Tight Verifiable Delay Functions

Nico Döttling[1], Sanjam Garg[2], Giulio Malavolta[3], and Prashant Nalini Vasudevan[2]

[1] CISPA Helmholtz Center for Information Security
[2] University of California, Berkeley
[3] Carnegie Mellon University

Abstract. A Verifiable Delay Function (VDF) is a function that takes at least T sequential steps to evaluate and produces a unique output that can be verified efficiently, in time essentially independent of T. In this work we study *tight* VDFs, where the function can be evaluated in time not much more than the sequentiality bound T.
On the negative side, we show the impossibility of a black-box construction from random oracles of a VDF that can be evaluated in time $T + O(T^\delta)$ for any constant $\delta < 1$. On the positive side, we show that any VDF with an inefficient prover (running in time cT for some constant c) that has a natural self-composability property can be generically transformed into a VDF with a tight prover efficiency of $T + O(1)$. Our compiler introduces only a logarithmic factor overhead in the proof size and in the number of parallel threads needed by the prover. As a corollary, we obtain a simple construction of a tight VDF from any succinct non-interactive argument combined with repeated hashing. This is in contrast with prior generic constructions (Boneh et al, CRYPTO 2018) that required the existence of incremental verifiable computation, which entails stronger assumptions and complex machinery.

1 Introduction

Verifiable Delay Functions (VDFs), introduced by Boneh et al [5], is a recent cryptographic primitive which allows one to put protocol parties on halt for a set amount of time. VDFs are functions that are characterized by three properties. For a time parameter T, (i) it should be possible to compute the function in sequential time T. Furthermore, a VDF should be T-*sequential* in the sense that (ii) it should not be possible to compute such a function in (possibly parallel) time significantly less than T. Finally, (iii) the function should produce a proof π which convinces a verifier that the function output has been correctly computed. Such a proof π should be succinct, in the sense that the proof size and the verification complexity are (essentially) independent of T. These properties enable a prover to prove to the verifier that a certain amount of time has elapsed, say, by computing the function on an input provided by the verifier.

After the seminal work of Boneh et al. [5], VDFs have rapidly generated interest in the community and several follow-up constructions have recently been proposed [17, 21], and there is active work in implementing and optimizing them

for near-term practical use [8]. This is partially motivated by the large range of applications of this primitive. As an example, VDFs can turn blockchains into randomness beacons – introducing a delay in the generation of the randomness prevents malicious miners from sampling blocks adaptively to bias the outcome of the beacon. VDFs are also useful as a computational time-stamp and have further applications in the context of proofs of replications [1] and resource-efficient blockchains [9].

One of the major efficiency metrics for a VDF is the prover's computational complexity, in relation to the time parameter T. This determines the time taken to evaluate a VDF by an honest prover, and therefore the gap with respect to the best possible successful malicious machine (which we bound to take time atleast T). In the ideal case this gap is non-existent, i.e., the prover can compute the VDF in time exactly T without resorting to massive parallelization. This work asks the followin question:

When do VDFs with tight *prover complexity exist?*

Our Negative Result: Motivated by concerns about concrete efficiency, we first investigate the possibility of black-box constructions of VDFs from other cryptographic primitives. In particular, given the prevalence of strong and efficient candidates for simple cryptographic primitives like one-way functions and collision-resistant hash functions (SHA256, for instance), we would ideally like to use these as black-boxes to get similarly strong and efficient VDFs. As a negative result, we show that it is impossible to construct a T-sequential VDF where the prover runtime is close to T (with any number of processors) in a black-box manner from random oracles (and thus one-way functions or collision-resistant hash functions).

Theorem 1 (Informal). *There is no black-box construction of a T-sequential VDF from a random oracle where the prover makes at most $T + O(T^{\delta})$ rounds of queries to the oracle for some constant $\delta < 1$.*

Our Positive Result: On the other hand, we find that the natural generic non-blackbox approach to constructing a VDF can actually be made tight. All known constructions of VDF proceed by iteratively applying some function f to the given input – that is, computing $f(x)$, $f(f(x))$, and so on. A proof that this was done correctly is computed either afterwards or during this computation. We show that, assuming a *modest* parallelism of the prover, we can bootstrap any such VDF where the prover complexity may not be tight into a VDF where the prover runtime matches the sequentiality bound T. More specifically, we construct VDFs that can be computed in parallel time $T + O(1)$ using $O(log(T))$ processors and space.

Our bootstrapping theorem consists of a compiler that transforms *any* VDF with a somewhat natural *self-composability* property into a VDF with a tight prover complexity. Roughly speaking, we require that the evaluation of a VDF on time parameter T can be decomposed into two (sequential) evaluations of the same VDF on parameters T_1 and T_2 such that $T_1 + T_2 = T$. This is satisfied

by all known VDF candidates. The resulting scheme is as secure as the underlying VDF. Furthermore, the transformation is practically efficient and simple to implement.

Theorem 2 (Informal). *If there exists a self-composable VDF with a prover runtime bounded by $c \cdot T$ for some constant c, then there exists a VDF with prover runtime bounded by $T + O(1)$.*

As our transformation mentioned above is black-box, Theorems 1 and 2 together rule out black-box constructions from random oracles of a self-composable VDF where the prover makes at most cT rounds of queries for some constant c. We highlight a few other interesting corollaries of our theorem:

(1) Assuming the existence of an inherently sequential function and (not necessarily incremental) succinct non-interactive arguments [14, 16], there exists a T-sequential VDF, for any T, where the prover runs in time T using a poly-logarithmic number of processors.

This improves over generic constructions from prior work [5], which required *incremental* verifiable computation [20, 3], which is a stronger primitive.[4] Next, we turn our attention to specific number-theoretic constructions. In this context we improve the prover efficiency of the construction of Pietrzak [17], where the prover runs in time approximately $T + \sqrt{T}$.

(2) Assuming the security of Pietrzak's VDF, there exists a T-sequential VDF with prover parallel runtime exactly T using $log log(T)$ processors and space. The proof size is increased by a factor of $log log(T)$ and the verifier parallel complexity is unchanged.

Our result generalizes a prior work by Wesolowski [21] which obtains a specific construction of a tight VDF. We discuss the consequences of our bootstrapping theorem in greater detail in Section 4. Regarding the additional parallelism used by our prover, we stress that a $log(T)$ parallelism by the prover is also (implicitly) assumed by prior work: In the circuit model, even computing a simple collision-resistant hash with security parameter λ already requires at least λ ($> log(T)$ in our setting) parallel processors. Our compiler adds an extra logarithmic factor, which is well in reach of modern GPUs.

1.1 Our Techniques

In this section we give a brief overview of the main ideas behind this work. To simplify the exposition, we first discuss our bootstrapping theorem and then we give some intuition behind our impossibility result.

Self-Composability and Weak Efficiency. In the standard syntax of VDFs, both the output value y and the proof π are computed by a single function Eval.

[4] Known constructions of incremental verifiable computation require succinct arguments with knowledge extraction.

For the purposes of this work, it is instructive to think of the algorithms which compute a VDF function values y and proofs π as separate algorithms Comp and Prove: Comp takes as input the time parameter T and an input x and outputs a function value y and auxiliary information α. On the other hand Prove takes as input α and outputs a proof π. The first property that we assume for the underlying VDF is *weak-efficiency*: On time parameter T, the Comp algorithm runs in time T whereas Prove runs in time at most cT, for some constant c. While our argument is generic, for the purpose of this outline we always assume that $c = 1$, i.e., it takes the same number of steps to compute the function via $(y, \alpha) \leftarrow \mathsf{Comp}(T, x)$ and to compute the proof $\pi \leftarrow \mathsf{Prove}(\alpha)$.

