Non-interactive Cryptographic Timestamping based on Verifiable Delay Functions

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

[1] CWI Amsterdam
[2] University of Amsterdam & QuSoft

esteban@cwi.nl

Abstract. We present the first treatment of non-interactive publicly-verifiable timestamping schemes in the Universal Composability framework. Similar to a simple construction by Mahmoody et al., we use non-parallelizable computational work that relates to elapsed time to avoid previous impossibility results on non-interactive timestamping. We extend these ideas to the UC-framework and show how to model verifiable delay functions (VDF) related to a global clock, and non-interactive timestamping, in the UC-framework. Furthermore, we present new constructions that are substantial improvements over Mahmoody et al.'s construction, such that any forged timestamps by the adversary are now limited to within a certain time-window that depends only on its ratio to compute VDFs more quickly and the time-window of corruption. Finally, we discuss natural applications for our construction in decentralized protocols.

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 [HS91] presented two different solutions that avoid having to fully trust a third party to validate timestamps. Their first solution uses a hashchain: a 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 been proposed, most notably substituting the individual records in a hashchain with a Merkle tree (or similar) [BHS93, BLS00]. These timestamping systems require many parties to maintain records and their timestamps, as well as the availability to answer validation queries. Other protocols require less space but stronger trust assumptions on the set of validators [BDM93].

While timestamping evokes the idea of fixing an event in a specific point in clocktime, in practice most timestamping schemes are relative; they provide only an ordering. A significant limitation of such ordering systems is that it is impossible to directly compare the timestamps to events outside the protocol. Clock-based timestamping generally requires stronger trust assumptions, as the notion of clocktime in a computational setting is problematic.

One of the goals of this paper is to provide the first treatment of non-interactive timestamping in the universal composability framework. In the classical literature, almost every timestamping service requires interaction with a group of validators and provides security guarantees only for relative timestamping. Non-interactive timestamping has been explored before in [MSTS04], where the authors present a generic impossibility result, as an adversarial prover would simply need to simulate the execution of an honest prover to generate a fake timestamp. They sidestep this result by working in the bounded-storage model where they construct a secure protocol. Another approach to sidestep this result is to relate computational work to elapsed time.

For instance, there are constructions that use the Bitcoin blockchain [Nak08] for timestamping in the absolute sense and not only relatively [CE12] [3]. Haber and Stornetta's hashchain protocol served as a fundamental building block for the Bitcoin blockchain. Nakamoto's blockchain consists of a hashchain where new elements of the chain, or blocks, are added if and only if they are solutions to a cryptographic puzzle known as a Proof-of-Work (PoW). The work required to create a new block ensures that the blockchain cannot be easily rewritten, especially for block created a long time ago [GKL15]. The protocol is tuned to create a block every ten minutes in average, meaning that one can roughly associate each block to a ten minute slot. There are other constructions that utilize the blockchain to provide timestamping [GMG15, SV17], which simply treat it as a trusted party and do not formally prove security.

Constructions that use PoW-based blockchain's computational work to encode time are interesting for timestamping purposes, but these require enormous computational resources which come with cost and sustainability concerns[4]. The main argument for this computational cost is its direct relation to the blockchain's security: an adversary must hold the majority of the computational power to significantly rewrite the blockchain. The adversary is still 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, where α is the adversary's fraction of the total hashing power. 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 the costs and complications of proof-of-work blockchains.

Due to the above issues, we specifically study constructions that use inherently sequential or non-parallelizable functions which have a long history in cryptography. Originally conceived for timed-release encryption [RSW96], these functions have been used as way to achieve pseudonymous [KMS14] or deniable authentication [Wes18] and to create non-malleable commitments [LPS17]. Lately, there has been interest in generating verifiable proofs of sequential computation to represent time for distributed protocols. One can distinguish between two similar but slightly different types of primitives: proof-of-sequential-work (PoSW) and verifiable delay functions (VDF).

In [MMV11], the authors present the concept of a *proof of sequential work* (PoSW) in order to verify that a number of computational steps have happened since something existed. The gap between computation and verification of PoSWs is exponential, which is fundamental for their practical use. However, by the nature of the problems the proofs are non-unique and grow with the length of the computation (size of the graph) in order to maintain security, even for the more efficient follow-up [MMV13]. Another construction based on graph pebbling [CP18] links the size of the proof only with the expected security parameter. However, for a reasonable security level, the proofs are still very large (in the order of 200kB).

Verifiable delay functions, that are similar to PoSW but have unique proofs, such as the *sloth* construction from [LW15]. Unfortunately *sloth*'s verification gap is linear at best. Since then, functions having an exponential gap were first presented and characterized in [BBBF18], although the proposals presented there were somewhat vulnerable to parallelization. Newer constructions based on modular exponentiation [Pie18, Wes18] are more resilient to parallelization. The main difference between the last two proposals is the proof size and the efficiency in computing a proof. Such slow functions have been postulated before in order to generate publicly verifiable randomness [LW15, PW16].

1.1 Our contributions

We study non-interactive cryptographic timestamping based on verifiable delay functions in the universal-composability framework and using the random-oracle model. This is the first treatment of non-interactive timestamping in the UC model. In contrast to interactive timestamping, the time of proof-generation and time of proof-verification may lie far apart. Since proofs do not magically adjust over time, it is easy to see that there is a fundamental limitation of non-interactive timestamping that only allows relative timestamping, namely proving the record's age at the time of proof-generation.

[3] Although with certain limitations as this is not its primary purpose `http://culubas.blogspot.nl/2011/05/timejacking-bitcoin_802.html`

[4] `https://digiconomist.net/bitcoin-energy-consumption`

Towards our goals, we first explore the modelling of parties' ability to compute VDFs at a certain rate with respect to some global clock. This is non-trivial since the UC-framework is based on a simple scheduling mechanism for all parties (machines) where only one party is active at a time. Based on this setting we define an ideal timestamping functionality $\mathcal{F}_{\text{ts}}^{\alpha}$ that maintains a timestamped record of bitstrings, which 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 power to compute VDFs faster as a *time-diluting* factor α, and the adversary can corrupt the timestamping functionality at any time. Our ideal functionality continues to perform its goal even after adversarial corruption, as the adversary is only allowed to forge proofs under certain constraints. Specifically, when the adversary corrupted the functionality time T ago, the adversary can for any record (of age *TrueAge*) only craft forged timestamps of age less than $\alpha \cdot \min(T, \mathit{TrueAge})$. And in particular this implies that it cannot create any forged timestamps with age larger than $T \cdot \alpha$, and that it cannot exploit honest VDF computations to create forged timestamps. This is a substantially stronger security guarantee than provided by Mahmoody et al.'s construction [MMV13], 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. Note that in Section 4 we provide a treatment of Mahmoody et al.'s construction in the UC-framework for easy comparison with our work.

We then first present a simple construction to demonstrate the basic techniques, However this construction is rather inefficient in practice as it requires individual continuous VDF computations for each record entry. Next we present a more efficient construction that combines our first construction with the hashchain technique in the original timestamping mechanism from [HS91]. For both constructions, the prover provides a non-interactive proof to a verifier in order to attest the age of a bitstring. The ability to create forged proofs depends solely on the period of corruption and the adversaries' ability to compute VDFs in less time, which can only be achieved through a faster VDF core, as VDFs are designed to be resistant to parallelization. We show that both constructions securely implement 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.

Finally, in Section 7, we discuss the impact and potential applications of our construction.

2 Model and Definitions

We construct our protocol in the universal-composability framework [Can18] 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. Representing time in the UC framework is a non-trivial issue, as the framework is fundamentally synchronous. In general, the ordering of events is dictated by the passing of messages, where only one machine (party) is active at a time, which does not correspond to the usual concepts of time and concurrent computing. However, the UC framework remains sufficiently general to represent distributed computations at any granularities (see [Can18]):

> *Indeed, by setting individual activations of ITMs to represent the appropriate granularity of "atomic sequential operations", it is possible to represent distributed computations at any desired level of abstraction or detail - from the granularity of physical interrupts in actual CPUs, to virtualized processes in a multi-process virtual system, to abstract processes that do not necessarily correspond to specific physical computations.*

We represent time through a global read-only functionality clock that any party can access. When queried, this functionality outputs a (strictly monotonously increasing) **time receipt** of constant length θ stating the current time. For instance, clock may increase by one for every step of the active machine. We do not specify exactly how clock increases, but any monotously increasing function is sufficient for our purposes as we show in the modeling of VDFs. This leaves our model sufficiently general and allows close representations of real-world clocks and distributed computing. As an abuse of notation, we perform operations over time receipts, where the result of these operations is a natural number.

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 saved into 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 accept. Otherwise, return reject.

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. The computation of these proofs is not inherently sequential and should not take too long to compute compared to the function evaluation. For simplification, we treats both outputs as one, assuming that the proof construction is effectively immediate. VDFs have unique proofs, so we can still assert than our construction is still a function.

More formally, we consider a verifiable delay function to be a triple of algorithms ($\mathsf{VDF.gen}$, $\mathsf{VDF.verify}, \mathsf{VDF.extend}$) with security parameter μ and parameters $g, v \in \mathbb{N}$ as defined below.

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

$\mathsf{VDF.verify}(x, s, p)$ 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 $(s, p) = \mathsf{VDF.gen}(x, s)$, and reject otherwise, in at most $s \cdot v$ time steps.

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

Our construction differs from the definitions in the literature in a couple of ways. The output of the VDF contains not only the final result of the computation but also a proof that allows for faster verification. We treat both outputs as one, which we can do as VDFs have unique proofs. This point of view assumes that computation of these proofs is effectively immediate. This approach is also outlined in [Wes18], which states that the computation of the proof can be considered as part of the overall computation, without noticeably affecting the computation time (as generally the proof generation can be parallelized). We have taken this road, as treating the output as two separate entities introduces uninteresting complications.

