW}}$ and H as follows.*

Whenever $\mathcal{V}$ receives sequence $S = ((x_0, s_0), \dots, (x_l, s_l))$, it returns $\perp$ if S is not a valid $H2T$-sequence. For a valid $H2T$-sequence $\mathcal{V}$ returns the average rate $\gamma = L/(t - t_0)$, where L is the total strength of all PoSW in S, t is the current time and t_0 is the root time of S.

The following game captures the challenge to compute an $H2T$-sequence that can be claimed τ older than it really is while still keeping minimum average rate γ.

Definition 4.2 (γ-time-lock game). *For $\tau > 0$, positive PoSW-rates $\gamma_{\mathcal{A}}, \gamma > 0$, we define the **$\gamma$-time-lock game** with respect to a time-lock verifier $\mathcal{V}$ as follows.*

Consider an adversary $\mathcal{A}$ with access to oracles $\mathcal{F}_{\gamma_{\mathcal{A}}}^{\mathsf{PoSW}}$ and H. At time t_0 the adversary gets access to an oracle O and queries it for a random bitstring $c_0 \in \{0,1\}^{\lambda}$, the adversary can make additional queries $c_1, \dots$ to O later on, but not before time t_0. The adversary constructs an $H2T$-sequence $S = ((x_0, s_0), \dots, (x_l, s_l))$ with root time $t_0 - \tau$ and where c_0 is a continuous substring of x_0. It sends S to $\mathcal{V}$ at time t_1 and wins if $\mathcal{V}$ outputs rate $r \geq \gamma$.

Here, the oracle O is used to enforce that the adversary can only legitimately start computing S from time t_0, which may seem slightly unnatural. However, in the remainder of the paper we will treat additional structured data (blockchains) that needs to satisfy additional constraints. One important case is that every block structure needs a signature by a certain party. In this case the oracle O can capture the signing capability of that party, to which the adversary only gets access after it has corrupted that party.

The following lemma lower-bounds the running time it takes an adversary with rate $\gamma_{\mathcal{A}}$ to compute an $H2T$-sequence with claimed root time that is τ older than the "real" root time, while still keeping minimum average rate γ, with non-negligible success probability.

Lemma 4.3 (γ-time-locked). *If $t_1 - t_0 < \tau \cdot \frac{\gamma}{\gamma_{\mathcal{A}} - \gamma}$ then the adversary wins the above game with probability at most*

$$2 \cdot (q_1 \cdot 2^{-\lambda} + q_2 \cdot 2^{-\mu}) \cdot (Q_1 + Q_2 + |S|),$$

where $\mathcal{A}$ made q_1 queries of total bitlength Q_1 to H and q_2 queries of total bitlength Q_2 to $\mathcal{F}_{\gamma_{\mathcal{A}}}^{\mathsf{PoSW}}$.

Proof. In the elapsed time $e := t_1 - t_0$ after corruption, the adversary can sequentially compute an $H2$-sequence of strength at most $e \cdot \gamma_{\mathcal{A}}$. The required minimum average rate of γ over age $T = e + \tau$ requires a minimum strength of $L = T \cdot \gamma$. It follows that the adversary can sequentially compute the $H2$-sequence with probability 1 if and only if $e \cdot \gamma_{\mathcal{A}} \geq T \cdot \gamma$ or equivalently $t_1 - t_0 \geq \tau \cdot \frac{\gamma}{\gamma_{\mathcal{A}} - \gamma}$. However, if $t_1 - t_0 < \tau \cdot \frac{\gamma}{\gamma_{\mathcal{A}} - \gamma}$ then the adversary cannot compute the $H2$-sequence sequentially and by Lemma 2.4 succeeds with probability at most

$$2 \cdot (q_1 \cdot 2^{-\lambda} + q_2 \cdot 2^{-\mu}) \cdot (Q_1 + Q_2 + |S|).$$

□

The above theorem shows that PoSW offers some sort of immutability for $H2T$-sequences by requiring a minimum average PoSW-rate in that the adversary needs a PoSW-rate that is a factor $\alpha = a/b$ higher to create an $H2T$-sequence of "age" a in real time b. Unlike for proof-of-work, the adversary cannot simply use more CPUs, but he needs to use a substantially faster CPU. It should be noted that this excludes any kind of protection against forking, *i.e.* when $a = b$ then the adversary does not need a higher PoSW-rate.

Furthermore, time-locked PoSW enforces a certain computational obstacle for any adversary in modifying an existing $H2T$-sequence with honest timestamps up to the point of the target modification, since in the malicious $H2T$-subsequence created by the adversary the earliest malicious timestamp must still be later than the last honest timestamp.

