# Verifiable Delay Functions

Dan Boneh[1], Joseph Bonneau[2], Benedikt Bünz[1], and Ben Fisch[1]

[1]Stanford University
[2]New York University

June 26, 2019

**Abstract**

We study the problem of building a *verifiable delay function* (VDF). A VDF requires a specified number of sequential steps to evaluate, yet produces a unique output that can be efficiently and publicly verified. VDFs have many applications in decentralized systems, including public randomness beacons, leader election in consensus protocols, and proofs of replication. We formalize the requirements for VDFs and present new candidate constructions that are the first to achieve an exponential gap between evaluation and verification time.

## 1 Introduction

Consider the problem of running a verifiable lottery using a *randomness beacon*, a concept first described by Rabin [66] as an ideal service that regularly publishes random values which no party can predict or manipulate. A classic approach is to apply an extractor function to a public entropy source, such as stock prices [24]. Stock prices are believed to be difficult to predict for a passive observer, but an active adversary could manipulate prices to bias the lottery. For example, a high-frequency trader might slightly alter the closing price of a stock by executing (or not executing) a few transactions immediately before the market closes.

Suppose the extractor takes only a single bit per asset (e.g. whether the stock finished up or down for the day) and suppose the adversary is capable of changing this bit for $k$ different assets using last-second trades. The attacker could read the prices of the assets it cannot control, quickly simulate $2^k$ potential lottery outcomes based on different combinations of the $k$ outcomes it can control, and then manipulate the market to ensure its preferred lottery outcome occurs.

One solution is to add a delay function after extraction, making it slow to compute the beacon outcome from an input of raw stock prices. With a delay function of say, one hour, by the time the adversary simulates the outcome of any potential manipulation strategy, the market will be closed and prices finalized, making it too late to launch an attack. This suggests the key security property for a delay function: it should be infeasible for an adversary to distinguish the function's output from random in less than a specified amount of wall-clock time, even given a potentially large number of parallel processors.

A trivial delay function can be built by iterating a cryptographic hash function. For example, it is reasonable to assume it is infeasible to compute $2^{40}$ iterations of SHA-256 in a matter of

seconds, even using specialized hardware. However, a lottery participant wishing to verify the output of this delay function must repeat the computation in its entirety (which might take many hours on a personal computer). Ideally, we would like to design a delay function which any observer can quickly verify was computed correctly.

**Defining delay functions.** In this paper we formalize the requirements for a *verifiable delay function* (VDF) and provide the first constructions which meet these requirements. A VDF consists of a triple of algorithms: $\mathsf{Setup}$, $\mathsf{Eval}$, and $\mathsf{Verify}$. $\mathsf{Setup}(\lambda, t)$ takes a security parameter $\lambda$ and delay parameter $t$ and outputs public parameters $\mathbf{pp}$ (which fix the domain and range of the VDF and may include other information necessary to compute or verify it). $\mathsf{Eval}(\mathbf{pp}, x)$ takes an input $x$ from the domain and outputs a value $y$ in the range and (optionally) a short proof $\pi$. Finally, $\mathsf{Verify}(\mathbf{pp}, x, y, \pi)$ efficiently verifies that $y$ is the correct output on $x$. Crucially, for every input $x$ there should be a *unique* output $y$ that will verify. Informally, a VDF scheme should satisfy the following properties:

- *sequential*: honest parties can compute $(y, \pi) \leftarrow \mathsf{Eval}(\mathbf{pp}, x)$ in $t$ sequential steps, while no parallel-machine adversary with a polynomial number of processors can distinguish the output $y$ from random in significantly fewer steps.

- *efficiently verifiable*: We prefer $\mathsf{Verify}$ to be as fast as possible for honest parties to compute; we require it to take total time $O(\text{polylog}(t))$.

- *unique*: for all inputs $x$, it is difficulty to find a $y$ for which $\mathsf{Verify}(\mathbf{pp}, x, y, \pi) = \mathit{Yes}$, but $y \neq \mathsf{Eval}(\mathbf{pp}, x)$.

A VDF should remain secure even in the face of an attacker able to perform polynomially bounded pre-computation.

Some VDFs may also offer additional useful properties:

- *decodable*: A VDF is decodable if there exists a decoding algorithm $\mathsf{Dec}$ such that $(\mathsf{Eval}, \mathsf{Dec})$ form a lossless encoding scheme. A lossless encoding scheme is a pair of algorithms $(\mathsf{Enc}, \mathsf{Dec})$ where $\mathsf{Enc} : X \to Y$ and $\mathsf{Dec} : Y \to X$ such that $\mathsf{Dec}(\mathsf{Enc}(x)) = x$ for all $x \in X$. We say that a VDF is *efficiently decodable* if it is decodable and $\mathsf{Dec}$ is efficient. In this case, $\mathsf{Eval}$ need not include a proof as $\mathsf{Dec}$ itself can be used to verify the output. There are many different kinds of encoding schemes with different properties, including compression schemes, error correcting codes, ciphers, etc. Of course any VDF can be turned in a trivial encoding scheme by appending the input $x$ to the output, however this would not have any interesting properties as an encoding scheme. In Section 2, we will describe one interesting application of an encoding scheme that is both an efficiently invertible ideal cipher and a VDF. The application is to *proofs-of-replication*.

- *incremental*: a single set of public parameters $\mathbf{pp}$ supports multiple hardness parameters $t$. The number of steps used to compute $y$ is specified in the proof, instead of being fixed during $\mathsf{Setup}$. The main benefit of incremental VDFs over a simple chaining of VDFs to increase the delay is a reduced aggregate proof size. This is particularly useful for applications of VDFs to computational time-stamping or blockchain consensus.

**Classic slow functions** Time-lock puzzles [68] are similar to VDFs in that they involve computing an inherently sequential function. An elegant solution uses repeated squaring in an RSA group as a time-lock puzzle. However, time-lock puzzles are not required to be universally verifiable and in all known constructions the verifier uses its secret state to prepare each puzzle and verify the results. VDFs, by contrast, may require an initial trusted setup but then must be usable on any randomly chosen input.

Another construction for a slow function dating to Dwork and Naor [31] is extracting modular square roots. Given a challenge $x \in \mathbb{Z}_p^*$ (with $p \equiv 3 \pmod 4$), computing $y = \sqrt{x} = x^{\frac{p+1}{4}} \pmod p$ can be efficiently verified by checking that $y^2 = x \pmod p$. There is no known algorithm for computing modular exponentiation which is sublinear in the bit-length of the exponent. However, the difficulty of puzzles is limited to $t = O(\log p)$ as the exponent can be reduced modulo $p-1$ before computation, requiring the use of a very large prime $p$ to produce a difficult puzzle. While it was not originally proposed for its sequential nature, it has subsequently been considered as such several times [41, 48]. In particular, Lenstra and Wesolowski [48] proposed chaining a series of such puzzles together in a construction called Sloth, with lotteries as a specific motivation. Sloth is best characterized as a *time-asymmetric encoding*, offering a trade-off in practice between computation and inversion (verification), and thus can be viewed as a pseudo-VDF. However, it does not meet our asymptotic definition of a VDF because it does not offer asymptotically efficient verification: the $t$-bit modular exponentiation can be computed in parallel time $t$, whereas the output (a $t$-bit number) requires $\Omega(t)$ time simply to read, and therefore verification cannot run in total time polylog$(t)$. We give a more complete overview of related work in Section 9.

**Our contributions:** In addition to providing the first formal definitions of VDFs, we contribute the following candidate constructions and techniques:

1. A theoretical VDF can be constructed using *incrementally verifiable computation* [70] (IVC), in which a proof of correctness for a computation of length $t$ can be computed in parallel to the computation with only polylog(t) processors. We prove security of this theoretical VDF using IVC as a black box. IVC can be constructed from succinct non-interactive arguments of knowledge (SNARKs) under a suitable extractor complexity assumption [14]. In an update (May 2019) we present a simpler construction based only on verifiable computation (Section 5).

2. We propose a construction based on injective polynomials over algebraic sets that cannot be inverted faster than computing polynomial GCDs. Computing polynomial GCD is sequential in the degree $d$ of the polynomials on machines with fewer than $O(d^2)$ processors. We propose a candidate construction of time-asymmetric encodings from a particular family of *permutation polynomials* over finite fields [39]. This construction is asymptotically a strict improvement on Sloth, and to the best of our knowledge is the first encoding offering an exponential time gap between evaluation and inversion. We call this a *weak VDF* because it requires the honest $\mathsf{Eval}$ to use greater than *polylog*$(t)$ parallelism to run in parallel time $t$ (the delay parameter).

3. In Section 7 we describe a practical efficiency boost to constructing VDFs from IVC using time-asymmetric encodings as the underlying sequential computation, offering up to a 7,000 fold improvement (in the SNARK efficiency) over naive hash chains. In this

construction the SNARK proof is only used to boost the efficiency of verification as the output $(y, \pi)$ of $\mathsf{Eval}$ on an input $x$ can also be verified directly without $\pi$ by computing the inverse of $y$, which is still faster than computing $y$ from $x$.

4. We construct a VDF secure against bounded pre-computation attacks following a generalization of time-lock puzzles based on exponentiation in a group of unknown order.

## 2 Applications

Before giving precise definitions and describing our constructions, we first informally sketch several important applications of VDFs.

**Randomness beacons.** VDFs are useful for constructing randomness beacons from sources such as stock prices [24] or proof-of-work blockchains (e.g. Bitcoin, Ethereum) [17, 63, 12]. Proof-of-work blockchains include randomly sampled solutions to computational puzzles that network participants (called *miners*) continually find and publish for monetary rewards. Underpinning the security of proof-of-work blockchains is the strong belief that these solutions have high computational min-entropy. However, similar to potential manipulation of asset prices by high-frequency traders, powerful miners could potentially manipulate the beacon result by refusing to post blocks which produce an unfavorable beacon output.

Again, this attack is only feasible if the beacon can be computed quickly, as each block is fixed to a specific predecessor and will become "stale" if not published. If a VDF with a suitably long delay is used to compute the beacon, miners will not be able to determine the beacon output from a given block before it becomes stale. More specifically, given the desired delay parameter $t$, the public parameters $\mathbf{pp} = (\mathsf{ek}, \mathsf{vk}) \xleftarrow{R} \mathsf{Setup}(\lambda, t)$ are posted on the blockchain, then given a block $b$ the beacon value is determined to be $r$ where $(r, \pi) = \mathsf{Eval}(\mathsf{ek}, b)$, and anyone can verify correctness by running $\mathsf{Verify}(\mathsf{vk}, b, r, \pi)$. The security of this construction, and in particular the length of delay parameter which would be sufficient to prevent attacks, remains an informal conjecture due to the lack of a complete game-theoretic model capturing miner incentives in Nakamoto-style consensus protocols. We refer the reader to [17, 63, 12] for proposed models for blockchain manipulation. Note that most formal models for Nakamoto-style consensus such as that of Garay et al. [36] do not capture miners with external incentives such as profiting from lottery manipulation.

Another approach for constructing beacons derives randomness from a collection of participants, such as all participants in a lottery [38, 48]. The simplest paradigm is "commit-and-reveal" paradigm where $n$ parties submit commitments to random values $r_1, ..., r_n$ in an initial phase and subsequently reveal their commitments, at which point the beacon output is computed as $r = \bigoplus_i r_i$. The problem with this approach is that a malicious adversary (possibly controlling a number of parties) might manipulate the outcome by refusing to open its commitment after seeing the other revealed values, forcing a protocol restart. Lenstra and Wesolowski proposed a solution to this problem (called "Unicorn"[48]) using a delay function: instead of using commitments, each participant posts their $r_i$ directly and $seed = H(r_1, \ldots, r_n)$ is passed through a VDF. The output of $\mathsf{Eval}$ is then posted and can be efficiently verified. The final beacon outcome is the hash of the output of $\mathsf{Eval}$. With a sufficiently long delay parameter (longer than the time period during which values may be submitted), even the last party to publish their $r_i$ cannot predict what its impact will be on the final beacon outcome. The beacon is unpredictable even

to an adversary who controls $n-1$ of the participating parties. It has linear communication complexity and uses only two rounds. This stands in contrast to coin-tossing beacons which use verifiable secret sharing and are at best resistant to an adversary who controls a minority of the nodes [1, 69, 23]. These beacons also use super-linear communication and require multiple rounds of interaction. In the two party setting there are tight bounds that an $r$-round coin-flipping protocol can be biased with $O(1/r)$ bias [57]. The "Unicorn" construction circumvents these bounds by assuming semi-synchronous communication, i.e. there exists a bound to how long an adversary can delay messages.

**Resource-efficient blockchains.** Amid growing concerns over the long-term sustainability of proof-of-work blockchains like Bitcoin, which consume a large (and growing) amount of energy, there has been concerted effort to develop resource-efficient blockchains in which miners invest an upfront capital expenditure which can then be re-used for mining. Examples include proof-of-stake [46, 55, 45, 28, 13], proof-of-space [61], and proof-of-storage [56, 2]. However, resource-efficient mining suffers from *costless simulation* attacks. Intuitively, since mining is not computationally expensive, miners can attempt to produce many separate forks easily.

One method to counter simulation attacks is to use a randomness beacon to select new leaders at regular intervals, with the probability of becoming a leader biased by the quality of proofs (i.e. amount of stake, space, etc) submitted by miners. A number of existing blockchains already construct beacons from tools such as *verifiable random functions*, *verifiable secret sharing*, or *deterministic threshold signatures* [45, 28, 23, 4]. However, the security of these beacons requires a non-colluding honest majority; with a VDF-based lottery as described above this can potentially be improved to participation of any honest party.

A second approach, proposed by Cohen [26], is to combine proofs-of-resources with incremental VDFs and use the product of resources proved and delay induced as a measure of blockchain quality. This requires a proof-of-resource which is costly to initialize (such as certain types of proof-of-space). This is important such that the resources are committed to the blockchain and cannot be used for other purposes. A miner controlling $N$ units of total resources can initialize a proof $\pi$ demonstrating control over these $N$ units. Further assume that the proof is non-malleable and that in each epoch there is a common random challenge $c$, e.g. a block found in the previous epoch, and let $H$ be a random oracle available to everyone. In each epoch, the miner finds $\tau = \min_{1 \leq i \leq N}\{H(c, \pi, i)\}$ and computes a VDF on input $c$ with a delay proportional to $\tau$. The first miner to successfully compute the VDF can broadcast their block successfully. Note that this process mimics the random delay to find a Bitcoin block (weighted by the amount of resources controlled by each miner), but without each miner running a large parallel computation.