Another difference lies on what we consider an execution of the VDF. As seen in [LW15], VDFs can be chained together for better efficiency. All constructions assume that the number of sequential executions is a fixed parameter, which we call *strength*. Instead of that, we assume that the function is iterated indefinitely until the party decides to stop it, receiving the number of iterations s in the output.

If we consider a VDF to be the sequential composition of a function, given an output it is possible to start executing the VDF from that output, as a continuation of the original execution. Therefore, we assume an additional algorithm $\mathsf{VDF.extend}$ that allows continuing an existing VDF output:

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

We use this mechanism to model the capability of running VDFs a variable number of iterations that is decided a posteriori.

Moreover, we use the ability to 'extend' VDF outputs to capture the adversaries capability to query the running VDF computations of any just-corrupted party and then continue VDF

computations with its own faster VDF-rate. Note that if we do not assume that proof computation is immediate, extending a VDF through this mechanism would require wasting time computing the proof more than once. We use this algorithm to allow the adversary to act in a realistic manner within the UC framework, honest parties have no need for it. Because our assumption benefits only the adversary in this case, the security oproperties are not affected by allowing VDF extension.

We require perfect **correctness**: $\mathsf{VDF.verify}(x, \mathsf{VDF.gen}(x, s)) = \mathsf{accept}$ for all $x \in \{0,1\}^*$ and $s \in \mathbb{N}$. The VDF is called **sound** 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 $\mathsf{VDF.verify}(x, s, p) = \mathsf{accept}$ with non-negligible probability. The **usability** of the VDF is the factor g/v by which verification is faster than generation of the proof.

To deal with VDFs and real-world time, we consider for each party its VDF-rate γ which represents the largest number of iterations of the VDF that that party can compute in a time unit given its computational resources. For our results to be meaningful, the honest prover must compute the VDF efficiently enough such that the advantage the adversary might get, represented by α, is not too large. 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. Studies in specialized hardware for [Wes18] have already started[5], rendering our assumption even closer to reality.

3.1 Candidate constructions

A candidate construction that satisfies our notion of a VDF is the *sloth* construction by Lenstra and Wesolowski [LW15] that iterates modular-square-root and (keyed)-binary-permutation functions. Unfortunately, this construction only has a logarithmic usability. Wesolowski's subsequent construction [Wes18] is the closest to our characterization. It is especially useful as the output consists of only two field elements. This succinctness of the proof is particularly interesting for our work, minimizing overheads in the produced timestamps.

While our constructions use verifiable delay functions, in essence they could be substituted by proofs of sequential work like the ones presented in [CP18, MMV13], although this would require slightly changing our VDF, as the outputs must be treated differently due to non-uniqueness of the proof. A more practical problem is that the security level of these constructions depend on the size of the proof, which consists of node labels. In contrast, most VDFs allow for constant-size proofs. Furthermore, timed-release cryptography primitives [RSW96] cannot be used for our purpose, even if the principle is similar to the preceding functions. The reason is that timed-release cryptography allows for a puzzle and a solution to be sampled at the same time, hence the generating party already knows the solution and does not have to solve the puzzle.

3.2 Modelling Time and Verifiable Delay Functions

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. To that end, we directly link each party's VDF rate to the global clock, which we have modeled as a read-only global oracle that returns monotously increasing time receipts. Throughout our construction, we simulate the execution of our VDF through the oracle functionality presented in Figure 1.

Our VDF is represented a random oracle VDF that cannot be directly queried by participants. Instead, they only interact with it through a functionality $\mathcal{F}_{\mathsf{VDF}}^{\gamma}$. The $\mathcal{F}_{\mathsf{VDF}}^{\gamma}$ oracle simulates the execution of the VDF across the passage of time. Participants can query this functionality to start a computation and again to output the result. Depending on the amount of time passed between these two events and the rate γ, participants get a proof of a certain strength. The strength represents the number of sequential iterations that were computed. By adding the notion of rate, we can compare the execution of the same function in different hardware and the implications this has for the timestamping protocol. In particular, the ability of the adversary to compute VDFs more efficiently than an honest party.

In addition, participants are able to input any string to the functionality, so if this is not the case for the particular VDF construction then we assume that a full-domain hash is applied to any

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

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

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

- `On input` (start, x):
  `Update` $Q \leftarrow Q \cup \{(x, \mathsf{clock}())\}$.
- `On input` (output, x) at time $t_o = \mathsf{clock}()$:
  `Let` $t_s \leftarrow \min_t\{(x,t) \in Q\}$, `return` $\perp$ if there is no such t_s;
  `Let` $s := (t_o - t_s) \cdot \gamma$; `/* the strength of the resulting proof */`
  `Let` $p := \mathsf{VDF}(x, s)$;
  `Return` (s, p)
- `On input` $(\mathsf{verify}, x, p, s)$:
  `If` $\mathsf{VDF}(x, s) = p$ `then return accept`, `else return reject`.
- `On input` $(\mathsf{extend}, x, p, s)$:
  `/* Continue computation from a given existing function output */`
  `If` $\mathsf{VDF}(x, s) \neq p$ `return` $\perp$.
  `Else update` $Q \leftarrow Q \cup \{(x, \mathsf{clock}() - {}^{s}/_{\gamma})\}$.

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

inputs. In the case of the verifier, its own VDF-rate is not important in which case we just refer to $\mathcal{F}_{\mathsf{VDF}}^{*}$.

This functionality replicates the desirable properties of a verifiable delay function. Correctness and soundness are achieved by default. Every valid proof is accepted ($\mathsf{VDF}(x, s) = p$) and an adversary is able to guess valid proofs by performing verify queries, which it can only successfully do with negligible probability in μ.

Note that these VDF oracles are only active when it responds to a query, but the returned outputs remain solely dependent on the difference between time receipts of the start and end of the VDF computation. This ensures that regardless of the UC-framework synchronous modeling where only one machine is active at a time, our VDF oracle faithfully models VDF computations linked to the passage of the clock.

Our construction of $\mathcal{F}_{\mathsf{VDF}}^{\gamma}$ is not only relevant for timestamping but something similar could be used to represent timed-release cryptography. Our approach fulfills the expected properties of the passage of time, effectively creating a relationship between sequential computation and clock time. Universal composability is framework that acts in a very high level of abstraction. Our construction allows us to use it to express timestamping and computational limitations, which the framework was not designed to handle. This approach might be of independent interest as it allows us to talk about physical properties of computation, as long as you accept that $\mathcal{F}_{\mathsf{VDF}}^{\gamma}$ accurately reflects the real-world properties of a VDF.

4 A trivial timestamping scheme from VDFs

In this section we present a trivial construction based on the one found in [MMV13], 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 $\mathcal{F}_{\text{triv}}^{\alpha}$, justifying the design choices made for the protocol presented in Section 5. 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 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.

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

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

Fig. 2: Simple timestamping functionality

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), $c \in \{0,1\}^*$:
  `Query` $\mathcal{F}^{\gamma}_{\mathsf{VDF}}(\mathsf{start}, c)$.
- `On input` (stamp, c), $c \in \{0,1\}^*$:
  `Query` $u \leftarrow \mathcal{F}^{\gamma}_{\mathsf{VDF}}(\mathsf{output}, c)$;
  `If` $u = \perp$ `then return` $(c, 0, \perp)$;
  `/* Here` $u \in \mathbb{N}^+ \times \{0,1\}^{\mu}$ `*/`
  `Let` $(s, p) = u$, $a = s/\gamma$;
  `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 $\mathcal{F}^{*}_{\mathsf{VDF}}$:

- `On input` $(\mathsf{verify}, c, a, u)$,
  $c \in \{0,1\}^*, a \in \mathbb{R}^+, u \in \{0,1\}^*$:
  `If` $u \neq (s, p) \in \mathbb{N}^+ \times \{0,1\}^{\mu}$ `then return` reject;
  `If` $a \neq s/\gamma$ `then return` reject;
  `Return` $\mathcal{F}^{*}_{\mathsf{VDF}}(\mathsf{verify}, c, s, p)$.

(b) Simple VDF verifier

Fig. 3: Simple timestamping prover and verifier using VDFs

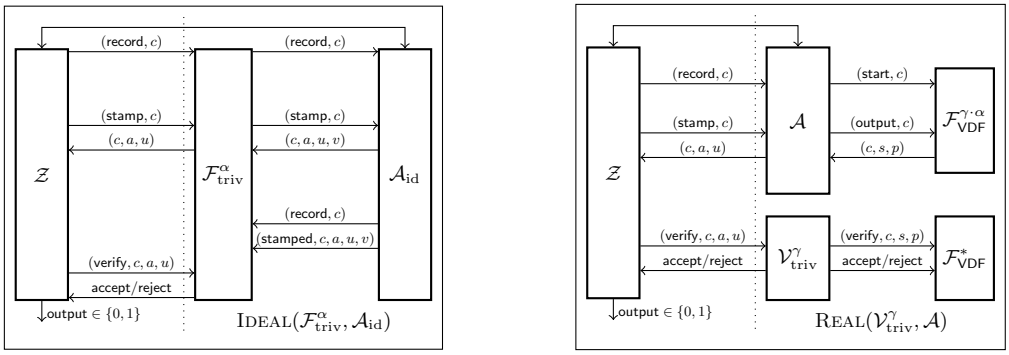
4.1 Security proof

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 4a), or interacts with the real world $\textsc{Real}(\mathcal{V}^{\gamma}_{\text{triv}}, \mathcal{A})$ (cf. Figure 4b), 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 $\mathcal{S}^{\mathcal{A},\mathcal{F}^{\gamma\cdot\alpha}_{\mathsf{VDF}}}_{\text{triv}}$, which has black-box access to the PPT real adversary $\mathcal{A}$ (and may observe its oracle calls).

