Non-interactive Cryptographic Timestamping based on Verifiable Delay Functions

Esteban Landerreche[1], Marc Stevens[1], and Christian Schaffner[2,3]

[1] Centrum Wiskunde & Informatica (CWI), Amsterdam
[2] Institute of Logic, Language and Computation, University of Amsterdam
[3] QuSoft, Amsterdam
esteban@cwi.nl

Abstract. We present the first treatment of non-interactive publicly-verifiable timestamping schemes in the Universal Composability framework. Inspired by the timestamping properties of Bitcoin, we use non-parallelizable computational work that relates to elapsed time to avoid previous impossibility results on non-interactive timestamping. We introduce models of verifiable delay functions (VDF) related to a clock and non-interactive timestamping in the UC-framework. These are used to present a secure construction that provides improvements over previous concrete constructions. Namely, timestamps forged by the adversary are now limited to a certain time-window that depends only on the adversary's ability to compute VDFs more quickly and on the length of corruption. Finally, we discuss how our construction can be added to non-PoW blockchain protocols to prevent costless simulation attacks.

Keywords: non-interactive cryptographic timestamping, universal composability, verifiable delay functions, time-lock cryptography

1 Introduction

In the digital domain, giving evidence that a certain amount of time has passed is more challenging than in the physical world. Exploring how to reliably create digital timestamps has been an active research area for the last thirty years. The first paper to deal with digital timestamping by Haber and Stornetta [9] presented solutions which utilized cryptography to limit the trust deposited on the party doing the timestamping. Their solution is based on a hashchain: a sequence of documents linked through a collision-resistant hash function.

In the literature, almost every timestamping service requires interaction with a group of validators and provides security guarantees only for the ordering of events, timestamping relative to other events. Non-interactive timestamping has been explored previously in [12], where the authors present a generic impossibility result. Because there is no interaction between prover and verifier, an adversarial prover could simply simulate the execution of an honest prover 'in the past' to generate a fake timestamp. They sidestep this result by working in the bounded-storage model where they construct a secure protocol, where the adversary is

unable to simulate an honest prover due to lack of storage. Another approach to sidestep this result is to relate computational work to elapsed time. This was mentioned in [11] as a possible application for their proof-of-sequential-work construction. The same idea is what allows Bitcoin to act as a decentralized timestamping service.

Haber and Stornetta's timestamping hashchain served as the inspiration for the blockchain that underlies Bitcoin [13]. Even when it was not its stated goal, Bitcoin achieved timestamps that are more trustworthy than those of the original construction. Proofs of work were introduced to prevent malicious adversaries from overwhelming honest parties through false identities. Each block of the Bitcoin blockchain contains a certificate that, on average, a certain amount of work has been invested in its creation. Given an idea of how much computational power is available to the network, it is possible to assert the age of any piece of data on the blockchain, with higher certainty than previous non-centralized systems. One can assume the age of a record approximately corresponds to its depth in the blockchain, unless an adversary's computational power exceeds the honest Bitcoin miners' computational power. Under non-standard yet realistic assumptions, Bitcoin provides long-term proofs of the age of its blocks.

While timestamping is mentioned as a goal of Bitcoin in the original whitepaper its security properties were only recently formalized in [1]. There, the authors study which timestamping guarantees can be achieved through interaction with an existing ledger. While there is a level of public verifiability in ledger-based timestamping, they require accepting the assumptions of the underlying blockchain protocol. Bitcoin was able to sidestep impossibility results about non-interactive timestamping by connecting time with work, but at a high price.

Unfortunately proof of work is incredibly wasteful, which is why there has been a search for more sustainable replacements. While there are many proposals for different Sybil-resistance mechanisms, none of them provide these additional timestamping guarantees. In fact, all of these solutions explicitly avoid the computational investment that allows for timestamping in Bitcoin. However, the large computational work is the mechanism behind Bitcoin's resilience to costless-simulation and long-range attacks. Non-proof-of-work solutions require additional cryptographic assumptions, like secure erasures [8], in order to maintain the same level of robustness as Bitcoin.

The main goal of this paper is to find another solution to base the security of timestamping on computational work in a manner that is not wasteful while being easy to apply. We take ideas from the same timestamping protocol that influenced Bitcoin to construct a hashchain-based protocol using verifiable delay functions (VDFs). Using non-parallelizable work allows us to have assumptions that are realistic, as adversaries only gain a computational advantage with faster processors, not with more processors. We quantify the advantage of the adversary in terms of the factor that she is able to compute VDFs faster compared to an honest prover. Our scheme offers similar timestamping guarantees of proofs of work for distributed ledgers that do not rely on proof of work, as we briefly discuss in the last section of this paper.

1.1 Our contributions

We study non-interactive cryptographic timestamping based on Verifiable Delay Functions (VDFs) in the Universal-Composability (UC) framework and the Random-Oracle Model (ROM). This is the first treatment of non-interactive timestamping in the UC model by introducing non-interactive computational proofs-of-age which act as a lower bound on the age of a record.

We define an ideal timestamping functionality $\mathcal{F}_{\mathrm{ts}}^{\alpha}$ that maintains a time-stamped record of messages. It can be queried to generate a non-interactive proof for the record's age at the time of proof-generation. We parametrize the adversary's advantage with a *time-diluting* factor α.

For a record of age *TrueAge*, the ideal functionality allows timestamps of age *StampAge* that are correct ($\mathit{StampAge} \leq \mathit{TrueAge}$) or forged timestamps with a claimed age bounded as $\mathit{StampAge} \leq \alpha \cdot \min(t_{\mathrm{corr}}, \mathit{TrueAge})$, where t_{corr} is the time since corruption (or $-\infty$ if uncorrupted).

In particular this implies that she cannot create any forged timestamps with age larger than $T \cdot \alpha$, which is possible in a similar construction by Mahmoody et al. [11], where for any record of any age the adversary can craft forged timestamps that are at most α times older than the record's true age. In Appendix C we provide a treatment of Mahmoody et al.'s construction in the UC-framework for easy comparison with our work.

Our main contribution is presented in Section 4. We define the functionality $\mathcal{F}_{\mathrm{ts}}^{\alpha}$ and present our hashchain-based protocol. We show our construction securely realizes the timestamping functionality in the random-oracle model and universal-composability framework against an adversary that can compute verifiable delay functions faster than the prover by the time-diluting factor α.

2 Model and Definitions

We construct our protocol in the universal-composability framework [5, 6] where two PPT algorithms $\mathcal{Z}$ and $\mathcal{A}$ interact with parties executing a protocol. We assume a hybrid model where parties have access to a global clock, random oracles, an unforgeable signature scheme and the $\mathcal{F}_{\mathsf{VDF}}^{\Gamma}$ functionality that represents our verifiable delay function.

Time. We will work in the synchronous model presented in [10] where parties have access to a clock functionality $\mathcal{F}_{\mathsf{clock}}$ (Appendix A.1). For the clock to advance, each party must input an instruction to the clock that they have done all their computations for the current round. Once every party and the adversary have finished their computations for the round, the clock ticks and the next round begins. The clock allows us to give meaningful statements aboutthe passage of time, but our protocol does not make any assumption on the synchronicity of communication between the parties. We will refer to the encoding of the round number as a **time receipt** with constant length θ.

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.

Sequences. We denote a sequence of ℓ elements from a set X as $S = \langle x_i \mid x_i \in X \rangle_\ell$, where the elements of the sequence are indexed by $i \in \{0, \dots, \ell - 1\}$. When it is more practical or clear from context, we may denote a sequence as $S = \langle x_0 \dots, x_{\ell-1} \rangle_\ell$ or simply $\langle x_i \rangle_\ell$. We also avoid writing the subscript ℓ when the length of the sequence is not relevant. When we wish to append an element x to the end of a sequence S we write $\langle S, x \rangle$.

Public-key signatures. We assume a EU-CMA signature scheme with security parameter κ. For consistency, we represent the computations related to this scheme as interaction with a signature oracle Σ in the following way:

- Each participant has a public/secret key pair (pk, sk) known to Σ.
- On query $\Sigma.\mathsf{sign}(sk, msg)$:
  A signature $sig \in \{0,1\}^\kappa$ is generated and the tuple (sk, msg, sig) is stored in memory. Return sig.
- On query $\Sigma.\mathsf{verify}(pk, msg, sig)$:
  If (sk, msg, sig) is in the memory of Σ and (pk, sk) is a valid keypair, return 1. Otherwise, return 0.

We assume the probability that any PPT adversary forges a signature without knowledge of the corresponding secret key is negligible in κ.

3 Verifiable Delay Functions

Informally, verifiable delay functions are functions that require inherently sequential computation and can be efficiently verified.

Definition 3.1. *A* ***verifiable delay function*** *(VDF) is a triple of algorithms* $(\mathsf{init}_{\mathsf{VDF}}, \mathsf{eval}_{\mathsf{VDF}}, \mathsf{verify}_{\mathsf{VDF}})$ *with security parameter* μ *and parameters* $g, v \in \mathbb{N}$

$\mathsf{init}_{\mathsf{VDF}}(\mu) \to pp$ *A probabilistic algorithm which, given a security parameter* μ *outputs public parameters* pp.

$\mathsf{eval}_{\mathsf{VDF}}(pp, x, s) \to (y, p)$ *is a slow cryptographic algorithm that given public parameters* pp*, input* $x \in \{0,1\}^*$ *and a strength parameter* s *computes an output* y *and a proof* p.

$\mathsf{verify}_{\mathsf{VDF}}(pp, x, s, y, p)$ *is a fast cryptographic algorithm that for public parameters* pp*, input* x*, strength parameter* s*, output* y *and proof* $p \in \{0,1\}^\mu$ *outputs* 1 *if* y *is the correct output of the VDF on input* x *given* pp *and* s.

For clarity, we avoid writing pp as inputs to $\mathsf{eval}_{\mathsf{VDF}}$ and $\mathsf{verify}_{\mathsf{VDF}}$. As verifiable delay functions are a recent invention, the current definitions differ slightly from each other. Our definition is closer to the definitions in [14, 15] where both the solving and verification algorithms get as input a parameter s that represents the number of sequential work necessary to execute $\mathsf{eval}_{\mathsf{VDF}}$. In other constructions,

this parameter is input only to the initialization algorithm [3, 4]. Choosing to use it as an input at the time of execution allows us to run VDFs for different strengths given the same initial parameters. While generally VDFs are computed with the strength determined in advance, it is possible to execute some of them [14, 15] and halting them at some unknown point in the future.

A VDF must satisfy the following security properties:

Correctness $\mathsf{verify}_{\mathsf{VDF}}(x, s, \mathsf{eval}_{\mathsf{VDF}}(x, s)) = 1$ for all $x \in \{0,1\}^*$ and $s \in \mathbb{N}$.

Soundness The probability that $\mathsf{verify}_{\mathsf{VDF}}(x, s, y, p) = 1$ when $y \neq \mathsf{eval}_{\mathsf{VDF}}(x, s)$ is negligible.

Sequentiality No efficient algorithm $\mathcal{A}$ that executes less than s sequential steps can compute values (y, p) for an input x (with sufficient min-entropy) such that $\mathsf{verify}_{\mathsf{VDF}}(x, s, y, p) = 1$ with non-negligible probability.

Uniqueness For each x there an unique y such that $\mathsf{verify}_{\mathsf{VDF}}(x, s, y, p) = 1$.

Additionally, we expect verification of the VDF to be efficient. The algorithm $\mathsf{verify}_{\mathsf{VDF}}$ must not take more than $O(\log(s))$ sequential computational steps.

Wesolowski presents a simple VDF in [15] based on iterated squaring in groups of unknown order. His construction is particularly practical because of its succinctness, as the proof consists of a single group element. Additionally, it does not require to choose the **strength** s before starting the computation. Computing the function consists of repeated squarings, which can be stopped at any moment to get an output of strength s after s squarings. Continuous execution has only a linear cost on space, in order to save enough values to efficiently compute the proof p.

Wesolowski's VDF fulfills all the security properties expected from a VDF and is the model from which we construct our functionality $\mathcal{F}^{\Gamma}_{\mathsf{VDF}}$. Additionally, the security of our scheme is based on assumptions on the possibility of computing the VDF quickly enough to ensure that the adversary's advantage through faster hardware is minimal. Research on optimized algorithms and specialized hardware for computing [15] is an active research topic[4][5] meaning that our assumptions should not be too far from reality.

3.1 The VDF Functionality

The primary goal of our construction is to generate non-interactive proofs that a certain amount of time has passed since a message was recorded by the prover. The first step to do that is representing our VDF in the UC framework.