We also assume that the function Comp is *self-composable*. Namely, let that $T = T_1 + T_2$ for any $T_1, T_2 > 0$. For any input x, we require that if $(y_1, \cdot) \leftarrow \mathsf{Comp}(T_1, x)$, $(y_2, \cdot) \leftarrow \mathsf{Comp}(T_2, y_1)$ and $(y, \cdot) \leftarrow \mathsf{Comp}(T, x)$, then it holds that $y_2 = y$. In other words, we require that the function $(y, \cdot) \leftarrow \mathsf{Comp}(T, x)$ can be computed in two smaller steps, where we feed the result of the first step back into the function to obtain the actual output.

We argue that these are mild requirements for VDFs, in fact all of the recent constructions fit into this paradigm and therefore can be used as input for our compiler. This includes the more structured approach of repeated squaring over hidden order groups [17, 21] and even the straightforward combination of repeated hashing and succinct arguments [5].

Bootstrapping VDFs. In favor of a simpler presentation, throughout the following informal discussion we assume that $T = 2^t$, for some integer t. The more general case is handled in the main body of the paper and follows with minor modifications. Recall that, by the self-composability property, we can split the computation $(y, \alpha) \leftarrow \mathsf{Comp}(2^t, x)$ into two separate blocks $(y_1, \alpha_1) \leftarrow \mathsf{Comp}(2^{t-1}, x)$ and $(y, \alpha_2) \leftarrow \mathsf{Comp}(2^{t-1}, y_1)$. Our main insight is that we do not need to compute a proof π for the full trace of the computation, instead we can compute two separate proofs for the corresponding subroutines. Then the final proof will consist of the concatenation of the two proofs. More specifically, we will set $\pi = (\pi_1, \pi_2)$, where $\pi_1 \leftarrow \mathsf{Prove}(\alpha_1)$ and $\pi_2 \leftarrow \mathsf{Prove}(\alpha_2)$.

This modification allows us to leverage parallelism. To evaluate the function on input x, one first computes $(y_1, \alpha_1) \leftarrow \mathsf{Comp}(2^{t-1}, x)$ in a single threaded computation. Once this step is reached, the computation forks into two parallel threads: A thread S_1 that computes $(y, \alpha_2) \leftarrow \mathsf{Comp}(2^{t-1}, y_1)$ and a thread S_2 which computes $\pi_1 \leftarrow \mathsf{Prove}(\alpha_1)$. Note that by the weak efficiency of the VDF, the runtime of the algorithm Prove is identical to that $\mathsf{Comp}(2^{t-1}, x)$, i.e., 2^{t-1} steps. It follows that S_1 and S_2 will terminate simultaneously. In other words, both y and π_1 will be available at the same time.

At this point only the computation of the proof π_2 is missing. If we were to do it naively, then we would need to add an extra 2^{t-1} steps to compute $\mathsf{Prove}(\alpha_2)$, after the computation of $(y, \alpha_2) \leftarrow \mathsf{Comp}(2^{t-1}, y_1)$ terminates. This would yield a total computation time of $T + T/2$, which is still far from optimal. However, observe that our trick has cut the original computation overhead by a factor of 2. This suggest a natural strategy to proceed: We recursively apply the same

algorithm as above on $\mathsf{Comp}(2^{t-1}, y_1)$ and further split the proof π_2 into two sub-proofs for two equal-length chunks of computation. The recursion proceeds up to the point where the computation of the function consists of a single step, and therefore there is no proof needed. Note that this happens after $t = log(T)$ iterations. Since we spawn a new thread for each level of the recursion, this also bounds the total amount of parallelism of the prover.

Our actual proof π now consists of $((y_1, \pi_1), \ldots, (y_t, \pi_t))$, i.e. the proof π consists of t components. We verify π in the canonical way, that is, setting $y_0 = x$ we compute $\mathsf{Vf}(2^{t-i}, y_{i-1}, y_i, \pi_i)$ for all $1 \leq i \leq t$ and accept if all verify and $y_t = y$.

Black-Box Impossibility. We show that there cannot be a black-box construction of a VDF from a random oracle if the overhead in computing the proof in the VDF is small. That is, if the number of sequential rounds of queries that the algorithm Eval makes to the oracle is less than $T + O(T^\delta)$ for some constant $\delta < 1$. Note that our transformation sketched above is itself black-box, and thus our result also rules out black-box constructions of self-composable VDFs with a constant-factor overhead in generating the proof.

The central idea behind the impossibility is observation that since the verification algorithm Vf makes only a small number of queries, it cannot tell whether Eval actually made all the queries it was supposed to. For simplicity, suppose that Eval makes exactly T sequential rounds of queries, and that the sequentiality of the VDF guarantees that its output cannot be computed in less than T rounds of queries. On input x, suppose $\mathsf{Eval}(x) = (y, \pi)$. Efficiency and completeness of the VDF require that the verification algorithm Vf, making only $poly(log(T))$ queries to the oracle, accepts when given (x, y, π). Whereas, soundness requires that the same Vf rejects when given (x, y', π') for any $y' \neq y$ and any π'. We show that all of these cannot happen at the same time while making only black-box use of the random oracle, if this oracle is the only source of computational hardness.

Consider an alternative evaluation algorithm $\overline{\mathsf{Eval}}$ that behaves the same as Eval except that on one of the rounds of queries that Eval makes, instead of making the queries to the oracle, it sets their answers on its own to uniformly random strings of the appropriate length. Otherwise it proceeds as Eval does, and outputs whatever Eval would output. Now, unless the algorithm Vf makes one of the queries that $\overline{\mathsf{Eval}}$ skipped, it should not be able to distiniguish between the outputs of Eval and $\overline{\mathsf{Eval}}$. As these skipped queries are only a $1/T$ fraction of the queries that $\overline{\mathsf{Eval}}$ made, Vf, which only makes $poly(log(T))$ queries, catches them with only this small probability. Thus, if Vf accepts the output (y, π) of $\mathsf{Eval}(x)$, then it should also mostly accept the output (y', π') of $\overline{\mathsf{Eval}}(x)$.

On the other hand, as $\overline{\mathsf{Eval}}$ made less than T rounds of queries to the oracle, sequentiality implies that y' should be different from y (except perhaps with negligible probability). Thus, with high probability, Vf accepts (y', π') where $y' \neq y$, contradicting soundness.

1.2 Related Work

A related concept is that of Proofs of Sequential Work (PoSW), originally introduced by Mahmoody, Moran, and Vadhan [15]. A PoSW can be seen as a relaxed version of a VDF where the output of the function is not necessarily unique: PoSW satisfy a loose notion of sequentiality where the prover is required to perform at least αT sequential steps, for some constant $\alpha \in [0, 1]$. In contrast with VDFs, PoSWs admit efficient instantiations that make only black-box calls to a random oracle [10, 11].

Time-lock puzzles [19] allow one to hide some information for a certain (polynomial) amount of time. This primitive is intimately related to sequential computation as it needs to withstand attacks from massively parallel algorithms. Time lock-puzzles have been instantiated in RSA groups [19] or assuming the existence of succinct randomized encodings and of a worst case non-parallelizable language [4]. Time-lock puzzles can be seen as VDFs with an additional encryption functionality, except that there is no requirement for an efficient verification algorithm. In this sense the two primitives are incomparable.