Theorem 4.1. *Let $\mu \in \mathbb{N}^{+}$ be the security parameter. For any real-world PPT adversary $\mathcal{A}$ with oracle access to $\mathcal{F}^{\gamma\cdot\alpha}_{\mathsf{VDF}}$, 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 3b,4b) 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 $\mathcal{Z}^{\textsc{Ideal}(\mathcal{F}^{\alpha}_{\text{triv}},\mathcal{A}_{\text{id}})}$

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

Fig. 4: Environment $\mathcal{Z}$ in ideal- and real-world experiment for simple timestamping

Proof. Let the ideal-world simulator $\mathcal{A}_{\text{id}} := \mathcal{S}^{\mathcal{A},\mathcal{F}^{\gamma\cdot\alpha}_{\mathsf{VDF}}}_{\text{triv}}$ be defined as in Figure 5. We consider all related execution transcripts in the ideal and real world

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

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:

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

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 $(t_{\text{bad}}, \mathcal{Z}, (\mathsf{verify}, c, a, u)) \in \Pi_{ideal} \cap \Pi_{real}$ 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_{verify}, and q_{VDF} be the maximum of the

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

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

Fig. 5: Simulator

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

Next consider the case when $\mathcal{S}_{\mathrm{triv}}^{\mathcal{A},\mathcal{F}_{\mathsf{VDF}}^{\gamma\cdot\alpha}}$ never outputted $(\mathsf{stamped}, c, a, u, 1)$ in Π_{ideal}. This implies that the real adversary never received $(s, p) \leftarrow \mathcal{F}_{\mathsf{VDF}}^{\gamma\cdot\alpha}(\mathsf{output}, c)$, where $(s, p) = u$. Note that the value $\mathsf{VDF}(c, s)$ can only be directly queried through a $\mathcal{F}_{\mathsf{VDF}}^{*}(\mathsf{output}, c)$ query, which thus did not happen. The only other way to learn that $p = \mathsf{VDF}(c, s)$ is indirectly through a $\mathcal{F}_{\mathsf{VDF}}^{*}(\mathsf{verify}, c, s, p) \in \{\mathsf{accept}, \mathsf{reject}\}$ query. Since VDF is a random oracle of bitlength μ, this event occurs with probability upper-bounded by $q_{\mathsf{verify}} \cdot 2^{-\mu}$.

What remains is when $o_i = \mathsf{reject}$, $o_r = \mathsf{accept}$ and $\mathcal{S}_{\mathrm{triv}}^{\mathcal{A},\mathcal{F}_{\mathsf{VDF}}^{\gamma\cdot\alpha}}$ has outputted $(\mathsf{stamped}, c, a, u, 1)$. Then since $o_i = \mathsf{reject}$ it must be caused by one of the two rules in `Procedure checkstamp` of the ideal functionality (cf. Figure 2) 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 $(s, p) \leftarrow \mathcal{F}_{\mathsf{VDF}}^{\gamma\cdot\alpha}(\mathsf{output}, c)$, at which point the simulator immediately makes a $(\mathsf{stamped}, c, s/\gamma, (s, p), 1)$ query to the functionality. By the definition of $\mathcal{F}_{\mathsf{VDF}}^{\gamma\cdot\alpha}$, the adversary $\mathcal{A}$ must have queried $\mathcal{F}_{\mathsf{VDF}}^{\gamma\cdot\alpha}(\mathsf{start}, c)$ exactly the amount of time $s/\gamma_{\mathcal{A}}$ before that, at which time the simulator made a (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 $(\mathsf{verify}, c, a, (s, p))$ query made earlier. Since VDF is a random oracle with bitlength μ, this event happens with probability upper-bounded by $q_{\mathsf{verify}} \cdot 2^{-\mu}$.

We conclude that

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

where q_{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. □

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

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

- `On input corrupt` by $\mathcal{A}_{\text{id}}$ at time $t = \mathsf{clock}()$:
  `If` $T_{\text{corr}} = \infty$ `then set` $T_{\text{corr}} \leftarrow t$.
- `On input` (record, c), $c \in \{0,1\}^*$ at time $t = \mathsf{clock}()$:
  `Send` (record, c) `to` $\mathcal{A}_{\text{id}}$;
  `Set` $\mathcal{R} \leftarrow \mathcal{R} \cup (c, t)$.
- `Procedure` $\mathsf{checkstamp}(c, a, u, v)$:
  `Let` $\hat{t} = \min\{t \mid (c,t) \in \mathcal{R}\} \cup \{\infty\}$.
  `If` $a > (\mathsf{clock}() - \hat{t})$: `/* Claimed age` a `incorrect given record? */`
    `If` $a > \alpha \cdot \min(\mathsf{clock}() - T_{\text{corr}}, \mathsf{clock}() - \hat{t})$ `then` $v \leftarrow 0$;
  `If` $\exists(c, a, u, \hat{v}) \in \mathcal{C}$ `then` $v \leftarrow \hat{v}$; `/* For consistency. */`
  `Let` $\mathcal{C} \leftarrow \mathcal{C} \cup (c, a, u, v)$.
- `On input` (stamp, c), $c \in \{0,1\}^*$:
  `Query` $(c, a, u, v) \leftarrow \mathcal{A}_{\text{id}}(\mathsf{stamp}, c)$; `/*` $\mathcal{A}_{\text{id}}$ `produces proof strings */`
  `Call` $\mathsf{checkstamp}(c, a, u, v)$;
  `Return` (c, a, u).
- `On input` $(\mathtt{verify}, c, a, u)$, $c \in \{0,1\}^*, a \in \mathbb{R}, p \in \{0,1\}^*$:
  `If` $\neg\exists\hat{v} : (c, a, u, \hat{v}) \in \mathcal{C}$ `then`
    `Query` $v \leftarrow \mathcal{A}_{\text{id}}(\mathsf{cnewstamp}, c, a, u)$.
    `Call` $\mathsf{checkstamp}(c, a, u, v)$;
  `If` $(c, a, u, 1) \in \mathcal{C}$ `then return accept`;
  `Else return reject`.

Fig. 6: Timestamping functionality $\mathcal{F}_{\text{ts}}^{\alpha}$

4.2 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.
  We solve this problem by using digital signatures to prevent the adversary from constructing valid timestamp triples with α-time-dilution unless it has corrupted the prover. Note that once it has corrupted the prover, the use of digital signatures still allows time-dilution for all prior records.
  We can actually limit the adversary 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 $\mathcal{F}_{\mathsf{VDF}}^{\gamma}.\mathsf{start}$ 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.

5 A UC non-interactive timestamping functionality

In this section change our protocol to realize a timestamping functionality that fulfills stronger security guarantees than $\mathcal{F}_{\text{triv}}^{\alpha}$.

Our protocol differs to the construction in [MMV13] in a simple manner. Given an existing timestamp, an adversary with a faster VDF rate could work over the existing timestamp to make

it seem older than it is. We prevent this by adding a time receipt to the timestamp, which forces the adversary to start computing a new timestamp and prevents them from taking advantage of the honestly computed work. We present a timestamping functionality $\mathcal{F}_{\text{ts}}^{\alpha}$ defined in Figure 6 (see also Figure 8a) which generates proofs for *age*, i.e., for the amount of elapsed time between the recording of the message and the generation of the proof.

Such a proof of age is not the same as a proof of the actual time when the message was recorded. For any generated proof the proven age is fixed and will stay the same while time continues to progress. It is possible to verify that a message was recorded around a particular time by querying a new proof and immediately verifying it. What our protocol does not do is prevent an attacker from post-dating a record, that is, pretending it was first recorded later than it was. In our functionality we assume the adversary has access to better (computational) resources than the honest player and can use these resources 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.

The functionality accepts four inputs, including a corruption input corrupt after which the adversary is allowed to input new records and amnipulate timestamps. The functionality receives records through the record query and saves when this happened. It generates a timestamp whenever it recieves a stamp input with the appropriate record. The adversary can choose how the timestamp looks like but can only modify the age if they have corrupted the functionality. Even then, they are limited by the checkstamp procedure, which checks whether the presented age of the timestamp is correct. If the functionality is not corrupted, it simply checks whether the age in the timestamp does not exceed the time elapsed since the record was originally queried. When the functionality is corrupted, the adversary can stretch a timestamp by a factor α but only if enough time has elapsed since corruption. That is, an adversary can only modify a timestamp if they have been in control of the functionality for at least the age of the timestamp divided by α. The procedure additionally checks whether the triple of record, timestamp and age has been registered before. Then, the triple is registered with a validitity 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 is not saved but is within acceptable parameters, it queries the adversary whether it is a valid timestamp or not and outputs the adversary's answer.

This functionality starts uncorrupted (time of corruption $T_{\text{corr}} = \infty$ is set to infinity) and produces timestamps that are faithful to the record. It allows a corruption message (setting $T_{\text{corr}} \leftarrow \mathsf{clock}()$) after which the adversary is allowed to exploit its time-dilution factor α in timestamps. However, any produced accepted timestamp that is unfaithful is not only bounded by α times the real age, but in certain cases, the bound is even stricter, preventing the creation of timestamps of a certain age within a period $\alpha \cdot (\mathsf{clock}() - T_{\text{corr}})$. This implies that any accepted timestamp produced by the adversary with claimed age older than $\alpha \cdot (\mathsf{clock}() - T_{\text{corr}})$ is faithful. This is a stronger result than the one found in Section 4.