**Proof of replication.** Another promising application of VDFs is *proofs of replication*, a special type of proof of data storage which requires the prover to dedicate unique storage even if the data is available from another source. For instance, this could be used to prove that a number of replicas of the same file are being stored. Classic *proofs of retrievability* [43] are typically defined in a private-key client/server setting, where the server proves to the client that it can retrieve the client's (private) data, which the client verifies using a private key.

Instead, the goal of a *proof of replication* [6, 2, 3] is to verify that a given server is storing a *unique replica* of some data which may be publicly available. An equivalent concept to proof-of-replication was first introduced by Sergio Demian Lerner in 2014 under the name "proof of

unique blockchain storage" [49]. Lerner proposed using time-asymmetric encodings to apply a slow transformation to a file using a unique identifier as a key. During a challenge-response protocol, a verifier periodically challenges the server for randomly sampled blocks of the uniquely encoded file. The basic idea is that the server should fail to respond quickly enough if it has deleted the encoding. Verifying that the received blocks of the encoding are correct is fast in comparison due to the time-asymmetry of the encoding. Lerner proposed using a Pohlig-Hellman cipher, using the permutation $x^3$ on $\mathbb{Z}_p^*$, which has asymmetry roughly equivalent to modular square roots. Armknecht et al. [6] proposed a similar protocol in the private verifier model using RSA time-lock puzzles. The verification of this protocol may be outsourced, but is still less transparent as it fundamentally requires a designated private-key verifier per file.

Efficiently decodable VDFs can be used to improve Lerner's publicly verifiable and transparent construction by using the VDF as the time-asymmetric encoding. As VDFs achieve an exponential gap between parallel-time computation and verification they would improve the challenge-response protocol by reducing the frequency of polling. The frequency needs to be set such that the server cannot delete and recompute the encoding between challenges. Technically, the security property we need the VDF to satisfy is that a stateless adversary running in parallel time less than $T$ cannot predict any component of the output on any given input, which is stronger than the plain sequentiality requirement of a VDF. A formal way to capture this requirement is to model the VDF as an *ideal delay cipher* [35], namely as an oracle that implements an ideal cipher and takes $T$ sequential steps to respond to queries on any point.

We review briefly the construction, now based on VDFs. The replicator is given an input file, a unique replicator identifier $id$, and public parameters $\mathbf{pp} \xleftarrow{R} \mathsf{Setup}(\lambda, t)$, and computes a *slow* encoding of the file using the VDF cipher. This takes sequential time $T$ to derive. In more detail, the encoding is computed by breaking the file into $b$-bit blocks $B_1, \ldots, B_n$ and storing $y_1, ..., y_n$ where $(y_i, \perp) = \mathsf{Eval}(\mathbf{pp}, B_i \oplus H(id||i))$ where $H$ is a collision-resistant hash function $H : \{0,1\}^* \to \{0,1\}^b$. To verify that the replicator has stored this unique copy, a verifier can query an encoded block $y_i$ (which must be returned in significantly less time than it is feasible to compute $\mathsf{Eval}$). The verifier can quickly decode this response and check it for correctness, proving that the replicator has stored (or can quickly retrieve from somewhere) an encoding of this block which is unique to the identifier $id$. If the unique block encoding $y_i$ has not been stored, the VDF ensures that it cannot be re-computed quickly enough to fool the verifier, even given access to $B_i$ . The verifier can query for as many blocks as desired; each query has a $1 - \rho$ chance of exposing a cheating prover that is only storing a fraction $\rho$ of the encoded blocks.

More generally, if the inputs $B_1, .., B_n$ are fixed and known to the verifier then this construction is also a *proof of space* [32]. A proof of space is an interactive protocol in which the prover can provide a compact proof that it is persistently using $\Omega(N)$ space. This construction is in fact a *tight* PoS, meaning that it requires an honest prover to use $N$ space and for any $\epsilon > 0$ it can be tuned so that an adversary who uses $(1 - \epsilon)N$ space will be caught with overwhelming probability. A proof-of-replication is a special kind of proof of space that is quite delicate to formally define and has been developed further in followup work [35].

**Computational timestamping.** All known proof-of-stake systems are vulnerable to long-range forks due to post-hoc stakeholder misbehavior [46, 55, 45, 13]. In proof-of-stake protocols, at any given time the current stakeholders in the system are given voting power proportionate to their stake in the system. An honest majority (or supermajority) is assumed because the current stakeholders are incentivized to keep the system running correctly. However, after stakeholders

have divested they no longer have this incentive. Once the majority (eq. supermajority) of stakeholders from a point in time in the past are divested, they can collude (or sell their key material to an attacker) in order to create a long alternate history of the system up until the present. Current protocols typically assume this is prevented through an external timestamping mechanism which can prove to users that the genuine history of the system is much older.

Incremental VDFs can provide computational evidence that a given version of the state's system is older (and therefore genuine) by proving that a long-running VDF computation has been performed on the genuine history just after the point of divergence with the fraudulent history. This potentially enables detecting long-range forks without relying on external timestamping mechanisms.

We note however that this application of VDFs is fragile as it requires precise bounds on the attacker's computation speed. For other applications (such as randomness beacons) it may be acceptable if the adversary can speed up VDF evaluation by a factor of 10 using faster hardware; a higher $t$ can be chosen until even the adversary cannot manipulate the beacon even with a hardware speedup. For computational timestamping, a 10-fold speedup would be a serious problem: once the fraudulent history is more than one-tenth as old as the genuine history, an attacker can fool participants into believing the fraudulent history is actually *older* than the genuine one.

# 3 Model and definitions

We now define VDFs more precisely. In what follows we say that an algorithm runs in *parallel time* $t$ with $p$ processors if it can be implemented on a PRAM machine with $p$ parallel processors running in time $t$. We say *total time* (eq. sequential time) to refer to the time needed for computation on a single processor.

**Definition 1.** *A VDF* $V = (\mathsf{Setup}, \mathsf{Eval}, \mathsf{Verify})$ *is a triple of algorithms as follows:*

- $\mathsf{Setup}(\lambda, t) \to \mathbf{pp} = (\mathsf{ek}, \mathsf{vk})$ *is a randomized algorithm that takes a security parameter* $\lambda$ *and a desired puzzle difficulty* $t$ *and produces public parameters* $\mathbf{pp}$ *that consists of an evaluation key* $\mathsf{ek}$ *and a verification key* $\mathsf{vk}$*. We require* $\mathsf{Setup}$ *to be polynomial-time in* $\lambda$*. By convention, the public parameters specify an input space* $\mathcal{X}$ *and an output space* $\mathcal{Y}$*. We assume that* $\mathcal{X}$ *is efficiently sampleable.* $\mathsf{Setup}$ *might need secret randomness, leading to a scheme requiring a trusted setup. For meaningful security, the puzzle difficulty* $t$ *is restricted to be sub-exponentially sized in* $\lambda$*.*

- $\mathsf{Eval}(\mathsf{ek}, x) \to (y, \pi)$ *takes an input* $x \in \mathcal{X}$ *and produces an output* $y \in \mathcal{Y}$ *and a (possibly empty) proof* $\pi$*.* $\mathsf{Eval}$ *may use random bits to generate the proof* $\pi$ *but not to compute* $y$*. For all* $\mathbf{pp}$ *generated by* $\mathsf{Setup}(\lambda, t)$ *and all* $x \in \mathcal{X}$*, algorithm* $\mathsf{Eval}(\mathsf{ek}, x)$ *must run in parallel time* $t$ *with* $\mathsf{poly}(\log(t), \lambda)$ *processors.*

- $\mathsf{Verify}(\mathsf{vk}, x, y, \pi) \to \{Yes, No\}$ *is a deterministic algorithm takes an input, output and proof and outputs Yes or No. Algorithm* $\mathsf{Verify}$ *must run in total time polynomial in* $\log t$ *and* $\lambda$*. Notice that* $\mathsf{Verify}$ *is much faster than* $\mathsf{Eval}$*.*

*Additionally* $V$ *must satisfy Correctness (Definition 2), Soundness (Definition 3), and Sequentiality (Definition 4).*

**Correctness and Soundness** Every output of $\mathsf{Eval}$ must be accepted by $\mathsf{Verify}$. We guarantee that the output $y$ for an input $x$ is unique because $\mathsf{Eval}$ evaluates a deterministic function on $\mathcal{X}$. Note that we do not require the proof $\pi$ to be unique, but we do require that the proof is sound and that a verifier cannot be convinced that some different output is the correct VDF outcome. More formally,

**Definition 2** (Correctness)**.** *A VDF $V$ is correct if for all $\lambda$, $t$, parameters $(\mathsf{ek}, \mathsf{vk}) \xleftarrow{R} \mathsf{Setup}(\lambda, t)$, and all $x \in \mathcal{X}$, if $(y, \pi) \xleftarrow{R} \mathsf{Eval}(\mathsf{ek}, x)$ then $\mathsf{Verify}(\mathsf{vk}, x, y, \pi) = \mathit{Yes}$.*

We also require that for no input $x$ can an adversary get a verifier to accept an incorrect VDF output.

**Definition 3** (Soundness)**.** *A VDF is sound if for all algorithms $\mathcal{A}$ that run in time $O(\mathsf{poly}(t, \lambda))$*

$$Pr\left[\begin{array}{c}\mathsf{Verify}(\mathsf{vk}, x, y, \pi) = \mathit{Yes} \\ y \neq \mathsf{Eval}(\mathsf{ek}, x)\end{array}\;\middle|\;\begin{array}{c}\mathbf{pp} = (\mathsf{ek}, \mathsf{vk}) \xleftarrow{R} \mathsf{Setup}(\lambda, t) \\ (x, y, \pi) \xleftarrow{R} \mathcal{A}(\lambda, \mathbf{pp}, t)\end{array}\right] \leq \mathsf{negl}(\lambda)$$

**Size restriction on $t$** Asymptotically $t$ must be subexponential in $\lambda$. The reason for this is that the adversary needs to be able to run in time at least $t$ ($\mathsf{Eval}$ requires this), and if $t$ is exponential in $\lambda$ then the adversary might be able to break the underlying computational security assumptions that underpin both the soundness as well as the sequentiality of the VDF, which we will formalize next.

**Parallelism in Eval** The practical implication of allowing more parallelism in $\mathsf{Eval}$ is that "honest" evaluators may be required to have this much parallelism in order to complete the challenge in time $t$. The sequentiality security argument will compare an adversary's advantage to this optimal implementation of $\mathsf{Eval}$. Constructions of VDFs that do not require any parallelism to evaluate $\mathsf{Eval}$ in the optimal number of sequential steps are obviously superior. However, it is unlikely that such constructions exist (without trusted hardware). Even computing an iterated hash function or modular exponentiation (used for time-lock puzzles) could be computed faster by parallelizing the hash function or modular arithmetic. In fact, for an decodable VDF it is necessary that $|\mathcal{Y}| > \mathsf{poly}(t)$, and thus the challenge inputs to $\mathsf{Eval}$ have size $\mathsf{poly}\log(t)$. Therefore, in our definition we allow algorithm $\mathsf{Eval}$ up to $\mathsf{poly}\log(t)$ parallelism.

### 3.1 VDF Security

We call the security property needed for a VDF scheme *$\sigma$-sequentiality*. Essentially, we require that no adversary is able to compute an output for $\mathsf{Eval}$ on a random challenge in parallel time $\sigma(t) < t$, even with up to "many" parallel processors and after a potentially large amount of precomputation. It is critical to bound the adversary's allowed parallelism, and we incorporate this into the definition. Note that for an efficiently decodable VDF, an adversary with $|\mathcal{Y}|$ processors can always compute outputs in $o(t)$ parallel time by simultaneously trying all possible outputs in $\mathcal{Y}$. This means that for efficiently decodable VDFs it is necessary that $|\mathcal{Y}| > \mathsf{poly}(t)$, and cannot achieve $\sigma$-sequentiality against an adversary with greater than $|\mathcal{Y}|$ processors.

We define the following sequentiality game applied to an adversary $\mathcal{A} := (\mathcal{A}_0, \mathcal{A}_1)$:

$$
\begin{aligned}
\mathbf{pp} &\xleftarrow{R} \mathsf{Setup}(\lambda, t) && // \text{ choose a random } \mathbf{pp} \\
L &\xleftarrow{R} \mathcal{A}_0(\lambda, \mathbf{pp}, t) && // \text{ adversary preprocesses } \mathbf{pp} \\
x &\xleftarrow{R} \mathcal{X} && // \text{ choose a random input } x \\
y_A &\xleftarrow{R} \mathcal{A}_1(L, \mathbf{pp}, x) && // \text{ adversary computes an output } y_A
\end{aligned}
$$

We say that $(\mathcal{A}_0, \mathcal{A}_1)$ wins the game if $y_A = y$ where $(y, \pi) := \mathsf{Eval}(\mathbf{pp}, x)$.

**Definition 4** (Sequentiality)**.** *For functions $\sigma(t)$ and $p(t)$, the VDF is $(p, \sigma)$-sequential if no pair of randomized algorithms $\mathcal{A}_0$, which runs in total time $O(\mathsf{poly}(t, \lambda))$, and $\mathcal{A}_1$, which runs in parallel time $\sigma(t)$ on at most $p(t)$ processors, can win the sequentiality game with probability greater than $\mathsf{negl}(\lambda)$.*

The definition captures the fact that even after $\mathcal{A}_0$ computes on the parameters $\mathbf{pp}$ for a (polynomially) long time, the adversary $\mathcal{A}_1$ cannot compute an output from the input $x$ in time $\sigma(t)$ on $p(t)$ parallel processors. If a VDF is $(p, \sigma)$-sequential for any polynomial $p$, then we simply say the VDF is $\sigma$-sequential. In the sequentiality game we do not require the online attack algorithm $\mathcal{A}_1$ to output a proof $\pi$. The reason is that in many of our applications, for example in a lottery, the adversary can profit simply by learning the output early, even without being able to prove correctness to a third party.

**Values of $\sigma(t)$** Clearly any candidate construction trivially satisfies $\sigma(t)$-sequentiality for *some* $\sigma$ (e.g. $\sigma(t) = 0$). Thus, security becomes more meaningful as $\sigma(t) \to t$. No construction can obtain $\sigma(t) = t$ because by design $\mathsf{Eval}$ runs in parallel time $t$. Ideal security is achieved when $\sigma(t) = t - 1$. This ideal security is in general unrealistic unless, for example, time steps are measured in rounds of queries to an ideal oracle (e.g. random oracle). In practice, if the oracle is instantiated with a concrete program (e.g. a hash function), then differences in hardware/implementation would in general yield small differences in the response time for each query. An almost-perfect VDF would achieve $\sigma(t) = t - o(t)$ sequentiality. Even $\sigma(t) = t - \epsilon t$ sequentiality for small $\epsilon$ is sufficient for most applications. Security degrades as $\epsilon \to 1$. The naive VDF construction combining a hash chain with succinct verifiable computation (i.e. producing a SNARG proof of correctness following the hash chain computation) cannot beat $\epsilon = 1/2$, unless it uses at least $\omega(t)$ parallelism to generate the proof in sublinear time (exceeding the allowable parallelism for VDFs, though see a relaxation to "weak" VDFs below).