It should be clear that for honest parties a higher minimum average PoSW-rate implies a stronger protection against adversaries. It does require though that parties actively maintain their $H2T$-sequences by continually extending them with PoSW, lest they fall below the minimum required rate.

4.2 Proof Borrowing

The underlying assumption behind our model is that the participant has access to a very fast processor that is always on and computing proofs of sequential work. The processor must be fast enough that it provides a reasonable time-lock compared with the fastest available processor. These assumptions are non-trivial and cannot be achieved by any individual. However, as we show it is possible for participants to *borrow* proofs of sequential work from other blockchains. Any number of blockchains can be time-locked by a single constantly-maintained blockchain with a high rate through the use of Merkle paths. More importantly, if this blockchain were to disappear or stop growing, the security guarantees provided by it to another chain are encoded in the latter chain and are therefore maintained. This also allows a blockchain to connect to another blockchain through a bilateral agreement but disconnect unilaterally if they so choose.

In this multi-agent model, we continue to work solely with blockchains that are maintained by individual agents, so no consensus mechanism is required. Interaction between parties consists solely in choosing to point to another party's blockchain or not. While the purpose of this section is to show how proofs of sequential work can be strengthened by connecting blockchains, it is also important to note the effect that this has on someone attempting to rewrite their own chain. Participants are incentivized to achieve a stronger time-lock by connecting to other chains. However, a well-connected block is harder to rewrite, as disappearing every trace of the rewritten block implies rewriting all the blockchains that point to it.

To borrow a proof of sequential work it is not sufficient to simply take the *posw* component of a block. The *posw* component of a block time-locks the preceding chain because they belong to the same $H2T$-sequence. In order to replicate this property for a block that is not part of the chain, we need to create an $H2T$-sequence from the block that we wish to time-lock to the proof of sequential work. Fortunately, this can be achieved by adding a pointer to the block that precedes the proof of sequential work, as the hash of this block is contained in the argument of PoSW. Then, it will be possible to link this sequence with the $H2T$-sequence time-locking the block B in order to time-lock B^*. Participants can then use these $H2T$-sequences to *borrow* the proof of sequential work computed by other participants in order to secure their own blocks.

In order to work with these proofs of sequential work, we modify the struture of our blocks. As we are in a setting with multiple parties $\mathcal{P}_1, \ldots, \mathcal{P}_n$, we add a superscript to identify the blocks and chains of a party. Thus, the blockchains maintained by $\mathcal{P}_j$ becomes C^j. For simplicity, we refer to j as the identifier of $\mathcal{P}_j$ and assume that all parties are aware of this relationship. Additionally, we divide the *content* component in each block, which is a generic string, in two different components: *incoming* and *content*. The pointers to each block are added in the former, while *content* remains unchanged. All pointers in *incoming* consist solely of the Merkle roots of the blocks. At the same time, the parties who are looking to borrow the proof of sequential work are able to record the $H2T$-sequence generated by the other party and encode it in their own chain. This allows participants to maintain the time-lock even if the blockchain from where they borrowed it disappears. Participants also have the freedom to connect to multiple chains to borrow proofs and the ability to disconnect from one at will. Whenever we have that a block B^k_r contains a pointer to B^j_i we write $B^j_i \in B^k_r.incoming$. Additionally, if $r_0 \leq r < r_l$ we write $B^j_i \in C^k[r_0, r_l).incoming$.

While *incoming* is used to add the pointers to blocks in order to generate $H2T$-sequences, we are also interested in storing said sequences in the block. We do this through $H2T$-sequences representing the links between all the blocks in the subchain from where the proofs are borrowed.

Definition 4.4 (Borrowed PoSW). *Given a block B^j_i, we say that a* borrowed proof of sequential work *(or* borrowed PoSW*) from a subchain $C^k[r_0, r_l)$ is a set of tuples $(k, r, S, r+1)$ where S is an $H2T$-sequence from B^k_r to B^k_{r+1} passing through $B^k_{r+1}.posw$ for $r \in [r_0, r_l)$ and tuples (j, h, S, r) where S is an $H2T$-sequence from B^j_h to B^k_r, with $h < i$ and $r \in [r_0, r_l)$.*

Similar to what we have with *incoming*, we say that $B^k_r \in C^j i.Bposw$ or $C^k[r_0, r_l) \in C^j i.Bposw$ whenever the borrowed proof contains a block or subchain respectively.