The universal composability framework is very powerful but requires multiple simplifying assumptions. In order to avoid explicitly dealing with simulation overheads and fine-grained complexity in general, UC deals with closed complexity classes. Any time an ITM is activated, it can perform arbitrary polytime computations. Without additional tools, it is impossible to quantify the amount of computation executed by any party. Moderately-hard functions, like VDFs

[4] https://vdfresearch.org/

[5] https://medium.com/@chia_network/chia-vdf-competition-guide-5382e1f4bd39

Oracle $\mathcal{F}_{\mathsf{VDF}}^{\Gamma}$

The functionality is parametrized by a set Γ and a parameter ψ and has access to the clock $\mathcal{F}_{\mathsf{clock}}$ and the random oracle $\mathsf{VDF} : \{0,1\}^* \times \mathbb{N} \to \{0,1\}^\mu \times \{0,1\}^\mu$. The functionality manages a set $\mathcal{P}$ of parties P_i. For every party, it manages a query set Q_i and a rate $\gamma_i \in \Gamma$. The functionality holds corresponding parameters $Q_\mathcal{A}$ and $\gamma_\mathcal{A}$ for the adversary. All Q_i as well as $\mathcal{P}$ are initialized as $\varnothing$.
Whenever $\mathcal{F}_{\mathsf{VDF}}^{\Gamma}$ is activated, it sets $t^* \leftarrow \mathcal{F}_{\mathsf{clock}}.\texttt{clock-read}$.

Computing the function

- On input $(\texttt{start}, x)$ from any party P_i or the adversary $\mathcal{A}$, the functionality adds (x, t^*) to Q_i. It then sends `started` back.
- On input $(\texttt{output}, x)$ from any party P_i or the adversary $\mathcal{A}$, the functionality checks if the party started the computation, that is $(x, t_s) \in Q_i$. If it did not, output $\perp$. If it did, compute the strength s by taking the time elapsed since the start and letting $s \leftarrow \lfloor {t^* - t_s}/{\gamma_i} \rfloor$. Let $(y, p) \leftarrow \mathsf{VDF}(x, s)$ and return (s, y).
- On input $(\texttt{proof}, x, s, y)$ from a party P_i or the adversary $\mathcal{A}$, check whether $(x, t') \in Q_i$ with $t^* - t' \leq \gamma_i s + \lceil \psi/\gamma_i \rceil$. If they did not, return $\perp$. Otherwise, let $(y, p) \leftarrow \mathsf{VDF}(x, s)$ and return p. Otherwise, return $\perp$.

Verification

- On input $(\texttt{verify}, x, y, s, p)$ from any party P_i or $\mathcal{A}$, checks whether $(y, p) = \mathsf{VDF}(x, s)$. If so, outputs 1. Otherwise, output 0.

Corruption

- Whenever $\mathcal{A}$ corrups a party P_i, additionally update the adversarial query list: $Q^*_\mathcal{A} \leftarrow Q^*_\mathcal{A} \cup \{(x, t^* - {\gamma_\mathcal{A}}/{\gamma_i}(t^* - t)) \mid (x, t) \in Q_i\}$. Then send Q_i to $\mathcal{A}$;

Fig. 1: The functionality $\mathcal{F}_{\mathsf{VDF}}^{\Gamma}$ is the only way to query the random oracle VDF.

as well as proofs-of-resource (work, space, replication etc.) are functions that will eventually be computed at a cost of time or some other resource. Universal composability is not natively equipped to handle such functions, as there is no natural way to represent this cost.

A way to utilize these functions in UC is through functionality wrappers, as exemplified in [2] where parties can only query the random oracle a certain number of times per timestep We follow a similar approach, where our VDF functionality is the only way parties can access an underlying random oracle. It is a natural question whether the VDF presented in [15] can actually UC-realize our functionality. This question goes beyond the scope of our paper, but further work is ongoing to understand whether it is possible to prove this within the constraints of the UC framework. On the other hand, the functionality fulfills the expected security properties of a VDF.

Our functionality $\mathcal{F}_{\mathsf{VDF}}^{\Gamma}$ is described in Figure 1 and simulates the continuous execution of the VDF. When the execution of the VDF is finished, the party gets back the output of the VDF after a certain number of iterations, which we

call *strength*. Whenever an execution of the VDF is completed, the strength is also output to the party, so they receive the pair (s, y). In order to calculate the appropriate strength of an execution after a certain amount of time we introduce the parameter γ_P which represents the time needed for an iteration of the underlying function for party P. We call this parameter the **rate of party** P. In this setting with only one party, we assume the adversary has rate $\gamma_{\mathcal{A}} = \gamma/\alpha$ where her advantage $\alpha \geq 1$ is not too large (≤ 2). While we cannot ensure the physical reality of this assumption, the impossibility of parallelization greatly limits this possible advantage, in contrast to generic (parallelizable) computation.

The functionality has access to the random oracle VDF which has an output of size 2μ which we parse as two distinct outputs: the result of the VDF (y) and its proof (p), which allows for verification. On occassion we will refer to the pair (y, p) as Y. Parties can only access this random oracle through $\mathcal{F}_{\mathsf{VDF}}^{\Gamma}$. Given an `output` query, they get the function output y, which constitutes the first half of the random oracle, as well as the strength. A `proof` query, requires the output y and returns p. This functionality depends on the clock functionality $\mathcal{F}_{\mathsf{clock}}$ presented in Appendix A.1.

The functionality $\mathcal{F}_{\mathsf{VDF}}^{\Gamma}$ captures the desirable properties of a verifiable delay function. Correctness, soundness and uniqeness follow from the use of a random oracle for VDF. The functionality models sequentiality because a certain amount of time must pass before a party can get certain output. The functionality extends the standard corruption mechanism [6, §7.1] of an adversary corrupting a party and allows not only the adversary taking over that party's existing VDF computations, but also to continue them with its faster rate. To do so, the functionality checks the current strength of the computed function ($t^* - t/\gamma$) and multiplies it by the adversary's rate to compute the hypothetical time in which the adversary would have started the computation to reach the strength at corruption. It then saves that time in the adversary's query log $Q_{\mathcal{A}}$.

Participants are able to input any string to the functionality, as a full-domain hash is applied to any inputs in our choice of VDF to prevent trivially re-using previously computed proofs. We require a canonical unambiguous encoding of integers $s \in \mathbb{N}$ as bitstrings, so (s, y, p) has a natural description as a bitstring $s||y||p \in \{0, 1\}^*$.

4 Creating a timestamping scheme

In this section we will first present the timestamping functionality $\mathcal{F}_{\mathrm{ts}}^{\alpha}$ and then a protocol that realizes it through our VDF functionality.

4.1 A Timestamping Functionality

The goal of the timestamping functionality $\mathcal{F}_{\mathrm{ts}}^{\alpha}$ is to generate proofs of *age* and prevent dishonest parties from claiming that something is older than it really is. Because we want to reflect the realities of an adversary having access to faster computational rates, the functionality allows for timestamps to be

forged under certain circumstances. The adversary will be able to dilute the proven age by at most a time-dilution factor $\alpha \geq 1$. We then show that we can efficiently and securely instantiate this functionality with the use of VDFs. What our functionality does not do is prevent an attacker from post-dating a record, that is, pretending the record is "younger" and was first recorded later than it was. A simpler functionality, based on the construction from [11] can be found in Appendix C. The main difference between the two is that the following functionality does not allow the adversary to take advantage of already existing timestamps. That is, an adversary must start from scratch whenever she tries to forge a timestamp and cannot take advantage of honestly-computed work. We present our functionality in Figure 2.

The functionality accepts three inputs from the parties and manages the corruption of the prover. When the prover is corrupted, the adversary can start forging timestamps by a factor α. The functionality receives records through the `record` query and stores them when this happens. It generates a timestamp whenever it receives a `stamp` input with the appropriate record. The adversary can choose how the timestamp looks like but can only modify the age if she has corrupted the prover. She is limited by the checkstamp procedure, which checks whether the presented age of the timestamp is correct. If the prover is honest, it simply checks whether the age in the timestamp does not exceed the time elapsed since the record was originally queried. When the prover has been corrupted, the adversary can stretch a timestamp by a factor α but only if enough time has elapsed since corruption of the prover. An adversary can only modify a timestamp if she has been in control of the functionality for at least the age of the timestamp divided by α. This implies that any accepted timestamp produced by the adversary with claimed age older than $\alpha \cdot (\mathcal{F}_{\mathsf{clock}}.\texttt{clock-read} - t_{\mathrm{corr}})$ is truthful. Such a assertion cannot be established for the scheme in [11]. The procedure additionally checks whether the triple of record, timestamp and age has been registered before. Then, the triple is registered with a validity bit v which states whether the timestamp is valid. The `verify` query simply checks whether a triple is in the list of generated timestamps and outputs the associated validity bit. If the timestamp was not previously generated but is within acceptable parameters, it queries the adversary whether it is a valid timestamp or not and outputs the adversary's answer.

4.2 Random Oracle Sequences

In order to construct a protocol that realizes our functionality, we need to introduce some additional concepts. Exclusively using verifiable delay functions for timestamping is not practical, as it requires continuous execution of the function in order to have an up-to-date timestamp. Additionally, each record would need a different instance of the VDF, making it impractical to timestamp multiple records. Instead, our construction is based on the hashchain originally presented in [9]. Instead of simply providing an ordering of events, the VDFs allow for a proof of age. Our timestamps consist of sequences of VDF-proofs that are linked to each other through cryptographic hash functions, modelled

Timestamping functionality $\mathcal{F}_{\mathbf{ts}}^{\alpha}$

The functionality is parametrized with an adversarial time-diluting factor $\alpha \geq 1$. It answers queries for a dummy prover $\mathcal{P}$ and a set $\mathcal{V}$ of dummy verifiers V_i and interacts with an adversary $\mathcal{A}$.
Let $t_{\text{corr}} \leftarrow \infty$ represent the time the adversary sends a corrupt message to the functionality. It maintains two internal lists: $R \subset \{0,1\}^* \times \{0,1\}^\theta$ for (record,time)-tuples and $C \subset \{0,1\}^* \times \{0,1\}^\theta \times \{0,1\}^* \times \{0,1\}$ for (record, age, proof, valid)-tuples. Whenever $\mathcal{F}_{\text{ts}}^{\alpha}$ is activated, it sets $t^* \leftarrow \mathcal{F}_{\mathsf{clock}}.\mathtt{clock\text{-}read}$.

Creating a timestamp

- On input $(\mathtt{record}, c)$, the functionality records (c, t^*) in R and sends a message $(\mathtt{record}, c)$ to the adversary. After she responds with `ok`, it sends `ok` back to the prover.
- On input $(\mathtt{stamp}, c)$, the functionality relays $(\mathtt{stamp}, c)$ to the adversary. After she responds with (c, a, u, v), the functionality runs the procedure $\mathsf{checkstamp}(c, a, u, v)$ and returns (c, a, u) to the prover, where u is the timestamp string certifying age a.

Verifying a timestamp

- On input $(\mathtt{verify}, c, a, u)$, if a tuple of the form $(c, a, u, \hat{v})$ exists in C then it outputs $\hat{v}$. If no such tuple is found then it inputs $(\mathtt{cnewstamp}, c, a, u)$ to the adversary and gets back (v') and runs $\mathsf{checkstamp}(c, a, u, v')$. After that, there exists a unique tuple $(c, a, u, \hat{v})$ in C and it outputs $\hat{v}$.

Party Corruption

- On input $(\mathtt{corrupt})$ to $\mathcal{P}$, `set` $t_{\text{corr}} \leftarrow t$, beyond standard corruption management [6, §7.2].

Procedure $\mathsf{checkstamp}(c, a, u, v)$

`Let` $\hat{t} = \min(\{t \mid (c, t) \in R\} \cup \{\infty\}\ )$.
`If` $a > (t^* - \hat{t})$: `/* Claimed age` a `larger than true age */`
  `If` $a > \alpha \cdot \min(t^* - t_{\text{corr}}, t^* - \hat{t})$ `then` $v \leftarrow 0$;
`If` $\exists (c, a, u, \hat{v}) \in C$ `then` $v \leftarrow \hat{v}$; `/* For consistency. */`
`Let` $C \leftarrow C \cup (c, a, u, v)$.

Fig. 2: Timestamping functionality $\mathcal{F}_{\text{ts}}^{\alpha}$

as random-oracle sequences as originally presented in [7]. We enhance these constructions by adding VDFs to the sequences, maintaining the property that dictates that such sequences can only be built in a sequential manner, allowing us to realize our timestamping functionality.