**Unpredictability and min-entropy** Definition 4 captures an unpredictability property for the output of the VDF, similar to a one-way function. However, similar to random oracles, the output of the VDF on a given input is never indistinguishable from random. It is possible that no depth $\sigma(t)$ circuit can distinguish the output on a randomly sampled challenge from random, but only if the VDF proof is not given to the distinguisher. Efficiently decodable VDFs cannot achieve this stronger property.

For the application to random beacons (e.g. for lotteries), it is only necessary that on a random challenge the output is unpredictable and also has sufficient min-entropy[1] conditioned

[1] A randomness extractor can then be applied to the output to map it to a uniform distribution.

on previous outputs for different challenges. In fact, $\sigma$-sequentiality already implies that min-entropy is $\Omega(\log \lambda)$. Otherwise some fixed output $y$ occurs with probability $1/\mathsf{poly}(\lambda)$ for randomly sampled input $x$; the adversary $\mathcal{A}_0$ can computes $O(\mathsf{poly}(\lambda))$ samples of this distribution in the preprocessing to find such a $y'$ with high probability, and then $\mathcal{A}_1$ could output $y'$ as its guess. Moreover, if $\sigma$-sequentiality is achieved for $t$ superpolynomial (sub-exponential) in $\lambda$, then the preprocessing adversary is allowed $2^{o(\lambda)}$ samples, implying some $o(\lambda)$ min-entropy of the output must be preserved. By itself, $\sigma$-sequentiality does not imply $\Omega(\lambda)$ min-entropy. Stronger min-entropy preservation can be demonstrated in other ways given additional properties of the VDF, e.g. if it is a permutation or collision-resistant. Under suitable complexity theoretic assumptions (namely the existence of subexponential $2^{o(n)}$ circuit lower bounds) a combination of Nisan-Wigderson type PRGs and extractors can also be used to generate $\mathsf{poly}(\lambda)$ pseudorandom bits from a string with min-entropy $\log \lambda$.

**Random "Delay" Oracle** In the random oracle model, *any* unpredictable string (regardless of its min-entropy) can be used to extract an unpredictable $\lambda$-bit uniform random string. For the beacon application, a random oracle $H$ would simply be applied to the output of the VDF to generate the beacon value. We can even model this construction as an ideal object itself, a *Random Delay Oracle*, which implements a random function $H'$ and on any given input $x$ it waits for $\sigma(t)$ steps before returning the output $H'(x)$. Demonstrating a construction from a $\sigma$-sequential VDF and random oracle $H$ that is provably *indifferentiable* [53] from a Random Delay Oracle is an interesting research question.[2]

**Remark:** Removing any single property makes VDF construction easy. We note the existence of well-known outputs if any property is removed:

- If $\mathsf{Verify}$ is not required to be fast, then simply iterating a one-way function $t$ times yields a trivial solution. Verification is done by re-computing the output, or a set of $\ell$ intermediate points can be supplied as a proof which can be verified in parallel time $\Theta(t/\ell)$ using $\ell$ processors, with total verification time remaining $\Theta(t)$.
- If we do not require *uniqueness*, then the construction of Mahmoody et al. [52] using hash functions and depth-robust graphs suffices. This construction was later improved by Cohen and Pietrzak [19]. This construction fails to ensure uniqueness because once an output $y$ is computed it can be easily mangled into many other valid outputs $y' \neq y$, as discussed in Section 9.1.
- If we do not require *$\sigma$-sequentiality*, many solutions are possible, such as finding the discrete log of a challenge group element with respect to a fixed generator. Note that computing an elliptic curve discrete log can be done in parallel time $o(t)$ using a parallel version of the Pollard rho algorithm [71].

**Weaker VDFs** For certain applications it is still interesting to consider a VDF that requires even more than polylog($t$) parallelism in $\mathsf{Eval}$ to compute the output in parallel time $t$. For

[2]The difficulty in proving indifferentiability arises because the distinguisher can query the VDF/RO construction and the RO itself separately, therefore the simulator must be able to simulate queries to the random oracle $H$ given only access to the Random Delay Oracle. Indifferentiability doesn't require the simulator to respond in exactly the same time, but it is still required to be *efficient*. This becomes an issue if the delay $t$ is superpolynomial.

example, in the randomness beacon application only one party is required to compute the VDF and all other parties can simply verify the output. It would not be unreasonable to give this one party a significant amount of parallel computing power and optimized hardware. This would yield a secure beacon as long as no adversary could compute the outputs of $\mathsf{Eval}$ in faster that $t$ steps given even more parallelism than this party. Moreover, for small values of $t$ it may be practical for anyone to use up to $O(t)$ parallelism (or more). With this in mind, we define a weaker variant of a VDF that allows additional parallelism in $\mathsf{Eval}$.

**Definition 5.** *We call a system $V = (\mathsf{Setup}, \mathsf{Eval}, \mathsf{Verify})$ a weak-VDF if it satisfies Definition 1 with the exception that $\mathsf{Eval}$ is allowed up to $poly(t, \lambda)$ parallelism.*

Note that $(p, \sigma)$-sequentiality can only be meaningful for a weak-VDF if $\mathsf{Eval}$ is allowed strictly less that $p(t)$ parallelism, otherwise the honest computation of $\mathsf{Eval}$ would require more parallelism than even the adversary is allowed.

# 4 VDFs from Incrementally Verifiable Computation

VDFs are by definition sequential functions. We therefore require the existence of sequential functions in order to construct any VDF. We begin by defining a sequential function.

**Definition 6** ($(t, \epsilon)$-Sequential function)**.** *$f : X \to Y$ is a $(t, \epsilon)$-sequential function if for $\lambda = O(\log(|X|))$, if the following conditions hold.*

1. *There exists an algorithm that for all $x \in X$ evaluates $f$ in parallel time $t$ using $\mathsf{poly}(\log(t), \lambda)$ processors.*
2. *For all $\mathcal{A}$ that run in parallel time strictly less than $(1 - \epsilon) \cdot t$ with $\mathsf{poly}(t, \lambda)$ processors:*

$$Pr\Big[y_A = f(x) \;\Big|\; y_A \xleftarrow{R} \mathcal{A}(\lambda, x),\ x \xleftarrow{R} X\Big] \leq \mathsf{negl}(\lambda)$$

.

In addition we consider *iterated sequential functions* that are defined as the iteration of some round function. The key property of an iterated sequential function is that iteration of the round function is the fastest way to evaluate the function.

**Definition 7** (Iterated Sequential Function)**.** *Let $g : X \to X$ be a $(t, \epsilon)$-sequential function. A function $f : \mathbb{N} \times X \to X$ defined as $f(k, x) = g^{(k)}(x) = \underbrace{g \circ g \circ \cdots \circ g}_{k \text{ times}}$ is said to be an* iterated sequential function *(with round function $g$) if for all $k = 2^{o(\lambda)}$ the function $h : X \to X$ defined by $h(x) = f(k, x)$ is $(k \cdot t, \epsilon)$-sequential as in Definition 6.*

It is widely believed that the function obtained from iterating a hash function like SHA-256 is an iterated sequential function with $t = O(\lambda)$ and $\epsilon$ negligible in $\lambda$. The sequentiality of such functions can be proved in the random oracle model [47, 52]. We will use the functions $g$ explicitly and require it to have an explicit arithmetic circuit representation. Modeling $g$ as an oracle, therefore, does not suffice for our construction.

Another candidate for an iterated sequential function is exponentiation in a finite group of unknown order, where the round function is squaring in the group. The fastest known way to compute this is by repeated squaring which is an iterative sequential computation.

Based on these candidates, we can make the following assumption about the existence of iterated sequential functions:

**Assumption 1.** *For all $\lambda \in \mathbb{N}$ there exist (1) an $\epsilon, t$ with $t = \mathsf{poly}(\lambda)$, and (2) a function $g_\lambda : X \to X$, where $\log_2 |X| = \lambda$ and $X$ can be sampled in time $\mathsf{poly}(\lambda)$. This $g_\lambda$ satisfies: (i) $g_\lambda$ is a $(t, \epsilon)$-sequential function, and (ii) the function $f : \mathbb{N} \times X \to X$ with round function $g_\lambda$ is an iterated sequential function.*

An iterated sequential function by itself gives us many of the properties needed for a secure VDF. It is sequential by definition and the trivial algorithm (iteratively computing $g$) uses only $\mathsf{poly}(\lambda)$ parallelism. Such a function by itself, however, does not suffice to construct a VDF. The fastest generic verification algorithm simply recomputes the function. While this ensures soundness it does not satisfy the efficient verification requirement of a VDF. The verifier of a VDF needs to be exponentially faster than the prover.

**SNARGs and SNARKs** A natural idea to improve the verification time is to use verifiable computation. In verifiable computation the prover computes a succinct argument (SNARG) that a computation was done correctly. The argument can be efficiently verified using resources that are independent of the size of the computation. A SNARG is a weaker form of a succinct non-interactive argument of knowledge (SNARK) [37] for membership in an NP language $\mathcal{L}$ with relation $R$ (Definition 8). The additional requirement of a SNARK is that for any algorithm that outputs a valid proof of membership of an instance $x \in \mathcal{L}$ there is also an extractor that "watches" the algorithm and outputs a witness $w$ such that $(x, w) \in R$. In the special case of providing a succinct proof that a (polynomial size) computation $F$ was done correctly, i.e. $y$ is the output of $F$ on $x$, the NP witness is empty and the NP relation simply consists of pairs $((x, y), \perp)$ such that $F(x) = y$.

**Definition 8** (Verifiable Computation / SNARK)**.** *Let $\mathcal{L}$ denote an NP language with relation $R_\mathcal{L}$, where $x \in \mathcal{L}$ iff $\exists w \; R_\mathcal{L}(x, w) = 1$. A SNARK system for $R_\mathcal{L}$ is a triple of polynomial time algorithms ($\mathsf{SNKGen}, \mathsf{SNKProve}, \mathsf{SNKVerify}$) that satisfy the following properties:*

***Completeness:***

$$\forall (x, w) \in R_\mathcal{L} : \; Pr\left[\mathsf{SNKVerify}(\mathsf{vk}, x, \pi) = 1 \;\middle|\; \begin{array}{l} (\mathsf{vk}, \mathsf{ek}) \xleftarrow{R} \mathsf{SNKGen}(1^\lambda) \\ \pi \xleftarrow{R} \mathsf{SNKProve}(\mathsf{ek}, x, w) \end{array}\right] = 1$$

***Succinctness:*** *The length of a proof and complexity of $\mathsf{SNKVerify}$ is bounded by $poly\big(\lambda, log(|y| + |w|)\big)$.*

***Knowledge extraction:****[sub-exponential adversary knowledge extractor] For all adversaries $\mathcal{A}$ running in time $2^{o(\lambda)}$ there exists an extractor $\mathcal{E}_\mathcal{A}$ running in time $2^{o(\lambda)}$ such that for all auxiliary inputs $z$ of size $poly(\lambda)$:*

$$Pr\left[\begin{array}{c} \mathsf{SNKVerify}(\mathsf{vk}, x, \pi) = 1 \\ R_\mathcal{L}(x, w) \neq 1 \end{array} \;\middle|\; \begin{array}{l} (\mathsf{vk}, \mathsf{ek}) \xleftarrow{R} \mathsf{SNKGen}(1^\lambda) \\ (x, \pi) \xleftarrow{R} \mathcal{A}(z, \mathsf{ek}) \\ w \xleftarrow{R} \mathcal{E}_\mathcal{A}(z, \mathsf{ek}) \end{array}\right] \leq \mathsf{negl}(\lambda)$$

**Impractical VDF from SNARGs.** Consider the following construction for a VDF from a $(t, \epsilon)$-sequential function $f$. Let $\mathbf{pp} = (\mathsf{ek}, \mathsf{vk}) \xleftarrow{R} \mathsf{SNKGen}(\lambda)$ be the public parameter of a SNARG scheme for proving membership in the language of pairs $(x, y)$ such that $f(x) = y$. On input $x \in X$ the $\mathsf{Eval}$ computes $y = f(x)$ and a succinct argument $\pi \xleftarrow{R} \mathsf{SNKProve}(\mathsf{ek}, (x, y), \perp)$. The prover outputs $((x, y), \pi)$. On input $((x, y), \pi)$ the verifier checks $y = f(x)$ by checking $\mathsf{SNKVerify}(\mathsf{vk}, (x, y), \pi) = 1$.

This construction clearly satisfies fast verification. All known SNARK constructions are quasi-linear in the length of the underlying computation $f$ [11]. Assuming the cost for computing a SNARG for a computation of length $t$ is $k \cdot t \log(t)$ then the SNARG VDF construction achieves $\sigma(t) = \frac{(1-\epsilon) \cdot t}{(k+1) \cdot \log(t)}$ sequentiality. This does not even achieve the notion of $(1 - \epsilon')t$ sequentiality for any adversary. This means that the adversary can compute the output of the VDF in a small fraction of the time that it takes the honest prover to convince an honest verifier. If, however, $\mathsf{SNKProve}$ is sufficiently parallelizable then it is possible to partially close the gap between the sequentiality of $f$ and the sequentiality of the VDF. The $\mathsf{Eval}$ simply executes $\mathsf{SNKProve}$ on a parallel machine to reduce the relative total running time compared to the computation of $f$. SNARK constructions can run in parallel time polylog$(t)$ on $O(t \cdot \text{polylog}(t))$ processors. This shows that a VDF can theoretically be built from verifiable computation.

The construction has, however, two significant downsides: First, in practice computing a SNARG is more than 100,000 times more expensive than evaluating the underlying computation [72]. This means that to achieve meaningful sequentiality, the SNARG computation would require massive parallelism using hundreds thousands of cores. The required parallelism additionally depends on the time $t$. Second, the construction does not achieve $(1 - \epsilon)t$ sequentiality, which is the optimal sequentiality that can be achieved by a construction which involves the evaluation of $f$.

We therefore, now give a VDF construction[3] with required parallelism independent of $t$ and $\sigma$-sequentiality asymptotically close to $(1 - \epsilon)t$ where $\epsilon$ will be defined by the underlying sequential computation.