Definition 4.5 (P2P-PoSW-block). *We define a* P2P-PoSW-block *for a party $\mathcal{P}_j$ with public key $pk_j \in \{0,1\}^*$ as a tuple $B^j = (data, sig)$, where $data = (pk, rnd, prev, posw, bposw, incoming, content)$ and*

1. *$pk \in \{0,1\}^*$ is the public key of the party maintaining the chain;*
2. *$rnd \in \mathbb{N}$ is the sequence number of the block;*

3. *$prev \in \{0,1\}^*$ is the root hash $\mathsf{MT.root}(B_p^j)$ of the previous block B_p^j, or $prev = \mathsf{H}(pk)$ when $rnd = 0$;*
4. *$posw = \{((t||prev),(p||s))\}$ includes both the proof of sequential work computed by $\mathcal{P}_j$, where $(p,s) = \mathsf{PoSW}(t||prev, s)$ and $t \in \{0,1\}^\theta$ is a timestamp the PoSW was started;*
5. *$Bposw = \{(k, r, S_0, r+1)\} \cup \{(j, h, S_1, r)\}$ where $k \neq j$ and S_0 and S_1 are $H2T$-sequences;*
6. *$incoming \subsetneq \{0,1\}^\lambda$ contains root pointers to other blocks;*
7. *$content \in \{0,1\}^*$ represents some content;*
8. *$sig \in \{0,1\}^*$ is a digital signature on $\mathsf{MT.root}(data)$,* i.e., *$\Sigma.\mathsf{verify}(pk, data, sig) = \mathsf{accept}$.*

Definition 4.6 (P2P-PoSW-chain). *We define a* P2P-PoSW-chain *C^j for a party $\mathcal{P}_j$ with public key $pk \in \{0,1\}^*$ as a sequence of SP-PoSW-blocks $C^j = (B_0^j, \dots, B_k^j)$ where for all $0 \leq i \leq k$:*

1. *$B_i^j.pk = pk_j$;*
2. *$B_i^j.rnd = i$;*
3. *$B_0^j.prev = \mathsf{H}(B_0^j.pk)$ and $B_i^j.prev = \mathsf{MT.root}(B_{i-1}^j)$ for $i > 0$;*
4. *$B_i^j.posw = ((B_i^j.t||B_i^j.prev),(p_i||s_i))$ and $(p_i, s_i) = \mathsf{PoSW}(B_i^j.t||B_i^j.prev, s_i)$;*
5. *If $(j, h, (x_0, s_0), \dots), r) \in B_i^j.Bposw$ then $h < B_i^j.rnd$ and $x_0 = a|\mathsf{MT}(B_h^j)|b$ for some $a, b \in \{0,1\}^*$ and there exists $(k, r, S, r+1) \in B_i^j.Bposw$ such that $\big((x_0, s_0), \dots\big) \bowtie S$ is a valid $H2T$-sequence;*
6. *For every $(k, r, S_r, r+1) \in B_i^j.Bposw$, S is an $H2T$-sequence of non-zero strength with timestamp $t_{r+1} < B_i^j.t$*
7. *For any two $(k, r, S_r, r+1), (k, r+1, S_{r+1}, r+2) \in B_i^j.Bposw$ with $k \neq j$ we have that $S_r \bowtie S_{r+1}$ is a valid $H2T$-sequence;*
8. *If $(r, h, P) \in B_i^j.M$ then $\mathsf{MT.verify}(P) = \mathsf{accept}$, $P = ((a|\mathsf{MT}(B_r^j)|b, s_0), \dots)$ for some strings a, b and $r < i$*
9. *$B_i^j.t < B_j^j.t$ for all $i < j \leq k$;*
10. *$\Sigma.\mathsf{verify}(pk, B_i^j.data, B_i^j.sig) = \mathsf{accept}$;*

*Let $\mathsf{len}(C^j) = k$ be the **length** of C^j. We define the notations $C^j[i] = B_i^j$ for block indexing, $C^j[ir, =)(B_i^j, \dots, B_{r-1}^j)$ for subchains, and $\mathsf{last}(C^j) = B_k^j$ for the last block of C^j.*

All the time that passes between the creation of two blocks in the same chain is encoded in a proof of sequential work, but we cannot say the same about the time that passes between blocks of different chains. This "wasted" time implies that in certain situations it is not advantageous to use the entirety of the borrowed proof, or cases when a proof borrowed from the participant with the highest rate is not as strong as the one provided by a different party with a slightly lower rate. In order to choose the correct $H2T$-sequence that maximally time-locks a block, we need to search all the possible $H2T$-sequences between the chains in the network. This process should not take too much time and while there can be many heuristics and design choices limit the number of blocks considered, we show that finding the strongest $H2T$-sequence requires linear time over the number of blocks.