The functionality also allows the adversary to submit records and stamps so that the functionality can potentially accept stamps for some message c for which the environment never called (record, c) nor (stamp, c). This case is possible because the environment and adversary can freely communicate, but even if they could not, they could use record and stamp queries as a covert channel. Nevertheless, note that valid stamps always have to fulfill the basic requirements from the functionality: the real age a_{real} and claimed age a satisfy $a \leq a_{real} \cdot \alpha$. The functionality is simplified since it does not distinguish between the public record query from the environment and the private one from the adversary. This distinction is not necessary because it is the adversary who decides how to respond to the public stamp queries.

5.1 Instantiation with VDFs

We can instantiate this functionality with an honest prover $\mathcal{P}_{\text{ts}}^{\gamma}$ (cf. Figure 7a) and an honest verifier $\mathcal{V}_{\text{ts}}^{\gamma}$ (cf. Figure 7b). Note that actual proofs are of the form $(t, \sigma, s, p) \in \{0,1\}^{\theta} \times \{0,1\}^{\kappa} \times \mathbb{N}^{+} \times \{0,1\}^{\mu}$, we assume an efficient unambiguous encoding to bitstring $\{0,1\}^{*}$ is fixed.

The role of the prover $\mathcal{P}_{\text{ts}}^{\gamma}$ is to start the execution of $\mathcal{F}_{\mathsf{VDF}}^{\gamma}$ with the input included in the record query whenever they receive one. The input to the VDF also includes a time receipt of the current time as well as a signature. When they receive a stamp input, they query for an output from the

VDF-based prover $\mathcal{P}^{\gamma}_{\mathbf{ts}}$

Let μ and κ be the security parameters of VDF and Σ, respectively, and γ the VDF-rate of $\mathcal{P}^{\gamma}_{\mathrm{ts}}$'s oracle oracle $\mathcal{F}^{\gamma}_{\mathsf{VDF}}$. Let $(pk_{\mathcal{P}}, sk_{\mathcal{P}}) \leftarrow \Sigma.\mathsf{keygen}()$, publish public key $pk_{\mathcal{P}}$. Let $\mathcal{R} = \varnothing$, then $\mathcal{P}^{\gamma}_{\mathrm{ts}}$ proceeds as follows:

- `On input` (record, c), $c \in \{0,1\}^*$:
  - `If` $\exists (t', c, \sigma') \in \mathcal{R}$ `then return`;
  - `Let` $t = \mathsf{clock}()$, $\sigma = \Sigma.\mathsf{sign}(t||c, sk_{\mathcal{P}})$;
  - `Set` $\mathcal{R} \leftarrow \mathcal{R} \cup \{(t, c, \sigma)\}$;
  - `Query` $\mathcal{F}^{\gamma}_{\mathsf{VDF}}(\mathsf{start}, t||c||\sigma)$;
- `On input` (stamp, c), $c \in \{0,1\}^*$:
  - `Let` t, σ such that $(t, c, \sigma) \in \mathcal{R}$,
  - `Else return` $(0, c, \perp)$;
  - `Query` $(s, p) \leftarrow \mathcal{F}^{\gamma}_{\mathsf{VDF}}(\mathsf{output}, c||t||\sigma)$;
  - `Let` $u = (t, \sigma, s, p)$ and $a = {}^{s}/_{\gamma}$;
  - `Return` (c, a, u);
- `On input` $(\mathsf{corrupt})$ from $\mathcal{A}$:
  - `Send` $sk_{\mathcal{P}}$ to $\mathcal{A}$.
  - `For each` $(t, c, \sigma) \in \mathcal{R}$:
    - `Query` $(s, p) \leftarrow \mathcal{F}^{\gamma}_{\mathsf{VDF}}(\mathsf{output}, c||t||\sigma)$;
    - `Send` (t, c, σ, s, p) `to` $\mathcal{A}$;
  - Transfer control to $\mathcal{A}$;

(a) VDF-based prover

VDF-based verifier $\mathcal{V}^{\gamma}_{\mathbf{ts}}$

Given parameter VDF rates γ and γ_0, and $pk_{\mathcal{P}}$. It answers queries from the environment $\mathcal{Z}$ and interacts with its oracle $\mathcal{F}^{*}_{\mathsf{VDF}}$:

- `On input` $(\mathsf{verify}, c, a, u)$,
  $c \in \{0,1\}^*, a \in \{0,1\}^{\theta}, u \in \{0,1\}^*$:
  - `If` $u \neq (t, \sigma, s, p)$ with $t \in \{0,1\}^{\theta}$, $\sigma \in \{0,1\}^{\kappa}$, $s \in \mathbb{N}^{+}$, $p \in \{0,1\}^{\mu}$ `then return reject`;
  - `If` $\Sigma.\mathsf{verify}(t||c, \sigma, pk_{\mathcal{P}}) = \mathsf{reject}$ `then return reject`;
  - `If not` ${}^{s}/_{\gamma} = a \leq (\mathsf{clock}() - t)$ `then return reject`.
  - `Return` $\mathcal{F}^{*}_{\mathsf{VDF}}.\mathsf{verify}(c, s, p)$.

(b) VDF-based verifier

Fig. 7: Timestamping prover and verifier

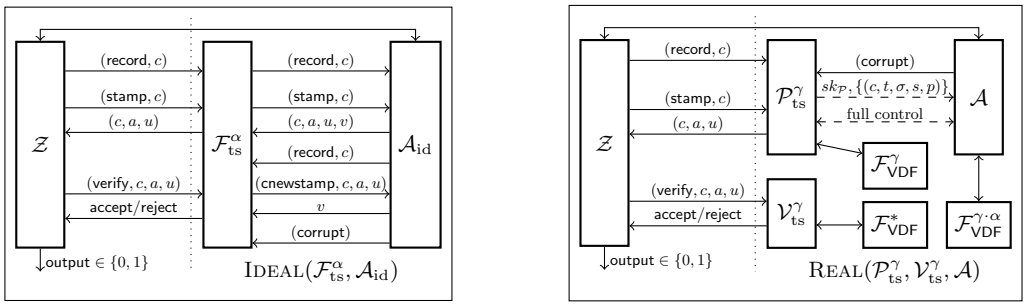
corresponding VDF computation and forwars the output of the VDF, which will be the timestamp that $\mathcal{V}^{\gamma}_{\mathrm{ts}}$ verifies.

As we are simply creating proofs of age, time receipts might seem superfluous. However, they prevent a trivial attack of *stretching* a previously seen timestamp proof, as the time receipt forces a maximum possible age for the record. Digital signatures, on the other hand, allow us to model the corruption of the stamper. Furthermore they also imply the unpredictability of VDF inputs, a deeper discussion on this design choice can be found in Section 7.

5.2 Security proof

We assume a PPT environment $\mathcal{Z}$ that either interacts with the ideal world $\textsc{Ideal}(\mathcal{F}^{\alpha}_{\mathrm{ts}}, \mathcal{A}_{\mathrm{id}})$ with the ideal functionality and an ideal adversary (cf. Figure 4a), or interacts with the real world $\textsc{Real}(\mathcal{P}^{\gamma}_{\mathrm{ts}}, \mathcal{V}^{\gamma}_{\mathrm{ts}}, \mathcal{A})$ (cf. Figure 4b), 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 $\mathcal{A}_{\mathrm{id}}$ by a similator $\mathcal{S}^{\mathcal{A}}_{\mathrm{ts}}$, which has black-box access to the PPT real adversary $\mathcal{A}$ (and may observe its oracle calls).



(a) Ideal world $\mathcal{Z}^{\text{IDEAL}(\mathcal{F}_{\text{ts}}^{\alpha}, \mathcal{A}_{\text{id}})}$

(b) Real world $\mathcal{Z}^{\text{REAL}(\mathcal{P}_{\text{ts}}^{\gamma}, \mathcal{V}_{\text{ts}}^{\gamma}, \mathcal{A})}$

Fig. 8: Ideal and Real world

Theorem 5.1. *Given security parameter $\mu \in \mathbb{N}^+$, there exists a PPT simulator $\mathcal{S}_{ts}^{\mathcal{A}}$ such that for any PPT adversary $\mathcal{A}$ with oracle access to $\mathcal{F}_{\mathsf{VDF}}^{\gamma \cdot \alpha}$ and any PPT environment $\mathcal{Z}$ the advantage that $\mathcal{Z}$ can distinguish between the ideal world with $\mathcal{F}_{ts}^{\alpha}$ and $\mathcal{A}_{id} := \mathcal{S}_{ts}^{\mathcal{A}}$ (cf. Figure 8a, 6, 9) and the real world with $\mathcal{P}_{ts}^{\gamma}$, $\mathcal{V}_{ts}^{\gamma}$, and $\mathcal{A}$ (cf. Figure 8b, 7a, 7b), is bounded as follows:*

$$\left|\Pr[\mathcal{Z}^{\text{IDEAL}(\mathcal{F}_{ts}^{\alpha}, \mathcal{S}_{ts}^{\mathcal{A}})} = 1] - \Pr[\mathcal{Z}^{\text{REAL}(\mathcal{P}_{ts}^{\gamma}, \mathcal{V}_{ts}^{\gamma}, \mathcal{A})} = 1]\right| \leq negl(\mu, \kappa),$$

where $negl(\mu)$ is a negligible function in μ.