**Incremental Verifiable Computation (IVC).** IVC provides a direction for circumventing the problem mentioned above. IVC was first studied by Valiant [70] in the context of computationally sound proofs [54]. Bitansky et al. [14] generalized IVC to distributed computations and to other proof systems such as SNARKs. IVC requires that the underlying computation can be expressed as an iterative sequence of evaluations of the same Turing machine. An iterated sequential function satisfies this requirement.

The basic idea of IVC is that at every incremental step of the computation, a prover can produce a proof that a certain state is indeed the current state of the computation. This proof is updated after every step of the computation to produce a new proof. Importantly, the complexity of each proof in proof size and verification cost is bounded by $\mathsf{poly}(\lambda)$ for any sub-exponential length computation. Additionally the complexity of updating the proof is independent of the total length of the computation.

**Towards VDFs from IVC.** Consider a VDF construction that runs a sequential computation and after each step uses IVC to update a proof that both this step and the previous proof were

[3]The construction is largely subsumed by the subsequently added simpler construction in Section 5. This simpler construction is built directly from verifiable computation.

correct. Unfortunately, for IVC that requires knowledge extraction we cannot prove soundness of this construction for $t > O(\lambda)$. The problem is that a recursive extraction yields an extractor that is exponential in the recursion depth [14].

The trick around this is to construct a binary tree of proofs of limited depth [70, 14]. The leaf proofs verify computation steps whereas the internal node proofs prove that their children are valid proofs. The verifier only needs to check the root proof against the statement that all computation steps and internal proofs are correct.

We focus on the special case that the function $f$ is an iterated sequential function. The regularity of the iterated function ensures that the statement that the verifier checks is succinct. We impose a strict requirement on our $IVC$ scheme to output both the output of $f$ and a final proof with only an additive constant number of additional steps over evaluating $f$ alone.

We define *tight* IVC for an iterated sequential functions, which captures the required primitive needed for our theoretical VDF. We require that incremental proving is almost overhead free in that the prover can output the proof almost immediately after the computation has finished. The definition is a special case of Valiant's definition [70].

**Definition 9** (Tight IVC for iterated sequential functions)**.** *Let $f_\lambda : \mathbb{N} \times \mathcal{X} \to \mathcal{X}$ be an iterated sequential function with $(t, \epsilon)$-sequential round function $g_\lambda$ iterated $k = 2^{o(\lambda)}$ times. An IVC system for $f_\lambda$ is a triple of polynomial time algorithms ($\mathsf{IVCGen}, \mathsf{IVCProve}, \mathsf{IVCVerify}$) that satisfy the following properties:*

***Completeness:***

$$\forall x \in \mathcal{X} : \; Pr\left[\mathsf{IVCVerify}(\mathsf{vk}, x, y, k, \pi) = \mathit{Yes} \;\middle|\; \begin{array}{c}(\mathsf{vk}, \mathit{ek}) \xleftarrow{R} \mathsf{IVCGen}(\lambda, f) \\ (y, \pi) \xleftarrow{R} \mathsf{IVCProve}(\mathit{ek}, k, x)\end{array}\right] = 1$$

***Succinctness:*** *The length of a proof and the complexity of $\mathsf{SNKVerify}$ is bounded by $\mathsf{poly}\big(\lambda, \log(k \cdot t)\big)$.*

***Soundness:****[sub-exponential soundness] For all algorithms $\mathcal{A}$ running in time $2^{o(\lambda)}$:*

$$Pr\left[\begin{array}{c}\mathsf{IVCVerify}(\mathsf{vk}, x, y, k, \pi) = \mathit{Yes} \\ f(k, x) \neq y\end{array}\;\middle|\; \begin{array}{l}(\mathsf{vk}, \mathit{ek}) \xleftarrow{R} \mathsf{IVCGen}(\lambda, f) \\ (x, y, k, \pi) \xleftarrow{R} \mathcal{A}(\lambda, \mathsf{vk}, \mathit{ek})\end{array}\right] \leq \mathsf{negl}(\lambda)$$

***Tight Incremental Proving****: There exists a $k'$ such that for all $k \geq k'$ and $k = 2^{o(\lambda)}$, $\mathsf{IVCProve}(\mathit{ek}, k, x)$ runs in parallel time $k \cdot t + O(1)$ using $\mathsf{poly}(\lambda, t)$-processors.*

**Existence of tight IVC.** Bitansky et al. [14] showed that any SNARK system such as [62] can be used to construct IVC. Under strong knowledge of exponent assumptions there exists an IVC scheme using a SNARK tree of depth less than $\lambda$ (Theorem 1 of [14]). In every computation step the prover updates the proof by computing $\lambda$ new SNARKs each of complexity $\mathsf{poly}(\lambda)$, each verifying another SNARK and one of complexity $t$ which verifies one evaluation of $g_\lambda$, the round function of $f_\lambda$. Ben Sasson et al. [10] discuss the parallel complexity of the Pinocchio SNARK [62] and show that for a circuit of size $m$ there exists a parallel prover using $O(m \cdot \log(m))$ processors that computes a SNARK in time $O(\log(m))$. Therefore, using these SNARKs we can construct an IVC proof system ($\mathsf{IVCGen}, \mathsf{IVCProve}, \mathsf{IVCVerify}$) where, for sufficiently large $t$, $\mathsf{IVCProve}$ uses

$\tilde{O}(\lambda+t)$ parallelism to produce each incremental IVC output in time $\lambda \cdot \log(t+\lambda) \leq t$. If $t$ is not sufficiently large, i.e. $t > \lambda \cdot \log(t+\lambda)$ then we can construct an IVC proof system that creates proofs for $k'$ evaluations of $g_\lambda$. The IVC proof system chooses $k'$ such that $t \leq \lambda \cdot \log(k' \cdot t + \lambda)$. Given this the total parallel runtime of $\mathsf{IVCProve}$ on $k$ iterations of an $(t, \epsilon)$-sequential function would thus be $k \cdot t + \lambda \cdot \log(k' \cdot t + \lambda) = k \cdot t + O(1)$. This shows that we can construct tight IVC from existing SNARK constructions.

**$\mathbf{VDF_{IVC}}$ construction.** We now construct a VDF from a tight IVC. By Assumption 1 we are given a family $\{f_\lambda\}$, where each $f_\lambda : \mathbb{N} \times X_\lambda \to X_\lambda$ is defined by $f_\lambda(k, x) = g_\lambda^{(k)}(x)$. Here $g_\lambda$ is a $(s, \epsilon)$-sequential function on an efficiently sampleable domain of size $O(2^\lambda)$.
Given a tight IVC proof system $(\mathsf{IVCGen}, \mathsf{IVCProve}, \mathsf{IVCVerify})$ for $f$ we can construct a VDF that satisfies $\sigma(t)$-sequentiality for $\sigma(t) = (1-\epsilon) \cdot t - O(1)$:

- $\mathsf{Setup}(\lambda, t)$ : Let $g_\lambda$ be a $(t, \epsilon)$-sequential function and $f_\lambda$ the corresponding iterated sequential function as described in Assumption 1. Run $(\mathsf{ek}, \mathsf{vk}) \xleftarrow{R} \mathsf{IVCGen}(\lambda, f_\lambda)$. Set $k$ to be the largest integer such that $\mathsf{IVCProve}(\mathsf{ek}, k, x)$ takes time less than $t$. Output $\mathbf{pp} = \big((\mathsf{ek}, k),\ (\mathsf{vk})\big)$.
- $\mathsf{Eval}((\mathsf{ek}, k), x)$: Run $(y, \pi) \xleftarrow{R} \mathsf{IVCProve}(\mathsf{ek}, k, x)$, output $(y, \pi)$.
- $\mathsf{Verify}(\mathsf{vk}, x, (y, \pi))$: Run and output $\mathsf{IVCVerify}(\mathsf{vk}, x, y, k, \pi)$.

Note that $t$ is fixed in the public parameters. It is, however, also possible to give $t$ directly to $\mathsf{Eval}$. $\mathrm{VDF}_{\mathrm{IVC}}$ is, therefore, *incremental*.

**Lemma 1.** *$VDF_{IVC}$ satisfies soundness (Definition 3)*

*Proof.* Assume that an $\mathsf{poly}(t, \lambda)$ algorithm $\mathcal{A}$ outputs (with non-negligible probability in $\lambda$) a tuple $(x, y, \pi)$ on input $\lambda$, $t$, and $\mathbf{pp} \xleftarrow{R} \mathsf{Setup}(\lambda, t)$ such that $\mathsf{Verify}(\mathbf{pp}, x, y, \pi) = \text{Yes}$ but $f_\lambda(k, x) \neq y$. We can then construct an adversary $\mathcal{A}'$ that violates IVC soundness. Given $(\mathsf{vk}, \mathsf{ek}) \xleftarrow{R} \mathsf{IVCGen}(\lambda, f_\lambda)$ the adversary $\mathcal{A}'$ runs $\mathcal{A}$ on $\lambda, t$, and $(\mathsf{vk}, \mathsf{ek})$. Since $(\mathsf{vk}, \mathsf{ek})$ is sampled from the same distribution as $\mathbf{pp} \xleftarrow{R} \mathsf{Setup}(\lambda, t)$ it follows that, with non-negligible probability in $\lambda$, $\mathcal{A}'$ outputs $(x, y, \pi)$ such that $\mathsf{Verify}(\mathbf{pp}, x, y, \pi) = \mathsf{IVCVerify}(\mathsf{vk}, x, y, k, \pi) = \text{Yes}$ and $f_\lambda(k, x) \neq y$, which directly violates the soundness of IVC. □

**Theorem 1** ($\mathrm{VDF}_{\mathrm{IVC}}$)**.** *$VDF_{IVC}$ is a VDF scheme with $\sigma(t) = (1-\epsilon)t - O(1)$ sequentiality.*

*Proof.* First note that the $\mathrm{VDF}_{\mathrm{IVC}}$ algorithms satisfy the definition of the VDF algorithms. $\mathsf{IVCProve}$ runs in time $(\frac{t}{s} - 1) \cdot s + s = t$ using $\mathsf{poly}(\lambda, s) = \mathsf{poly}(\lambda)$ processors. $\mathsf{IVCVerify}$ runs in total time $\mathsf{poly}(\lambda, \log(t))$. Correctness follows from the correctness of the IVC scheme. Soundness was proved in Lemma 1. The scheme is $\sigma(t)$-sequential because $\mathsf{IVCProve}$ runs in time $k \cdot s + O(1) < t$. If any algorithm that uses $\mathsf{poly}(t, \lambda)$ processors can produce the VDF output in time less than $(1-\epsilon)t - O(1)$ he can directly break the $t, \epsilon$-sequentiality of $f_\lambda$. Since $s$ is independent of $t$ we can conclude that $\mathrm{VDF}_{\mathrm{IVC}}$ has $\sigma(t) = (1-\epsilon)t - O(1)$ sequentiality.
□

# 5 VDFs from Verifiable Computation (added May 2019)

We next present a VDF construction from verifiable computation. This construction is simpler than the one in the previous section in that it does not require tight incremental verifiable computation. Verifiable computation is sufficient.

The core idea is that we can run a sequential computation and, in parallel, compute multiple SNARGs that prove the correctness of different parts of the computation. The protocol we propose computes $\log N$ SNARGs in parallel for segments of geometrically decreasing size.

Let $f$ be an iterated sequential function obtained from iterating a round function $k$ times. As a warmup, assume that constructing a SNARG for an evaluation of $f$ takes exactly the same time as computing the function $f$. The VDF prover first iterates the round function $k/2$ times, to complete half the computation of $f$ on a given input. It then iterates the round function $k/2$ more times, and *in parallel*, computes a SNARG for the first $k/2$ iterations. This way the SNARG computation and the function evaluation will complete at the same time. The prover then continues by constructing a SNARG for the next $k/4$ iterations of the round function, then the next $k/8$ iterations, and so on. All these SNARGs are constructed in parallel to computing the function. The final iteration requires no SNARG as the verifier can check it directly. All these SNARG computations are done in parallel. Using $\log_2(k)$ processors, the evaluation of $f$ and all the SNARG computations will complete at exactly the same time. Hence, constructing the SNARGs adds no time to the evaluation of $f$, but requires $\log_2 k$ parallelism at the evaluator.

We will now describe a more general construction that allows for arbitrary gaps between the SNARG prover time and the function evaluation time. The construction uses an iterated sequential function and a SNARG proof system. However, an analogous construction can be built from any underlying VDF scheme. The construction can amplify the VDF scheme to a "tight" VDF scheme in which the prover outputs the proof concurrently with the output. The amplification has a logarithmic overhead in proof size, verifier time, and prover parallelism. This is described in more detail in concurrent work by Döttling, Garg, Malavolta and Vasudevan[33].

**$\mathbf{VDF_{VC}}$ construction** As in the $\mathrm{VDF_{IVC}}$ we use a family of sequential iterated functions $\{f_\lambda\}$ such that each $f_\lambda : \mathbb{Z} \times X_\lambda \to X_\lambda$ is defined as $f_\lambda(k, x) = g_\lambda^{(k)}(x)$ for an $(s, \epsilon)$ sequential function $g_\lambda$. We are also given a $\mathsf{SNARK}$ systems ($\mathsf{SNKGen}$,$\mathsf{SNKProve}$,$\mathsf{SNKVerify}$) for the family of relations $R_{f_\lambda,k} := \{\big((x, y), \bot\big) : f_\lambda(k, x) = y\}$. The $\mathsf{SNARK}$ only needs to satisfy the soundness definition and not the knowledge extraction so a SNARG suffices. We slightly modify the system compared to Definition 8 by letting $\mathsf{SNKGen}_{f_\lambda,k}$ be the setup for a SNARG corresponding to $R_{f_\lambda,k}$. We also let $\alpha \in \mathbb{R}^+$ denote how much slower the SNARG prover runs compared to the evaluation of $f_\lambda$. That is, if $f_\lambda(k, x)$ can be evaluated in time $t$ then $\mathsf{SNKProve}$ runs in time at most $\alpha \cdot t$ on the same machine. Note that we implicitly require that $\mathsf{SNKProve}$ is a linear algorithm but the construction can easily modified for quasilinear algorithms. $\mathrm{VDF_{VC}}$ works by running the computation of $f_\lambda$ until $t \cdot (\frac{1}{\alpha+1})$ time has passed. That is, compute $\frac{k}{\alpha+1}$ iterations of $g_\lambda$ on the input. Then the prover continues the computation and in parallel computes a SNARG for that first $\frac{1}{\alpha+1}$ fraction of the computation. The same process is repeated using geometrically decreasing parts of the computation. Namely, the prover produce a SNARG for the next $\frac{1}{\alpha+1}$ fraction of the remaining computation and in parallel continue the computation of $f_\lambda$ as well as all other executing SNARG computations. After $\ell$ steps a $\left(\frac{\alpha}{\alpha+1}\right)^\ell$ fraction of the computation remains and $\ell$ SNARGs are being computed in parallel. Thus after $n = \log_{\frac{\alpha}{\alpha+1}}(k)$ steps only 1 invocation of $g$ remains. The verifier can check this invocation directly. For simplicity we assume that $k$ is a power of $\frac{\alpha}{\alpha+1}$. Note that all SNARG computations will finish at the same time, precisely when the computation of $f_\lambda$ is completed.