If we consider a borrowed PoSW from $C^k[r_0, r_l)$ as a pointer to $B_{r_l-1}^k$, we can see that we can create a directed graph where blocks are vertices and the $H2T$-sequences encoded in them (be them borrowed PoSW, incoming pointers or pointers to the previous blocks) are edges. Even more, we can have each edge have the strength of the $H2T$-sequence as a weight whenever two blocks are in the same chain. This way we can construct a graph of the entire network, with all the $H2T$ -sequences connecting blocks acting as edges. We can use this graph to find the strongest $H2T$-sequence for any block.

Lemma 4.7. *Given n blockchains that are maintained by parties $\mathcal{P}_1, \dots, \mathcal{P}_n$, the pointers in a block induce a weighted DAG $D = (V, E, W)$ with $V = \bigcup_{j \in [n]}\{B_i^j \mid i \in [\mathsf{len}(C^j)]\}$ and $E \times W$ represents all $H2T$-sequences between blocks where W are the strengths of these sequences except with negligible probability in λ.*

Proof. First, let $V = \bigcup_{j \in [n]}\{B_i^j \mid i \in [\mathsf{len}(C^j)]\}$. Then, for each $B_i^j \in V$, with $i \neq 0$, add a weighted edge to $E \times W$ from its predecessor, $\big((B_{i-1}^j, B_i^j), \mathsf{str}(B_i^j.posw)\big)$. Then for every block in $B_r^k \in B_i^j.incoming$, add an edge $\big((B_i^j, B_r^k), 0\big)$. Finally, if $C^k[r_0, r_l) \in B_i^j.Bposw$ and $C^k[r_0, r_l + 1) \notin B_i^j.Bposw$, add an edge $\big((B_{r_l}^k, B_i^j), 0\big)$. For every edge in $E \times W$, the edge corresponds to either

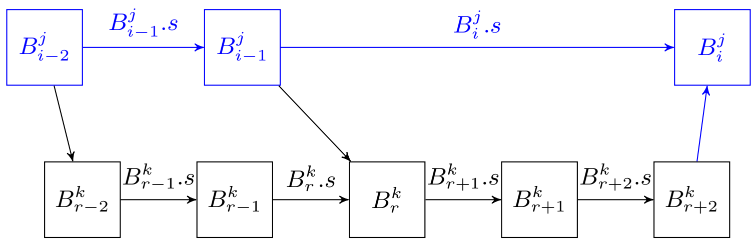

Fig. 1. An example of the DAG induced by two blockchains. Arrows are of the color of the block that contains the pointer.

a Merkle path (when the weight is 0) or a proof of sequential work (where the weight equals the strength of the PoSW). Therefore, all edges correspond to $H2T$-sequences. For a cycle to appear in this graph, a block would need to have a pointer for a block that appeared after it. This is only possible if there is a collision in H and two blocks have the same root pointer, which has negligible probability in λ. □

The fact that this graph is a DAG allow us to find the best $H2T$-sequence that time-locks a block in an efficient manner.

Lemma 4.8. *Given n blockchains that are maintained by parties $\mathcal{P}_1, \ldots, \mathcal{P}_n$, finding the $H2T$-sequence $S = \big((\mathsf{MT}(B^j_i), \perp), \ldots, (\mathsf{MT}(\mathsf{last}(C^j)), \perp)\big)$ of maximum strength between two blocks takes linear time over the total number of blocks.*

Proof. We can see that every edge of the graph induced by the executions presented in Lemma 4.7 corresponds to an $H2T$-sequence. As the graph is a DAG, finding the longest path from can be done in linear time [16]. The longest path between $\mathsf{last}(C^j)$ and B^j_i corresponds to the strongest $H2T$-sequence between blocks. □

An important characteristic about the construction of our blockchains is that after the pointers have been created, the relevant $H2T$-sequences to time-lock a block B^j_i are all encoded in C^j. Therefore, we present an algorithm to extract the relevant subgraph D^j_i of D in order to find the strongest $H2T$-sequence time-locking B^j_i in a blockchain of length $i + r$. We remember that given an $H2T$-sequence that includes an instance of PoSW, it is possible to extract a timestamp t from any term with non-zero strength.

Algorithm $\mathsf{makeDAG}$

Set $t^* \leftarrow B^j_i.t$, $V \leftarrow \{B^j_i\}$ and $E \times W \leftarrow \varnothing$ For each block B^j_{i+h} in $C^j[i+1, i+r)$

1. $V \leftarrow V \cup \{B^j_{i+h}\}$, $E \times W \leftarrow E \times W \cup \{\big((B^j_{i+h-1}, B^j_{i+h}), B^j_{i+h}.s\big)\}$
2. For each $(j, i+b, S, l) \in B^j_{i+h}.posw$, with $b > 0$, let $V \leftarrow V \cup \{B^k_l\}$ and $E \times W \leftarrow E \times W \cup \{\big((B^k_l, B^j_{i+b}), 0\big)\}$
3. For each $(k, l, S, l+1) \in B^j_{i+h}.posw$, if the timestamp t encoded in S is greater than t^*, let $V \leftarrow V \cup \{B^k_l, B^k_{l+1}\}$ and $E \times W \leftarrow E \times W \cup \{\big((B^k_l, B^k_{l+1}), 0\big)\}$