Two constructions of tight VDFs were proposed in the seminal work of Boneh et al. [5], one assuming the existence of incremental verifiable computation and the other from permutation polynomials (shown secure against a new assumption). The latter scheme achieves only a weak form of prover efficiency since it requires T parallel processors to be evaluated tightly. Shortly after, two number theoretic constructions have been presented independently by Pietrzak [17] and Wesolowski [21], based on squaring in groups of unknown order. In their original presentation, neither of these schemes was tight as the prover required additional $\sqrt{T}$ and $T/logT$ extra steps, respectively [6].

However, Wesolowski [21] later updated his paper with a paragraph that sketches a method to improve the prover complexity of his construction from $T+O(T/logT)$ to $T+O(1)$, using techniques similar to the ones in our compiler. Our bootstrapping theorem can be seen as a generalization of these techniques and can be applied to a broader class of constructions. Finally we mention of a new VDF instance from supersingular isogenies [12] where the validity of the function output can be checked without the need to compute any extra proof, however the public parameters of such a scheme grow linearly with the time parameter T.

2 Verifiable Delay Functions

We denote by $\lambda \in \mathbb{N}$ the security parameter. We we say that a function *negl* is negligible if it vanishes faster than any polynomial. Given a set S, we denote by $s \leftarrow_{\$} S$ the uniform sampling from S. We say that and algorithm runs in *parallel time* T with with P-many processors if it can be implemented by a PRAM machine with P parallel processors running in time T. We say that an algorithm is PPT if it can be implemented by a probabilistic machine running in time polynomial in λ.

Here we recall the definition of verifiable delay functions (VDF) from [5].

Definition 1 (Verifiable Delay Function). *A VDF* $\mathfrak{V} = (\mathsf{Setup}, \mathsf{Gen}, \mathsf{Eval}, \mathsf{Vf})$ *is defined as the following tuple of algorithms.*

- $\mathsf{Setup}(1^\lambda) \to pp$: *On input the security parameter* 1^λ*, the setup algorithm returns the public parameters* pp*. By convention, the public parameters encode an input domain* $\mathcal{X}$ *and an output domain* $\mathcal{Y}$*.*
- $\mathsf{Gen}(pp) \to x$: *On input the public parameters* pp*, the instance generation algorithm sample a random input* $x \leftarrow_\$ \mathcal{X}$*.*
- $\mathsf{Eval}(pp, x, T) \to (y, \pi)$: *On input the public parameters* pp*, an input* $x \in \mathcal{X}$*, and a time parameter* T*, the evaluation algorithm returns an output* $y \in \mathcal{Y}$ *together with a proof* π*. The evaluation algorithm may use random coins to compute* π *but not for computing* y*.*
- $\mathsf{Vf}(pp, x, y, \pi, T) \to \{0, 1\}$: *On input the public parameters* pp*, an input* $x \in \mathcal{X}$*, an output* $y \in \mathcal{Y}$*, a proof* π*, and a time parameter* T*, the verification algorithm output a bit* $\{0, 1\}$*.*

Efficiency. We require that the setup and the instance generation algorithms run in time $poly(\lambda)$, whereas the running time of the verification algorithm must be bounded by $poly(log(T), \lambda)$. For the evaluation algorithm, we require it to run in parallel time *exactly* T. We also consider less stringent notions of efficiency where its (parallel) running time is bounded by cT, for some constant c.

Completeness. The completeness property requires that correctly generated proofs always cause the verification algorithm to accept.

Definition 2 (Completeness). *A VDF* $\mathfrak{V} = (\mathsf{Setup}, \mathsf{Gen}, \mathsf{Eval}, \mathsf{Vf})$ *is complete if for all* $\lambda \in \mathbb{N}$ *and all* $T \in \mathbb{N}$ *it holds that*

$$\Pr\left[\mathsf{Vf}(pp, x, y, \pi, T) = 1 \,\middle|\, \begin{array}{l} pp \leftarrow \mathsf{Setup}(1^\lambda) \\ x \leftarrow \mathsf{Gen}(pp) \\ (y, \pi) \leftarrow \mathsf{Eval}(pp, x, T) \end{array}\right] = 1.$$

Sequentiality. We require a VDF to be sequential in the sense that no machine should be able to gain a noticeable speed-up in terms of parallel running time, when compared with the honest evaluation algorithm.

Definition 3 (Sequentiality). *A VDF* $\mathfrak{V} = (\mathsf{Setup}, \mathsf{Gen}, \mathsf{Eval}, \mathsf{Vf})$ *is sequential if for all* $\lambda \in \mathbb{N}$ *and for all pairs of PPT machines* $(\mathcal{A}_1, \mathcal{A}_2)$*, such that the parallel running time of* $\mathcal{A}_2$ *(with any polynomial amount of processors in* T*) less then* T*, there exists a negligible function negl such that*

$$\Pr\left[(y, \cdot) = \mathsf{Eval}(pp, x, T) \,\middle|\, \begin{array}{l} pp \leftarrow \mathsf{Setup}(1^\lambda) \\ (T, \tau) \leftarrow \mathcal{A}_1(pp) \\ x \leftarrow \mathsf{Gen}(pp) \\ y \leftarrow \mathcal{A}_2(pp, x, T, \tau) \end{array}\right] = negl(\lambda).$$

Soundness. For soundness we require that it is computationally hard to find two valid outputs for a single instance of the VDF. Note that here we do not constrain the running time of the adversary, except for being polynomial in λ and T.

Definition 4 (Soundness). *A VDF* $\mathfrak{V} = (\mathsf{Setup}, \mathsf{Gen}, \mathsf{Eval}, \mathsf{Vf})$ *is sound if for all* $\lambda \in \mathbb{N}$ *and for all PPT machines* $\mathcal{A}$ *there exists a negligible function negl such that*

$$\Pr\left[\mathsf{Vf}(pp, x, y, \pi, T) = 1 \text{ and } (y, \cdot) \neq \mathsf{Eval}(pp, x, T) \middle| \begin{array}{l} pp \leftarrow \mathsf{Setup}(1^\lambda) \\ (T, x, y, \pi) \leftarrow \mathcal{A}_1(pp) \end{array}\right] = negl(\lambda).$$

3 A Bootstrapping Theorem for VDFs

In this section we propose a compiler that takes as input any weakly efficient VDF (that satisfies some natural composability properties) and turns it into a fully-fledged efficient scheme. We first characterize the exact requirements of the underlying VDF, then we describe our construction and we show that it preserves all of the properties of the underlying scheme.

3.1 Building Block

We require that the underlying VDF can be composed with itself (possibly using different time parameters) arbitrarily many times, without altering the function output, i.e., $\mathsf{Eval}(pp, \cdot, T_1) \circ \mathsf{Eval}(pp, \cdot, T_2) = \mathsf{Eval}(pp, \cdot, T_1 + T_2)$. More concretely, we assume the existence of a VDF that satisfies the following.