Proof. Let $\mathcal{S}_{\text{ts}}^{\mathcal{A}}$ be as defined in Figure 9. Effectively what $\mathcal{S}_{\text{ts}}^{\mathcal{A}}$ does is forward the queries to a simulated prover running the protocol and back. The only thing of note is that it can check whether a prioof is invalid, in which case it tells the functionality to save it as an invalid statement. We consider all related execution transcripts in the ideal and real world

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

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:

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

Note that in the ideal world all record and stamp queries by $\mathcal{Z}$ and their answers by $\mathcal{P}_{\text{ts}}^{\gamma}$ (potentially under control of $\mathcal{A}$) are perfectly forwarded by the functionality and the simulator. Hence 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, consider the first query for which the answer differs between the ideal world and real world. Let $(t_{\text{bad}}, \mathcal{Z}, (\mathsf{verify}, c, a, u)) \in \Pi_{ideal} \cap \Pi_{real}$ be this first query by $\mathcal{Z}$, where t_{bad} is the clocktime of the query, and 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_{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 $o_i = \mathsf{accept}$, this is only possible if $\mathcal{S}_{\text{ts}}^{\mathcal{A}}$ has output $(c, a, u, 1)$ (as answer to a stamp query or as a $\mathsf{cnewstamp}$ query). That can only happen when $\mathcal{V}_{\text{ts}}^{\gamma}.(\mathsf{verify}, c, a, u) = \mathsf{accept}$ and thus that $o_r = \mathsf{accept}$, which is a contradiction. It follows that $o_i = \mathsf{reject}$ and $o_r = \mathsf{accept}$ and u is of the form $(t, \sigma, s, p) \in \{0,1\}^{\theta} \times \{0,1\}^{\kappa} \times \mathbb{N}^+ \times \{0,1\}^{\mu}$, where:

$$\Sigma.\mathsf{verify}(c||t, \sigma, pk_{\mathcal{P}}) = \mathsf{accept}, \quad p = \mathsf{VDF}(c||t||\sigma, s), \quad s/\gamma = a, \quad a \leq (t_{\text{bad}} - t).$$

Next consider the case when $\mathcal{S}_{\text{ts}}^{\mathcal{A}}$ never outputted 0 after a $(\mathsf{cnewstamp}, c, a, u)$ query in Π_{ideal}. This implies that the real adversary never received $(s, p) \leftarrow \mathcal{F}_{\mathsf{VDF}}^{\gamma \cdot \alpha}(\mathsf{output}, c||t||\sigma)$, otherwise $\mathcal{S}_{\text{ts}}^{\mathcal{A}}$

Simulator $\mathcal{S}_{\mathrm{ts}}^{\mathcal{A}}$

Given any real world PPT $\mathcal{A}$, the simulator $\mathcal{S}_{\mathrm{ts}}^{\mathcal{A}}$ runs the $\mathcal{A}$ in a black box manner, the honest prover $\mathcal{P}_{\mathrm{ts}}^{\gamma}$ and the verifier $\mathcal{V}_{\mathrm{ts}}^{\gamma}$. It maintains a list of (record,age,proof)-triples R. It sees all VDF queries they make, but faithfully and immediately forwards all queries and responses. The simulator acts as follows:

- `On input` (record, c), $c \in \{0,1\}^*$:
  `Send` (record, c) `to` $\mathcal{P}_{\mathrm{ts}}^{\gamma}$. `/* potentially under` $\mathcal{A}$ `control */`
- `On input` (stamp, c), $c \in \{0,1\}^*$:
  `/* Query proof from` $\mathcal{P}_{\mathrm{ts}}^{\gamma}$ `(potentially under` $\mathcal{A}$ `control). */`
  `Query` $(c, a, u) \leftarrow \mathcal{P}_{\mathrm{ts}}^{\gamma}(\mathsf{stamp}, c)$;
  `/* Check validity of the proof */`
  `If` $\mathcal{V}_{\mathrm{ts}}^{\gamma}.\mathsf{verify}(c, a, u) = \mathsf{accept}$ `then` $v \leftarrow 1$ `else` $v \leftarrow 0$;
  `Return` (c, a, u, v);
- `When` $\mathcal{A}$ outputs query $\mathsf{corrupt}$ to $\mathcal{P}_{tl}^{\gamma}$:
  `Send` $\mathsf{corrupt}$ `to` $\mathcal{F}_{\mathrm{ts}}^{\alpha}$;
- `When` $\mathcal{A}$ outputs query $(\mathsf{start}, c||t||\sigma)$ to $\mathcal{F}_{\mathsf{VDF}}^{\gamma\cdot\alpha}$:
  `If` $\Sigma.\mathsf{verify}(c||t, \sigma, pk_{\mathcal{P}}) = \mathsf{accept}$ `then send` (record, c) `to` $\mathcal{F}_{\mathrm{ts}}^{\alpha}$;
- `When` $\mathcal{A}$ receives (s, p) for query $(\mathsf{output}, c||t||\sigma)$ from $\mathcal{F}_{\mathsf{VDF}}^{\gamma\cdot\alpha}$:
  `If` $\mathcal{V}_{\mathrm{ts}}^{\gamma}.\mathsf{verify}(c||t||\sigma, {}^{s}/_{\gamma}, u) = \mathsf{accept}$ `then` $v \leftarrow 1$ `else` $v \leftarrow 0$;
  `Let` $R \leftarrow R \cup (c, {}^{s}/_{\gamma}, (t, \sigma, s, p), v)$;
- `On input` $(\mathsf{cnewstamp}, c, a, u)$, $c \in \{0,1\}^*$:
  `If` $(c, a, u, v) \in R$ `then return` v `else return` 0.

Fig. 9: Simulator

would hav recorded the triple in R. Note that the value $\mathsf{VDF}(c||t||\sigma, s)$ can only be directly queried through a $\mathcal{F}_{\mathsf{VDF}}^{\gamma*}(\mathsf{output}, c||t||\sigma)$ query, which thus did not happen. The only other way to learn that $p = \mathsf{VDF}(c||t||\sigma, s)$ is indirectly through a $\mathcal{F}_{\mathsf{VDF}}^{\gamma*}(\mathsf{verify}, c||t||\sigma, s, p) \in \{\mathsf{accept}, \mathsf{reject}\}$ query. Since VDF is a random oracle of bitlength μ, this event occurs with probability upper-bounded by $q_{\mathsf{verify}} \cdot 2^{-\mu}$.

What remains is when $o_i = \mathsf{reject}$, $o_r = \mathsf{accept}$ and $\mathcal{S}_{\mathrm{ts}}^{\mathcal{A}}$ has outputted 1 to a $(\mathsf{cnewstamp}, c, a, u)$ query. Then since $o_i = \mathsf{reject}$ it must be caused by one of the 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 α times the real age a_r:
   Assume that $\mathcal{A}$ received $(s, p) \leftarrow \mathcal{F}_{\mathsf{VDF}}^{\gamma\cdot\alpha}(\mathsf{output}, c||t||\sigma)$, at which point the simulator immediately makes a $(, c, {}^{s}/_{\gamma}, (t, \sigma, s, p), 1)$ to R. By the definition of $\mathcal{F}_{\mathsf{VDF}}^{\gamma\cdot\alpha}$, the adversary $\mathcal{A}$ must have queried $\mathcal{F}_{\mathsf{VDF}}^{\gamma\cdot\alpha}(\mathsf{start}, c||t||\sigma)$ exactly the amount of time $s/\gamma_{\mathcal{A}}$ before that, at which time the simulator made a (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. If claimed age a is older than α times the time a_{corr} since corruption:
   Assume that $\mathcal{A}$ received $(s, p) \leftarrow \mathcal{F}_{\mathsf{VDF}}^{\gamma\cdot\alpha}(\mathsf{output}, c||t||\sigma)$. By the definition of $\mathcal{F}_{\mathsf{VDF}}^{\gamma\cdot\alpha}$, the adversary $\mathcal{A}$ must have queried $\mathcal{F}_{\mathsf{VDF}}^{\gamma\cdot\alpha}(\mathsf{start}, c||t||\sigma)$ exactly the amount of time $s/(\gamma \cdot \alpha)$ before that, which is before corruption. The adversary cannot yet legitimately query $\sigma = \Sigma.\mathsf{sign}(sk_{\mathcal{P}}, c||t)$, and $\sigma = \Sigma.\mathsf{sign}(sk_{\mathcal{P}}, c||t)$ also has not appeared legitimaly before (otherwise $o_i = \mathsf{accept}$). It follows that the probability of this signature forging event is negligible in κ.
   Otherwise $\mathcal{A}$ never received $(s, p) \leftarrow \mathcal{F}_{\mathsf{VDF}}^{\gamma\cdot\alpha}(\mathsf{output}, c||t||\sigma)$ and successfully guessed p. This event is upper-bounded by $2^{-\mu} \cdot q_{\mathsf{VDF}}$.
3. The case that $(c, a, (t, \sigma, s, p), 0) \in \mathcal{P}$:
   The environment $\mathcal{Z}$ correctly guessed $p = \mathsf{VDF}(c||t||\sigma, s)$ in a $(\mathsf{verify}, c, a, (t, \sigma, s, p))$ query made earlier. This event happens with probability upper-bounded by $q_{\mathsf{verify}} \cdot 2^{-\mu}$.

We conclude that

$$\Pr[\textsc{View}(\mathcal{Z}, \Pi_{ideal}) \neq \textsc{View}(\mathcal{Z}, \Pi_{real})]$$

is bounded by the finite sum of the above probabilities that are all negligible in μ and κ, which proves the theorem. □

We have shown that our protocol can realize our ideal functionality and therefore allows us to create secure timestamps based on our VDF.

6 A more efficient timestamping scheme

The protocol we constructed in the previous section securely realizes our timestamping functionality. However, the construction is not practical as it does not scale. Each new timestamp requires running another independent instance of the VDF. In particular, if any prover wants to continue producing timestamps for a certain record entry then it must continue to compute that VDF instance indefinitely. In order to sidestep this issue, we turn to one of the constructions in Haber and Stornetta's seminal timestamping paper: the hashchain.