Note that D^i_j only includes blocks that are directly connected to blocks in $C^j[i, i+l)$, this is not strictly necessary as we can naturally extend our definition of a borrowed PoSW to include also all the borrowed PoSWs contained in the blocks from where the proofs are being borrowed. In practice, parties would only borrow proofs from blockchains with a higher average rate than them, limiting the number of $H2T$-sequences that are encoded in a block. Additionally, participants are able to add $H2T$-sequences to blockchains they are not directly connected by *hopping* between blockchains by linking $H2T$-sequences. However, for this presentation we assume that $B^j_i.Bposw$ contains only $H2T$-sequences that directly connect to C^j.

4.3 A model with n blockchains

We have shown that a participant can generate a stronger proof of sequential work for their blockchains by borrowing the proofs computed by other participants through Merkle pointers. We

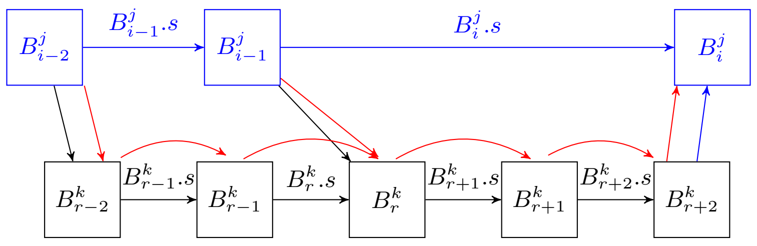

Fig. 2. An example of the DAG induced by two blockchains. The red arrows represent all of the edges contained in $B^j_i.Bposw$.

also showed that this information can be directly encoded in the blockchain itself, meaning that once a proof is borrowed, it will always be valid even if the source for it disappears. This contrasts with blockchain-based timestamping protocols in practice, where the timestamp is only valid as long as the blockchain maintaining it continues to exist and considered trustworthy. This section focuses on the creation of a model for this setting, showing that sharing proofs of work allows for a realization of $\mathcal{F}^{\alpha}_{\texttt{time-lock}}$ for a lower α.

In order to model the communication between the parties, we use a broadcast functionality $\mathcal{F}^{\delta}_{\mathsf{comm}}$ through which parties share their blocks. At any time, a party may input a block B to $\mathcal{F}^{\delta}_{\mathsf{comm}}$, which the functionality then sends to $\mathcal{A}$. The adversary is tasked with putting messages in the inbox of each participant and can delay messages for at most δ time steps, after which it must add the message to the inbox. Whenever a paprty inputs `listen` to $\mathcal{F}^{\delta}_{\mathsf{comm}}$, they receive all messages in their inbox in the form of a set M.

Participants decide how to construct the *incoming* and *Bposw* components of their blocks from M in the following way, given a $\gamma_{\min}$:

Algorithm pointers

```
Set $t^* :=$ clock, $incoming := \varnothing$ and $Bposw := \varnothing$
    Divide $M$ into $n-1$ sequences $M_k$ $(k \neq j)$ ordered by $B.i$, such that $B \in M_k$ iff $\Sigma.\mathsf{verify}(pk_k, \mathsf{MT}(B.data), B.sig) = \mathsf{accept}$ and $B$ is a valid block.
    For each $M_k$:
  – For each $B_r$ in $M_k = (B_f, B_{f+1}, \ldots, B_l)$:
      • If $B_r.t \geq t^*$: discard $B_r$ and empty $M_k$;
      • Else If $B_r = B_f$:
          ∗ If $\gamma_{\min} > {}^{B_r.s}/_{(t^*-B_r.t)}$: discard $B_r$;
          ∗ Else If ${}^{B_r.s}/_{t^*-B_r.t} \leq \gamma_j$: set $incoming \leftarrow incoming \cup \{\mathsf{MT}(B_r)\}$ ;
          ∗ Otherwise: Let $P$ be the borrowed PoSW from $M_k$, set $Bposw \leftarrow Bposw \cup P$;
      • Else If ${}^{B_{r+1}.s}/_{B_{r+1}.t-B_r.t} < \gamma_{\min}$: discard $B_r$ and empty $M_k$;
      • Otherwise skip;
Output $(Bposw, incoming)$
```

The purpose of $\gamma_{\min}$ is to prevent an attack from the adversary where it timestamps a block in the future and starts computing a proof of sequential work until this point in the future passes. This would allow $\mathcal{A}$ to create a block that would have an exaggerated strength. While this brings no advantage to the adversary, as the average rate of the chain continues to be constant, the previous has a very weak proof. This selection mechanism punishes a participant that attempts to perform this attack.