Definition 5 (Self-Composability). *A VDF* $\mathfrak{V} = (\mathsf{Setup}, \mathsf{Gen}, \mathsf{Eval}, \mathsf{Vf})$ *is self-composable if, for all pp in the support of* $\mathsf{Setup}(1^\lambda)$*, for all* $x \in \mathcal{X}$*, all* (T_1, T_2) *bounded by a sub-exponential function in* λ*, we have that* $\mathsf{Eval}(pp, x, T_1 + T_2) = \mathsf{Eval}(pp, y, T_2)$*, where* $(y, \cdot) = \mathsf{Eval}(pp, x, T_2)$*.*

Note that this also implies that the domain and the range of the function are identical, i.e., $\mathcal{X} = \mathcal{Y}$. We stress that this is property is satisfied by all known candidate VDF constructions [5, 17, 21]. To characterize the second requirement, it is going to be useful to refine the syntax of our primitive. We assume that the evaluation algorithm $\mathsf{Eval}(pp, x, T)$ is split in the following subroutines:

- $\mathsf{Eval}(pp, x, T)$: On input the public parameters pp, and input $x \in \mathcal{X}$, and a time parameter T, execute the subroutine $(y, \alpha) \leftarrow \mathsf{Comp}(pp, x, T)$. Then compute $\pi \leftarrow \mathsf{Prove}(\alpha)$ and return (y, π).

This captures the compute-and-prove approach, where the prover evaluates some inherently sequential function and then computes a short proof of correctness, potentially using some information from the intermediate steps of the computation (α). Note that this refinement is done without loss of generality since one can recover the original syntax by encoding the proof π in the advice α and set the prove algorithm to be the identity function. We are now in the position of stating the efficiency requirements of the input VDF.

Definition 6 (Weak Efficiency). *A VDF* $\mathfrak{V} = (\mathsf{Setup}, \mathsf{Gen}, \mathsf{Eval}, \mathsf{Vf})$ *is weakly efficient if there exists a function* $\Psi : \mathbb{N} \to \mathbb{N}$ *and a non-negative constant c such that for all* $T \in \mathbb{N}$ *it holds that* $0 \leq \Psi(T) \leq cT$ *and* Prove *runs in parallel time* $\Psi(T)$*, where* T *is the parallel running time of* $\mathsf{Eval}(\cdot, \cdot, T)$*.*

Note that the total running time of the evaluation algorithm is bounded by $(c+1)T$, for some constant c. This condition is again met by all known VDF instances [5, 17, 21], since they are all based on the compute-and-prove paradigm: Given a long sequential computation, the corresponding proof π can be computed in parallel time at most linear in T by using essentially any verifiable computation scheme.

3.2 Scheme Description

Let $\mathfrak{V} = (\mathsf{Setup}, \mathsf{Gen}, \mathsf{Eval}, \mathsf{Vf})$ be a weakly efficient and self-composable VDF and let Ψ be the corresponding function such that, on input T, the running time of the subroutine Prove is bounded by $\Psi(T)$. Our construction $\overline{\mathfrak{V}} = (\overline{\mathsf{Setup}}, \overline{\mathsf{Gen}}, \overline{\mathsf{Eval}}, \overline{\mathsf{Vf}})$ is shown below.

$\overline{\mathsf{Setup}}(1^\lambda)$: Return $\mathsf{Setup}(1^\lambda)$.

$\overline{\mathsf{Gen}}(pp)$: Return $\mathsf{Gen}(pp)$.

$\overline{\mathsf{Eval}}(pp, x, T)$: Set S to be the smallest non-negative integer such that $S + \Psi(S) \geq T$.

(1) If $S \leq 1$:
  (a) Compute $(y, \alpha) \leftarrow \mathsf{Comp}(pp, x, T)$.
  (b) Return $(y, \emptyset)$.

(2) Else:
  (a) Compute $(y, \alpha) \leftarrow \mathsf{Comp}(pp, x, S)$.
  (b) Spawn a parallel thread to compute $\pi \leftarrow \mathsf{Prove}(\alpha)$.
  (c) Compute $(\tilde{y}, L) \leftarrow \overline{\mathsf{Eval}}(pp, y, T - S)$ in the main thread.
  (d) Return $(\tilde{y}, (y, \pi) \cup L)$.

$\overline{\mathsf{Vf}}(pp, x, y, \pi, T)$:

(1) Parse π as $((y_1, \pi_1), \ldots, (y_n, \pi_n))$.

(2) Set $S_0 = T$ and $y_0 = x$.

(3) For all $1 \leq i \leq n$:
  (a) Define S_i to be the smallest integer such that $S_i + \Psi(S_i) \geq S_0 - \sum_{j=1}^{i-1} S_j$.
  (b) Compute in parallel $b_i = \mathsf{Vf}(pp, y_{i-1}, y_i, S_i)$.

(4) Return $(b_1 \wedge \cdots \wedge b_n \wedge (y, \cdot) = \mathsf{Eval}(pp, y_n, T - \sum_{j=1}^{n} S_j))$.

The setup and the instance generation algorithms are unchanged. The new evaluation algorithm is defined recursively: On input some time T, the algorithm defines S to be the smallest non-negative integer such that $S + \Psi(S) \geq T$. Recall that the correctness of the evaluation of the Comp algorithm on time parameter S can be proven in time $\Psi(S)$, by the weak efficiency condition of the VDF. Thus,

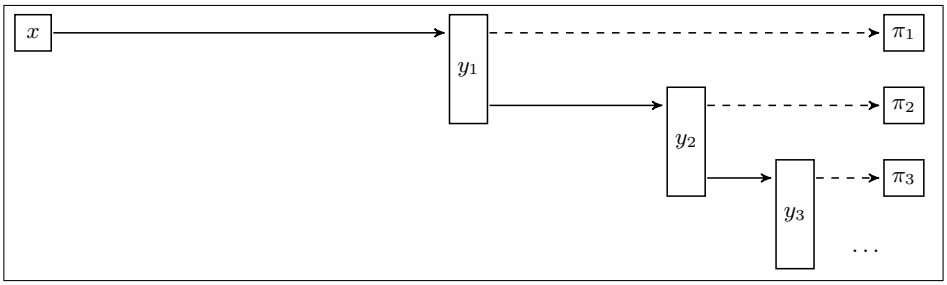

Fig. 1: Example of the execution trace of the evaluation algorithm for Ψ being the identity function. Solid lines denote the computation of the iterated function (Comp) and dashed lines denote the computation of the corresponding proof (Prove).

S approximates from above the maximum amount of computation that can be performed and proven within time T. The algorithm then branches, depending on the value of S:

(1) $S \leq 1$: In this case the algorithm simply computes $\mathsf{Comp}(pp, x, T)$ and outputs the result of the computation, without computing a proof. Observe that since $S = 1$ it holds that $T \leq (1 + c)$, where c is the constant such that $\Psi(S) \leq cS$. Thus the algorithm runs for at most $c+1$ steps. This corresponds to the last step of the recursion.

(2) $S > 1$: In this case the algorithm computes the underlying VDF on time parameter S and outputs the resulting $(y, \alpha) \leftarrow \mathsf{Comp}(pp, x, S)$. At this point the algorithm branches in two parallel threads:

   (a) The first thread computes the proof $\pi \leftarrow \mathsf{Prove}(\alpha)$.

   (b) The second thread calls the evaluation algorithm recursively on input $(pp, y, T - S)$, which returns an output $\tilde{y}$ and a list L.

   The algorithm returns the function output $\tilde{y}$ and the list L augmented (from the left) with the intermediate pair (y, π), i.e., $(y, \pi) \cup L$.

The output of the computation consists of n pairs $(y_1, \pi_1), \ldots, (y_n, \pi_n)$, where n is the depth of the recursion, and a function output $\tilde{y}$. Each output-proof pair can be verified independently and the correctness of the function output $\tilde{y}$ can be checked by recomputing at most c steps of the VDF on input y_n.