First we introduce Merkle trees, which will allow us to construct our desired sequences. 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 a canonical deterministic way.

$\mathsf{MT.root}(T)$ computes the root hash h of the Merkle Tree for some finite ordered sequence $T = \langle x_i \mid x_i \in \{0,1\}^* \rangle$ 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 $\langle x_i \mid x_i \in \{0,1\}^\lambda \rangle_\ell$ where $x_0 = v$, $x_{\ell-1} = \mathsf{MT.root}(T)$, $x_i \in \{0,1\}^\lambda$ 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 $P = \langle x_i \mid x_i \in \{0,1\}^\lambda \rangle_\ell$ outputs 1 if P is a valid Merkle path. It outputs 0 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. For example, 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.

Our timestamps are based on random-oracle sequences, originally presented in [7]. In contrast with that work, our sequences are generated by two distinct random oracles, H and VDF, which is why we call them $H2$-sequences:

Definition 4.1 (H2-sequence). *Given functions* $\mathsf{H} : \{0,1\}^* \to \{0,1\}^\lambda$ *and* $\mathsf{VDF} : \{0,1\}^* \times \mathbb{N} \to \{0,1\}^\mu$, *an H2-sequence of length* ℓ *is defined as a sequence* $S = \langle (s_i, x_i) \mid s_i \in \mathbb{N} \cup \{\bot\}, x_i \in \{0,1\}^* \rangle_\ell$, *where the following holds for each* $0 \leq i < \ell$: *if* $s_i = \bot$ *then* $\mathsf{H}(x_i)$ *is contained in* x_{i+1} *as continuous substring*[6]; *otherwise* $s_i \in \mathbb{N}$ *and* $\mathsf{VDF}(s_i, x_i)$ *is contained in* x_{i+1} *as a continuous substring. We let* I_{VDF} *be the index set of all elements* $(s_i, x_i) \in S$ *such that* $s_i \neq \bot$ *and call it the* VDF-index set *of* S *and we call* $S[I_{\mathsf{VDF}}] = \langle (s_i, x_i) \in S \mid i \in I_{\mathsf{VDF}} \rangle$ *the* VDF-subsequence *of* S. *We refer to* $\mathsf{str}(S) = \sum_{i \in I_{\mathsf{VDF}}} s_i$ *as the* strength *of the* $H2$*-sequence* S.

Additionally, our sequences must contain time receipts, which we need to ensure that adversaries cannot take advantage of existing timestamps in order to forge new timestamps, forcing them to start from scratch.

Definition 4.2 ($H2T$-sequence). *Let* $S = \langle (s_i, x_i) \rangle_\ell$ *be an* $H2$*-sequence of length* ℓ *with* I_{VDF} *the* VDF*-index set of* S *and* $I_{\mathsf{VDF}}^{-1} = \{i - 1 \mid i \in I_{\mathsf{VDF}}, s_{i-1} = \bot\}$. *We call* S *an* ***H2T-sequence*** *if the following properties hold:*

1. *For* $i \in I_{\mathsf{VDF}}^{-1} \cup I_{\mathsf{VDF}}$: $x_i = t_i || r_i$ *where* $t_i \in \{0,1\}^\theta$ *is a time receipt and* r_i *is an arbitrary string.*
2. *For all* $i, j \in I_{\mathsf{VDF}}^{-1} \cup I_{\mathsf{VDF}}$: *if* $i < j$ *then* $t_i \leq t_j$.
3. *For all* $i, j \in I_{\mathsf{VDF}}$: *if* $i < j$ *then* $t_i < t_j$.

[6] That is, $x_{i+1} = a || \mathsf{H}(x_i) || b$ for some $a, b \in \{0,1\}^*$

We say S has ε ***delay*** *if for all $i \in I_{\mathsf{VDF}}$, if $i-1 \in I^{-1}_{\mathsf{VDF}}$ then we have that $t_i - t_{i-1} \leq \varepsilon$. If $I \neq \varnothing$ then we call the first element of $S[I^{-1}_{\mathsf{VDF}}]$ the* root *of S ($\mathsf{root}(S)$) and the time receipt $t_{\min}$ in $\mathsf{root}(S)$ the* root time *of S and we call $\mathsf{age}(S) = t_{\max} - t_{\min}$ the* age *of the sequence, where $t_{\max} = \max\{t_i \mid i \in I_{\mathsf{VDF}}\}$ is the last time receipt.*

It is important to make sure that these sequences can only be constructed sequentially, one element at a time. When she has access to a faster rate, the adversary will be able to *forge* a sequence's age up to a certain point. We can bound the ability of the adversary to create a sequence which seems τ time steps older than it really is.

Lemma 4.3 (Unforgeable $H2T$-sequences). *If $t_1 - t_0 < \tau \cdot \frac{\gamma}{\gamma_{\mathcal{A}} - \gamma}$ then an adversary can create an $H2T$-sequence S of age $\mathsf{age}(S) = t_1 - (t_0 - \tau)$ within $t_1 - t_0$ time steps 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 $\mathcal{F}^{\Gamma}_{\mathsf{VDF}}$.

Proof. The proof can be found in the appendix (Lemma B.7).

Additionally, we use signatures to prevent the adversary to output valid sequences without corrupting the prover.

Definition 4.4 ($H2TS$-sequence). *Let pk be a public key for a signature scheme Σ and $S = \langle (s_i, x_i) \rangle_\ell$ be an $H2T$-sequence of length ℓ with I_{VDF} and I^{-1}_{VDF} defined the same way as Definition 4.2. We call S an $H2TS$****-sequence for*** *pk if for $i \in I^{-1}_{\mathsf{VDF}}$: $x_i = t_i || r_i || \sigma_i$ where $t_i \in \{0,1\}^\theta$ is a time receipt and $\sigma_i \in \{0,1\}^\kappa$ is such that $\Sigma.\mathsf{verify}(pk, t_i||r_i, \sigma_i) = 1$.*

The aforementioned sequences provide the security properties that we expect our timestamps to have. Instead of dealing with them directly, our prover maintains a list of blocks, each containing the VDF proof and the record to be timestamped as well as additional information. These blocks are chained through the use of hash functions, each block containing a hash of the previous block, similar to a blockchain.

Definition 4.5 (Block). *We define a* block *for a party $\mathcal{P}$ with public key $pk \in \{0,1\}^*$ as a tuple $B = (rnd, prev, vi, vo, t, c)$ 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. *$vi = (t^u, sig) \in \{0,1\}^\theta \times \{0,1\}^\kappa$ is a (time receipt, signature)-pair such that $\Sigma.\mathsf{verify}(pk, t^u||prev, sig) = 1$;*
4. *$vo = (s, Y)$ is a VDF output: $Y = \mathsf{VDF}(t^u||prev||sig, s)$*
5. *$t \in \{0,1\}^\theta$ is a time receipt of the creation of the block;*

6. $c \in \{0,1\}^*$ *is the entry to be timestamped;*

For convenience we use the notation $B.pk$, $B.rnd$, $B.prev$, $B.vi$, $B.vo$, $B.t^u$, $B.sig$, $B.t$, $B.s$, $B.Y$ and $B.c$ to refer to these elements in block B. Note that $B.Y$ is the pair (y, p) which is an output of VDF.

We assume that there is a canonical construction for the Merkle tree of a block such that $B.t||B.Y$ is a leaf of the tree. This choice comes from the fact that elements of $S[I_{\mathsf{VDF}}]$ look like this, where each element of the sequence consists of the output of a VDF (Y) preceded by a time receipt (t).

Additionally, we have that *prev* and c must also be leaves, for similar reasons. These assumptions allow for an easy characterization of the link between these instances and the next VDF input.

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

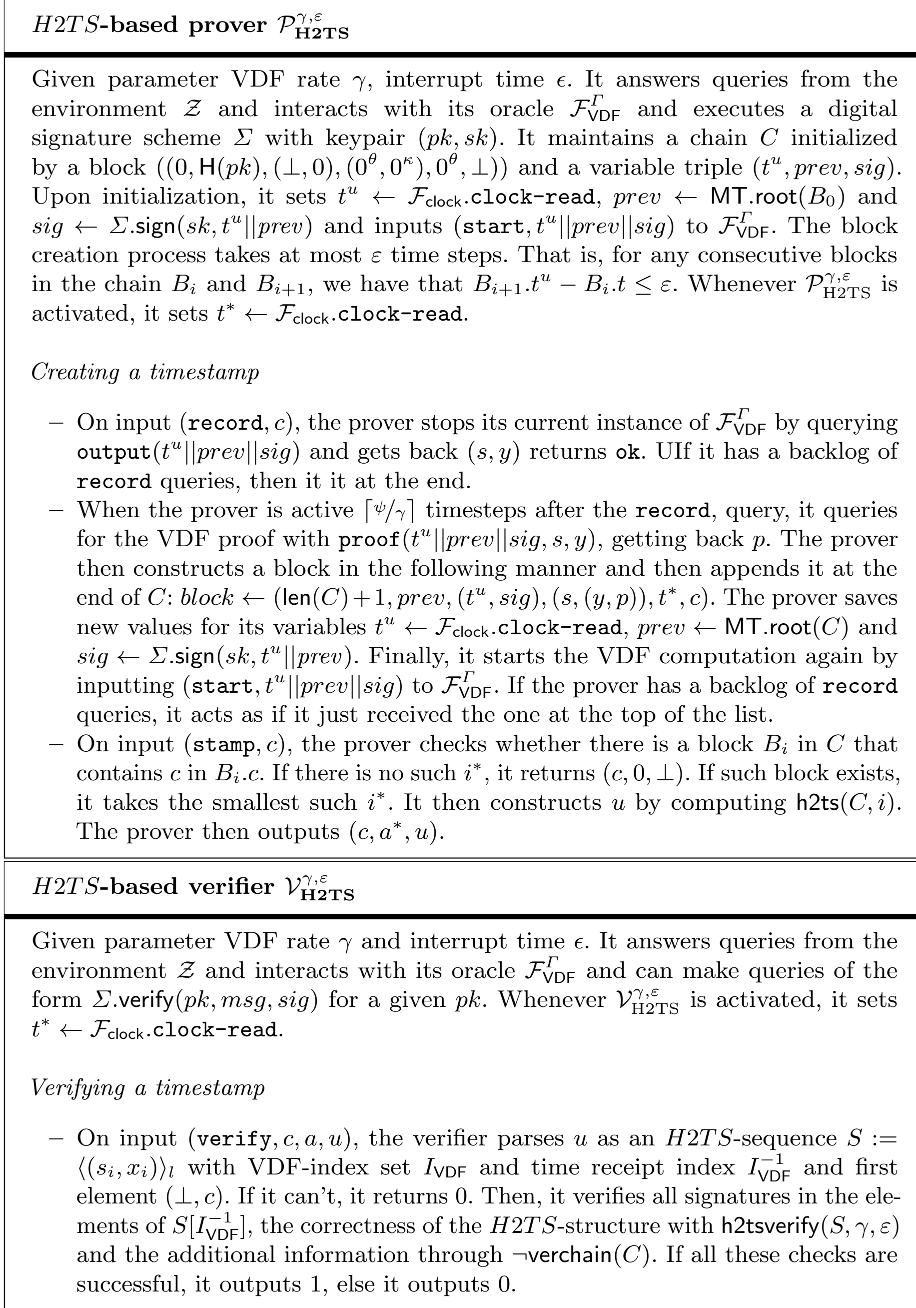
1. $B_i.rnd = i$;
2. $B_0.prev = \mathsf{H}(pk)$ *and* $B_i.prev = \mathsf{MT.root}(B_{i-1})$ *for* $i > 0$;
3. $B_i.Y = \mathsf{VDF}(B_i.t^u||B_i.prev||B_i.sig, B_i.s)$ *for* $i \geq 0$;
4. $\Sigma.\mathsf{verify}(pk, B_i.t^u||B_i.prev, B_i.sig) = 1$;
5. $B_i.t < B_j.t$ *for all* $i < j \leq k$;

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

Having $B_i.prev = \mathsf{MT.root}(B_{i-1})$ (and *prev* being a leaf of the canonical Merkle tree) allows us to generate a Merkle tree for the entire chain by concatenating Merkle trees through *prev*. This construction allows us to create an $H2TS$-sequence starting from any element in a block (in particular c) and ending at the end of the chain, passing through every VDF proof and including every time receipt t and t^u. This allows a party to attest an age of c. For a detailed explanation of this construction, see Appendix B.1. Given a chain C we refer to the $H2TS$-sequence starting from $B_i.c$ and going through the entire subchain $C[i :]$ as $\mathsf{h2ts}(C, i)$. Note that if we want to keep the timestamped records secret, we can ensure that $B.c$ is simply a hash of the record. The first element of the $H2TS$-sequence would simply be the record, followed by its hash.