Alternatively, if the participants know the rate of the other parties, an alternate algorithm can be chosen such that as long as the blocks are constructed correctly and have timely timestamps, if the rate of the other party is lower or equal than γ_j, the root hash is added to pointers and otherwise the borrowed proof is added to *Bposw*.

We define the following protocol $\mathsf{P2PLipwig}$ with n parties $\mathcal{P}_1, \ldots, \mathcal{P}_n$ who each maintain a P2P-PoSW-chain:

Algorithm $\mathsf{P2PLipwig}$

```
Assume that party $\mathcal{P}_j$ with public key $pk_j$ has access to $\mathsf{H}$, $\mathcal{F}^{\mathsf{PoSW}}_{\gamma_j}$ for PoSW-rate $\gamma_j > 0$, a broadcast functionality $\mathcal{F}^{\delta}_{\mathsf{comm}}$ and content oracle $\mathsf{O}_j$ which outputs the content bitstring for each block. $\mathcal{P}_j$ sets $prev = \mathsf{H}(pk)$, $t_0 = \mathsf{clock}()$, inputs $\mathsf{start}(t_0||prev)$ to $\mathcal{F}^{\mathsf{PoSW}}_{\gamma}$ and does the following for rounds $i = 0, \ldots$:

1. waits till $\mathsf{O}_j$ outputs $content$;
2. lets $comm \leftarrow \mathcal{F}^{\delta}_{\mathsf{comm}}$ and $(out, in) \leftarrow \mathsf{pointers}(comm)$;
3. retrieves $(p_i, s_i)$ by querying $\mathsf{output}(t_i||prev)$ to $\mathcal{F}^{\mathsf{PoSW}}_{\gamma}$;
4. queries $sig_i \leftarrow \Sigma.\mathsf{sign}(pk, \mathsf{MT.root}(data_i))$,
   where $data_i = (pk, i, prev, ((t_i||prev), (p_i||s_i)), out, in, content)$;
5. sets $B^j_i \leftarrow (data_i, sig_i)$, $prev \leftarrow \mathsf{MT.root}(B^j_i)$, and $t_{i+1} \leftarrow \mathsf{clock}()$;
6. inputs $\mathsf{start}(t_{i+1}||prev)$ to $\mathcal{F}^{\mathsf{PoSW}}_{\gamma}$;
7. outputs SP-PoSW-chain $C^j_i = (B^j_0, \ldots, B^j_i)$ together with time $t_{i+1}$ and inputs $B^j_i$ to $\mathcal{F}^{\delta}_{\mathsf{comm}}$;

   On input prove in round $i$, $\mathcal{P}$ do the following:

1. retrieves $(p'_i, s'_i)$ by querying $\mathsf{output}(t_i||prev)$ to $\mathcal{F}^{\mathsf{PoSW}}_{\gamma}$;
2. queries $sig'_i \leftarrow \Sigma.\mathsf{sign}(pk, \mathsf{MT.root}(data'_i))$,
   where $data'_i = (pk, i, prev, ((t_i||prev), (p'_i||s'_i)), \varnothing)$;
3. sets $\widehat{B_i} \leftarrow (data'_i, sig'_i)$ and $t'_{i+1} = \mathsf{clock}()$;
4. outputs SP-PoSW-chain $C_i = (B_0, \ldots, B_{i-1}, \widehat{B_i})$ together with time $t'_{i+1}$;
```

Note that contrary to previous models, the oracle O_j is different for all parties. Having multiple oracles implies two fundamental differences: parties do not need to agree on content and their block schedules are independent from each other.

Lemma 4.9. *Given a time-lock verifier $\mathcal{V}$ (Definition 4.1) and parties $\mathcal{P}_1, \mathcal{P}_2$ executing* $\mathsf{P2PLipwig}$ *with oracles O_j which independently output a string every $t \overset{R}{\leftarrow} (T_{\min}, T_{\max})$ time steps, with $T_{\min} \gg \delta$ and $T_{\max}$ and rates $\gamma_1 < \gamma_2$. Given a block B^1_i created at time t_0, at time $T_{borrow} = T_{\max} + 1 + t_0 + {}^{\gamma_2(T_{\max}-\delta)}/_{\gamma_2-\gamma_1}$ $\mathcal{P}_1$ can generate an H2T-sequence $\big((\mathsf{MT}(B^1_i), s)\big), \ldots$ such that $\mathcal{V}$ outputs γ*