Note that the only parameter that affects the efficiency of the prover and the verifier and the size of the proof is the depth of the recursion n. Intuitively, each step of the recursion slices off a chunk of the computation without affecting the total runtime of the algorithm, until only a constant number of steps is left. If S and $\Psi(S)$ are not too far apart, then n can be shown to be bounded by a poly-logarithmic factor in T. We give a pictorial representation of the steps of the computation in Figure 1.

3.3 Analysis

The completeness of our scheme follows from the self-composability of the underlying VDF. In the following we analyze the efficiency of our construction and we show that the properties of the underlying VDF are preserved under self-composition.

Efficiency. Recall that S is always a non-negative integers and therefore each step of the recursion is executed on input some integer $\leq T$. Thus we can bound the size of each proof π_i by the size of the proof of the underlying VDF on time parameter T. It follows that the proof size of our scheme is at most a factor n larger, where n is the depth of the recursion. In the following we bound the depth of the recursion, thus establishing also a bound also on the proof size. We begin by stating and proving the following instrumental lemma.

Lemma 1. *Let $\Psi : \mathbb{N} \to \mathbb{N}$ be a function such that there exists a constant c such that for all $S \in \mathbb{N}$ it holds that $\Psi(S) \leq cS$. Fix an $S_0 \in \mathbb{N}$ and define S_i to be the smallest non-negative integer such that $S_i + \Psi(S_i) \geq S_0 - \sum_{j=1}^{i-1} S_j$. Then for all $S_0 \in \mathbb{N}$ and for all $i \geq 0$ it holds that $S_0 - \sum_{j=1}^{i} S_j \leq S_0 \left(\frac{c}{c+1}\right)^i$.*

Proof. We prove the claimed bound by induction over i. For the base case $i = 0$ we have $S_0 \leq S_0$, which is trivial. For the induction step, recall that S_i is defined to be the smallest integer such that $S_i + \Psi(S_i) \geq S_0 - \sum_{j=1}^{i-1} S_j$. Since $\Psi(S_i) \leq cS_i$, for some non-negative c, we have that

$$\begin{aligned} S_i + cS_i &\geq S_0 - \sum_{j=1}^{i-1} S_j \\ S_i(c+1) &\geq S_0 - \sum_{j=1}^{i-1} S_j \\ S_i &\geq \frac{S_0 - \sum_{j=1}^{i-1} S_j}{c+1}. \end{aligned}$$

By induction hypothesis, it follows that

$$\begin{aligned} S_0 - \sum_{j=1}^{i} S_j &= S_0 - \sum_{j=1}^{i-1} S_j - S_i \\ &\leq S_0 - \sum_{j=1}^{i-1} S_j - \frac{S_0 - \sum_{j=1}^{i-1} S_j}{c+1} \\ &= \frac{(S_0 - \sum_{j=1}^{i-1} S_j)c}{c+1} \\ &\leq S_0 \left(\frac{c}{c+1}\right)^{i-1} \left(\frac{c}{c+1}\right) \\ &= S_0 \left(\frac{c}{c+1}\right)^{i}. \end{aligned}$$

The bound on the depth of the recursion is obtained by observing that the i-th copy of the evaluation algorithm is called on time parameter exactly $T - \sum_{j=1}^{i} S_j$, where S_j is defined as above. Note that if $T - \sum_{j=1}^{i} S_j \leq 1$, then the recursion always stops, since the condition $1 + \Psi(1) \geq 1$ is satisfied for all non-negative Ψ. If we set $S_0 = T$ and apply Lemma 1 we obtain the following relation

$$T - \sum_{j=1}^{i} S_j \leq T \left(\frac{c}{c+1} \right)^i \leq 1.$$

Solving for i, we have that $log_{\left(\frac{c+1}{c}\right)}(T)$ iterations always suffice for the algorithm to terminate. This also implies a bound on the number of processors needed to evaluate the function. We now establish a bound on the parallel runtime of the evaluation algorithm.

Lemma 2. *Let $\mathfrak{V}$ be a weakly efficient VDF. Then, for all pp in the support of $\overline{\mathsf{Setup}}(1^\lambda)$, for all $x \in \mathcal{X}$, and for all $T \in \mathbb{N}$, the algorithm $\overline{\mathsf{Eval}}(pp, x, T)$ terminates in at most $T + c$ steps.*

Proof. We first consider the main thread of the execution. Set $S_0 = T$ and define S_i to be the smallest integer such that $S_i + \Psi(S_i) \geq S_0 - \sum_{j=1}^{i-1} S_j$. Observe that the main thread consists of n applications of the algorithm Comp on time parameter S_i, where n is the depth of the recursion, and one application on input $S_0 - \sum_{j=1}^{n} S_j$. Thus, by the weak efficiency of $\mathfrak{V}$, the total running time of the main thread is exactly

$$\sum_{j=1}^{n} S_j + S_0 - \sum_{j=1}^{n} S_j = S_0 = T.$$

To bound the runtime of the parallel threads we bound the difference with respect to the amount of steps needed for the main thread to terminate, starting from the moment the execution forks. We show a bound assuming $\Psi(S) = cS$, which implies a bound for the general case since the proving algorithm can only get faster. Consider the i-th recursive instance of the algorithm: After computing $(\alpha_i, y_i) \leftarrow \mathsf{Comp}(pp, x_i, S_i)$, the main thread proceeds by calling $\overline{\mathsf{Eval}}$ on input $T - S_i$ and spawns a new thread to compute a proof on input α_i. As discussed above, we know that the main thread will terminate within $T - S_i$ steps, thus all we need to do is bounding the amount of extra steps needed for the computation of the proof. Note that we have

$$(S_i - 1) + \Psi(S_i - 1) < T$$

since we assumed that S_i was the smallest integer that satisfies $S_i + \Psi(S_i) \geq T$. Substituting,

$$\begin{aligned}(S_i - 1) + \Psi(S_i - 1) &< T \\ \Psi(S_i - 1) &< T - S_i + 1 \\ c(S_i - 1) &< T - S_i + 1 \\ cS_i - c &< T - S_i + 1 \\ \Psi(S_i) &< T - S_i + c + 1\end{aligned}$$

where $\Psi(S_i)$ is exactly the number of steps needed to compute π_i. This holds for all recursive instances of the algorithms and concludes our proof.

We remark that the extra c steps needed for our algorithm to terminate are due to the rounding that happens when S does not divide T. For the common case where T is a power of 2 and Ψ is the identity function, then $T = 2S$ in all of the recursive calls of the algorithm and the process terminates in exactly T steps. The verifier complexity is bounded by that of n parallel calls to the verifier of the underlying VDF on input some time parameter $\leq T$, where n is poly-logarithmic in T (see discussion above), plus an extra parallel instance that recomputes at most $c + 1$ steps of the Comp algorithm.

Sequentiality. In the following we show that our transformation preserves the sequentiality of the underlying VDF.

Theorem 3 (Sequentiality). *Let $\mathfrak{V}$ be a self-composable sequential VDF, then $\overline{\mathfrak{V}}$ is sequential.*

Proof. Let $\mathcal{A}$ be and adversary that, on input a random instance x, finds the corresponding image y in time less than T. By definition of our evaluation algorithm, y is computed as

$$(y, \cdot) \leftarrow \mathsf{Comp}\left(pp, y_n, T - \sum_{j=1}^{n} S_j\right),$$

where $(y_i, \cdot) \leftarrow \mathsf{Comp}(pp, y_{i-1}, S_i)$, for all $1 \leq i \leq n$, setting $y_0 = x$. Invoking the self-composability of the underlying VDF, we have that

$$(y, \cdot) = \mathsf{Comp}\,(pp, x, T) = \mathsf{Eval}\,(pp, x, T)$$

twhich implies that y is the correct image of the underlying VDF for the same time paramter T. Thus the existence of $\mathcal{A}$ contradicts the sequentiality of $\mathfrak{V}$.