4.3 Realizing the Functionality

In Figure 3 we present a timestamping protocol based on the chains presented in Definition 4.6 and $H2TS$-sequences. We then show that this protocol realizes the functionality $\mathcal{F}_{\text{ts}}^{\alpha}$ from Figure 2. Each time that the prover gets a new `record` query, they stop their current execution of $\mathcal{F}_{\mathsf{VDF}}^{\Gamma}$, then create a block containing the input from the `record` query and the output of $\mathcal{F}_{\mathsf{VDF}}^{\Gamma}$. Finally, they query the VDF functionality again with this block as an input. When creating a



$H2TS$-based prover $\mathcal{P}^{\gamma,\varepsilon}_{\mathbf{H2TS}}$

Given parameter VDF rate γ, interrupt time ϵ. It answers queries from the environment $\mathcal{Z}$ and interacts with its oracle $\mathcal{F}^{\Gamma}_{\mathsf{VDF}}$ and executes a digital signature scheme Σ with keypair (pk, sk). It maintains a chain C initialized by a block $((0, \mathsf{H}(pk), (\perp, 0), (0^{\theta}, 0^{\kappa}), 0^{\theta}, \perp))$ and a variable triple $(t^u, prev, sig)$. Upon initialization, it sets $t^u \leftarrow \mathcal{F}_{\mathsf{clock}}.\mathtt{clock\text{-}read}$, $prev \leftarrow \mathsf{MT.root}(B_0)$ and $sig \leftarrow \Sigma.\mathsf{sign}(sk, t^u||prev)$ and inputs $(\mathtt{start}, t^u||prev||sig)$ to $\mathcal{F}^{\Gamma}_{\mathsf{VDF}}$. The block creation process takes at most ε time steps. That is, for any consecutive blocks in the chain B_i and B_{i+1}, we have that $B_{i+1}.t^u - B_i.t \leq \varepsilon$. Whenever $\mathcal{P}^{\gamma,\varepsilon}_{\mathrm{H2TS}}$ is activated, it sets $t^* \leftarrow \mathcal{F}_{\mathsf{clock}}.\mathtt{clock\text{-}read}$.

Creating a timestamp

- On input $(\mathtt{record}, c)$, the prover stops its current instance of $\mathcal{F}^{\Gamma}_{\mathsf{VDF}}$ by querying $\mathtt{output}(t^u||prev||sig)$ and gets back (s, y) returns $\mathtt{ok}$. UIf it has a backlog of $\mathtt{record}$ queries, then it it at the end.
- When the prover is active $\lceil \psi/\gamma \rceil$ timesteps after the $\mathtt{record}$, query, it queries for the VDF proof with $\mathtt{proof}(t^u||prev||sig, s, y)$, getting back p. The prover then constructs a block in the following manner and then appends it at the end of C: $block \leftarrow (\mathsf{len}(C)+1, prev, (t^u, sig), (s, (y, p)), t^*, c)$. The prover saves new values for its variables $t^u \leftarrow \mathcal{F}_{\mathsf{clock}}.\mathtt{clock\text{-}read}$, $prev \leftarrow \mathsf{MT.root}(C)$ and $sig \leftarrow \Sigma.\mathsf{sign}(sk, t^u||prev)$. Finally, it starts the VDF computation again by inputting $(\mathtt{start}, t^u||prev||sig)$ to $\mathcal{F}^{\Gamma}_{\mathsf{VDF}}$. If the prover has a backlog of $\mathtt{record}$ queries, it acts as if it just received the one at the top of the list.
- On input $(\mathtt{stamp}, c)$, the prover checks whether there is a block B_i in C that contains c in $B_i.c$. If there is no such i^*, it returns $(c, 0, \perp)$. If such block exists, it takes the smallest such i^*. It then constructs u by computing $\mathsf{h2ts}(C, i)$. The prover then outputs (c, a^*, u).

$H2TS$-based verifier $\mathcal{V}^{\gamma,\varepsilon}_{\mathbf{H2TS}}$

Given parameter VDF rate γ and interrupt time ϵ. It answers queries from the environment $\mathcal{Z}$ and interacts with its oracle $\mathcal{F}^{\Gamma}_{\mathsf{VDF}}$ and can make queries of the form $\Sigma.\mathsf{verify}(pk, msg, sig)$ for a given pk. Whenever $\mathcal{V}^{\gamma,\varepsilon}_{\mathrm{H2TS}}$ is activated, it sets $t^* \leftarrow \mathcal{F}_{\mathsf{clock}}.\mathtt{clock\text{-}read}$.

Verifying a timestamp

- On input $(\mathtt{verify}, c, a, u)$, the verifier parses u as an $H2TS$-sequence $S := \langle (s_i, x_i) \rangle_l$ with VDF-index set I_{VDF} and time receipt index I^{-1}_{VDF} and first element $(\perp, c)$. If it can't, it returns 0. Then, it verifies all signatures in the elements of $S[I^{-1}_{\mathsf{VDF}}]$, the correctness of the $H2TS$-structure with $\mathsf{h2tsverify}(S, \gamma, \varepsilon)$ and the additional information through $\neg\mathsf{verchain}(C)$. If all these checks are successful, it outputs 1, else it outputs 0.

Fig. 3: $H2TS$-based Timestamping Protocol

timestamp, the prover finds the block with the expected record and extracts the $H2TS$-sequence from it up to the end of the chain.

It is important to ensure the correct structure of the timestamp sequences to meaningfully realize $\mathcal{F}_{\mathrm{ts}}^{\alpha}$. Timestamps can be split and recombined, creating new timestamps without interacting with the functionality. While this is a desirable feature in general, it introduces complications in UC. Our functionality can deal with cases of timestamps that are merged to create a longer timestamp for a certain value. In these cases, the functionality asks the adversary whether the new proof is valid. The adversary is still not able to make the functionality accept a proof that is longer than it should be (the timestamp must pass checkstamp).

However, our functionality is not equipped to deal with other cases that would occur naturally. For example, take the following $H2TS$-sequence, where $z = \mathsf{H}(x)||\mathsf{H}(y)$:

$$\langle(\perp, x), (\perp, \mathsf{H}(x)||\mathsf{H}(y)), (\perp, t||\mathsf{H}(z)||sig), (s, t^*||\mathsf{VDF}(x_2, s))\rangle.$$

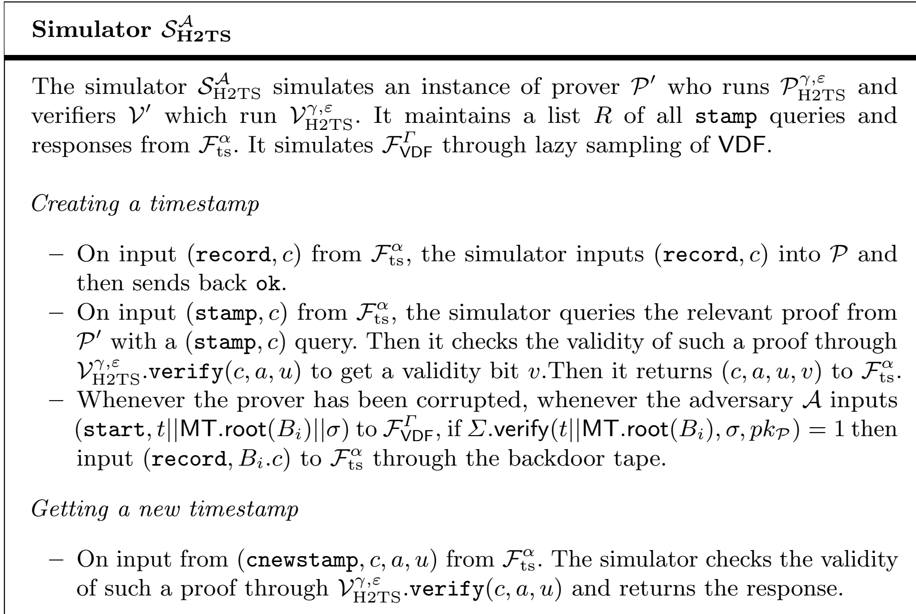
It is clear that substituting the first element of the sequence with $(\perp, y)$ results in a valid $H2TS$-sequence of strength s for y. In order to properly construct a protocol that realizes this functionality we must either give the functionality understanding of the structure of Merkle trees or "artificially" require additional parameters for verification. We choose to do the latter. The verifier can check whether an $H2TS$-sequence corresponds to the canonical Merkle tree of a chain. This makes cases like the previous example invalid (assuming that x was the content c of the block). We call this verification function verchain.

We constructed our $H2TS$-sequences with an ε-delay, representing the time between VDF executions. Such a delay is required as we need to take into account the time spent creating a new block. For simplicity, we assume that the adversary also has an advantage constructing the chain. Instead of taking ε time steps, the adversary takes $\varepsilon_{\mathcal{A}} = \lceil \varepsilon/\alpha \rceil$. In this setting, we consider that $\lceil \psi/\gamma \rceil < \varepsilon$ as it is necessary to add the proof of the VDF for quick verification. In practice, the proof may be presented in a different place, allowing us to make ε smaller.

Theorem 4.7. *Let $\mu \in \mathbb{N}^+$ be the security parameter. For any real-world PPT adversary $\mathcal{A}$ with oracle access to $\mathcal{F}_{\mathsf{VDF}}^{\Gamma}$, there exists a black-box PPT simulator $\mathcal{A}_{id}$ such that for any PPT environment $\mathcal{Z}$ the probability that $\mathcal{Z}$ can distinguish between the ideal world with $\mathcal{F}_{ts}^{\alpha}$ and $\mathcal{A}_{id}$ (cf. Figure 2,5) and the real world with $\mathcal{A}$, $\mathcal{P}_{H2TS}^{\gamma,\varepsilon}$ and $\mathcal{V}_{H2TS}^{\gamma,\varepsilon}$ (cf. Figure 3) is negligible in μ, λ and κ.*

Proof. We omit the proof for space reasons, it can be found in Appendix B.2 and is based on the simulator presented in Figure 4.

We have shown that we can create secure timestamps through random-oracle sequences. This result also allows for an efficient way to create timestamps for a large number of records through Merkle trees. Our analysis naturally extends to that context, as we only require the existence of an $H2TS$-sequence.


Simulator $\mathcal{S}^{\mathcal{A}}_{\mathbf{H2TS}}$

The simulator $\mathcal{S}^{\mathcal{A}}_{\text{H2TS}}$ simulates an instance of prover $\mathcal{P}'$ who runs $\mathcal{P}^{\gamma,\varepsilon}_{\text{H2TS}}$ and verifiers $\mathcal{V}'$ which run $\mathcal{V}^{\gamma,\varepsilon}_{\text{H2TS}}$. It maintains a list R of all `stamp` queries and responses from $\mathcal{F}^{\alpha}_{\text{ts}}$. It simulates $\mathcal{F}^{\Gamma}_{\mathsf{VDF}}$ through lazy sampling of VDF.

Creating a timestamp

- On input $(\texttt{record}, c)$ from $\mathcal{F}^{\alpha}_{\text{ts}}$, the simulator inputs $(\texttt{record}, c)$ into $\mathcal{P}$ and then sends back `ok`.
- On input $(\texttt{stamp}, c)$ from $\mathcal{F}^{\alpha}_{\text{ts}}$, the simulator queries the relevant proof from $\mathcal{P}'$ with a $(\texttt{stamp}, c)$ query. Then it checks the validity of such a proof through $\mathcal{V}^{\gamma,\varepsilon}_{\text{H2TS}}.\texttt{verify}(c, a, u)$ to get a validity bit v.Then it returns (c, a, u, v) to $\mathcal{F}^{\alpha}_{\text{ts}}$.
- Whenever the prover has been corrupted, whenever the adversary $\mathcal{A}$ inputs $(\texttt{start}, t||\mathsf{MT.root}(B_i)||\sigma)$ to $\mathcal{F}^{\Gamma}_{\mathsf{VDF}}$, if $\Sigma.\mathsf{verify}(t||\mathsf{MT.root}(B_i), \sigma, pk_{\mathcal{P}}) = 1$ then input $(\texttt{record}, B_i.c)$ to $\mathcal{F}^{\alpha}_{\text{ts}}$ through the backdoor tape.