- $\mathsf{Setup}(\lambda, t)$ : Let $g_\lambda$ be a $(s, \epsilon)$ sequential function. Let $k = \frac{t}{s}$. For $i = 1$ to $n = \log_{\frac{\alpha}{\alpha+1}}(k)$ the

setup generates $(\mathsf{vk}_i, \mathsf{ek}_i) \leftarrow \mathsf{SNKGen}_{f_\lambda, k_i}(\lambda)$, where $k_i$ is defined as

$$k_i = \left( (\frac{\alpha}{\alpha+1})^{i-1} - (\frac{\alpha}{\alpha+1})^{i} \right) \cdot \frac{t}{s}.$$

Output $\mathbf{pp} = \left\{ \left( (\mathsf{ek}_i, k_i),\ \mathsf{vk}_i \right) \right\}_{i=1,\ldots,n}$.

- $\mathsf{Eval}(\mathbf{pp}, x)$ : Let $x_0 = x$. For $i = 1$ to $n$ compute $x_i = g^{(k_i)}(x_{i-1})$ and in parallel start the computation of $\pi_i = \mathsf{SNKProve}(\mathsf{ek}_i, (x_{i-1}, x_i), \perp)$. Finally let $y = g(x_n)$. Output $\{y, (x_1, \pi_1 \ldots, x_n, \pi_n)\}$
- $\mathsf{Verify}(\{\mathsf{vk}_1, \ldots, \mathsf{vk}_n\}, x, (y, (x_1, \pi_1 \ldots, x_n, \pi_n)))$: Verify the proofs by running $\mathsf{SNKVerify}(\mathsf{vk}_i, (x_{i-1}, x_i), \pi_i)$. If any verification fails, reject. Else output 'yes' if $g(x_n) = y$ and reject otherwise.

**Lemma 2** (soundness)**.** *Given a sound SNARG system as defined by Definition 8, $\mathrm{VDF}_{VC}$ satisfies soundness (Definition 3)*

*Proof.* Assume that an $\mathsf{poly}(t, \lambda)$ algorithm $\mathcal{A}$ outputs (with non-negligible probability in $\lambda$) a tuple $(x, y, \pi)$ on input $\lambda$, $t$, and $\mathbf{pp}$ such that $\mathsf{Verify}(\mathbf{pp}, x, y, \pi) = \text{Yes}$ but $f_\lambda(k, x) \neq y$. We can then construct an adversary $\mathcal{A}'$ that violates the SNARG soundness. Note that the proof contains the intermediate computation steps $x_0, \ldots, x_n$ with $x_0 = x$. The verification guarantees that $g(x_n) = y$. However, if $f(x) \neq y$ then there must be an $i \in [0, n-1]$ such that $g^{(k_i)}(x_i) \neq x_{i+1}$ for $k_i = \frac{\alpha^{i-1}}{(\alpha+1)^i} \cdot \frac{t}{s}$. Note that this directly contradicts the soundness of the underlying SNARG. $\mathcal{A}'$ therefore simply runs $\mathcal{A}$ using honestly generated $(\mathsf{vk}_i, \mathsf{ek}_i)$ for all $n$ SNARG proof systems. $\mathcal{A}'$ is able to break at the soundness of at least one of the proof systems simply using the output of $\mathcal{A}$, i.e. $\pi_i$ and $x_i, x_{i+1}$. Since by assumption $\mathcal{A}'$ can only succeed with negligible probability $\mathcal{A}'$ also only succeeds with negligible probability. This shows that $\mathrm{VDF}_{\mathrm{VC}}$ is sound. □

**Theorem 2** ($\mathrm{VDF}_{\mathrm{VC}}$ is a VDF)**.** *Given a $(s, \epsilon)$ sequential function $f_\lambda$ and a SNARG proof system with perfect completeness, $\mathrm{VDF}_{VC}$ is a VDF scheme with $\sigma(t) = (1-\epsilon)t$ sequentiality.*

*Proof.* The algorithms of $\mathrm{VDF}_{\mathrm{VC}}$ satisfy the definition of the VDF algorithms. Treating $\alpha$ and $s$ as constants, the verifier checks only a logarithmic (in $t$) number of succinct proofs and one evaluation of $g_\lambda$. The prover requires $\log(t)$ parallelism for the computation of the proofs. Correctness is immediate from the construction and the completeness of the SNARG. Soundness was proved in Lemma 2. It remains to prove sequentiality. The $\mathsf{Eval}$ algorithm runs exactly in the time that it takes to compute $f_\lambda(\frac{t}{s}, x)$ as all the proof computation runs in parallel and by assumption completes the same moment the computation of $f_\lambda$ completes. Any adversary that can output the VDF value in time less than $(1-\epsilon)t$ will therefore directly break the $(t, \epsilon)$ sequentiality of $f_\lambda$. □

### 5.1 Discussion

We note that both the proof size, the verifier time, and the required parallelism of $\mathrm{VDF}_{\mathrm{VC}}$ are highly dependent on $\alpha$. If $\alpha > 1$ the number of iterations, i.e. the number of proofs and required parallelism is close to $\log(t/s) \cdot \alpha$, i.e. linear in $\alpha$. If $\alpha \approx 100,000$, as is the case for modern SNARGs on arbitrary computations, then this may become prohibitively large. In Section 6 and Section 7 we show how we can boost the construction to significantly reduce the prover

overhead. In particular we construct specific instantiations of $g$ and $f$ that a) are more efficient to verify than to evaluate, allowing a SNARG proof to assert a simpler statement and b) that are specifically optimized for modern SNARG systems. With these optimizations and certain parameter selection it is feasible to bring $\alpha$ closer to 1 or possibly even below 1.

We also note that the same technique of computing several proofs in parallel can be used to boost subsequent VDF constructions such as the RSA based constructions by Pietrzak [64] and Wesolowski [73]. These VDFs, in particular Wesolowski's construction, have a significant prover overhead. This leads to a suboptimal $\sigma(t)$-sequentiality. Using the same technique of computing proofs in parallel we can create "tight" VDFs with only a logarithmic overhead in terms of required parallelism, proof size, and verifier overhead.

# 6 A weak VDF based on injective rational maps

In this section we explore a framework for constructing a weak VDF satisfying $(t^2, o(t))$-sequentiality based on the existence of degree $t$ injective rational maps that cannot be inverted faster than computing polynomial greatest common denominators (GCDs) of degree $t$ polynomials, which we conjecture cannot be solved in parallel time less than $t - o(t)$ on fewer than $t^2$ parallel processors. Our candidate map will be a permutation polynomial over a finite field of degree $t$. The construction built from it is a weak VDF because the Eval will require $O(t)$ parallelism to run in parallel time $t$.

## 6.1 Injective rational maps

**Rational maps on algebraic sets.** An *algebraic rational function* on finite vector spaces is a function $F : \mathbb{F}_q^n \to \mathbb{F}_q^m$ such that $F = (f_1, \ldots, f_m)$ where each $f_i : \mathbb{F}_q^n \to \mathbb{F}_q$ is a rational function in $\mathbb{F}_q(X_1, \ldots, X_n)$, for $i = 1, \ldots, m$. An *algebraic set* $\mathcal{Y} \subseteq \mathbb{F}_q^n$ is the complete set of points on which some set $S$ of polynomials simultaneously vanish, i.e. $\mathcal{Y} = \{x \in \mathbb{F}_q^n | f(x) = 0 \text{ for all } f \in S\}$ for some $S \subset \mathbb{F}_q[X_1, \ldots, X_n]$. An *injective rational map* of algebraic sets $\mathcal{Y} \subseteq \mathbb{F}_q^n$ to $\mathcal{X} \subseteq \mathbb{F}_q^m$ is an algebraic rational function $F$ that is injective on $\mathcal{Y}$, i.e. if $\mathcal{X} := F(\mathcal{Y})$, then for every $\bar{x} \in \mathcal{X}$ there exists a unique $\bar{y} \in \mathcal{Y}$ such that $F(\bar{y}) = \bar{x}$.

**Inverting rational maps.** Consider the problem of inverting an injective rational map $F = (f_1, ...., f_m)$ on algebraic sets $\mathcal{Y} \subseteq \mathbb{F}_q^n$ to $\mathcal{X} \subseteq \mathbb{F}_q^m$. Here $\mathcal{Y} \subseteq \mathbb{F}_q^n$ is the set of vanishing points of some set of polynomials $S$. For $x \in \mathbb{F}_q^m$, a solution to $F(\bar{y}) = \bar{x}$ is a point $\bar{y} \in \mathbb{F}_q^n$ such that all polynomials in $S$ vanish at $\bar{y}$ and $f_i(\bar{y}) = x_i$ for $i = 1, ..., m$. Furthermore, each $f_i(\bar{y}) = g(\bar{y})/h(\bar{y}) = x_i$ for some polynomials $g, h$, and hence yields a polynomial constraint $z_i(\bar{y}) := g(\bar{y}) - x_i h(\bar{y}) = 0$. In total we are looking for solutions to $|S| + m$ polynomial constraints on $\bar{y}$.

We illustrate two special cases of injective rational maps that can be inverted by a univariate polynomial GCD computation. In general, inverting injective rational maps on $\mathbb{F}_q^d$ for constant $d$ can be reduced to a univariate polynomial GCD computation using resultants.

- *Rational functions on finite fields.* Consider any injective rational function $F(X) = g(X)/h(X)$, for univariate polynomials $h, g$, on a finite field $\mathbb{F}_q$. A finite field is actually a special case of an algebraic set over itself; it is the set of roots of the polynomial $X^q - X$. Inverting $F$ on a

point $c \in \mathbb{F}_q$ can be done by calculating $GCD(X^q - X, g(X) - c \cdot h(X))$, which outputs $X - s$ for the unique $s$ such that $F(s) = c$.

- *Rational maps on elliptic curves.* An elliptic curve $E(\mathbb{F}_q)$ over $\mathbb{F}_q$ is a 2-dimensional algebraic set of vanishing points in $\mathbb{F}_q^2$ of a bivariate polynomial $E(y, x) = y^2 - x^3 - ax - b$. Inverting an injective rational function $F$ on a point in the image of $F(E(\mathbb{F}_q))$ involves computing the GCD of three bivariate polynomials: $E, z_1, z_2$, where $z_1$ and $z_2$ come from the two rational function components of $F$. The resultant $R = Res_y(z_1, z_2)$ is a univariate polynomial in $x$ of degree $deg(z_1) \cdot deg(z_2)$ such that $R(x) = 0$ iff there exists $y$ such that $(x, y)$ is a root of both $z_1$ and $z_2$. Finally, taking the resultant again $R' = Res_y(R, E)$ yields a univariate polynomial such that any root $x$ of $R'$ has a corresponding coordinate $y$ such that $(x, y)$ is a point on $E$ and satisfies constraints $z_1$ and $z_2$. Solving for the unique root of $R'$ reduces to a Euclidean GCD computation as above. Then given $x$, there are two possible points $(x, y) \in E$, so we can try them both and output the unique point that satisfies all the constraints.

**Euclidean algorithm for univariate polynomial GCD.** Univariate polynomials over a finite field form a Euclidean domain, and therefore the GCD of two polynomials can be found using the Euclidean algorithm. For two polynomials $f$ and $g$ such that $deg(f) > deg(g) = d$, one first reduces $f$ mod $g$ and then computes $GCD(f, g) = GCD(f \bmod g, g)$. In the example $f = X^q - X$, the first step of reducing $X^q$ mod $g$ requires $O(\log(q))$ multiplications of degree $O(deg(g))$ polynomials. Starting with $X$, we run the sequence of repeated squaring operations to get $X^q$, reducing the intermediate results mod $g$ after each squaring operation. Then running the Euclidean algorithm to find $GCD(f \bmod g, g)$ involves $O(d)$ sequential steps where in each step we subtract two $O(d)$ degree polynomials. On a sequential machine this computation takes $O(d^2)$ time, but on $O(d)$ parallel processors this can be computed in parallel time $O(d)$.

**NC algorithm for univariate polynomial GCD.** There is an algorithm for computing the GCD of two univariate polynomials of degree $d$ in $O(\log^2(d))$ parallel time, but requires $O(d^{3.8})$ parallel processors. This algorithm runs $d$ parallel determinant calculations on submatrices of the Sylvester matrix associated with the two polynomials, each of size $O(d^2)$. Each determinant can be computed in parallel time $O(\log^2(d))$ on $M(d) \in O(d^{2.85})$ parallel processors [25]. The parallel advantage of this method over the euclidean GCD method kicks in after $O(d^{2.85})$ processors. For any $c \leq d/\log^2(d)$, it is possible to compute the GCD in $O(d/c)$ steps on $c\log^2(d)M(d)$ processors.

**Sequentiality of univariate polynomial GCD.** The GCD can be calculated in parallel time $d$ using $d$ parallel processors via the Euclidean algorithm. The NC algorithm only beats this bound on strictly greater than $d^{2.85}$ processors, but a hybrid of the two methods can gain an $o(d)$ speedup on only $d^2$ processors. Specifically, we can run the Euclidean method for $d - d^{2/3}$ steps until we are left with two polynomials of degree $d^{2/3}$, then we can run the NC algorithm using $\log^3(d)M(d^{2/3}) < (d^{2/3})^3 = d^2$ processors to compute the GCD of these polynomials in $O(d^{2/3}/\log(d))$ steps, for a total of $d - \epsilon d^{2/3}$ steps. This improvement can be tightened further, but generally results in $d - o(d)$ steps as long as $M(d) \in \omega(d^2)$.

We pose the following assumption on the parallel complexity of calculating polynomials GCDs on fewer that $O(d^2)$ processors. This assumption would be broken if there is an NC

algorithm for computing the determinant of a $n \times n$ matrix on $o(n^2)$ processors, but this would require a significant advance in mathematics on a problem that has been studied for a long time.