Soundness. The following theorem establishes the soundness of our scheme. Note that the reduction is tight, which means that our construction is exactly as secure as the input VDF.

Theorem 4. *Let $\mathfrak{V}$ be a self-composable and sound VDF, then $\overline{\mathfrak{V}}$ is sound.*

Proof. Let $\mathcal{A}$ be an adversary that, on input the honestly generated public parameters pp, outputs some tuple (T, x, y, π) such that π is a valid proof, but $(y, \cdot) \neq \overline{\mathsf{Eval}}(pp, x, T)$. Let $\pi = ((y_1, \pi_1), \ldots, (y_n, \pi_n))$ and set $y_0 = x$. Then we claim that one of the following conditions must be satisfied:

(1) There exists some $i \in \{1, \ldots, n\}$ such that $(y_i, \cdot) \neq \mathsf{Comp}(pp, y_{i-1}, S_i)$, where S_i is defined as in the verification algorithm, or
(2) $(y, \cdot) \neq \mathsf{Comp}(pp, y_n, T - \sum_{j=1}^{n} S_j)$.

If none of the above conditions is met, then we have that

$$(y, \cdot) = \mathsf{Comp}\,(pp, x, T) = \overline{\mathsf{Eval}}\,(pp, x, T)$$

by the self-composability of $\mathfrak{V}$, which contradicts the initial hypothesis. It follows that a successful attack implies at least one of the above conditions. However, (1) implies that we can extract a tuple $(y_{i-1}, y_i, \pi_i, S_i)$, such that π_i is a valid proof but $(y_i, \cdot) \neq \mathsf{Eval}(pp, y_{i-1}, S_i)$, which contradicts the soundness of $\mathfrak{V}$. On the other hand, if (2) happens then the verifier always rejects. It follows that the existence of $\mathcal{A}$ implies that the underlying VDF is not sound.

4 Instantiations

In the following we survey the existing candidate VDF schemes and we discuss the implications of our results.

4.1 Compute-and-Prove VDF

The original work of Boneh et al. [5] discusses an instantiation for VDF based on any (conjectured) inherently sequential function and a succinct non-interactive argument system (SNARG) [14, 16]. The prover simply evaluates the function on a randomly chosen input and computes a short proof that the computation is done correctly. However, such an approach is dismissed since the time to compute a SNARG is typically much longer than that needed for the corresponding relation. Therefore, to achieve meaningful sequentiality guarantees, the prover needs to resort to massive parallelization which requires a number of processors linear in the time parameter T.

For this reason they turned their attention to incremental verifiable computation schemes [20]. Such a primitive derives from the recursive composition of SNARGs and allow one to compute the proof incrementally as the computation proceeds. However, this feature comes at a cost: The number of recursions introduces an exponential factor in the running time of the extractor and therefore the schemes can be shown sound only for a constant amount of iterations. Other constructions [3] circumvent this issue by constructing computation trees of constant depth, however the overhead given by the recursive application of a SNARG is typically the bottleneck of the computation.

Our approach can be seen as a lightweight composition theorem for VDFs and rehabilitates the compute-and-prove paradigm using standard SNARGs in

conjunction with iterated sequential functions: Most of the existing SNARG schemes can be computed in time quasi-linear in T [2] and can be parallelized to meet our weak efficiency requirements using a poly-logarithmic amount of processors (in the time parameter T). Our compiler shows that the combination of SNARGs and iterated sequential functions already gives a tightly sequential VDF, for any value of T.

4.2 Wesolowski's VDF

A recent work by Wesolowski [21] builds an efficient VDF exploiting the conjectured sequentiality of repeated squaring in groups of unknown order, such as RSA groups [18] or class groups of imaginary quadratic order [7]. Loosely speaking, given a random instance $x \in \mathbb{G}$ and a time parameter T, the sequential function is defined as $f(x) = x^{2^T}$. Wesolowski proposes a succinct argument for the corresponding language

$$\mathcal{L} = \left\{ (\mathbb{G}, x, y, T) : y = x^{2^T} \right\}$$

where the verification is much faster than recomputing the function from scratch. The argument goes as follows:

(1) The verifier samples a random prime p from the set of the first 2^λ primes.
(2) The prover computes $q, r \in \mathbb{Z}$ such that $2^T = pq + r$ and outputs $\pi = x^q$ as the proof.
(3) The proof π can be verified by checking that $\pi^p x^r = y$, where r is computed as $2^T \mod p$.

The argument can be made non-interactive using the Fiat-Shamir transformation [13]. Note that the value of q cannot be computed by the prover explicitly since the order of the group is unknown, however it can be computed in the exponent of x in time close to T.

Wesolowski's proof consists of a single group element and the verifier workload is essentially that of two exponentiations in $\mathbb{G}$. The main shortcoming of the scheme is that the time to compute a valid proof is proportional to the time to compute the function. However, Wesolowski briefly explains how to reduce this overhead to a constant factor using parallel processors. The modification sketched in his paper is essentially an ad-hoc version of our compiler.

4.3 Pietrzak's VDF

Recently, Pietrzak [17] also showed an efficient succinct argument for the same language $\mathcal{L}$, taking a slightly different route. In the following we briefly recall the backbone of the argument:

(1) If $T = 1$, the verifier simply checks that $x^2 = y$.
(2) Else the prover computes $z = x^{2^{T/2}}$ and sends it to the verifier.

(3) The verifier samples some $r \in \{1, \ldots, 2^\lambda\}$.
(4) The prover and the verifier recurse on input $(\mathbb{G}, x^r z, z^r y, T/2)$.

The resulting argument is less efficient than Wesolowski's approach in terms of proof size and verifier complexity by a factor of $log(T)$. However Pietrzak's argument can be computed in time approximately $\sqrt{T}$ using roughly $\sqrt{T}$ memory by storing some intermediate values of the function evaluation.

It is clear that such a VDF fulfills the conditions to apply our compiler and allows us to truncate the additional $\sqrt{T}$ factor from the proof computation. Due to the increased proof size, it might appear that the resulting scheme is strictly worse than that obtained by combining our compiler with Wesolowski's approach. However the significantly shorter proving time allows us to give a sharper bound on the number of recursion of our algorithm: In each iteration the new time parameter is computed as $1/2\sqrt{4T+1}-1$ and therefore approximately $loglog(T)$ iterations suffice to hit the bottom of the recursion. As a consequence, Pietrzak's argument needs less parallelism to achieve optimal prover runtime. We also point out that Pietrzak's argument rests on a weaker assumption, as discussed in [6].

5 Black-Box Impossibility

In this section, we show that it is not possible to have a (tight) VDF whose only source of computational hardness is a random oracle. This implies that VDFs cannot be constructed by making just black-box use of several cryptographic primitives such as One-Way Functions and Collision-Resistant Hash Functions.

In this setting, we give all the algorithms in a VDF construction (that is, Setup, Gen, Eval, and Vf), as well as any adversaries, access to a random oracle that we denote by H. Our measure of the complexity of an algorithm will be the number of queries that it makes to H, and sequentiality is also measured by the number of rounds of queries made to H adaptively.