Proof. If the parties are following the protocol, the longest it can take for a block B^1_i to be pointed to in a block $B^2_{k_i}$ is $T_{\max} + \delta$. Therefore, for $\mathcal{V}$ to output $\gamma_2 - \epsilon$, it would be necessary that $\gamma_2 \cdot (T_{borrow} - t_0 - (T_{\max} + \delta)) > \gamma_1(T_{borrow} - t_0)$. After this, the faster chain can take an additional $T_{\max} - 1$ time steps to issue a block containing the requisite proof. □

Lemma 4.10. *Given a time-lock verifier $\mathcal{V}$ and parties $\mathcal{P}_1, \mathcal{P}_2$ executing* $\mathsf{P2PLipwig}$ *with rates $\gamma_1 < \gamma_2$ and oracles O_1 and O_2 which independently output a string every $t \overset{R}{\leftarrow} (T_{\min}, T_{\max})$ time steps. Given a block B^1_i created at time t_0, for any $\gamma = \gamma_2 - \xi$ for ξ arbitrarily small, there exists a time $T_\xi = t_0 + T_{\max} - 1 + {}^{\gamma_2 \cdot T_{\max} + \gamma_2\delta}/_{(-\gamma+\xi+\gamma_2)}$ such that T_ξ that time $\mathcal{P}_1$ can generate an H2T-sequence $\big((\mathsf{MT}(B^1_i), s)\big), \ldots$ such that $\mathcal{V}$ outputs γ.*

Proof. If the parties are following the protocol, the longest it can take for a block B^1_i to be pointed to in a block $B^2_{k_i}$ is $T_{\max} + \delta$. Therefore, for $\mathcal{V}$ to output $\gamma_2 - \xi$, it would be necessary that $(\gamma - \xi) \cdot (T_\xi - t_0) \geq \gamma_2 \cdot ((T_\xi - t_0) - T_{\max} + \delta)$. After this, the faster chain can take an additional $T_{\max} - 1$ time steps to issue a block containing the requisite proof. □

These simple lemmas show that as long as the faster chain has a bound on the space between blocks, a party with a slower rate can always benefit from having pointers to it.

5 Applications

Given the problems inherent in proof-of-work consensus, research has has turned towards the search for more scalable and less wasteful consensus algorithms. While there are multiple protocols that have been presented that achieve consensus, none manage to replicate all the desirable properties from the Nakamoto protocol. In particular, when there is no cost associated to the construction of a blockchain, emulation attacks become possible if enough private keys become compromised. For example, a new party does not have any certainty of whether the blockchain that they are presented with is the real one. In particular, Proof-of-Stake proposals have a natural vulnerability to posterior attacks, where participants who no longer own stake in the blockchain (but did in the past) are incentivized to rewrite the blockchain in some point in the past in order to regain their past fortune. While solutions to this problem already exist, like forward-secure signature schemes [10], they do not entirely capture the temporal properties existent in Bitcoin.

We believe that our work is of particular importance in the realm of permissioned blockchains which operate in a small network. There have been concerns that when a blockchain is maintained by a small group which uses the same software, the possibility that the whole network can corrupted by an adversary becomes relatively high [25]. The guarantees presented in our work show that even in the case of complete corruption of the network, only some limited tail of the chain can be rewritten, depending on the time that the network stays under adversarial control. By design, while time intensive, the computations needed to compute the proofs of sequential work will be done by a single processor, meaning that higher security does not imply higher energy waste.

In Section 4, we showed that the parties maintaining the blockchain do not need to directly compute the proof themselves. If we wish to secure a blockchain, it is only necessary to have a blockchain with proofs of sequential work which contains pointers to it. The guarantees achieved by this process are greater than ones provided by simply adding a hash to a Bitcoin transaction, as the maintainers of the blockchain will be able to extract the proof and encode it back into the blockchain. This could lead to a creation of a blockchain containing solely pointers to other blockchains maintained by multiple parties with their own PoSW blockchains, as seen in [15]. Maintaining such a blockchain would not be too expensive, as only one (arguably very fast) processor would be needed per participant.

While it might seem that depending on centralized groups to provide time-locking might betray the decentralized vision for blockchains, many of these groups could exist, allowing a blockchain to freely choose to change their time-lock provider if they so choose.

Finally, such a blockchain containing pointers from multiple separate blockchains could be a good candidate to realize a trusted randomness beacon. While there have been numerous proposals to use blockchains as randomness beacons, the one presented in [22] relies on similar assumptions to the ones used in this work.