**Assumption 2.** *There is no general algorithm for computing the GCD of two univariate polynomials of degree $d$ over a finite field $\mathbb{F}_q$ (where $q > d^3$) in less than parallel time $d - o(d)$ on $O(d^2)$ parallel processors.*

On the other hand, evaluating a polynomial of degree $d$ can be logarithmic in its degree, provided the polynomial can be expressed as a small arithmetic circuit, e.g. $(ax + b)^d$ can be computed with $O(\log(d))$ field operations.

**Abstract weak VDF from an injective rational map.** Let $F : \mathbb{F}_q^n \to \mathbb{F}_q^m$ be a rational function that is an injective map from $\mathcal{Y}$ to $\mathcal{X} := F(\mathcal{Y})$. We further require that $\mathcal{X}$ is efficiently sampleable and that $F$ can be evaluated efficiently for all $\bar{y} \in \mathcal{Y}$. When using $F$ in a VDF we will require that $|\mathcal{X}| > \lambda t^3$ to prevent brute force attacks, where $t$ and $\lambda$ are given as input to the Setup algorithm.

We will need a family $\mathcal{F} := \{(q, F, \mathcal{X}, \mathcal{Y})\}_{\lambda,t}$ parameterized by $\lambda$ and $t$. Given such a family we can construct a weak VDF as follows:

- $\mathsf{Setup}(\lambda, t)$: choose a $(q, F, \mathcal{X}, \mathcal{Y}) \in \mathcal{F}$ specified by $\lambda$ and $t$, and output $\mathbf{pp} := ((q, F), (q, F))$.
- $\mathsf{Eval}((q, F), \bar{x})$: for an output $\bar{x} \in \mathcal{X} \subseteq \mathbb{F}_q^m$ compute $\bar{y} \in \mathcal{Y}$ such that $F(\bar{y}) = \bar{x}$; The proof $\pi$ is empty.
- $\mathsf{Verify}((q, F), \bar{x}, \bar{y}, \pi)$ outputs Yes if $F(\bar{y}) = \bar{x}$.

The reason we require that $F$ be injective on $\mathcal{Y}$ is so that the solution $\bar{y}$ be unique.
The construction is a weak $(p(t), \sigma(t))$-VDF for $p(t) = t^2$ and $\sigma(t) = t - o(t)$ assuming that there is no algorithm that can invert of $F \in \mathcal{F}$ on a random value in less than parallel time $d - o(d)$ on $O(d^2)$ processors. Note that this is a stronger assumption than 2 as the inversion reduces to a specific GCD computation rather than a general one.

**Candidate rational maps.** The question, of course, is how to instantiate the function family $\mathcal{F}$ so that the resulting weak VDF system is secure. There are many examples of rational maps on low dimensional algebraic sets among which we can search for candidates. Here we will focus on the special case of efficiently computable permutation polynomials over $\mathbb{F}_q$, and one particular family of permutation polynomials that may be suitable.

### 6.2 Univariate permutation polynomials

The simplest instantiation of the VDF system above is when $n = m = 1$ and $\mathcal{Y} = \mathbb{F}_q$. In this case, the function $F$ is a univariate polynomial $f : \mathbb{F}_q \to \mathbb{F}_q$. If $f$ implements an injective map on $\mathbb{F}_q$, then it must be a permutation of $\mathbb{F}_q$, which brings us to the study of *univariate permutation polynomials* as VDFs.

The simplest permutation polynomials are the monomials $x^e$ for $e \geq 1$, where $gcd(e, q-1) = 1$. These polynomials however, can be easily inverted and do not give a secure VDF. Dickson polynomials [50] $D_{n,\alpha} \in \mathbb{F}_p[x]$ are another well known family of polynomials over $\mathbb{F}_p$ that permute $\mathbb{F}_p$. Dickson polynomials are defined by a recurrence relation and can be evaluated efficiently.

Dickson polynomials satisfy $D_{t,\alpha^n}(D_{n,\alpha}(x)) = x$ for all $n, t, \alpha$ where $n \cdot t = 1 \bmod p-1$, hence they are easy to invert over $\mathbb{F}_p$ and again do not give a secure VDF.

A number of other classes of permutation polynomials have been discovered over the last several decades [40]. We need a class of permutation polynomials over a suitably large field that have a tunable degree, are fast to evaluate (i.e. have polylog($d$) circuit complexity), and cannot be inverted faster than running the parallelized Euclidean algorithm on O(d) processors.

**Candidate permutation polynomial.** We consider the following polynomial of Guralnick and Muller [39] over $\mathbb{F}_{p^m}$:

$$\frac{(x^s - ax - a) \cdot (x^s - ax + a)^s + ((x^s - ax + a)^2 + 4a^2x)^{(s+1)/2}}{2x^s} \tag{6.1}$$

where $s = p^r$ for odd prime $p$ and $a$ is not a $(s-1)st$ power in $\mathbb{F}_{p^m}$. This polynomial is a degree $s^3$ permutation on the field $\mathbb{F}_{p^m}$ for all $s, m$ chosen independently.

Below we discuss why instantiating a VDF with nearly all other examples of permutation polynomials would not be secure and why attacks on these other polynomials do not work against this candidate.

**Attacks on other families of permutation polynomials.** We list here several other families of permutation polynomials that can be evaluated in $O(\text{polylog}(d))$ time, yet would not yield a secure VDF. We explain why each of these attacks do not work against the candidate polynomial.

1. *Sparse permutation polynomials.* Sparse polynomials have a constant number of terms and therefore can be evaluated in time $O(\log(d))$. There exist families of non-monomial sparse permutation polynomials, e.g. $X^{2^{t+1}+1} + X^3 + X \in \mathbb{F}_{2^{2t+1}}[X]$ [40, Thm 4.12]. The problem is that the degree of this polynomial is larger than the square root of the field size, which allows for brute force parallel attacks. Unfortunately, all known sparse permutation polynomials have this problem. In our candidate the field size can be made arbitrarily large relative to the degree of the polynomial.

2. *Linear algebraic attacks.* A classic example of a sparse permutation polynomial of tunable degree over an arbitrarily large field, due to Mathieu [34], is the family $x^{p^i} - ax$ over $\mathbb{F}_{p^m}$ where $a$ is not a $p-1$st power. Unfortunately, this polynomial is easy to invert because $x \mapsto x^{p^i}$ is a linear operator in characteristic $p$ so the polynomial can be written as a linear equation over an $m$-dimensional vector space. To prevent linear algebraic attacks the degree of at least one non-linear term in the polynomial cannot be divisible by the field characteristic $p$. In our candidate there are many such non-linear terms, e.g. of degree $s+1$ where $s = p^r$.

3. *Exceptional polynomials co-prime to characteristic.* An *exceptional polynomial* is a polynomial $f \in \mathbb{F}_q[X]$ which is a permutation on $\mathbb{F}_{q^m}$ for infinitely many $m$, which allows us to choose sufficiently large $m$ to avoid brute force attacks. Any permutation polynomial of degree at most $q^{1/4}$ over $F_q$ is exceptional [74]. Since we want $q$ to be exponential in the security parameter and the degree to be sub-exponential we can restrict the search for candidate polynomials to exceptional polynomials. However, all exceptional polynomials

over $\mathbb{F}_q$. of degree co-prime to $q$ can be written as the composition of Dickson polynomials and linear polynomials, which are easy to invert [59]. In our candidate, the degree $s^3$ of the polynomial and field size are both powers of $p$, and are therefore not co-prime.

**Additional application: a new family of one-way permutations.** We note that a sparse permutation polynomial of sufficiently high degree over a sufficiently large finite field may be a good candidate for a one-way permutation. This may give a secure one-way permutation over a domain of smaller size than what is possible by other methods.

### 6.3 Comparison to square roots mod p

A classic approach to designing a sequentially slow verifiable function, dating back to Dwork and Naor [31], is computing modular square roots. Given a challenge $x \in \mathbb{Z}_p^*$, computing $y = x^{\frac{p+1}{4}} \pmod p$ can be efficiently verified by checking that $y^2 = x \pmod p$ (for $p \equiv 3 \pmod 4$). There is no known way to compute this exponentiation in faster than $\log(p)$ sequential field multiplications.

This is a special case of inverting a rational function over a finite field, namely the polynomial $f(y) = y^2$, although this function is not injective and therefore cannot be calculated with GCDs. An injective rational function with nearly the same characteristics is the permutation $f(y) = y^3$. Since the inverse of 3 mod $p-1$ will be $O(\log p)$ bits, this requires $O(\log p)$ squaring operations to invert. Viewed another way, this degree 3 polynomial can be inverted on a point $c$ by computing the GCD$(y^p - y, y^2 - c)$, where the first step requires reducing $y^p - y$ mod $y^3 - c$, involving $O(\log p))$ repeated squarings and reductions mod $y^3 - c$.

While this approach appears to offer a delay parameter of $t = \log(p)$, as $t$ grows asymptotically the evaluator can use $O(t)$ parallel processors to gain a factor $t$ parallel speedup in field multiplications, thus completing the challenge in parallel time equivalent to one squaring operation on a sequential machine. Therefore, there is asymptotically no difference in the parallel time complexity of the evaluation and the total time complexity of the verification, which is why this does not even meet our definition of a weak VDF. Our approach of using higher degree injective rational maps gives a strict (asymptotic) improvement on the modular square/cubes approach, and to the best of our knowledge is the first concrete algebraic candidate to achieve an exponential gap between parallel evaluation complexity and total verification complexity.

## 7 Practical improvements on VDFs from IVC

In this section we propose a practical boost to constructing VDFs from IVC (Section 4). In an IVC construction the prover constructs a SNARK which verifies a SNARK. Ben-Sasson et al. [11] showed an efficient construction for IVC using "cycles of Elliptic curves". This construction builds on the pairing-based SNARK [62]. This SNARK system operates on arithmetic circuits defined over a finite field $\mathbb{F}_p$. The proof output consists of elements of an elliptic curve group $E/\mathbb{F}_q$ of prime order $p$ (defined over a field $\mathbb{F}_q$). The SNARK verification circuit, which computes a pairing, is therefore an arithmetic circuit over $\mathbb{F}_q$. Since $q \neq p$, the prover cannot construct a new SNARK that directly operates on the verification circuit, as the SNARK operates on circuits defined over $\mathbb{F}_p$. Ben-Sasson et. al. propose using two SNARK systems where the curve order of one is equal to the base field of the other, and vice versa. This requires finding a pair

of pairing-friendly elliptic curves $E_1, E_2$ (defined over two different base fields $\mathbb{F}_1$ and $\mathbb{F}_2$) with the property that the order of each curve is equal to the size of the base field of the other.

The main practical consideration in $\text{VDF}_{\text{IVC}}$ is that the evaluator needs to be able to update the incremental SNARK proofs at the same rate as computing the underlying sequential function, and without requiring a ridiculous amount of parallelism to do so. Our proposed improvements are based on two ideas:

1. In current SNARK/IVC constructions (including [62], [11]) the prover complexity is proportional to the multiplicative arithmetic complexity of the underlying statement over the field $\mathbb{F}_p$ used in the SNARK ($p \approx 2^{128}$). Therefore, as an optimization, we can use a "SNARK friendly" hash function (or permutation) as the iterated sequential function such that the verification of each iteration has a lower multiplicative arithmetic complexity over $\mathbb{F}_p$.

2. We can use the Eval of a weak VDF as the iterated sequential function, and compute a SNARK over the Verify circuit applied to each incremental output instead of the Eval circuit. This should increase the number of sequential steps required to evaluate the iterated sequential function relative to the number of multiplication gates over which the SNARK is computed.

An improvement of type (1) alone could be achieved by simply using a cipher or hash function that has better multiplicative complexity over the SNARK field $\mathbb{F}_q$ than AES or SHA256 (e.g., see MiMC [5], which has 1.6% complexity of AES). We will explain how using square roots in $\mathbb{F}_q$ or a suitable permutation polynomial over $\mathbb{F}_q$ (from Section 6) as the iterated function achieve improvements of both types (1) and (2).

### 7.1 Iterated square roots in $\mathbb{F}_q$

**Sloth** A recent construction called Sloth [48] proposed a secure way to chain a series of square root computations in $\mathbb{Z}_p$ interleaved with a simple permutation [4] such that the chain must be evaluated sequentially, i.e. is an iterated sequential function (Definition 7). More specifically, Sloth defines two permutations on $\mathbb{F}_p$: a permutation $\rho$ such that $\rho(x)^2 = \pm x$, and a permutation $\sigma$ such that $\sigma(x) = x \pm 1$ depending on the parity of $x$. The parity of $x$ is defined as the integer parity of the unique $\hat{x} \in \{0, ..., p-1\}$ such that $\hat{x} = x \mod p$. Then Sloth iterates the permutation $\tau = \rho \circ \sigma$.

The verification of each step in the chain requires a single multiplication over $\mathbb{Z}_p$ compared to the $O(\log(p))$ multiplications required for evaluation. Increasing the size of $p$ amplifies this gap, however it also introduces an opportunity for parallelizing multiplication in $\mathbb{Z}_p$ for up to $O(\log(p))$ speedup.

Using Sloth inside $\text{VDF}_{\text{IVC}}$ would only achieve a practical benefit if $p = q$ for the SNARK field $\mathbb{F}_q$, as otherwise implementing multiplication in $\mathbb{Z}_p$ in an arithmetic circuit over $\mathbb{F}_q$ would have $O(\log^2(p))$ complexity. On modern architectures, multiplication of integers modulo a 256-bit prime is near optimal on a single core, whereas multi-core parallelized algorithms only offer speed-ups for larger primes [8]. Computing a single modular square root for a 256-bit prime

[4] If square roots are iterated on a value $x$ without an interleaved permutation then there is a shortcut to the iterated computation that first computes $v = (\frac{p+1}{4})^\ell \mod p$ and then the single exponentiation $x^v$.

takes approximately 45,000 cycles[5] on an Intel Core i7 [48], while computing SHA256 for 256-bit outputs takes approximately 864 cycles[6].

The best known arithmetic circuit implementation of SHA256 has 27,904 multiplication gates[9]. In stark contrast, the arithmetic circuit over $\mathbb{F}_p$ for verifying a modular square root is a single multiplication gate. Verifying the permutation $\sigma$ is more complex as it requires a parity check, but this requires at most $O(\log(p))$ complexity.

**Sloth++ extension** Replacing SHA256 with Sloth as the iterated function in $\text{VDF}_{\text{IVC}}$ already gives a significant improvement, as detailed above. Here we suggest yet a further optimization, which we call Sloth++. The main arithmetic complexity of verifying a step of Sloth comes from the fact that the permutation $\sigma$ is not naturally arithmetic over $\mathbb{F}_p$, which was important for preventing attacks that factor $\tau^\ell(x)$ as a polynomial over $\mathbb{F}_p$. Our idea here is to compute square roots over a degree 2 extension field $\mathbb{F}_{p^2}$ interleaved with a permutation that is arithmetic over $\mathbb{F}_p$ but not over $\mathbb{F}_{p^2}$.