Theorem 5. *There is no black-box construction of a VDF* $(\mathsf{Setup}, \mathsf{Gen}, \mathsf{Eval}, \mathsf{Vf})$ *from a random oracle where* $\mathsf{Eval}(\cdot, \cdot, T)$ *makes at most* $T + O(T^\delta)$ *rounds of queries for some* $\delta < 1$*, and is also* T*-sequential.*

Noting that our transformation from Section 3 (of any self-composable weakly efficient VDF into a VDF with constant proof overhead) is black-box, we can extend this impossibility as follows.

Corollary 1. *There is no black-box construction of a self-composable VDF* $(\mathsf{Setup}, \mathsf{Gen}, \mathsf{Eval}, \mathsf{Vf})$ *from a random oracle where* $\mathsf{Eval}(\cdot, \cdot, T)$ *makes at most* cT *rounds of queries (for some constant* c*) and is also* T*-sequential.*

Our approach is as follows. Imagine replacing the answers to all but $(T-1)$ rounds of queries that Eval makes to H with something completely random (independent of H). Since we only replaced a small fraction of the queries made, the output of Eval in this case should look the same to Vf, which makes very

few queries to H. On the other hand, if this replacement did not change the value of y output by Eval, then it means that we could have computed Eval by answering these queries ourselves, and thus making only $(T-1)$ rounds of queries to H, contradicting sequentiality. Thus, if Eval is indeed T-sequential, then this replacement allows us to break soundness.

Proof (Theorem 5). Fix any alleged VDF $(\mathsf{Setup}, \mathsf{Gen}, \mathsf{Eval}, \mathsf{Vf})$ with access to a random oracle H. Suppose, for some $\delta < 1$, there is a function $p(\lambda, T)$ that is $O(poly(\lambda) \cdot T^{\delta})$ such that, for any λ, T, and $pp \leftarrow \mathsf{Gen}^H(1^\lambda)$, the evaluation algorithm $\mathsf{Eval}(pp, \cdot, T)$ makes at most $T + p(\lambda, T)$ rounds of queries to H. Without loss of generality (since the algorithms are not bounded in the memory they use), we assume that an algorithm does not make the same query twice to H.

We construct an adversarial evaluator $\overline{\mathsf{Eval}}$ that works as follows on input (pp, x, T):

1. Pick a uniformly random set $I \subseteq [T + p(\lambda, T)]$ of size $(p(\lambda, T) + 1)$.
2. Run $\mathsf{Eval}(pp, x, T)$ as is whenever it is not making oracle queries.
3. When Eval makes the i^{th} round of oracle queries $(q_1, \ldots, q_m)$,
   - if $i \notin I$, respond with $(H(q_1), \ldots, H(q_m))$.
   - if $i \in I$, respond with $(a_1, \ldots, a_m)$, where the a_i's are uniformly random strings of the appropriate length.
4. Output the (y, π) that Eval produces at the end.

The following claim states that if Vf makes only a small number of queries (which it has to for efficiency), it cannot distinguish between the outputs of Eval and $\overline{\mathsf{Eval}}$.

Claim (Indistinguishability). Suppose pp and x are generated from $\mathsf{Setup}^H(1^\lambda)$ and $\mathsf{Gen}^H(pp)$, respectively. Let $(y, \pi) \leftarrow \mathsf{Eval}^H(pp, x, T)$, and $(\overline{y}, \overline{\pi}) \leftarrow \overline{\mathsf{Eval}}^H(pp, x, T)$. If the algorithm $\mathsf{Vf}^H(pp, \cdot, \cdot, \cdot, T)$ makes at most $T/8p(\lambda, T)$ queries to H, then, for all $\lambda \in \mathbb{N}$ and all $T \in \mathbb{N}$, it holds that:

$$\left| \Pr\left[\mathsf{Vf}^H(pp, x, y, \pi, T) = 1\right] - \Pr\left[\mathsf{Vf}^H(pp, x, \overline{y}, \overline{\pi}, T) = 1\right] \right| \leq \frac{1}{4}$$

We defer the proof of the above to later in this section. The next claim states that, if the given VDF is sequential, then the output y as computed by $\overline{\mathsf{Eval}}$ has to differ with high probability from that computed by Eval. This follows immediately from the observation that $\overline{\mathsf{Eval}}$ makes at most $(T-1)$ rounds of queries to H, and so if outputs that same y as Eval with non-negligible probability, this would immediately contradict T-sequentiality.

Claim (Sensitivity). Suppose the given VDF is T-sequential, and that pp and x are generated from $\mathsf{Setup}^H(1^\lambda)$ and $\mathsf{Gen}^H(pp)$, respectively. Let $(y, \pi) \leftarrow \mathsf{Eval}^H(pp, x, T)$, and $(\overline{y}, \overline{\pi}) \leftarrow \overline{\mathsf{Eval}}^H(pp, x, T)$. Then, there exists a negligible function $negl$ such that for all $\lambda \in \mathbb{N}$ and all $T \in \mathbb{N}$ it holds that:

$$\Pr[y = \overline{y}] \leq negl(\lambda)$$

We now construct an adversary $\mathcal{A}$ that breaks the soundness of the supposed VDF. $\mathcal{A}$, on input the parameters pp, works as follows:

1. Generate $x \leftarrow \mathsf{Gen}^H(pp)$, and set $T = 2^\lambda$.
2. Compute $(\overline{y}, \overline{\pi}) \leftarrow \overline{\mathsf{Eval}}^H(pp, x, T)$.
3. Output $(T, x, \overline{y}, \overline{\pi})$.

Our argument now is based on the following three points:

- By the efficiency of the VDF, Vf makes much fewer than $T/8p(\lambda, T)$ $(= 2^{\Omega(\lambda)})$ queries.
- So, by the correctness of the VDF and Claim 5, the probability that $\mathsf{Vf}(pp, x, \overline{y}, \overline{\pi}, T)$ does not output 1 is at most $1/4$.
- By Claim 5, the probability that $\overline{y}$ agrees with the output of Eval is at most $negl(\lambda)$.

Together, by the union bound, we have:

$$\Pr\left[\mathsf{Vf}(pp, x, \overline{y}, \overline{\pi}, T) = 1 \text{ and } (y, \cdot) \neq \mathsf{Eval}(pp, x, T)\right] \geq 1 - \left(\frac{1}{4} + negl(\lambda)\right)$$

Noting that $\mathcal{A}$ runs in nearly the same time as Eval, this contradicts the claim that we started with a VDF that is both sequential and sound, proving the theorem.