Getting a new timestamp

- On input from $(\texttt{cnewstamp}, c, a, u)$ from $\mathcal{F}^{\alpha}_{\text{ts}}$. The simulator checks the validity of such a proof through $\mathcal{V}^{\gamma,\varepsilon}_{\text{H2TS}}.\texttt{verify}(c, a, u)$ and returns the response.

Fig. 4: Simulator $\mathcal{S}^{\mathcal{A}}_{\text{H2TS}}$

5 Beyond Timestamping

We have constructed a timestamping protocol based on verifiable delay functions. Our motivation, however, was to replicate the robustness of proof-of-work blockchains with respect to costless-simulation and long-range attacks. Our timestamping protocol adds a sequential-computation cost that does not have the associated economic and environmental costs of proof-of-work. Fortunately, our construction already looks like a blockchain where the entries of the ledger are encoded in c as defined in Definition 4.5. As long as a blockchain contains the outputs of verifiable delay functions over the block hashes it will be possible to extract the necessary $H2TS$-sequences. We leave it to further work to find the best way to implement this while allowing for network delays and forks.

References

[1] Abadi, A., Ciampi, M., Kiayias, A., Zikas, V.: Timed signatures and zero-knowledge proofs -timestamping in the blockchain era-. Cryptology ePrint Archive, Report 2019/644 (2019), `https://eprint.iacr.org/2019/644`

[2] Badertscher, C., Maurer, U., Tschudi, D., Zikas, V.: Bitcoin as a transaction ledger: A composable treatment. In: Annual International Cryptology Conference. pp. 324–356. Springer (2017)

[3] Boneh, D., Bonneau, J., Bünz, B., Fisch, B.: Verifiable delay functions. In: Annual International Cryptology Conference. pp. 757–788. Springer (2018)

[4] Boneh, D., Bünz, B., Fisch, B.: A survey of two verifiable delay functions. Cryptology ePrint Archive, Report 2018/712 (2018), http://eprint.iacr.org/2018/712
[5] 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)
[6] Canetti, R.: Universally composable security: A new paradigm for cryptographic protocols. Cryptology ePrint Archive, Report 2000/067 (2019), v.20190826:041954 https://eprint.iacr.org/eprint-bin/getfile.pl?entry=2000/067&version=20190826:041954&file=067.pdf
[7] 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)
[8] David, B., Kiayias, A., Russell, A.: Ouroboros praos: An adaptively-secure, semi-synchronous proof-of-stake protocol (2017), http://eprint.iacr.org/2017/573
[9] 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
[10] Katz, J., Maurer, U., Tackmann, B., Zikas, V.: Universally composable synchronous computation. In: Theory of cryptography, pp. 477–498. Springer (2013)
[11] 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)
[12] Moran, T., Shaltiel, R., Ta-Shma, A.: Non-interactive timestamping in the bounded storage model. In: Annual International Cryptology Conference. pp. 460–476. Springer (2004)
[13] Nakamoto, S.: Bitcoin: A peer-to-peer electronic cash system (2008)
[14] Pietrzak, K.: Simple verifiable delay functions. Cryptology ePrint Archive, Report 2018/627 (2018), http://eprint.iacr.org/2018/627
[15] Wesolowski, B.: Efficient verifiable delay functions. Cryptology ePrint Archive, Report 2018/623 (2018), http://eprint.iacr.org/2018/623

A Functionalities

A.1 Clock Functionality

This is the clock functionality originally presented in [10], formulation from [2].

Clock Functionality $\mathcal{F}_{\mathsf{clock}}$

The functionality manages the set $\mathcal{P}$ of parties $\mathcal{P} = \{P_i\}$ which is initialized as $\varnothing$.

The clock maintains a variable τ. For each party $P_i \in \mathcal{P}$ it manages variable d_i which is initialized as 0.

- Upon receiving (`clock-update`) from some party $P_i \in \mathcal{P}$ set $d_i := 1$. If $d_j = 1$ for all honest parties $P_j \in \mathcal{P}$, then set $\tau := \tau + 1$ and reset $d_j := 0$ for all parties $P_j \in \mathcal{P}$.
  In any case, forward (`clock-update`, P_i) to $\mathcal{A}$.
- Upon receiving (`clock-read`) from any participant return (`clock-read`, τ).

B Proofs

B.1 Random-Oracle Sequences

We analyze recursive calls to the random oracle H and to the random oracle VDF underlying all oracles $\mathcal{F}^{\Gamma}_{\mathsf{VDF}}$ and analyze the cumulative strength of the verifiable delay functions that are found in the sequence. We have adapted the following lemmas from [7] to this setting.

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

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

A note on notation: here we consider the output p of the random oracle $\mathsf{VDF}(x, s)$, instead of output (s, p) for query $\mathcal{F}^{\Gamma}_{\mathsf{VDF}}(\texttt{output}, x)$ that includes s in the output.

It is simple to verify that any Merkle path $\mathsf{MT.path}(T, v) = \langle x_0, \ldots, x_{\ell-1}\rangle$ induces an $H2$-sequence of the form $\langle(\perp, x'_0), (\perp, x'_1), \ldots, (\perp, x'_{\ell-2}), (\perp, x_{\ell-1})\rangle$ of length ℓ, where x_i is a substring of x'_i for $0 \leq i < \ell - 1$. With an abuse of notation, we refer to Merkle path $\mathsf{MT.path}(T, v)$ as the induced $H2$-sequence of that path whenever it is relevant.

Definition B.2 (linking H2-sequences). *We define* linking *$H2$-sequence $S_2 = \langle(s_2, x_2), \widehat{\ldots}\rangle$ to $H2$-sequence $S_1 = \langle\widetilde{\ldots}(s_0, x_0), (s_1, x_1)\rangle$ where x_1 is a continuous substring of x_2 to result in the $H2$-sequence $S_1 \bowtie S_2 = \langle\widetilde{\ldots}, (s_0, x_0), (s_2, x_2), \widehat{\ldots}\rangle$.*

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

Lemma B.3 (Random Oracles are sequential). *Consider any adversary $\mathcal{A}^{(\mathsf{H},\mathsf{VDF})}$ 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 $\mathsf{VDF} : \{0,1\}^* \times \mathbb{N} \to \{0,1\}^\mu$ 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 VDF, then the probability that it outputs an $H2$-sequence $(\langle (s_0, x_0), \dot{(}s_{\ell-1}, x_{\ell-1})\rangle_\ell$ without making the queries $(s_0, x_0), \ldots, (s_{\ell-1}, x_{\ell-1})$ to respectively H and VDF sequentially is at most*

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

Proof. Constructing an $H2$-sequence without making the queries $(s_0, x_0), \ldots,$ $(s_{\ell-1}, x_{\ell-1})$ sequentially can happen when there is a cycle such that the adversary can repeat previous queries without making them again. I.e., there exist $0 \leq i \leq j < \ell$ such that $\mathsf{H}(x_j)$ (when $s_j = \perp$) or $\mathsf{VDF}(s_j, x_j)$ (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 VDF are uniform randomly selected, the probability of a cycle is upper bounded by $q_\mathsf{H}(Q_\mathsf{H} + Q_\mathsf{VDF})2^{-\lambda} + q_\mathsf{VDF}(Q_\mathsf{H} + Q_\mathsf{VDF})2^{-\mu}$.

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 VDF 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 VDF are uniform randomly selected, the probability of this event is upper bounded by $q_\mathsf{H}(Q_\mathsf{H} + Q_\mathsf{VDF} + \sum_{i=0}^{\ell} |x_i|)2^{-\lambda} + q_\mathsf{VDF}(Q_\mathsf{H} + Q_\mathsf{VDF} + \sum_{i=0}^{\ell} |x_i|)2^{-\mu}$.

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

Thus when an adversary outputs an $H2$-sequence of strength L where $2 \cdot (q_\mathsf{H}2^{-\lambda} + q_\mathsf{VDF}2^{-\mu}) \cdot (Q_\mathsf{H} + Q_\mathsf{VDF} + \sum_{i=0}^{\ell} |x_i|)$ is negligibly small, we can assume that it made all queries sequentially. (In practice this is certainly the case for output lengths λ and μ of 256 bits and larger.) In particular, if the adversary can query $\mathsf{VDF}(x, s)$ only through $\mathcal{F}^{\Gamma}_{\mathsf{VDF}}$ with a rate of γ then each query $\mathsf{VDF}(x, s)$ takes 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 [7] as we aggregate all the calls to VDF 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 VDF. We effectively treat calls to H as “free” with regards to time although the cost of executing them might be relevant in certain contexts.

Using $H2$-sequences is not enough to create timestamps, as an adversary can add things at the end of existing timestamps in order to stretch them. An

adversary wishing to stretch a timestamp can truncate the chain at any point and then append another sequence with a higher strength. This allows an adversary with access to $\mathcal{F}_{\mathsf{VDF}}^{\Gamma}$ to easily create an α-diluted proof, while taking advantage of the work already encoded in the sequence. A first step to prevent this attack is by adding time receipts to the sequences We embed unchangeable time receipts into $H2$-sequences which we then call $H2T$-sequences. These time receipts are enforced by each VDF in the sense that altering the time receipt requires redoing the VDF.

Definition B.4 **($H2T$-sequence).** *Let $S = \langle (s_i, x_i) \rangle_\ell$ be an $H2$-sequence of length ℓ with I_{VDF} the* VDF*-index set of S and $I_{\mathsf{VDF}}^{-1} = \{i-1 \mid i \in I_{\mathsf{VDF}}, s_{i-1} = \bot\}$. We call S an $H2T$-**sequence** if the following properties hold:*

1. *For $i \in I_{\mathsf{VDF}}^{-1} \cup I_{\mathsf{VDF}}$: $x_i = t_i || r_i$ where $t_i \in \{0,1\}^\theta$ is a time receipt.*
2. *For all $i, j \in I_{\mathsf{VDF}}^{-1} \cup I_{\mathsf{VDF}}$: if $i < j$ then $t_i \le t_j$.*
3. *For all $i, j \in I_{\mathsf{VDF}}$: if $i < j$ then $t_i < t_j$.*

*We say S has ε **delay** if for all $i \in I_{\mathsf{VDF}}$, if $i - 1 \in I_{\mathsf{VDF}}^{-1}$ then we have that $t_i - t_{i-1} \le \varepsilon$. If $I \neq \varnothing$ then we call the first element of $S[I_{\mathsf{VDF}}^{-1}]$ the* root *of S ($\mathsf{root}(S)$) and the time receipt $t_{\min}$ in $\mathsf{root}(S)$ the* root time *of S and we call $\mathsf{age}(S) = t_{\max} - t_{\min}$ the* age *of the sequence, where $t_{\max} = \max\{t_i \mid i \in I_{\mathsf{VDF}}\}$ is the last time receipt.*

Due to the inherent sequentiality of the $H2T$-sequences, it is natural to assume that time elapses between the output of one VDF and the input of the next VDF. This leads us to have two different time receipts, the ones representing the time of a VDF output (I_{VDF}) and the ones representing the time of input (I_{VDF}^{-1}). To extract meaningful timestamping, we require this delay to be bounded by ε, which we assume to be negligibly small in relation to the time elapsed during the execution of the VDF.

In order to verify whether these sequences create timestamps with the expected properties, we construct a verification procedure and a sequence-forging game. The verification procedure checks structural correctness of $H2T$-sequences and estimates the VDF-rate with respect to the claimed *root time* of S.

Definition B.5 **($H2T$-verification).** *Given a sequence $S = \langle (s_i, x_i) \rangle_\ell$, VDF-rate γ and interrupt-time ϵ, we define the following $H2T$-verification algorithm* $\mathsf{h2tsverify}(S, \gamma, \varepsilon)$ *given access to $\mathcal{F}_{\mathsf{VDF}}^{\Gamma}$:*

- *Check whether S is a valid H2-sequence* i.e. *for all $i < \ell$, either $x_{i+1} = a || \mathsf{H}(x_i) || b$ and $s_{i+1} = \bot$ or $x_{i+1} = a || \mathsf{VDF}(x_i, s_{i+1}) || b$ and $s_{i+1} \neq \bot$ for some $a, b \in \{0,1\}^*$*[7].

[7] In the second case, the verifier must make a $(\mathtt{verify}, x_i, y, s)$ query, as they must not compute the VDF themselves. This requires a particular structure for the entries in $S[I_{\mathsf{VDF}}]$ in order to know what to choose as y. Because these are $H2T$-sequences, we assume that $a \in \{0,1\}^\theta$ and b is the empty string. The verifier prunes the first θ bits from the beginning of x_{i+1} and takes the remaining string as the y to be be input in the verify query.