In any degree $r$ extension field $\mathbb{F}_{p^r}$ of $\mathbb{F}_p$ for a prime $p = 3 \mod 4$ a square root of an element $x \in \mathbb{F}_{p^r}$ can be found by computing $x^{(p^r+1)/4}$. This is computed in $O(r\log(p))$ repeated squaring operations in $\mathbb{F}_p^r$. Verifying a square root requires a single multiplication over $\mathbb{F}_{p^r}$. Elements of $\mathbb{F}_{p^r}$ can be represented as length $r$ vectors over $\mathbb{F}_p$, and each multiplication reduces to $O(r^2)$ arithmetic operations over $\mathbb{F}_p$. For $r = 2$ the verification *multiplicative* complexity over $\mathbb{F}_p$ is exactly 4 gates.

In Sloth++ we define the permutation $\rho$ exactly as in Sloth, yet over $\mathbb{F}_{p^2}$. Then we define a simple non-arithmetic permutation $\sigma$ on $\mathbb{F}_{p^2}$ that swaps the coordinates of elements in their vector representation over $\mathbb{F}_p$ and adds a constant, i.e. maps the element $(x, y)$ to $(y+c_1, x+c_2)$. The arithmetic circuit over $\mathbb{F}_p$ representing the swap is trivial: it simply swaps the values on the input wires. The overall multiplicative complexity of verifying an iteration of Sloth++ is only 4 gates over $\mathbb{F}_p$. Multiplication can be parallelized for a factor 2 speedup, so 4 gates must be verified roughly every 89,000 parallel-time evaluation cycles. Thus, even if an attacker could manage to speedup the modular square root computation by a factor 100 using an ASIC designed for 256-bit multiplication, for parameters that achieve the same wall-clock delay, the SNARK verification complexity of Sloth++ is over a 7,000 fold improvement over that of a SHA256 chain.

**Cube roots** The underlying permutation in both Sloth and Sloth++ can be replaced by cube roots over $\mathbb{F}_q$ when $gcd(3, q-1) = 1$. In this case the slow function is computing $\rho(x) = x^v$ where $3v = 1 \mod q-1$. The output can be verified as $\rho(x)^3 = x$.

## 7.2 Iterated permutation polynomials

Similar to Sloth+, we can use our candidate permutation polynomial (Equation 6.1) over $\mathbb{F}_q$ as the iterated function in $\text{VDF}_{\text{IVC}}$. Recall that $\mathbb{F}_q$ is an extension field chosen independently from the degree of the polynomial. We would choose $q \approx 2^{256}$ and use the same $\mathbb{F}_q$ as the field used for the SNARK system. For each $O(d)$ sequential provers steps required to invert the polynomial on a point, the SNARK only needs to verify the evaluation of the polynomial on the inverse,

[5] This is extrapolated from [48], which reported that 30 million iterations of a modular square root computation for a 256-bit prime took 10 minutes on a single 2.3 GHz Intel Core i7.

[6] http://www.ouah.org/ogay/sha2/

which has multiplicative complexity $O(\log(d))$ over $\mathbb{F}_q$. Concretely, for each $10^5$ parallel-time evaluation cycles a SNARK needs to verify approximately 16 gates. This is yet another factor 15 improvement over Sloth+. The catch is that the evaluator must use $10^5$ parallelism[7] to optimize the polynomial GCD computation. We must also assume that an adversary cannot feasibly amass more than $10^{14}$ parallel processors to implement the NC parallelized algorithm for polynomial GCD.

From a theory standpoint, using permutation polynomials inside $\text{VDF}_{\text{IVC}}$ reduces it to a weak VDF because the degree of the polynomial must be super-polynomial in $\lambda$ to prevent an adversary from implementing the NC algorithm on $poly(\lambda)$ processors, and therefore the honest evaluator is also required to use super-polynomial parallelism. However, the combination does yield a better weak VDF, and from a practical standpoint appears quite promising for many applications.

# 8 Towards VDFs from exponentiation in a finite group

The sequential nature of large exponentiation in a finite group may appear to be a good source for secure VDF systems. This problem has been used extensively in the past for time-based problems such as time-lock puzzles [68], benchmarking [21], timed commitments [16], and client puzzles [31, 48]. Very recently, Pietrzak [65] showed how to use this problem to construct a VDF that requires a trusted setup. The trusted setup can be eliminated by instead choosing a sufficiently large random number $N$ so that $N$ has two large prime factors with high probability. However, the large size of $N$ provides the adversary with more opportunity for parallelizing the arithmetic. It also increases the verifier's running time. Alternatively, one can use the class group of an imaginary quadratic order [20], which is an efficient group of unknown order with a public setup [51].

## 8.1 Exponentiation-based VDFs with bounded pre-computation

Here we suggest a simple exponentiation-based approach to constructing VDFs whose security would rely on the assumption that the adversary cannot run a long pre-computation between the time that the public parameters **pp** are made public and the time when the VDF needs to be evaluated. Therefore, in terms of security this construction is subsumed by the more recent solution of Pietrzak [65], however it yields much shorter proofs. We use the following notation to describe the VDF:

- let $L = \{\ell_1, \ell_2, \ldots, \ell_t\}$ be the first $t$ odd primes, namely $\ell_1 = 3$, $\ell_2 = 5$, etc. Here $t$ is the provided delay parameter.
- let $P$ be the product of the primes in $L$, namely $P := \ell_1 \cdot \ell_2 \cdots \ell_t$. This $P$ is a large integer with about $t \log t$ bits.

With this notation, the trusted setup procedure works as follows: construct an RSA modulus $N$, say 4096 bits long, where the prime factors are strong primes. The trusted setup algorithm knows the factorization of $N$, but no one else will. Let $\mathbb{G} := (\mathbb{Z}/N\mathbb{Z})^*$. We will also need a random hash function $H : \mathbb{Z} \to \mathbb{G}$.
Next, for a given preprocessing security parameter $B$, say $B = 2^{30}$, do:

[7] This is reasonable if the evaluator has an NVIDIA Titan V GPU, which can compute up to $10^{14}$ pipelined arithmetic operations per second (https://www.nvidia.com/en-us/titan/titan-v/).

– for $i = 1, \ldots, B$: compute $h_i \leftarrow H(i) \in \mathbb{G}$ and then compute $g_i := h_i^{1/P} \in \mathbb{G}$.
– output

$$\mathsf{ek} := (\mathbb{G},\ H,\ g_1, \ldots, g_B) \qquad \text{and} \qquad \mathsf{vk} := (\mathbb{G},\ H).$$

Note that the verifier's public parameters are short, but the evaluators parameters are not.

Solving a challenge $x$: Algorithm $\mathsf{Eval}(\mathbf{pp}_{\text{eval}}, x)$ takes as input the public parameters $\mathbf{pp}_{\text{eval}}$ and a challenge $x \in \mathcal{X}$.
– using a random hash function, map the challenge $x$ to a random subset $L_x \subseteq L$ of size $\lambda$, and a random subset $S_x$ of $\lambda$ values in $\{1, \ldots, B\}$.
– Let $P_x$ be the product of all the primes in $L_x$, and let $g$ be $g := \prod_{i \in S_x} g_i \in \mathbb{G}$.
– the challenge solution $y$ is simply $y \leftarrow g^{P/P_x} \in \mathbb{G}$, which takes $O(t \log t)$ multiplications in $\mathbb{G}$.

Verifying a solution $y$: Algorithm $\mathsf{Verify}(\mathbf{pp}_{\text{verify}}, x, y)$ works as follows:
– Compute $P_x$ and $S_x$ as in algorithm $\mathsf{Eval}(\mathbf{pp}_{\text{eval}}, x)$.
– let $h$ be $h := \prod_{i \in S_x} H(i) \in \mathbb{G}$.
– output yes if and only if $y^{P_x} = h$ in $\mathbb{G}$.
Note that exactly one $y \in \mathbb{G}$ will be accepted as a solution for a challenge $x$. Verification takes only $\tilde{O}(\lambda)$ group operations.

**Security.** The scheme does not satisfy the definition of a secure VDF, but may still be useful for some of the applications described in Section 2. In particular, the system is not secure against an adversary who can run a large pre-computation once the parameters $\mathbf{pp}$ are known. There are several pre-computation attacks possible that require $tB$ group operations in $\mathbb{G}$. Here we describe one such instructive attack. It uses space $O(sB)$, for some $s > 0$, and gives a factor of $s$ speed up for evaluating the VDF.

Consider the following pre-computation, for a given parameter $s$, say $s = 100$. Let $b = \lfloor P^{1/s} \rfloor$, then the adversary computes and stores a table of size $sB$:

$$\text{for all } i = 1, \ldots, B: \qquad g_i^b,\ g_i^{(b^2)},\ \ldots,\ g_i^{(b^s)} \quad \in \mathbb{G}. \tag{8.1}$$

Computing these values is comparable to solving $B$ challenges. Once computed, to evaluate the VDF at input $x$, the adversary uses the precomputed table to quickly compute

$$g^b,\ g^{(b^2)},\ \ldots,\ g^{(b^s)} \in \mathbb{G}.$$

Now, to compute $g^{P/P_x}$, it can write $P/P_x$ in base $b$ as:
$P/P_x = \alpha_0 + \alpha_1 b + \alpha_2 b^2 + \ldots + \alpha_s b^s$ so that

$$g^{P/P_x} = g^{\alpha_0} \cdot (g^b)^{\alpha_1} \cdot (g^{(b^2)})^{\alpha_2} \cdots (g^{(b^s)})^{\alpha_s}.$$

This expression can be evaluated in parallel and gives a parallel adversary a factor of $s$ speed-up over a sequential solver, which violates the sequentiality property of the VDF.

To mount this attack, the adversary must compute the entire table (8.1) for all $g_1, \ldots, g_B$, otherwise it can only gain a factor of two speed-up with negligible probability in $\lambda$. Hence, the scheme is secure for only $B$ challenges, after which new public parameters need to be generated. This may be sufficient for some applications of a VDF.

# 9 Related work

Taking a broad perspective, VDFs can be viewed as an example of *moderately hard* cryptographic functions. Moderately hard functions are those whose difficulty to compute is somewhere in between 'easy' (designed to be as efficient as possible) and 'hard' (designed to be so difficult as to be intractable). The use of moderately hard cryptographic functions dates back at least to the use of a deliberately slow DES variant for password hashing in early UNIX systems [58]. Dwork and Naor [31] coined the term *moderately hard* in a classic paper proposing client puzzles or "pricing functions" for the purpose of preventing spam. Juels and Brainard proposed the related notion of a *client puzzle*, in which a TCP server creates a puzzle which must be solved before a client can open a connection [44]. Both concepts have been studied for a variety of applications, including TLS handshake requests [7, 29], node creation in peer-to-peer networks [30], creation of digital currency [67, 27, 60] or censorship resistance [18]. For interactive client puzzles, the most common construction is as follows: the server chooses a random $\ell$-bit value $x$ and sends to the client $\mathrm{H}(x)$ and $x[\ell - \log_2 t - 1]$. The client must send back the complete value of $x$. That is, the server sends the client $\mathrm{H}(x)$ plus all of the bits of $x$ except the final $\log_2 t + 1$ bits, which the client must recover via brute force.

## 9.1 Inherently sequential puzzles

The simple interactive client puzzle described above is embarrassingly parallel and can be solved in constant time given $t$ processors. In contrast, the very first construction of a client puzzle proposed by Dwork and Naor involved computing modular square roots and is believed to be inherently sequential (although they did not discuss this as a potential advantage).

The first interest in designing puzzles that require an inherently sequential solving algorithm appears to come for the application of hardware benchmarking. Cai et al. [21, 22] proposed the use of inherently sequential puzzles to verify claimed hardware performance as follows: a customer creates an inherently-sequential puzzle and sends it to a hardware vendor, who then solves it and returns the solution (which the customer can easily verify) as quickly as possible. Note that this work predated the definition of client puzzles. Their original construction was based on exponentiation modulo an RSA number $N$, for which the customer has created $N$ and therefore knows $\varphi(N)$. They later proposed solutions based on a number of other computational problems not typically used in cryptography, including Gaussian elimination, fast Fourier transforms, and matrix multiplication.

**Time-lock puzzles** Rivest, Shamir, and Wagner [68] constructed a time-lock encryption scheme, also based on the hardness of RSA factoring and the conjectured sequentiality of repeated exponentiation in a group of unknown order. The encryption key $K$ is derived as $K = x^{2^t} \in \mathbb{Z}_N$ for an RSA modulus $N$ and a published starting value $x$. The encrypting party, knowing $\varphi(N)$, can reduce the exponent $e = 2^t \bmod \varphi(N)$ to quickly derive $K = x^e \bmod N$. The key $K$ can be publicly recovered slowly by $2^t$ iterated squarings. Boneh and Naor [16] showed that the puzzle creator can publish additional information enabling an efficient and sound proof that $K$ is correct. In the only alternate construction we are aware of, Bitansky et al. [15] show how to construct time-lock puzzles from randomized encodings assuming any inherently-sequential functions exist.

Time-lock puzzles are similar to VDFs in that they involve computing an inherently sequential function. However, time-lock puzzles are defined in a private-key setting where the verifier uses its private key to prepare each puzzle (and possibly a verification proof for the eventual answer). In contrast to VDFs, this trusted setup must be performed per-puzzle and each puzzle takes no unpredictable input.

**Proofs of sequential work** Mahmoody et al.[52] proposed publicly verifiable proofs of sequential work (PoSW) which enable proving to any challenger that a given amount of sequential work was performed on a specific challenge. As noted, time-lock puzzles are a type of PoSW, but they are not publicly verifiable. VDFs can be seen as a special case of publicly verifiable proofs of sequential work with the additional guarantee of a unique output (hence the use of the term "function" versus "proof").

Mahmoody et al.'s construction uses a sequential hash function $H$ (modeled as a random oracle) and depth robust directed-acyclic graph $G$. Their puzzle involves computing a *labeling* of $G$ using $H$ salted by the challenge $c$. The label on each node is derived as a hash of all the labels on its parent nodes. The labels are committed to in a Merkle tree and the proof involves opening a randomly sampled fraction. Very briefly, the security of this construction is related to graph pebbling games (where a pebble can be placed on a node only if all its parents already have pebbles) and the fact that depth robust graphs remain sequentially hard to pebble even if a constant fraction of the nodes are removed (in this case corresponding to places where the adversary cheats). Mahmoody et. al. proved security unconditionally in the random oracle model. Depth robust graphs and parallel pebbling hardness are use similarly to construct memory hard functions [42] and proofs of space [32]. Cohen and Pietrzak [19] constructed a similar PoSW using a simpler non-depth-robust graph based on a Merkle tree.