Proof (Indistinguishability Claim). Recall that $\overline{\mathsf{Eval}}$ generates $(\overline{y}, \overline{\pi})$ just by altering the oracle that Eval has access to. Denoting this altered oracle by $\overline{H}$, note that $\overline{H}$ is also a random oracle, and that if Vf also had access to $\overline{H}$ instead of H, then the behavior of the whole system would not change. That is,

$$\Pr\left[\mathsf{Vf}^{\overline{H}}(pp, x, \overline{y}, \overline{\pi}, T) = 1\right] = \Pr\left[\mathsf{Vf}^{H}(pp, x, y, \pi, T) = 1\right] \tag{1}$$

Suppose, when given input (pp, x, y, π, T), the algorithm Vf^H makes N queries $q_1, \ldots, q_N$ to the oracle. Its behavior when given access to $\overline{H}$ instead of H is different only if the two oracles disagree on at least one of these queries. For any query q_i, the algorithm $\overline{\mathsf{Eval}}$ alters the oracle at this query if it happens to be made by Eval in a round contained in I. This happens with probability less than $(p(\lambda, T) + 1)/(T + p(\lambda, T))$. This implies that, for any $i \in [N]$,

$$\Pr\left[H(q_i) \neq \overline{H}(q_i)\right] \leq \frac{p(\lambda, T) + 1}{T + p(\lambda, T)} \leq \frac{2p(\lambda, T)}{T}$$

Thus, by the union bound, the probability that $\mathsf{Vf}^{\overline{H}}$ behaves differently from Vf^H on any input is at most $2Np(\lambda, T)/T$. If $N \leq T/8p(\lambda, T)$, together with (1), this implies that:

$$\begin{aligned}&\left|\Pr\left[\mathsf{Vf}^{H}(pp, x, y, \pi, T) = 1\right] - \Pr\left[\mathsf{Vf}^{H}(pp, x, \overline{y}, \overline{\pi}, T) = 1\right]\right| \\ &\quad = \left|\Pr\left[\mathsf{Vf}^{\overline{H}}(pp, x, \overline{y}, \overline{\pi}, T) = 1\right] - \Pr\left[\mathsf{Vf}^{H}(pp, x, \overline{y}, \overline{\pi}, T) = 1\right]\right| \leq \frac{1}{4}.\end{aligned}$$

Acknowledgements

S. Garg and P. N. Vasudevan are supported in part from DARPA/ARL SAFEWARE Award W911NF15C0210, AFOSR Award FA9550-15-1-0274, AFOSR YIP Award, DARPA and SPAWAR under contract N66001-15-C-4065, a Hellman Award and research grants by the Okawa Foundation, Visa Inc., and Center for LongTerm Cybersecurity (CLTC, UC Berkeley).

G. Malavolta is supported in part by a gift from Ripple, a gift from DoS Networks, a grant from Northrop Grumman, a Cylab seed funding award, and a JP Morgan Faculty Fellowship.

References

1. Frederik Armknecht, Ludovic Barman, Jens-Matthias Bohli, and Ghassan O Karame. Mirror: Enabling proofs of data replication and retrievability in the cloud. In *25th {USENIX} Security Symposium ({USENIX} Security 16)*, pages 1051–1068, 2016.
2. Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. Scalable zero knowledge via cycles of elliptic curves. In Juan A. Garay and Rosario Gennaro, editors, *Advances in Cryptology – CRYPTO 2014, Part II*, volume 8617 of *Lecture Notes in Computer Science*, pages 276–294, Santa Barbara, CA, USA, August 17–21, 2014. Springer, Heidelberg, Germany.
3. Nir Bitansky, Ran Canetti, Alessandro Chiesa, and Eran Tromer. Recursive composition and bootstrapping for SNARKS and proof-carrying data. In Dan Boneh, Tim Roughgarden, and Joan Feigenbaum, editors, *45th Annual ACM Symposium on Theory of Computing*, pages 111–120, Palo Alto, CA, USA, June 1–4, 2013. ACM Press.
4. Nir Bitansky, Shafi Goldwasser, Abhishek Jain, Omer Paneth, Vinod Vaikuntanathan, and Brent Waters. Time-lock puzzles from randomized encodings. In Madhu Sudan, editor, *ITCS 2016: 7th Conference on Innovations in Theoretical Computer Science*, pages 345–356, Cambridge, MA, USA, January 14–16, 2016. Association for Computing Machinery.
5. Dan Boneh, Joseph Bonneau, Benedikt Bünz, and Ben Fisch. Verifiable delay functions. In Hovav Shacham and Alexandra Boldyreva, editors, *Advances in Cryptology – CRYPTO 2018, Part I*, volume 10991 of *Lecture Notes in Computer Science*, pages 757–788, Santa Barbara, CA, USA, August 19–23, 2018. Springer, Heidelberg, Germany.
6. Dan Boneh, Benedikt Bünz, and Ben Fisch. A survey of two verifiable delay functions. Cryptology ePrint Archive, Report 2018/712, 2018. `https://eprint.iacr.org/2018/712`.
7. Johannes Buchmann and Hugh C. Williams. A key-exchange system based on imaginary quadratic fields. *Journal of Cryptology*, 1(2):107–118, June 1988.
8. Chia network second vdf competition. `https://www.chia.net/2019/04/04/chia-network-announces-second-vdf-competition-with-in-total-prize-money.en.html`. Accessed: 2019-04-22.
9. Bram Cohen. Proofs of space and time. blockchain protocol analysis and security engineering. `https://cyber.stanford.edu/sites/default/files/bramcohen.pdf`, 2017.

10. Bram Cohen and Krzysztof Pietrzak. Simple proofs of sequential work. In Jesper Buus Nielsen and Vincent Rijmen, editors, *Advances in Cryptology – EUROCRYPT 2018, Part II*, volume 10821 of *Lecture Notes in Computer Science*, pages 451–467, Tel Aviv, Israel, April 29 – May 3, 2018. Springer, Heidelberg, Germany.
11. Nico Döttling, Russell W. F. Lai, and Giulio Malavolta. Incremental proofs of sequential work. In *EUROCRYPT 2019 (to appear)*, 2019.
12. Luca De Feo, Simon Masson, Christophe Petit, and Antonio Sanso. Verifiable delay functions from supersingular isogenies and pairings. Cryptology ePrint Archive, Report 2019/166, 2019. `https://eprint.iacr.org/2019/166`.
13. Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In Andrew M. Odlyzko, editor, *Advances in Cryptology – CRYPTO'86*, volume 263 of *Lecture Notes in Computer Science*, pages 186–194, Santa Barbara, CA, USA, August 1987. Springer, Heidelberg, Germany.
14. Joe Kilian. A note on efficient zero-knowledge proofs and arguments (extended abstract). In *24th Annual ACM Symposium on Theory of Computing*, pages 723–732, Victoria, BC, Canada, May 4–6, 1992. ACM Press.
15. Mohammad Mahmoody, Tal Moran, and Salil P. Vadhan. Publicly verifiable proofs of sequential work. In Robert D. Kleinberg, editor, *ITCS 2013: 4th Innovations in Theoretical Computer Science*, pages 373–388, Berkeley, CA, USA, January 9–12, 2013. Association for Computing Machinery.
16. Silvio Micali. CS proofs (extended abstracts). In *35th Annual Symposium on Foundations of Computer Science*, pages 436–453, Santa Fe, NM, USA, November 20–22, 1994. IEEE Computer Society Press.
17. Krzysztof Pietrzak. Simple verifiable delay functions. In Avrim Blum, editor, *ITCS 2019: 10th Innovations in Theoretical Computer Science Conference*, volume 124, pages 60:1–60:15, San Diego, CA, USA, January 10–12, 2019. LIPIcs.
18. Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman. A method for obtaining digital signature and public-key cryptosystems. *Communications of the Association for Computing Machinery*, 21(2):120–126, 1978.
19. Ronald L Rivest, Adi Shamir, and David A Wagner. Time-lock puzzles and timed-release crypto. *Technical Report MIT/LCS/TR-684*, 1996.
20. Paul Valiant. Incrementally verifiable computation or proofs of knowledge imply time/space efficiency. In Ran Canetti, editor, *TCC 2008: 5th Theory of Cryptography Conference*, volume 4948 of *Lecture Notes in Computer Science*, pages 1–18, San Francisco, CA, USA, March 19–21, 2008. Springer, Heidelberg, Germany.
21. Benjamin Wesolowski. Efficient verifiable delay functions. Cryptology ePrint Archive, Report 2018/623, 2018. `https://eprint.iacr.org/2018/623`.