- *Verify that for all* $i \in I_{\mathsf{VDF}}^{-1} \cup I_{\mathsf{VDF}}$ *we have that* $x_i = t_i || y_i$ *with* $t_i \in \{0,1\}^\theta$*;*
- *For all* $i \in I_{\mathsf{VDF}}^{-1}$*:* $t_{i+1} - t_i \leq s_i/\gamma$*, i.e. every VDF proof has sufficient strength;*
- *Let* $I_{\mathsf{VDF}} = \{i_0, \ldots, i_k\}$ *then for all* $0 < j \leq k$*:* $t_{(i_j - 1)} - t_{i_{(j-1)}} \leq \epsilon$*, i.e. between VDF proofs there is a time gap at most* ϵ*.*

If any of these checks fail, output 0*. Otherwise, output* 1*.*

It is not enough to check the average strength of the sequence over the total age. This verification would allow for timestamp stretching, when one particular VDF instance was computed with a considerably faster rate.

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

Definition B.6 (Sequence-forging game). *For* $\tau > 0$*, positive VDF-rate* $\gamma > 0$*, and adversarial time-dilution factor* $\alpha \geq 1$*, we define the* ***sequence-forging game*** *with respect to a* $H2T$*-verifier* $\mathcal{V}$ *as follows.*

Consider an adversary $\mathcal{A}$ *with access to oracles* $\mathcal{F}_{\mathsf{VDF}}^{\Gamma}$ *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* $c_1, \ldots$ *to* O *later on, but not before time* t_0*. The adversary constructs an* $H2T$*-sequence* $S = \langle (x_i, s_i) \rangle_\ell$ *with root time* $t_0 - \tau$ *and where* c_0 *is a continuous substring of* x_0*. It sends* S *to a verifier* $\mathcal{V}$ *at time* t_1 *and wins if* $\mathcal{V}$ *outputs* 1*.*

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. In the case of timestamping, similar oracles are used to represent the situation when the thing to be timestamped is unpredictable (such as an invention). Interestingly, this oracle can also be interpreted as a signing oracle, where gaining access to the oracle represents the corruption of an honest party and gaining access to the secret key.

In this game, the adversary has access to $\mathcal{F}_{\mathsf{VDF}}^{\Gamma}$ before t_0, so they are allowed to precompute a polynomial amount of $H2T$-sequences. However, for any of these sequences to be useful to $\mathcal{A}$ when constructing a block, they have to be able to link the output of O to these sequences, which can only be done through random oracle collisions. The $H2T$-sequence S that needs to be constructed requires root time $t_0 - \tau$, which must be the time receipt in the first element in $S[I_{\mathsf{VDF}}^{-1}]$: $(\perp, t_0 - \tau || x_k)$. By construction of $H2T$-sequences, there must be a random oracle (H) chain between c_0 to x_k, which implies that precomputation of a sequence with the correct root time is no use. More importantly, the time receipts ensure that having access to a $H2T$-sequence starting with c_0 of the requisite strength but with root time after $t_0 - \tau$ can also not be taken advantage of to create a correct $H2T$-sequence.

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 τ older than the "real" root time, while still keeping minimum average rate γ, with non-negligible success probability.

Lemma B.7 (Lemma 4.3 restated). *If* $t_1 - t_0 < \tau \cdot \frac{\gamma}{\gamma_{\mathcal{A}} - \gamma}$ *then the adversary wins the sequence-forging 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* $\mathcal{F}^{\Gamma}_{\mathsf{VDF}}\mathcal{A}$.

Proof. In the elapsed time $e := t_1 - t_0$ after corruption, the adversary can sequentially compute an $H2T$-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 $H2T$-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 $H2T$-sequence sequentially and by Lemma B.3 succeeds with probability at most $2 \cdot (q_1 \cdot 2^{-\lambda} + q_2 \cdot 2^{-\mu}) \cdot (Q_1 + Q_2 + |S|)$. □

While an adversary can still simply construct an $H2T$-sequence from scratch, it can no longer stretch an already exisiting one as the time receipts cannot be modified without recomputing the VDFs.

In order to correctly realize the timestamping functionality the adversary must not be able to precompute the VDFs. In order to prevent precomputation, we use a secure digital signature scheme. At the same time, our proofs for the unforgeability of the $H2T$-sequences require the content of the first element of the sequence to come from an unpredictable oracle. These two ideas can be combined by assuming that this oracle generates a signature for the rest of the content and can only be accessed by the adversary by corrupting the original stamper. This reasoning leads us to our next construction, where the signatures are added as part of the sequence. We now define $H2TS$-sequences which are simply $H2T$-sequences where every VDF input is signed.

Having $B_i.prev = \mathsf{MT.root}(B_{i-1})$ (and $prev$ being a leaf of the canonical Merkle tree) allows us to generate a Merkle tree for the entire chain by concatenating Merkle trees through $prev$. Additionally, this allows us to say that $\mathsf{MT.root}(C) = \mathsf{MT.root}(\mathsf{last}(C))$. More interestingly, because we use the root of the previous block as part of the input to our VDF, we can do a similar concatenation of Merkle trees through the signed VDF inputs that actually results in an $H2TS$-sequence.

In order to construct this $H2TS$-sequence S, we take a chain C and a record c stored in B_i and proceed as follows:

1. Start with the $H2$-sequence induced by $\mathsf{MT.path}(B_i, c)$,
2. For $j = i + 1, \ldots$:
   (a) Append $(B_j.s, B_j.t^u || \mathsf{MT.root}(B_{j-1}) || B_j.sig)$
   (b) Append $\mathsf{MT.path}(B_j, B_j.t || B_j.y)$

This construction allows us to create an $H2TS$-sequence starting from any element in a block (in particular c) and ending at the end of the chain, passing

through every VDF proof and including every time receipt t and t^u. This allows a party to attest an age of c. Given a chain C we refer to the $H2TS$-sequence starting from $B_i.c$ and going through the entire subchain $C[i :)$ as $\mathsf{h2ts}(C, i)$. Note that if we want to keep the timestamped records secret, we can ensure that $B.c$ is simply a hash of the record. The first element of the $H2TS$-sequence would simply be the record, followed by its hash.

B.2 Main Theorem

Here we prove our main theorem Theorem 4.7

Theorem B.8. *Let $\mu \in \mathbb{N}^+$ be the security parameter. For any real-world PPT adversary $\mathcal{A}$ with oracle access to $\mathcal{F}^{\Gamma}_{\mathsf{VDF}}$, there exists a black-box PPT simulator $\mathcal{A}_{id}$ such that for any PPT environment $\mathcal{Z}$ the probability that $\mathcal{Z}$ can distinguish between the ideal world with $\mathcal{F}^{\alpha}_{ts}$ and $\mathcal{A}_{id}$ (cf. Figure 2,5) and the real world with $\mathcal{A}$, $\mathcal{P}^{\gamma,\varepsilon}_{H2TS}$ and $\mathcal{V}^{\gamma,\varepsilon}_{H2TS}$ (cf. Figure 3) is negligible in μ, λ and κ.*

Proof. Let $\mathcal{S}^{\mathcal{A}}_{\text{H2TS}}$ be as defined in Figure 5. We consider all related execution transcripts in the ideal and real world

$$(\Pi_{ideal}, \Pi_{real}) \leftarrow \text{EXEC}(\mathcal{Z}^{\text{IDEAL}(\mathcal{F}^{\alpha}_{\text{ts}}, \mathcal{S}^{\mathcal{A}}_{\text{H2TS}})}, \mathcal{Z}^{\text{REAL}(\mathcal{P}^{\gamma,\varepsilon}_{\text{H2TS}}, \mathcal{V}^{\gamma,\varepsilon}_{\text{H2TS}}, \mathcal{A})}),$$

where all parties including $\mathcal{Z}$ and $\mathcal{A}$ receive the same starting input tape and randomness tape.

If these executions are identical from the viewpoint of $\mathcal{Z}$, then $\mathcal{Z}$ will output the same bit. It follows that to prove the theorem we only have to bound the probability that the two views of $\mathcal{Z}$ are not identical:

$$\left|\Pr[\mathcal{Z}^{\text{IDEAL}(\mathcal{F}^{\alpha}_{\text{ts}}, \mathcal{S}^{\mathcal{A}}_{\text{H2TS}})} = 1] - \Pr[\mathcal{Z}^{\text{REAL}(\mathcal{P}^{\gamma,\varepsilon}_{\text{H2TS}}, \mathcal{V}^{\gamma,\varepsilon}_{\text{H2TS}}, \mathcal{A})} = 1]\right| \\ \leq \Pr[\text{VIEW}(\mathcal{Z}, \Pi_{ideal}) \neq \text{VIEW}(\mathcal{Z}, \Pi_{real})].$$

One can verify that in the ideal world all queries by $\mathcal{Z}$ and their answers by $\mathcal{P}^{\gamma,\varepsilon}_{\text{H2TS}}$ (potentially under control of $\mathcal{A}$) are perfectly forwarded by the functionality and the simulator. Actually the only way for $\mathcal{Z}$'s view to be different is when a `verify` query by $\mathcal{Z}$ results in a different outcome between the ideal world and real world. We now bound the probability that this event occurs.

Given some related pair $(\Pi_{ideal}, \Pi_{real})$ of execution transcripts, let

$$(t_{\text{bad}}, \mathcal{Z}, (\texttt{verify}, c, a, u)) \in \Pi_{ideal} \cap \Pi_{real}$$

be the first query for which the answer o_i in the ideal world differs from the output $o_r \neq o_i$ in the real world. Let $q_{\texttt{verify}}$, and q_{VDF} be the maximum of the amount of `verify` and VDF queries, respectively, made in Π_{real} or Π_{ideal}. Below we only consider what happened up to time t_{bad} and disregard anything afterwards.

Assume $o_i = 1$, this is only possible if $\mathcal{S}^{\mathcal{A}}_{\text{H2TS}}$ has output $(c, a, u, 1)$ (as answer to a `stamp` query or as a `stamped` query). That can only happen when

Simulator $\mathcal{S}^{\mathcal{A}}_{\mathbf{H2TS}}$

The simulator $\mathcal{S}^{\mathcal{A}}_{\text{H2TS}}$ simulates an instance of prover $\mathcal{P}'$ who runs $\mathcal{P}^{\gamma,\varepsilon}_{\text{H2TS}}$ and verifiers $\mathcal{V}'$ which run $\mathcal{V}^{\gamma,\varepsilon}_{\text{H2TS}}$. It maintains a list R of all `stamp` queries and responses from $\mathcal{F}^{\alpha}_{\text{ts}}$. It simulates $\mathcal{F}^{\Gamma}_{\mathsf{VDF}}$ through lazy sampling of VDF.

Creating a timestamp

- On input $(\texttt{record}, c)$ from $\mathcal{F}^{\alpha}_{\text{ts}}$, the simulator inputs $(\texttt{record}, c)$ into $\mathcal{P}$ and then sends back `ok`.
- On input $(\texttt{stamp}, c)$ from $\mathcal{F}^{\alpha}_{\text{ts}}$, the simulator queries the relevant proof from $\mathcal{P}'$ with a $(\texttt{stamp}, c)$ query. Then it checks the validity of such a proof through $\mathcal{V}^{\gamma,\varepsilon}_{\text{H2TS}}.\texttt{verify}(c, a, u)$ to get a validity bit v.Then it returns (c, a, u, v) to $\mathcal{F}^{\alpha}_{\text{ts}}$.
- Whenever the prover has been corrupted, whenever the adversary $\mathcal{A}$ inputs $(\texttt{start}, t||\mathsf{MT.root}(B_i)||\sigma)$ to $\mathcal{F}^{\Gamma}_{\mathsf{VDF}}$, if $\Sigma.\mathsf{verify}(t||\mathsf{MT.root}(B_i), \sigma, pk_{\mathcal{P}}) = 1$ then input $(\texttt{record}, B_i.c)$ to $\mathcal{F}^{\alpha}_{\text{ts}}$ through the backdoor tape.

Getting a new timestamp

- On input from $(\texttt{cnewstamp}, c, a, u)$ from $\mathcal{F}^{\alpha}_{\text{ts}}$. The simulator checks the validity of such a proof through $\mathcal{V}^{\gamma,\varepsilon}_{\text{H2TS}}.\texttt{verify}(c, a, u)$ and returns the response.