A hashchain is exactly what its name suggests, a chain of records linked together through cryptographic hashes. Each record is hashed in conjunction with the hash of the previous record to create a chain. With the right choice of hash function, we can ensure that each element can only be created after all the preceding ones are known. It is natural to combine this concept with our VDF-based timestamps to be able to generate timestamps while computing only a single instance of our VDF at each moment in time. The price we pay for this benefit is longer proof strings, as they have to consist of a chain of hashes and VDFs. On the other hand, this allows for faster verification, as it is possible to verify each VDF in parallel. Our construction enhances the original hashchain from [HS91], as it now provides more than a simple ordering.

In order to construct our new protocol, we need to introduce a couple of additional concepts. Our timestamps consist of sequences of VDF-proofs that are linked to each other through cryptographic hash functions, modelled as random-oracle sequences as originally presented in [CP18]. 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 preserve the security guarantees from our previous construction.

First, we define these sequences and show what properties we require from them and how they can be enhanced. Then, we define the data structure used by the prover to save (and generate) their timestamps, which consist of so-called blocks chained through hash functions.

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

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 a canonical deterministic way.

$\mathsf{MT.root}(T)$ computes the root hash h of the Merkle Tree for some ordered finite 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 $\langle x_i \mid x_i \in \{0,1\}^\lambda \rangle_l$ where $x_0 = v$, $x_{l-1} = \mathsf{MT.root}(T)$, $x_i \in \{0,1\}^\lambda$ 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 $P = \langle x_i \mid x_i \in \{0,1\}^\lambda \rangle_l$ 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. 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.

Later in this section, we define the concept of a block, which we assume to have a canonical representation as a Merkle tree T.

6.1 Random Oracle Sequences

We analyze recursive calls to the random oracle H and to the random oracle VDF underlying all oracles $\mathcal{F}^*_{\mathsf{VDF}}$ and analyze the cumulative strength of the verifiable delay functions that are found in the sequence. We have adapted the following lemmas from [CP18] to this setting.

Lemma 6.1 (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 $\mathcal{F}^{\gamma}_{\mathsf{VDF}}(\mathsf{output}, x)$ that includes s in the output.

Definition 6.2 (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 $S = \langle (s_i, x_i) \mid s_i \in \mathbb{N} \cup \{\bot\}, x_i \in \{0,1\}^* \rangle_l$, where the following holds for each $0 \leq i < l$: 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.*

It is simple to verify that any Merkle path $\mathsf{MT.path}(T, v) = \langle x_0, \ldots, x_{l-1} \rangle$ induces an $H2$-sequence of the form $\langle (\bot, x'_0), (\bot, x'_1), \ldots, (\bot, x'_{l-2}), (\bot, x_{l-1}) \rangle$ of length l, where x_i is a substring of x'_i for $0 \leq i < l - 1$. 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 6.3 (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 6.4 (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 $(\langle (s_0, x_0), (s_{l-1}, x_{l-1}) \rangle_l$ without making the queries $(s_0, x_0), \ldots, (s_{l-1}, x_{l-1})$ to respectively H and VDF sequentially is at most*

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

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

Proof. Constructing an $H2$-sequence without making the queries $(s_0, x_0), \ldots, (s_{l-1}, x_{l-1})$ 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 $\mathsf{VDF}(s_j, x_j)$ (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 $q_{\mathsf{H}}(Q_{\mathsf{H}} + Q_{\mathsf{VDF}} + \sum_{i=0}^{l} |x_i|)2^{-\lambda} + q_{\mathsf{VDF}}(Q_{\mathsf{H}} + Q_{\mathsf{VDF}} + \sum_{i=0}^{l} |x_i|)2^{-\mu}$.

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

Thus when an adversary outputs an $H2$-sequence of strength L where $2 \cdot (q_{\mathsf{H}}2^{-\lambda} + q_{\mathsf{VDF}}2^{-\mu}) \cdot (Q_{\mathsf{H}} + Q_{\mathsf{VDF}} + \sum_{i=0}^{l} |x_i|)$ 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 [CP18] 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 incertain contexts.

Using $H2$-sequences is not enough to create timestamps for the same reasons discussed in Section 5. 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 $\mathcal{F}^{\gamma \cdot \alpha}_{\mathsf{VDF}}$ 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 6.5 ($H2T$-sequence). *Let $S = \langle (s_i, x_i) \rangle_l$ be an $H2$-sequence of length l with I_{VDF} the VDF-index set of S and $I^{-1}_{\mathsf{VDF}} = \{i - 1 \mid i \in I_{\mathsf{VDF}}, s_{i-1} = \perp\}$. We call S an $H2T$-**sequence** if the following properties hold:*

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

*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.*

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^{-1}_{VDF}). 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 6.6 ($H2T$-verification). *Given a sequence $S = \langle (s_i, x_i) \rangle_l$, VDF-rate γ and interrupt-time ϵ, we define the following $H2T$-verification algorithm $\mathsf{h2tsverify}(S, \gamma, \varepsilon)$ given access to $\mathcal{F}^{*}_{\mathsf{VDF}}$:*

- *Check whether S is a valid H2-sequence* i.e. *for all $i < l$, either $x_{i+1} = a||\mathsf{H}(x_i)||b$ and $s_{i+1} = \perp$ or $x_{i+1} = a||\mathsf{VDF}(x_i, s_{i+1})||b$ and $s_{i+1} \neq \perp$ for some $a, b \in \{0,1\}^*$*[7].
- *Verify that for all $i \in I^{-1}_{\mathsf{VDF}} \cup I_{\mathsf{VDF}}$ we have that $x_i = t_i||y_i$ with $t_i \in \{0,1\}^\theta$;*
- *For all $i \in I^{-1}_{\mathsf{VDF}}$: $t_{i+1} - t_i \leq {}^{s_i}/_{\gamma}$, 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 reject. *Otherwise, output* accept.

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 6.7 (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 $\mathcal{F}^{\gamma\cdot\alpha}_{\mathsf{VDF}}$ 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 $S = \langle (x_i, s_i) \rangle_l$ 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 accept.

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 $\mathcal{F}^{\gamma\cdot\alpha}_{\mathsf{VDF}}$ 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 $S[I^{-1}_{\mathsf{VDF}}]$: $(\perp, t_0 - \tau||x_k)$. 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 6.8 (Unforgeable $H2T$-sequences). *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 $\mathcal{F}^{\gamma_{\mathcal{A}}}_{\mathsf{VDF}}$.

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

[7] In the second case, the verifier must make a $(\mathsf{verify}, x_i, y, s)$ 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.

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 6.4 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.

As we saw in Section 5, 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.

Definition 6.9 ($H2TS$-sequence). *Let pk be a public key for a signature scheme Σ and $S = \langle (s_i, x_i) \rangle_l$ be an $H2T$-sequence of length l with I_{VDF} the VDF-index set of S and $I_{\mathsf{VDF}}^{-1} = \{i - 1 \mid i \in I_{\mathsf{VDF}}\}$. We call S an $H2TS$-**sequence for** pk if for $i \in I_{\mathsf{VDF}}^{-1}$: $x_i = t_i || r_i || \sigma_i$ where $t_i \in \{0,1\}^\theta$ is a time receipt and $\sigma_i \in \{0,1\}^\kappa$ is such that $\Sigma.\mathsf{verify}(pk, t_i || r_i, \sigma_i) = \mathsf{accept}$.*

6.2 Constructing the Sequences

While the aforementioned sequences provide the security properties that we expect our timestamps to have, our protocol utilizes different tools. 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, in a similar way as the hashchain by Haber and Stornetta [HS91] and similar to blockchains [Nak08].

Definition 6.10 (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 $\Sigma.\mathsf{verify}(pk, t^u || prev, sig) = \mathsf{accept}$;*
4. *$vo = (s, p)$ is a VDF output: $p = \mathsf{VDF}(t^u || prev || sig, s)$*
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 $B.pk$, $B.rnd$, $B.prev$, $B.vi$, $B.vo$, $B.t^u$, $B.sig$, $B.t$, $B.s$, $B.p$ and $B.c$ to refer to these elements in block B.

We assume that there is a canonical construction for the Merkle tree of a block. The exact construction is not important, but we make some characterizations to simplify the construction. We are interested in having $t||p$ be a leaf of the tree. Eagle-eyed readers will note that this string is similar to what we expect from an element of $S[I_{\mathsf{VDF}}]$, where each element of the sequence must consist of the output of a VDF (p) preceded by a time receipt (t). Additionally, we have that $prev$ and c are leaves. These assumptions allow for an easy characterization of the link between these instances and the next VDF input.

Definition 6.11 (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. *$B_i.p = \mathsf{VDF}(B_i.t^u || B_i.prev || B_i.sig, B_i.s)$ for $i \geq 0$;*
4. *$\Sigma.\mathsf{verify}(pk, B_i.t^u || B_i.prev, B_i.sig) = \mathsf{accept}$;*

$H2TS$-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 $((0, \mathsf{H}(pk), (\bot, 0), (0^{\theta}, 0^{\kappa}), \mathsf{clock}(), \bot))$ and a variable triple $(t^u, prev, sig)$. Upon initialization, it initializes $t^u \leftarrow \mathsf{clock}()$, $prev \leftarrow \mathsf{MT.root}(B_0)$ and $sig \leftarrow \Sigma.\mathsf{sign}(sk, t^u||prev)$ and inputs $(\mathsf{start}, t^u||prev||sig)$ to $\mathcal{F}^{\gamma}_{\mathsf{VDF}}$. The block creation process takes at most ε time steps. That is, for every consecutive blocks in the chain, B_i and B_{i+1}, we have that $B_{i+1}.t^u - B_i.t \leq \varepsilon$.