PoSWs based on graph labeling don't naturally provide a VDF because removing any single edge in the graph will change the output of the proof, yet is unlikely to be detected by random challenges.

**Sequentially hard functions** The most popular solution for a slow function which can be viewed as a proto-VDF, dating to Dwork and Naor [31], is computing modular square roots. Given a challenge $x \in \mathbb{Z}_p^*$, computing $y = x^{\frac{p+1}{4}} \pmod p$ can be efficiently verified by checking that $y^2 = x \pmod p$ (for $p \equiv 3 \pmod 4$). There is no known algorithm for computing modular exponentiation which is sublinear in the exponent. However, the difficulty of puzzles is fixed to $t = \log p$ as the exponent can be reduced modulo $p - 1$ before computation, requiring the use of a very large prime $p$ to produce a difficult puzzle.

This puzzle has been considered before for similar applications as our VDFs, in particular randomness beacons [41, 48]. Lenstra and Wesolowski [48] proposed creating a more difficult puzzle for a small $p$ by chaining a series of such puzzles together (interleaved with a simple permutation) in a construction called Sloth. We proposed a simple improvement of this puzzle in Section 7. Recall that this does not meet our asymptotic definition of a VDF because it does not offer (asymptotically) efficient verification, however we used it as an important building block to construct a more practical VDF based on IVC. Asymptotically, Sloth is comparable to a hash chain of length $t$ with $t$ checkpoints provided as a proof, which also provides $O(\text{polylog}(t))$-time verification (with $t$ processors) and a solution of size $\Theta(t \cdot \lambda)$.

## 10 Conclusions

Given their large number of interesting applications, we hope this work stimulates new practical uses for VDFs and continued study of theoretical constructions. We still lack a theoretically optimal VDF, consisting of a simple inherently sequential function requiring low parallelism to compute but yet being very fast (e.g. logarithmic) to invert. These requirements motivate the search for new problems which have not traditionally been used in cryptography. Ideally, we want a VDF that is also post-quantum secure.

## Acknowledgments

We thank Micheal Zieve for his help with permutation polynomials. We thank the CRYPTO reviewers for their helpful comments. This work was supported by NSF, a grant from ONR, the Simons Foundation, and a Google faculty fellowship.

## References

[1] Randao: A dao working as rng of ethereum. Technical report, 2016.

[2] Filecoin: A decentralized storage network. Protocol Labs, 2017. `https://filecoin.io/filecoin.pdf`.

[3] Proof of replication. Protocol Labs, 2017. `https://filecoin.io/proof-of-replication.pdf`.

[4] Threshold relay. Dfinity, 2017. `https://dfinity.org/pdfs/viewer.html?file=../library/threshold-relay-blockchain-stanford.pdf`.

[5] M. R. Albrecht, L. Grassi, C. Rechberger, A. Roy, and T. Tiessen. Mimc: Efficient encryption and cryptographic hashing with minimal multiplicative complexity. In *ASIACRYPT*, pages 191–219, 2016.

[6] F. Armknecht, L. Barman, J.-M. Bohli, and G. O. Karame. Mirror: Enabling proofs of data replication and retrievability in the cloud. In *USENIX Security Symposium*, pages 1051–1068, 2016.

[7] T. Aura, P. Nikander, and J. Leiwo. Dos-resistant authentication with client puzzles. In *International workshop on security protocols*, pages 170–177. Springer, 2000.

[8] S. Baktir and E. Savas. Highly-parallel montgomery multiplication for multi-core general-purpose microprocessors. In *Computer and Information Sciences III*, pages 467–476. Springer, 2013.

[9] E. Ben-Sasson, A. Chiesa, C. Garman, M. Green, I. Miers, E. Tromer, and M. Virza. Zerocash: Decentralized anonymous payments from Bitcoin. In *IEEE Symposium on Security and Privacy*, 2014.

[10] E. Ben-Sasson, A. Chiesa, D. Genkin, E. Tromer, and M. Virza. SNARKs for C: Verifying program executions succinctly and in zero knowledge. In *CRYPTO*, 2013.

[11] E. Ben-Sasson, A. Chiesa, E. Tromer, and M. Virza. Scalable zero knowledge via cycles of elliptic curves. In *Algorithmica*, pages 1102–1160, 2014.

[12] I. Bentov, A. Gabizon, and D. Zuckerman. Bitcoin beacon. *arXiv preprint arXiv:1605.04559*, 2016.

[13] I. Bentov, R. Pass, and E. Shi. Snow white: Provably secure proofs of stake. *IACR Cryptology ePrint Archive*, 2016, 2016.

[14] N. Bitansky, R. Canetti, A. Chiesa, and E. Tromer. Recursive composition and bootstrapping for snarks and proof-carrying data. In *Proceedings of the forty-fifth annual ACM symposium on Theory of computing*, pages 111–120. ACM, 2013.

[15] N. Bitansky, S. Goldwasser, A. Jain, O. Paneth, V. Vaikuntanathan, and B. Waters. Time-lock puzzles from randomized encodings. In *ACM Conference on Innovations in Theoretical Computer Science*, 2016.

[16] D. Boneh and M. Naor. Timed commitments. In *Advances in CryptologyCrypto 2000*, pages 236–254. Springer, 2000.

[17] J. Bonneau, J. Clark, and S. Goldfeder. On bitcoin as a public randomness source. *URL https://eprint.iacr.org/2015/1015.pdf*, 2015.

[18] J. Bonneau and R. Xu. Scrambling for lightweight censorship resistance. In *International Workshop on Security Protocols*. Springer, 2011.

[19] K. P. Bram Cohen. Simple proofs of sequential work. In *EUROCRYPT*, 2018.

[20] J. Buchmann and H. C. Williams. A key-exchange system based on imaginary quadratic fields. *Journal of Cryptology*, 1(2):107–118, 1988.

[21] J. Cai, R. J. Lipton, R. Sedgewick, and A. C. Yao. Towards uncheatable benchmarks. In *Structure in Complexity Theory*, 1993.

[22] J.-Y. Cai, A. Nerurkar, and M.-Y. Wu. The design of uncheatable benchmarks using complexity theory, 1997.

[23] I. Cascudo and B. David. Scrape: Scalable randomness attested by public entities. Cryptology ePrint Archive, Report 2017/216, 2017. `http://eprint.iacr.org/2017/216`.

[24] J. Clark and U. Hengartner. On the Use of Financial Data as a Random Beacon. *Usenix EVT/WOTE*, 2010.

[25] B. Codenottia, B. N. Datta, K. Datta, and M. Leoncini. Parallel algorithms for certain matrix computations. In *Theoretical Computer Science*, 1997.

[26] B. Cohen. Proofs of space and time. Blockchain Protocol Analysis and Security Engineering, 2017. `https://cyber.stanford.edu/sites/default/files/bramcohen.pdf`.

[27] W. Dai. B-money. *Consulted*, 1:2012, 1998.

[28] B. David, P. Gazi, A. Kiayias, and A. Russell. Ouroboros praos: An adaptively-secure, semi-synchronous proof-of-stake blockchain. In *Eurocrypt*. Springer, 2018.

[29] D. Dean and A. Stubblefield. Using client puzzles to protect tls. In *USENIX Security Symposium*, volume 42, 2001.

[30] J. R. Douceur. The sybil attack. In *International Workshop on Peer-to-Peer Systems*, pages 251–260. Springer, 2002.

[31] C. Dwork and M. Naor. Pricing via processing or combatting junk mail. In *CRYPTO*, 1992.

[32] S. Dziembowski, S. Faust, V. Kolmogorov, and K. Pietrzak. Proofs of space. In *CRYPTO*, 2015.

[33] N. Dttling, S. Garg, G. Malavolta, and P. N. Vasudevan. Tight verifiable delay functions. Cryptology ePrint Archive, Report 2019/659, 2019. `https://eprint.iacr.org/2019/659`.

[34] Émile Mathieu. Mémoire sur l'étude des fonctions de plusieurs quantités sur la manière de les former et sur les substitutions qui les laissent invariables. In *J. Math. Pures Appl. (2) 6*, 1861.

[35] B. Fisch. Poreps: Proofs of space on useful data. Cryptology ePrint Archive, Report

2018/678, 2018. https://eprint.iacr.org/2018/678.

[36] J. Garay, A. Kiayias, and N. Leonardos. The Bitcoin Backbone Protocol: Analysis and Applications. Cryptology ePrint Archive # 2014/765, 2014.

[37] R. Gennaro, C. Gentry, B. Parno, and M. Raykova. Quadratic span programs and succinct nizks without pcps. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 626–645. Springer, 2013.

[38] D. M. Goldschlag and S. G. Stubblebine. Publicly Verifiable Lotteries: Applications of Delaying Functions. In *Financial Cryptography*, 1998.

[39] R. M. Guralnick and P. Müller. Exceptional polynomials of affine type. *Journal of Algebra*, 194(2):429–454, 1997.

[40] X.-d. Hou. Permutation polynomials over finite fieldsa survey of recent advances. *Finite Fields and Their Applications*, 32:82–119, 2015.

[41] Y. I. Jerschow and M. Mauve. Non-parallelizable and non-interactive client puzzles from modular square roots. In *Availability, Reliability and Security (ARES)*, 2011.

[42] J. B. Joël Alwen and K. Pietrzak. Depth-robust graphs and their cumulative memory complexity. In *Eurocrypt*, 2017.

[43] A. Juels and B. S. Kaliski Jr. Pors: Proofs of retrievability for large files. In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 584–597. Acm, 2007.

[44] A. Jules and J. Brainard. Client-puzzles: a cryptographic defense against connection depletion. In *Proc. Network and Distributed System Security Symp.(NDSS'99)*, pages 151–165, 1999.

[45] A. Kiayias, A. Russell, B. David, and R. Oliynykov. Ouroboros: A provably secure proof-of-stake blockchain protocol. In *CRYPTO*, 2017.

[46] S. King and S. Nadal. Peercoin–secure & sustainable cryptocoin. *Aug-2012 [Online]. Available: https://peercoin. net/whitepaper*.

[47] D. Kogan, N. Manohar, and D. Boneh. T/key: Second-factor authentication from secure hash chains. In *ACM Conference on Computer and Communications Security*, 2017.

[48] A. K. Lenstra and B. Wesolowski. A random zoo: sloth, unicorn, and trx. *IACR Cryptology ePrint Archive*, 2015, 2015.

[49] S. D. Lerner. Proof of unique blockchain storage, 2014. https://bitslog.wordpress.com/2014/11/03/proof-of-local-blockchain-storage/.

[50] R. Lidl, G. L. Mullen, and G. Turnwald. *Dickson polynomials*, volume 65. Chapman & Hall/CRC, 1993.

[51] H. Lipmaa. Secure accumulators from euclidean rings without trusted setup. In *International Conference on Applied Cryptography and Network Security*, pages 224–240. Springer, 2012.

[52] M. Mahmoody, T. Moran, and S. Vadhan. Publicly verifiable proofs of sequential work. In *Proceedings of the 4th conference on Innovations in Theoretical Computer Science*. ACM, 2013.

[53] U. Maurer, R. Renner, and C. Holenstein. Indifferentiability, impossibility results on reductions, and applications to the random oracle methodology. In *TCC*, 2004.

[54] S. Micali. Cs proofs. In *Foundations of Computer Science, 1994 Proceedings., 35th Annual Symposium on*, pages 436–453. IEEE, 1994.

[55] S. Micali. Algorand: the efficient and democratic ledger. *arXiv preprint arXiv:1607.01341*, 2016.

[56] A. Miller, A. Juels, E. Shi, B. Parno, and J. Katz. Permacoin: Repurposing bitcoin work for data preservation. In *Security and Privacy (SP), 2014 IEEE Symposium on*, pages 475–490. IEEE, 2014.

[57] T. Moran, M. Naor, and G. Segev. An optimally fair coin toss. In *Theory of Cryptography Conference*, pages 1–18. Springer, 2009.

[58] R. Morris and K. Thompson. Password security: A case history. *Communications of the ACM*, 22(11):594–597, 1979.

[59] P. Müller. A weil-bound free proof of schur's conjecture. *Finite Fields and Their Applications*, 3(1):25–32, 1997.

[60] S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2008.

[61] S. Park, K. Pietrzak, A. Kwon, J. Alwen, G. Fuchsbauer, and P. Gai. Spacemint: A cryptocurrency based on proofs of space. Cryptology ePrint Archive, Report 2015/528, 2015. `http://eprint.iacr.org/2015/528`.

[62] B. Parno, J. Howell, C. Gentry, and M. Raykova. Pinocchio: Nearly practical verifiable computation. In *IEEE Security and Privacy*, 2013.

[63] C. Pierrot and B. Wesolowski. Malleability of the blockchains entropy. 2016.

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

[65] K. Pietrzak. Unique proofs of sequential work from time-lock puzzles, 2018. Manuscript.

[66] M. O. Rabin. Transaction protection by beacons. *Journal of Computer and System Sciences*, 1983.

[67] R. L. Rivest and A. Shamir. Payword and micromint: Two simple micropayment schemes. In *International Workshop on Security Protocols*, pages 69–87. Springer, 1996.

[68] R. L. Rivest, A. Shamir, and D. A. Wagner. Time-lock puzzles and timed-release crypto. 1996.

[69] E. Syta, P. Jovanovic, E. K. Kogias, N. Gailly, L. Gasser, I. Khoffi, M. J. Fischer, and B. Ford. Scalable bias-resistant distributed randomness. In *Security and Privacy (SP), 2017 IEEE Symposium on*, pages 444–460. Ieee, 2017.

[70] P. Valiant. Incrementally verifiable computation or proofs of knowledge imply time/space efficiency. *Theory of Cryptography*, pages 1–18, 2008.

[71] P. C. Van Oorschot and M. J. Wiener. Parallel collision search with application to hash functions and discrete logarithms. In *ACM Conference on Computer and Communications Security*, 1994.

[72] R. S. Wahby, S. T. Setty, Z. Ren, A. J. Blumberg, and M. Walfish. Efficient ram and control flow in verifiable outsourced computation. In *NDSS*, 2015.

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

[74] M. Zieve. Exceptional polynomials. `http://dept.math.lsa.umich.edu/~zieve/papers/epfacts.pdf`.