Fig. 5: Simulator $\mathcal{S}^{\mathcal{A}}_{\text{H2TS}}$ (same as the one in Figure 4)

$\mathcal{V}^{\gamma,\varepsilon}_{\text{H2TS}}(\texttt{verify}, c, a, u) = 1$ and thus that $o_r = 1$, which is a contradiction. It follows that $o_i = 0$ and $o_r = 1$ and u is of the form

$$\langle((s_i, p_i), t_i, c_i, \sigma_i)|s_i \in \mathbb{N}^+, p_i \in \{0,1\}^\mu, t_i \in \{0,1\}^\theta, \sigma_i \in \{0,1\}^\kappa\rangle_l$$

where:

$$\Sigma.\mathsf{verify}((s_i, p_i)||t_i||c_i, \sigma_i, pk_{\mathcal{P}}) = 1 \qquad a = \textstyle\sum s_i/\gamma$$
$$\mathsf{VDF}((s_i, p_i)||t_i||c_i||\sigma_i, s_{i+1}) = p_{i+1} \qquad a \leq (t_{\text{bad}} - t_0)$$

Now consider the case when (c, a, u) was not legitimately constructed through the functionality in the ideal world. Then, the simulator has to more actively deal with proofs that were not generated through `stamp` but constructed by the adversary/environment. If u was constructed by truncating and recombining previous proof chains in a valid way, they would have been accepted through the $\mathsf{checknewstamp}$ query to $\mathcal{S}^{\mathcal{A}}_{\text{H2TS}}$. Additionally, proofs legitimately computed by the adversary would be accepted through this mechanism, as the simulator ensures that the appropriate `record` query is created whenever a `start` query is input into $\mathcal{F}^{\Gamma}_{\mathsf{VDF}}$.

To continue, we bound the probabality that the adversary has constructed the proof in a non-sequential manner by Lemma 4.3 as at most $2 \cdot (q_{\mathsf{H}} \cdot 2^{-\lambda} + q_{\mathsf{VDF}} \cdot 2^{-\mu}) \cdot (Q_{\mathsf{H}} + Q_{\mathsf{VDF}} + |S|)$. Thus in the remainder of the proof we can assume the adversary has constructed the proof sequentially.

Since $o_i = 0$, it must be caused by one of the rules in `Procedure checkstamp` resulting in $v = 0$ for (c, a, u):

1. The case that c was not recorded by $\mathcal{F}_{\text{ts}}^{\alpha}$:
   Whenever $\mathcal{A}$ wants to construct a valid timestamp for a record c, they must take a particular hash r as part of the input to $\mathcal{F}_{\mathsf{VDF}}^{\Gamma}$ such that there is a particular sequence of H calls such that they form a Merkle path of a fixed length and parity[8] from c to r. Any other sequence of hashes, e.g., with wrong parity, will not be accepted by $\mathcal{V}_{\text{H2TS}}^{\gamma,\varepsilon}$. As $\mathcal{S}_{\text{H2TS}}^{\mathcal{A}}$ has access to all random oracle calls, it can check the c used to construct a particular r that is input to $\mathcal{F}_{\mathsf{VDF}}^{\Gamma}$ and generate the respective `record` to $\mathcal{F}_{\text{ts}}^{\alpha}$. Therefore, $\mathcal{A}$ can only construct a timestamp without triggering a `record` query through a collision in H, which has probability at most $q_{\mathsf{H}} \cdot 2^{-\lambda} \cdot Q_{\mathsf{H}}$.
2. The case that the claimed age a is older than the real age a_r times α:
   As $\mathcal{A}$ only has access to $\mathcal{F}_{\mathsf{VDF}}^{\Gamma}$ then they can only create $H2TS$-sequences diluted by a factor α. As the time-dilution factor acts the same over γ and ε, the adversary gains no additional advantage by changing the spacing of the time receipts in the sequence. Hence, it is impossible for the adversary to have created this proof in a sequential manner.
3. The case that the claimed age a is older than the time a_{corr} since corruption times α:
   The adversary cannot stretch VDF strengths of an honest chain in order to make it seem older because of the honest time receipts. Hence, if the adversary created this proof in a sequential manner and it started with a VDF of the form $\mathsf{VDF}(t||x||\sigma)$ with a valid signature then it is clear the adversary succeeded in forging a digital signature before corruption. The probability of this event is negligible in κ.
4. The case that $(c, a, u, 0) \in R$:
   The same analysis holds, but then for the first time $\mathsf{checkstamp}(c, a, u)$ was called.

As the number to all queries are polynomially bounded by μ, λ or κ, the probability of distinguishing is negligible in μ, λ and κ. □

C A trivial timestamping scheme from VDFs

In this section we present a trivial construction based on the one found in [11], using verifiable delay functions instead of proofs of sequential work. The purpose of this construction is primarily didactic, as it can only realize a weaker functionality $\mathcal{F}_{\text{triv}}^{\alpha}$. This section also allows the reader to become familiar with the proof structures in the remainder of this paper.

The functionality models a very pessimistic view, namely that the prover is always corrupted and it gives the adversary tremendous power, since for every

[8] Whether the tree branches left or right.

Timestamping functionality $\mathcal{F}^{\alpha}_{\textbf{triv}}$

The functionality is parametrized with an adversarial time-dilution factor $\alpha \geq 1$. It answers queries from the environment $\mathcal{Z}$ and interacts with an adversary $\mathcal{A}_{\text{id}}$. It maintains two internal lists: $\mathcal{R} \subset \{0,1\}^* \times \mathbb{R}^+$ for (record,time)-tuples and $\mathcal{C} \subset \{0,1\}^* \times \mathbb{R}^+ \times \{0,1\}^* \times \{0,1\}$ for (record,age,proof,valid)-tuples.

- `On input` $(\texttt{record}, c)$, $c \in \{0,1\}^*$:
  `If` $\mathcal{A}_{\text{id}}$ is not sender `then send` $(\texttt{record}, c)$ `to` $\mathcal{A}_{\text{id}}$;
  `If` $\nexists t : (c,t) \in \mathcal{R}$ `then set` $\mathcal{R} \leftarrow \mathcal{R} \cup (c, \mathcal{F}_{\mathsf{clock}})$.
- `Procedure` checkstamp(c, a, u, v):
  /* Claimed age a older than α times real age is not allowed. */
  `If` $\{(c', t') \in \mathcal{R} \mid c' = c \ \wedge \ a \leq (\mathcal{F}_{\mathsf{clock}} - t') \cdot \alpha\} = \emptyset$ `then` $v \leftarrow 0$.
  /* Once (in)valid is always (in)valid. */
  `If` $\exists (c, a, u, \hat{v}) \in \mathcal{C}$ `then` $v \leftarrow \hat{v}$.
  `Let` $\mathcal{C} \leftarrow \mathcal{C} \cup (c, a, u, v)$.
- `On input` $(\texttt{stamp}, c)$, $c \in \{0,1\}^*$:
  /* Let the adversary produce a claimed age and proof string */
  `Query` $(c, a, u, v) \leftarrow \mathcal{A}_{\text{id}}(\texttt{stamp}, c)$;
  `Call` checkstamp(c, a, u, v);
  `Return` (c, a, u).
- `On input` $(\texttt{stamped}, c, a, u, v)$:
  /* The adversary produced a stamp for a record unasked */
  `Call` checkstamp(c, a, u, v).
- `On input` $(\texttt{verify}, c, a, u)$, $c \in \{0,1\}^*, a \in \mathbb{R}, p \in \{0,1\}^*$:
  `If` $(c, a, u, 1) \in \mathcal{C}$ `then return` 1.
  `Let` $\mathcal{C} \leftarrow \mathcal{C} \cup (c, a, u, 0)$;
  `Return` 0.

Fig. 6: Simple timestamping functionality

stamp query, it is the adversary that generates the claimed age a and proof string u. Nevertheless, he cannot make the functionality accept (c, a, u) as valid, unless the claimed age $a \leq a_{real} \cdot \alpha$ is bounded by the time-dilution factor α multiplied by the real record's age at the time when the proof was generated.

Simple VDF prover $\mathcal{P}^{\gamma}_{\mathbf{triv}}$

Given parameter VDF rate γ. It answers queries from the environment $\mathcal{Z}$ and interacts with its oracle $\mathcal{F}^{\Gamma}_{\mathsf{VDF}}$:

```
– On input (record, c), c ∈ {0,1}*:
      Query F^Γ_VDF(start, c).
– On input (stamp, c), c ∈ {0,1}*:
      Query u ← F^Γ_VDF(output, c);
      If u = ⊥ then return (c, 0, ⊥);
      /* Here u ∈ N+ × {0,1}^μ */
      Let (s, p) = u, a = s/γ;
      Return (c, a, u);
```

(a) Simple VDF prover

Simple VDF verifier $\mathcal{V}^{\gamma}_{\mathbf{triv}}$

Given parameter VDF rate γ. It answers queries from the environment $\mathcal{Z}$ and interacts with its oracle $\mathcal{F}^{\Gamma}_{\mathsf{VDF}}$:

```
– On input (verify, c, a, u),
    c ∈ {0,1}*, a ∈ R+, u ∈ {0,1}*:
      If u ≠ (s, p) ∈ N+ × {0,1}^μ
      then return 0;
      If a ≠ s/γ then return 0;
      Return F^Γ_VDF(verify, c, s, p).
```

(b) Simple VDF verifier

Fig. 7: Simple timestamping prover and verifier using VDFs

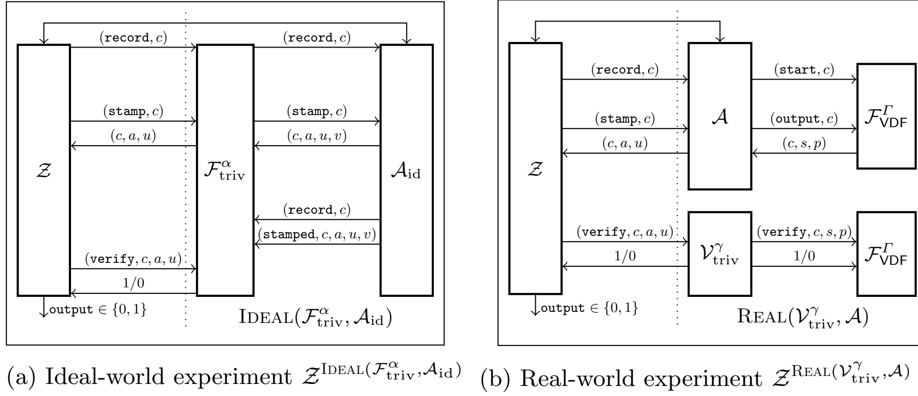
C.1 Security proof

The ideal functionality assumes an always corrupted prover, so our real world consists of an arbitrary PPT adversary $\mathcal{A}$ (instead of $\mathcal{P}^{\gamma}_{\text{triv}}$) and the honest verifier $\mathcal{V}^{\gamma}_{\text{triv}}$. We assume a PPT environment $\mathcal{Z}$ that either interacts with the ideal world $\textsc{Ideal}(\mathcal{F}^{\alpha}_{\text{triv}}, \mathcal{A}_{\text{id}})$ consisting of the ideal functionality and an ideal adversary (cf. Figure 8a), or interacts with the real world $\textsc{Real}(\mathcal{V}^{\gamma}_{\text{triv}}, \mathcal{A})$ (cf. Figure 8b), and that at the end outputs a guessing bit.

The below theorem states that we can bound the advantage of $\mathcal{Z}$ to distinguish between the real world and ideal world for a certain instantiation of the ideal adversary $\mathcal{A}_{\text{id}}$ by a simulator $\mathcal{S}^{\mathcal{A},\mathcal{F}^{\Gamma}_{\mathsf{VDF}}}_{\text{triv}}$, which has black-box access to the PPT real adversary $\mathcal{A}$ (and may observe its oracle calls).