- `On input` (record, c), $c \in \{0,1\}^*$:
  `Query` $vo = (s, p) \leftarrow \mathcal{F}^{\gamma}_{\mathsf{VDF}}.\mathsf{output}(t^u||prev||sig)$;
  `Let` $t \leftarrow \mathsf{clock}()$, $block \leftarrow (\mathsf{len}(C)+1, prev, (t^u, sig), (s, p), t, c)$;
  `Append` $block$ to C as $B_{\mathsf{len(C)}+1}$;
  `Let` $t^u \leftarrow \mathsf{clock}()$, $prev \leftarrow \mathsf{MT.root}(C)$ and $sig \leftarrow \Sigma.\mathsf{sign}(sk, t^u||prev)$;
  `On input` $(\mathsf{start}, t^u||prev||sig)$ to $\mathcal{F}^{\gamma}_{\mathsf{VDF}}$;
  `Return` $t^u||prev||sig$.
- `On input` (stamp, c), $c \in \{0,1\}^*$:
  `Let` $i^* = \min\{i | B_i.c = c\}$;
  `If` $i^* = \bot$ `then return` $(c, 0, \bot)$;
  `Else Query` $vo = (s, p) \leftarrow \mathcal{F}^{\gamma}_{\mathsf{VDF}}.\mathsf{output}(t^u||prev||sig)$;
  `Let` $u \leftarrow \langle \mathsf{h2ts}(C, i), (s, \mathsf{clock}()||vo)\rangle$ and $a^* \leftarrow {}^{s}/_{\gamma} + \sum_{C[i^*+1:)} B_j.t - B_j.t^u$
  `Return` (c, a^*, u)

(a) $H2TS$-based prover

$H2TS$-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 $\mathcal{F}^{*}_{\mathsf{VDF}}$ and can make queries of the form $\Sigma.\mathsf{verify}(pk, msg, sig)$ for a given pk:

- `On input` $(\mathsf{verify}, c, a, u)$, $c \in \{0,1\}^*, a \in \{0,1\}^{\theta}$:
  `Parse` $S := \langle (s_i, x_i)\rangle_l \leftarrow u$ as an $H2TS$-sequence with VDF-index set I_{VDF} and time receipt index I^{-1}_{VDF} and first element $(\bot, c)$;
  `If` unable, `return` reject;
  `For each` $(s_i, t_i||x_i||sig_i) \in I^{-1}_{\mathsf{VDF}}$:
    `if` $\Sigma.\mathsf{verify}(pk, t_i||x_i, sig_i) =$ reject `return` reject;
  `If` $\neg\mathsf{h2tsverify}(S, \gamma, \varepsilon) \vee \neg\mathsf{verchain}(C)$ `then return` reject;
  `Return` accept

(b) $H2TS$-based verifier

Fig. 10: $H2TS$-based Timestamping Protocol

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 $prev$. 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 $\mathsf{MT.path}(B_i, c)$ (cf. Section 6.1),
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 $\mathsf{MT.path}(B_j, B_j.t || B_j.p)$

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 a certain age of c, as Lemma 6.4 states that this sequence must have been created sequentially except with negligible probability. Given a chain C we refer to the $H2TS$-sequence starting from $B_i.c$ and going through the entire subchain $C[i:]$ as $\mathsf{h2ts}(C, i)$.

In Figure 10 we present a timestamping protocol based on the chains presented in Definition 6.11 and $H2TS$-sequences. We show that this protocol realizes the functionality $\mathcal{F}_{\text{ts}}^{\alpha}$ presented in Figure 6. Opposed to the previous protocol, every time that a prover gets a new record query, they stop their current execution of $\mathcal{F}_{\mathsf{VDF}}^{\gamma}$ and start a new on, create a block with the input in the query and the output of $\mathcal{F}_{\mathsf{VDF}}^{\gamma}$ and query the VDF functionality again with this block as an input. When creating a timestamp, the prover finds the block with the expected block and extracts the $H2TS$-sequence from it up to the end of the chain.

Requiring the timestamp sequence to have a specific structure related to the chain is necessary to meaningfully realize $\mathcal{F}_{\text{ts}}^{\alpha}$. In this setting, timestamps can be by split and recombined, creating new timestamps without interacting with the functionality. Our functionality can deal with cases of timestamps that are merged to create a longer timestamp for a certain value, in the form of Definition 6.3. In these cases, the functionality asks the adversary whether the new proof is valid, but the adversary is not able to make the functionality accept a proof that is longer than it should be (that is, the timestamp is still checked by checkstamp).

However, our functionality is not equipped to deal with other cases that would occur naturally. For example, take the following $H2TS$-sequence:

$$\langle (\bot, x), (\bot, \mathsf{H}(x) || \mathsf{H}(y)), (\bot, t || \mathsf{H}(x_1) || sig), (s, t^* || \mathsf{VDF}(x_2, s)) \rangle.$$

It is clear that substituting the first element of the sequence with $(\bot, y)$ 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. As we have a previously stated that blocks have a canonical Merkle tree associated to them, the verifier can check whether an $H2TS$-sequence is constructed through $\mathsf{h2t}(C, i)$. 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. This is needed 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 can consider that ε also includes the time needed to generate the proof of the VDF, this allows us to extend our result to the alternative formulation of $\mathcal{F}_{\mathsf{VDF}}^{\gamma}$ as presented in Section 3.

Theorem 6.12. *Let $\mu \in \mathbb{N}^+$ be the security parameter. For any real-world PPT adversary $\mathcal{A}$ with oracle access to $\mathcal{F}_{\mathsf{VDF}}^{\gamma \cdot \alpha}$, 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 6,11) and the real world with $\mathcal{A}$, $\mathcal{P}_{H2TS}^{\gamma,\varepsilon}$ and $\mathcal{V}_{H2TS}^{\gamma,\varepsilon}$ (cf. Figure 10a,10b) is negligible in μ, λ and κ.*

Proof. Let $\mathcal{S}_{\text{H2TS}}^{\mathcal{A}}$ be as defined in Figure 11. We consider all related execution transcripts in the ideal and real world

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

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:

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

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

Given any real world PPT $\mathcal{A}$, the simulator $\mathcal{S}^{\mathcal{A}}_{\mathrm{H2TS}}$ runs the $\mathcal{A}$ in a black box manner, the honest prover $\mathcal{P}^{\gamma,\varepsilon}_{\mathrm{H2TS}}$ and the verifier $\mathcal{V}^{\gamma,\varepsilon}_{\mathrm{H2TS}}$. It maintains a list R of all stamp queries to $\mathcal{F}^{\gamma}_{\mathsf{VDF}}$. It sees all VDF queries they make, but faithfully and immediately forwards all queries and responses. The simulator acts as follows:

- `On input` (record, c), $c \in \{0,1\}^*$:
  `Send` (record, c) `to` $\mathcal{P}^{\gamma,\varepsilon}_{\mathrm{H2TS}}$;
- `On input` (stamp, c), $c \in \{0,1\}^*$:
  `/* Query proof from` $\mathcal{P}^{\gamma,\varepsilon}_{\mathrm{H2TS}}$. `*/`
  `Query` $(c, a, u) \leftarrow \mathcal{P}^{\gamma,\varepsilon}_{\mathrm{H2TS}}(\mathsf{stamp}, c)$;
  `/* Check validity of the proof */`
  `If` $\mathcal{V}^{\gamma,\varepsilon}_{\mathrm{H2TS}}.\mathsf{verify}(c, a, u) = \mathsf{accept}$ `then` $v \leftarrow 1$ `else` $v \leftarrow 0$;
  `Return` (c, a, u, v);
- `When` $\mathcal{A}$ inputs query $(\mathsf{corrupt})$ to $\mathcal{P}^{\gamma,\varepsilon}_{\mathrm{H2TS}}$:
  `Send` $(\mathsf{corrupt})$ `to` $\mathcal{F}^{\alpha}_{\mathrm{ts}}$;
- `When` $\mathcal{A}$ inputs $(\mathsf{start}, t||\mathsf{MT.root}(B_i)||\sigma)$ to $\mathcal{F}^{\gamma\cdot\alpha}_{\mathsf{VDF}}$:
  `If` $\Sigma.\mathsf{verify}(t||\mathsf{MT.root}(B_i), \sigma, pk_{\mathcal{P}}) = \mathsf{accept}$ `then send` $(\mathsf{record}, B_i.c)$ `to` $\mathcal{F}^{\alpha}_{\mathrm{ts}}$;
- `When` $\mathcal{F}^{\alpha}_{\mathrm{ts}}$ inputs $(\mathsf{cnewstamp}, c, a, u)$:
  `Input` $(\mathtt{verify}, c, a, u)$ to $\mathcal{V}^{\gamma,\varepsilon}_{\mathrm{H2TS}}$;
  `If` $\mathcal{V}^{\gamma,\varepsilon}_{\mathrm{H2TS}}$ outputs `accept return` 1 to $\mathcal{F}^{\alpha}_{\mathrm{ts}}$.
  `Else`, `return` 0 to $\mathcal{F}^{\alpha}_{\mathrm{ts}}$.