References

1. Bitcoin energy consumption index (4 April 2018), `https://digiconomist.net/bitcoin-energy-consumption`
2. Bayer, D., Haber, S., Stornetta, W.S.: Improving the efficiency and reliability of digital time-stamping. Sequences II: Methods in Communication, Security and Computer Science pp. 329–334 (1993)
3. Benaloh, J., De Mare, M.: One-way accumulators: A decentralized alternative to digital signatures. In: Workshop on the Theory and Application of of Cryptographic Techniques. pp. 274–285. Springer (1993)
4. Bitansky, N., Goldwasser, S., Jain, A., Paneth, O., Vaikuntanathan, V., Waters, B.: Time-lock puzzles from randomized encodings. In: Proceedings of the 2016 ACM Conference on Innovations in Theoretical Computer Science. pp. 345–356. ACM (2016)
5. Buldas, A., Lipmaa, H., Schoenmakers, B.: Optimally efficient accountable time-stamping. In: International Workshop on Public Key Cryptography. pp. 293–305. Springer (2000)
6. Buldas, A., Saarepera, M.: On provably secure time-stamping schemes. In: International Conference on the Theory and Application of Cryptology and Information Security. pp. 500–514. Springer (2004)
7. Canetti, R.: Universally composable security: A new paradigm for cryptographic protocols. In: Foundations of Computer Science, 2001. Proceedings. 42nd IEEE Symposium on. pp. 136–145. IEEE (2001)
8. Canetti, R.: Universally composable signature, certification, and authentication. In: Computer Security Foundations Workshop, 2004. Proceedings. 17th IEEE. pp. 219–233. IEEE (2004)
9. Cohen, B., Pietrzak, K.: Simple proofs of sequential work. In: Annual International Conference on the Theory and Applications of Cryptographic Techniques. pp. 451–467. Springer (2018)
10. David, B., Kiayias, A., Russell, A.: Ouroboros praos: An adaptively-secure, semi-synchronous proof-of-stake protocol (2017), `http://eprint.iacr.org/2017/573`
11. Garay, J., Kiayias, A., Leonardos, N.: The bitcoin backbone protocol: Analysis and applications. Cryptology ePrint Archive, Report 2014/765 (2014), `http://eprint.iacr.org/2014/765`
12. Gipp, B., Meuschke, N., Gernandt, A.: Decentralized trusted timestamping using the crypto currency bitcoin. arXiv preprint arXiv:1502.04015 (2015)

13. Haber, S., Stornetta, W.S.: How to time-stamp a digital document. Journal of Cryptology 3(2), 99–111 (Jan 1991), https://doi.org/10.1007/BF00196791
14. Katz, J., Miller, A., Shi, E.: Pseudonymous broadcast and secure computation from cryptographic puzzles. Cryptology ePrint Archive, Report 2014/857 (2014), http://eprint.iacr.org/2014/857
15. Landerreche, E.: Leaning on impossible-to-parallelise work for immutability guarantees in the blockchain (2017), https://www.illc.uva.nl/Research/Publications/Reports/MoL-2017-29.text.pdf
16. Lawler, E.L.: Combinatorial optimization: networks and matroids. Courier Corporation (1976)
17. Lenstra, A.K., Wesolowski, B.: A random zoo: sloth, unicorn, and trx. Cryptology ePrint Archive, Report 2015/366 (2015), http://eprint.iacr.org/2015/366
18. Lin, H., Pass, R., Soni, P.: Two-round and non-interactive concurrent non-malleable commitments from time-lock puzzles. Cryptology ePrint Archive, Report 2017/273 (2017), http://eprint.iacr.org/2017/273
19. Mahmoody, M., Moran, T., Vadhan, S.: Time-lock puzzles in the random oracle model. In: CRYPTO. vol. 6841, pp. 39–50. Springer (2011)
20. Mahmoody, M., Moran, T., Vadhan, S.: Publicly verifiable proofs of sequential work. In: Proceedings of the 4th conference on Innovations in Theoretical Computer Science. pp. 373–388. ACM (2013)
21. Nakamoto, S.: Bitcoin: A peer-to-peer electronic cash system (2008)
22. Pierrot, C., Wesolowski, B.: Malleability of the blockchain's entropy. In: ArcticCrypt 2016 (2016)
23. Poelstra, A.: Distributed consensus from proof of stake is impossible (2014)
24. Rivest, R.L., Shamir, A., Wagner, D.A.: Time-lock puzzles and timed-release crypto (1996)
25. Sirer, E.G.: What could go wrong? when blockchains fail (2017), http://events.technologyreview.com/video/watch/emin-gun-sirer-cornell-when-blockchains-fail/, business of Blockchain
26. Stavrou, A., Voas, J.: Verified time. Computer 50(3), 78–82 (Mar 2017)