Theorem C.1. *Let $\mu \in \mathbb{N}^+$ be the security parameter. For any real-world PPT adversary $\mathcal{A}$ with oracle access to $\mathcal{F}^{\Gamma}_{\mathsf{VDF}}$, there exists a black-box PPT simulator $\mathcal{A}_{id}$ such that for any PPT environment $\mathcal{Z}$ there exists a negligible function $negl(\mu)$ such that the the probability that $\mathcal{Z}$ can distinguish between the ideal world with $\mathcal{F}^{\alpha}_{triv}$ and $\mathcal{A}_{id}$ and the real world with $\mathcal{A}$ and $\mathcal{V}^{\gamma}_{triv}$ (cf. Figure 7b,8b) is negligible:*

$$\left|\Pr[\mathcal{Z}^{\textsc{Ideal}(\mathcal{F}^{\alpha}_{triv},\mathcal{A}_{id})} = 1] - \Pr[\mathcal{Z}^{\textsc{Real}(\mathcal{V}^{\gamma}_{triv},\mathcal{A})} = 1]\right| \leq negl(\mu)\,.$$



(a) Ideal-world experiment $\mathcal{Z}^{\textsc{Ideal}(\mathcal{F}_{\text{triv}}^{\alpha}, \mathcal{A}_{\text{id}})}$ (b) Real-world experiment $\mathcal{Z}^{\textsc{Real}(\mathcal{V}_{\text{triv}}^{\gamma}, \mathcal{A})}$

Fig. 8: Environment $\mathcal{Z}$ in ideal- and real-world experiment for simple timestamping

Proof. Let the ideal-world simulator $\mathcal{A}_{\text{id}} := \mathcal{S}_{\text{triv}}^{\mathcal{A}, \mathcal{F}_{\mathsf{VDF}}^{\Gamma}}$ be defined as in Figure 9. We consider all related execution transcripts in the ideal and real world

$$(\Pi_{ideal}, \Pi_{real}) \leftarrow \textsc{Exec}(\mathcal{Z}^{\textsc{Ideal}(\mathcal{F}_{\text{triv}}^{\alpha}, \mathcal{S}_{\text{triv}}^{\mathcal{A}, \mathcal{F}_{\mathsf{VDF}}^{\Gamma}})}, \mathcal{Z}^{\textsc{Real}(\mathcal{V}_{\text{triv}}^{\gamma}, \mathcal{A})}),$$

where $\mathcal{Z}$ and $\mathcal{A}$ receive the same input and use the same random coin toss outcomes. If these executions are identical from the viewpoint of $\mathcal{Z}$, then $\mathcal{Z}$ will output the same bit. It follows that to prove the theorem we only have to bound the probability that these views are not identical:

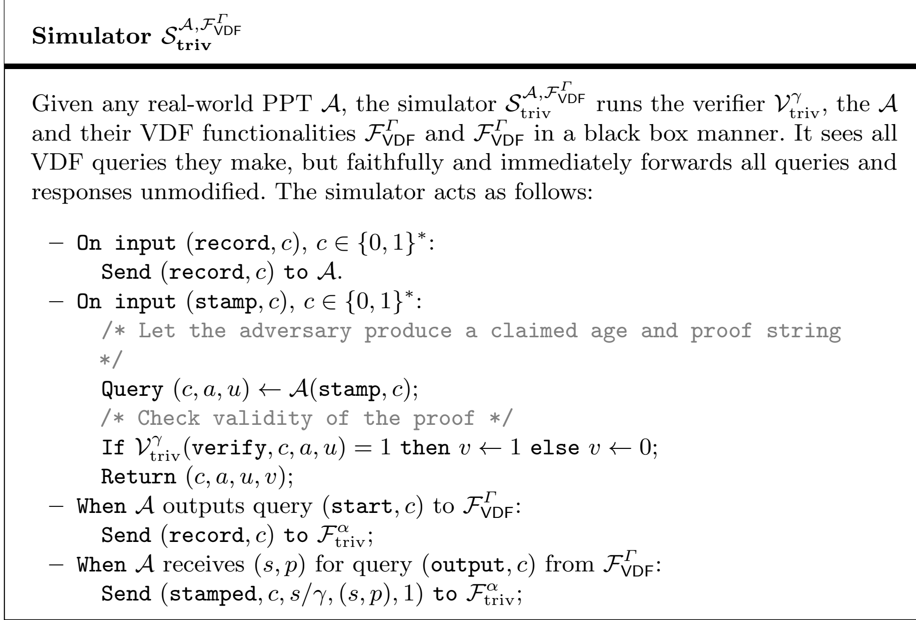
$$\left| \Pr[\mathcal{Z}^{\textsc{Ideal}(\mathcal{F}_{\text{triv}}^{\alpha}, \mathcal{S}_{\text{triv}}^{\mathcal{A}, \mathcal{F}_{\mathsf{VDF}}^{\Gamma}})} = 1] - \Pr[\mathcal{Z}^{\textsc{Real}(\mathcal{V}_{\text{triv}}^{\gamma}, \mathcal{A})} = 1] \right| \leq \Pr[\textsc{View}_{\mathcal{Z}}(\Pi_{ideal}) \neq \textsc{View}_{\mathcal{Z}}(\Pi_{real})].$$

One can verify that in the ideal world all `record` and `stamp` queries by $\mathcal{Z}$ and their answers by $\mathcal{A}$ are perfectly forwarded by the functionality and the simulator without changing content or timing. Hence the only way for $\mathcal{Z}$'s view to be different is when a `verify` query results in a different outcome between the ideal world and real world. We now bound the probability that this event occurs.

Given some related pair $(\Pi_{ideal}, \Pi_{real})$ of execution transcripts, let $(t_{\text{bad}}, \mathcal{Z}, (\texttt{verify}, c, a, u)) \in \Pi_{ideal} \cap \Pi_{real}$ be the first query for which the answer o_i in the ideal world differs from the output $o_r \neq o_i$ in the real world. Let $q_{\texttt{verify}}$, and q_{VDF} be the maximum of the amount of `verify` and VDF queries, respectively, made in Π_{real} or Π_{ideal}. Below we only consider what happened up to time t_{bad} and disregard anything afterwards.

Assume $o_i = 1$, this is only possible if $\mathcal{S}_{\text{triv}}^{\mathcal{A}, \mathcal{F}_{\mathsf{VDF}}^{\Gamma}}$ has output $(c, a, u, 1)$ (as answer to a `stamp` query or as a `stamped` query). That can only happen when $\mathcal{V}_{\text{triv}}^{\gamma}(\texttt{verify}, c, a, u) = 1$ and thus that $o_r = 1$, which is a contradiction. It follows that $o_i = 0$ and $o_r = 1$ and $u = (s, p)$ for some s and p.

Next consider the case when $\mathcal{S}_{\text{triv}}^{\mathcal{A}, \mathcal{F}_{\mathsf{VDF}}^{\Gamma}}$ never outputted $(\texttt{stamped}, c, a, u, 1)$ in Π_{ideal}. This implies that the real adversary never received $(s, p) \leftarrow \mathcal{F}_{\mathsf{VDF}}^{\Gamma}(\texttt{output}, c)$,



Simulator $\mathcal{S}_{\mathbf{triv}}^{\mathcal{A},\mathcal{F}_{\mathsf{VDF}}^{\Gamma}}$

Given any real-world PPT $\mathcal{A}$, the simulator $\mathcal{S}_{\mathrm{triv}}^{\mathcal{A},\mathcal{F}_{\mathsf{VDF}}^{\Gamma}}$ runs the verifier $\mathcal{V}_{\mathrm{triv}}^{\gamma}$, the $\mathcal{A}$ and their VDF functionalities $\mathcal{F}_{\mathsf{VDF}}^{\Gamma}$ and $\mathcal{F}_{\mathsf{VDF}}^{\Gamma}$ in a black box manner. It sees all VDF queries they make, but faithfully and immediately forwards all queries and responses unmodified. The simulator acts as follows:

- On input $(\texttt{record}, c)$, $c \in \{0,1\}^*$:
  Send $(\texttt{record}, c)$ to $\mathcal{A}$.
- On input $(\texttt{stamp}, c)$, $c \in \{0,1\}^*$:
  /* Let the adversary produce a claimed age and proof string */
  Query $(c, a, u) \leftarrow \mathcal{A}(\texttt{stamp}, c)$;
  /* Check validity of the proof */
  If $\mathcal{V}_{\mathrm{triv}}^{\gamma}(\texttt{verify}, c, a, u) = 1$ then $v \leftarrow 1$ else $v \leftarrow 0$;
  Return (c, a, u, v);
- When $\mathcal{A}$ outputs query $(\texttt{start}, c)$ to $\mathcal{F}_{\mathsf{VDF}}^{\Gamma}$:
  Send $(\texttt{record}, c)$ to $\mathcal{F}_{\mathrm{triv}}^{\alpha}$;
- When $\mathcal{A}$ receives (s, p) for query $(\texttt{output}, c)$ from $\mathcal{F}_{\mathsf{VDF}}^{\Gamma}$:
  Send $(\texttt{stamped}, c, s/\gamma, (s, p), 1)$ to $\mathcal{F}_{\mathrm{triv}}^{\alpha}$;

Fig. 9: Simulator

where $(s, p) = u$. Note that the value $\mathsf{VDF}(c, s)$ can only be directly queried through a $\mathcal{F}_{\mathsf{VDF}}^{\Gamma}(\texttt{output}, c)$ query, which thus did not happen. The only other way to learn that $p = \mathsf{VDF}(c, s)$ is indirectly through a $\mathcal{F}_{\mathsf{VDF}}^{\Gamma}(\texttt{verify}, c, s, p) \in \{1, 0\}$ query. Since VDF is a random oracle of bitlength μ, this event occurs with probability upper-bounded by $q_{\texttt{verify}} \cdot 2^{-\mu}$.

What remains is when $o_i = 0$, $o_r = 1$ and $\mathcal{S}_{\mathrm{triv}}^{\mathcal{A},\mathcal{F}_{\mathsf{VDF}}^{\Gamma}}$ has outputted $(\texttt{stamped}, c, a, u, 1)$. Then since $o_i = 0$ it must be caused by one of the two rules in `Procedure checkstamp` of the ideal functionality (cf. Figure 6) resulting in $v = 0$ for (c, a, u):

1. The case that the claimed age a is older than the real age a_r times α:
   Assume that $\mathcal{A}$ received $(s, p) \leftarrow \mathcal{F}_{\mathsf{VDF}}^{\Gamma}(\texttt{output}, c)$, at which point the simulator immediately makes a $(\texttt{stamped}, c, s/\gamma, (s, p), 1)$ query to the functionality. By the definition of $\mathcal{F}_{\mathsf{VDF}}^{\Gamma}$, the adversary $\mathcal{A}$ must have queried $\mathcal{F}_{\mathsf{VDF}}^{\Gamma}(\texttt{start}, c)$ exactly the amount of time $s/\gamma_{\mathcal{A}}$ before that, at which time the simulator made a $(\texttt{record}, c)$ query to the functionality. Thus, $a_r = s/(\gamma \cdot \alpha)$ and $a = s/\gamma = a_r \cdot \alpha$, which is a contradiction.
   It follows that $\mathcal{A}$ guessed the value $p = \mathsf{VDF}(c, s)$, the probability of this event is upper-bounded by $2^{-\mu} \cdot q_{\mathsf{VDF}}$.
2. The case that $(c, a, (s, p), 0) \in \mathcal{C}$:
   This implies that the environment $\mathcal{Z}$ correctly guessed $p = \mathsf{VDF}(c, s)$ in a $(\texttt{verify}, c, a, (s, p))$ query made earlier. Since VDF is a random oracle with bitlength μ, this event happens with probability upper-bounded by $q_{\texttt{verify}} \cdot 2^{-\mu}$.

We conclude that

$$\Pr[\textsc{View}_{\mathcal{Z}}(\Pi_{ideal}) \neq \textsc{View}_{\mathcal{Z}}(\Pi_{real})] \leq 2^{-\mu} \cdot (2 \cdot q_{\texttt{verify}} + q_{\mathsf{VDF}}),$$

where $q_{\texttt{verify}}$ and q_{VDF} are polynomially upper-bounded by μ. It follows that the right hand side is negligible in μ, which proves the theorem. □

C.2 Issues

The above simple construction using VDFs has several important flaws that we try to mitigate in this work.

- The functionality is in always corrupted state and α-time-dilution is always possible, since the real adversary can always construct a valid timestamp triple (c, a, u) with α-time-dilution and then pass this triple to the environment to verify.
  An approach to solve this problem is by using digital signatures to prevent the adversary from constructing valid timestamp triples with α-time-dilution, until it has corrupted the prover. Unfortunately, once it has corrupted the prover, the use of digital signatures still allows time-dilution for all prior records.
  In this work we achieve even more: the adversary is limited from producing valid time-diluted timestamp proofs with claimed age only at most $A_{corr} \cdot \alpha$, where A_{corr} is the amount of time passed since corruption. It follows that all valid timestamp proofs with greater claimed age cannot be time-diluted. We achieve this property by adding both the time and the signature on the $\mathcal{F}^{\Gamma}_{\mathsf{VDF}}.\texttt{start}$ input, making any time-diluted age claim invalid with respect to the recording time.
- The construction is very inefficient as it requires any honest prover to run a separate VDF computation per record. We get rid of this problem in our final construction that uses a blockchain structure and requires only one VDF computation at all times for an honest prover.