Fig. 11: Simulator

One can verify that in the ideal world all queries by $\mathcal{Z}$ and their answers by $\mathcal{P}^{\gamma,\varepsilon}_{\mathrm{H2TS}}$ (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

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

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_{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 $o_i = \mathsf{accept}$, this is only possible if $\mathcal{S}^{\mathcal{A}}_{\mathrm{H2TS}}$ has output $(c, a, u, 1)$ (as answer to a stamp query or as a stamped query). That can only happen when $\mathcal{V}^{\gamma,\varepsilon}_{\mathrm{H2TS}}(\mathsf{verify}, c, a, u) = \mathsf{accept}$ and thus that $o_r = \mathsf{accept}$, which is a contradiction. It follows that $o_i = \mathsf{reject}$ and $o_r = \mathsf{accept}$ 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:

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

Now consider the case when (c, a, u) was not legitimately constructed through the functionality in the ideal world. Then, in contrast with our previous construction, cf. Section 5, 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 checknewstamp query to $\mathcal{S}^{\mathcal{A}}_{\mathrm{H2TS}}$. 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 $\mathcal{F}^{\gamma\cdot\alpha}_{\mathsf{VDF}}$.

To continue, we bound the probabality that the adversary has constructed the proof in a non-sequential manner by Lemma 6.8 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 $o_i = \mathsf{reject}$, 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 $\mathcal{F}_{\mathsf{VDF}}^{\gamma \cdot \alpha}$ 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 $\mathcal{F}_{\mathsf{VDF}}^{\gamma \cdot \alpha}$ 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 $\mathcal{F}_{\mathsf{VDF}}^{\gamma \cdot \alpha}$ 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. □

We have shown that we can create secure timestamps through random oracle sequences. Besides the practical advantages of only having to run one VDF instance, this result also allows for an efficient way to create timestamps for a large number of records through hash-based accumulators. Our analysis naturally extends to that context, as we would still be able to extract an $H2TS$-sequence.

7 Discussion

7.1 Beyond timestamping

Up to this point, we have presented a timestamping functionality and a pair of protocols that realize the functionality using VDFs. However, building a timestamping service is not the only motivation for this work. Publicly verifiable timestamping can serve as a valuable tool in a trustless distributed setting, for example, in blockchain protocols. One of the main motivations (and inspirations) for the construction of Bitcoin was that of a public timestamp server which guarantees the immutability of the records of transactions of the cryptocurrency [Nak08]. Bitcoin allows for verifiable timestamping[9] but only as a means to an end and using numerous assumptions and a very high resource cost. What Bitcoin has done is highlight the usefulness of timestamping in this new setting.

Proofs-of-Work in blockchains serve two distinct purposes. The most well-known is Sybil prevention in a setting where there is no barrier to the creation of fake identities. PoW serves as a way to simulate the identities needed for leader election in the consensus protocol. The second purpose is the addition of a computational cost to the creation of each individual block. Due to the wastefulness of PoW-based blockchains, there has been a strong research push to find alternatives.

[8] Whether the tree branches left or right.

[9] Up to a certain point, as there exist attacks to modify the timestamps in the Bitcoin blockchain.

These alternate proposals fulfill the first role of the proofs of work but not the second. This is by design, as the computational cost is exactly what they want to avoid. Erasing the computational cost allows for attacks that exploit the ability of an adversary to costlessly simulate the execution of the blockchain protocol on their own. Blockchains then become vulnerable to attempts to break the consistency of the blockchain by rewriting it starting from a point in the far past. Such attacks are specially relevant in a proof-of-stake construction, as parties may no longer have stake in the preseent but they held enough stake at some point in the past. Timestamping through VDFs can fulfill this function of proof-of-work at considerably lower cost and in a sustainable manner.

Blockchain protocols can also benefit from cryptographic timestamping beyond preventing low-cost simulation attacks. So called "layer 2" scaling solutions for blockchains, such as [PD16], are based on timing assumptions that are represented by the addition of blocks in the native chain. Tying time to the block schedule only works within the context of one specific blockchain, as the number of blocks of one chain has no meaning in any other protocol. The lack of a protocol-independent notion of time prevents these constructions from being used to transact between two different chains, In general, inter-blockchain protocols have struggled with finding a common notion of time. Because VDFs are publicly verifiable and protocol-independent, they fulfill this purpose, allowing for cryptographic proofs of age. For this reason, they can also find a place in blockchain sharding, as a way to synchronize the multiple chains. VDFs have also been discussed as a source for randomness for (and from) blockchain protocols [LW15, PW16] but our construction is orthogonal to this use, as trustworthy randomness requires a pre-determined choice of strength for the VDF, as otherwise it is vulnerable to grinding attacks.

For our security assumptions to hold, we require honest parties to have a fast enough rate such that the adversarial advantage α is not too high. This idea seems to contrast with the premise that VDFs can be used in permissionless blockchain protocols, as access to a fast enough rate becomes a barrier for entry. However, the two ideas are not necessarily at odds. VDF-based proofs of age are transferable. More importantly, they can directly be combined with cryptographic accumulators in order to timestamp batches of records.

7.2 Immutability beacon

In a similar vein of a public randomness beacon, we propose an *immutability beacon* which consists of a party (or a set of parties) collecting hash pointers and computing VDFs over a Merkle root of the tree of these pointers. Any party wishing to have a timestamp can send a pointer of the record to be timestamped to the beacon, which will use it as part of the input to an execution of a VDF. In essence, the beacon would run the protocol from Figure 10a where $B_i.c$ is substituted by the Merkle tree of all pointers. This beacon would then regularly publish the outputs of the VDF as well as the Merkle tree, allowing for any party to construct their own timestamps using this output. Additionally, once a timestamp has been generated, the client does not need to rely on the prover anymore.

Besides being publicly verifiable, VDF timestamps are also naturally linkable through hash sequences. Such links allow users of an immutability beacon to place little trust in the beacon. If an immutability beacon stops computing VDFs or starts behaving erratically, the users can simply switch to a different beacon without this change affecting their existing timestamps. Even better,a party may make use of different beacons simultaneously, which also allows them to create the strongest possible timestamps, by choosing the appropriate VDF proofs in order to maximize the strength of a timestamp.

In practice, timestamps are not able to represent the totality of the elapsed time, as immutability beacons do not immediately create a block after receiving a stamping query. Not being able to create timestamps on-demand is the price of not doing the computation oneself. To counteract this problem, a party might want to have access to multiple immutability beacons.

References

[BBBF18] Dan Boneh, Joseph Bonneau, Benedikt Bünz, and Ben Fisch. Verifiable delay functions. In *Annual International Cryptology Conference*, pages 757–788. Springer, 2018.

[BDM93] Josh Benaloh and Michael De Mare. One-way accumulators: A decentralized alternative to digital signatures. In *Workshop on the Theory and Application of of Cryptographic Techniques*, pages 274–285. Springer, 1993.

[BHS93] Dave Bayer, Stuart Haber, and W Scott Stornetta. Improving the efficiency and reliability of digital time-stamping. *Sequences II: Methods in Communication, Security and Computer Science*, pages 329–334, 1993.

[BLS00] Ahto Buldas, Helger Lipmaa, and Berry Schoenmakers. Optimally efficient accountable time-stamping. In *International Workshop on Public Key Cryptography*, pages 293–305. Springer, 2000.

[Can18] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. Cryptology ePrint Archive, Report 2000/067, 2018. `http://eprint.iacr.org/2000/067`.

[CE12] Jeremy Clark and Aleksander Essex. Commitcoin: Carbon dating commitments with bitcoin. In *International Conference on Financial Cryptography and Data Security*, pages 390–398. Springer, 2012.

[CP18] Bram Cohen and Krzysztof Pietrzak. Simple proofs of sequential work. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 451–467. Springer, 2018.

[GKL15] Juan Garay, Aggelos Kiayias, and Nikos Leonardos. The bitcoin backbone protocol: Analysis and applications. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 281–310. Springer, 2015.

[GMG15] Bela Gipp, Norman Meuschke, and André Gernandt. Decentralized trusted timestamping using the crypto currency bitcoin. *arXiv preprint arXiv:1502.04015*, 2015.

[HS91] Stuart Haber and W. Scott Stornetta. How to time-stamp a digital document. *Journal of Cryptology*, 3(2):99–111, Jan 1991.

[KMS14] Jonathan Katz, Andrew Miller, and Elaine Shi. Pseudonymous broadcast and secure computation from cryptographic puzzles. Cryptology ePrint Archive, Report 2014/857, 2014. `http://eprint.iacr.org/2014/857`.

[LPS17] Huijia Lin, Rafael Pass, and Pratik Soni. Two-round and non-interactive concurrent non-malleable commitments from time-lock puzzles. In *Foundations of Computer Science (FOCS), 2017 IEEE 58th Annual Symposium on*, pages 576–587. IEEE, 2017.

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

[MMV11] Mohammad Mahmoody, Tal Moran, and Salil Vadhan. Time-lock puzzles in the random oracle model. In *CRYPTO*, volume 6841, pages 39–50. Springer, 2011.

[MMV13] Mohammad Mahmoody, Tal Moran, and Salil Vadhan. Publicly verifiable proofs of sequential work. In *Proceedings of the 4th conference on Innovations in Theoretical Computer Science*, pages 373–388. ACM, 2013.

[MSTS04] Tal Moran, Ronen Shaltiel, and Amnon Ta-Shma. Non-interactive timestamping in the bounded storage model. In *Annual International Cryptology Conference*, pages 460–476. Springer, 2004.

[Nak08] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2008.

[PD16] Joseph Poon and Thaddeus Dryja. The bitcoin lightning network: Scalable off-chain instant payments. 2016.

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

[PW16] Cécile Pierrot and Benjamin Wesolowski. Malleability of the blockchain's entropy. In *ArcticCrypt 2016*, 2016.

[RSW96] Ronald L Rivest, Adi Shamir, and David A Wagner. Time-lock puzzles and timed-release crypto. 1996.

[SV17] A. Stavrou and J. Voas. Verified time. *Computer*, 50(3):78–82, Mar 2017.

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