Continuous Verifiable Delay Functions

Naomi Ephraim[*] Cody Freitag[†] Ilan Komargodski[‡] Rafael Pass[§]

Abstract

We introduce the notion of a *continuous verifiable delay function* (cVDF): a function g which is (a) iteratively sequential—meaning that evaluating the iteration $g^{(t)}$ of g (on a random input) takes time roughly t times the time to evaluate g, even with many parallel processors, and (b) (iteratively) verifiable—the output of $g^{(t)}$ can be efficiently verified (in time that is essentially independent of t). In other words, the iterated function $g^{(t)}$ is a verifiable delay function (VDF) (Boneh et al., CRYPTO '18), having the property that intermediate steps of the computation (i.e., $g^{(t')}$ for $t' < t$) are publicly and continuously verifiable.

We demonstrate that cVDFs have intriguing applications: (a) they can be used to construct *public randomness beacons* that only require an initial random seed (and no further unpredictable sources of randomness), (b) enable *outsourceable VDFs* where any part of the VDF computation can be verifiably outsourced, and (c) have deep complexity-theoretic consequences: in particular, they imply the existence of *depth-robust moderately-hard* Nash equilibrium problem instances, i.e. instances that can be solved in polynomial time yet require a high sequential running time.

Our main result is the construction of a cVDF based on the repeated squaring assumption and the soundness of the Fiat-Shamir (FS) heuristic for *constant-round proofs.* We highlight that when viewed as a (plain) VDF, our construction requires a weaker FS assumption than previous ones (earlier constructions require the FS heuristic for either super-logarithmic round proofs, or for arguments).

[*]Cornell Tech, `nephraim@cs.cornell.edu`
[†]Cornell Tech, `cfreitag@cs.cornell.edu`
[‡]NTT Research, `ilan.komargodski@ntt-research.ac.il`
[§]Cornell Tech, `rafael@cs.cornell.edu`

Contents

1 Introduction **1**
- 1.1 Our Results . . . 2
- 1.2 Related Work . . . 5

2 Technical Overview **6**
- 2.1 Adapting Pietrzak's VDF . . . 8
- 2.2 Constructing a Continuous VDF . . . 10
- 2.3 The Efficiency of our Construction . . . 11

3 Preliminaries **12**
- 3.1 Verifiable, Sequential, and Iteratively Sequential Functions . . . 13
- 3.2 Repeated Squaring Assumption . . . 14
- 3.3 Interactive Protocols and the Fiat-Shamir Heuristic . . . 15

4 Interactive Proof for Repeated Squaring **16**
- 4.1 Protocol . . . 17
- 4.2 Proofs . . . 17

5 Unique Verifiable Delay Function **21**
- 5.1 Construction . . . 21
- 5.2 Proofs . . . 23

6 Continuous Verifiable Delay Function **26**
- 6.1 Construction . . . 28
- 6.2 Proofs . . . 31

7 Applications **40**
- 7.1 Public Randomness Beacons . . . 40
- 7.2 PPAD Hardness . . . 44

References **47**

A Number Theory Facts **50**

1 Introduction

A fundamental computational task is to simulate "real time" via computation. This was first suggested by Rabin [Rab83] in 1983, who introduced a notion called *randomness beacon* to describe an ideal functionality that publishes unpredictable and independent random values at fixed intervals. This concept has received a substantial amount of attention since its introduction, and even more so in recent years due to its many applications to more efficient and reliable consensus protocols in the context of blockchain technologies.

One natural approach, which is the focus of this work, is to implement a randomness beacon by using an *iteratively sequential function*.[1] An iteratively sequential function g inherently takes some time ℓ to compute and has the property that there are no shortcuts to compute sequential iterations of it. That is, computing the t-wise composition of g for any t should take roughly time $t \cdot \ell$, even with parallelism. Using an iteratively sequential function g with an initial seed x, we can construct a randomness beacon where the output at interval t is computed as the hash of

$$g^{(t)}(x) = \underbrace{g \circ g \circ \ldots \circ g}_{t \text{ times}}(x).$$

After $t \cdot \ell$ time has elapsed (at which point we know the first t values), the beacon's output should be unpredictable sufficiently far in the future.[2] The original candidate iteratively sequential function is based on (repeated) squaring in a finite group of unknown order [CLSY93, RSW96]. It is also conjectured that any secure hash function (such as SHA-256) gives an iteratively sequential function; this was suggested in [Kal00] and indeed, as shown in [MMV13], a random oracle is iteratively sequential.

Continuous VDFs. The downside of using an iteratively sequential function as a randomness beacon is that to verify the current value of the beacon, one needs to recompute its entire history which is time consuming by definition. In particular, a party that joins late will never be able to catch up. Rather, we would like the output at each step to be both *publicly* and *efficiently* verifiable. It is also desirable for the randomness beacon to be generated without any private state so that *anyone* can compute it, meaning that each step can be computed based solely on the output of the preceding step. Indeed, if we have an iteratively sequential function that is also *(iteratively) verifiable*—in the sense that one can efficiently verify the output of $g^{(t)}(x)$ in time polylog(t)—then such a function could be used to obtain a *public randomness beacon*. In this paper, we introduce and construct such a function and refer to it as a *continuous verifiable delay function* (cVDF). As the name suggests, it can be viewed as enabling continuous evaluation and verification of a verifiable delay function (VDF) [BBBF18] as we describe shortly.[3]

Continuous VDFs are related to many previously studied time-based primitives. One classical construction is the time-lock puzzle of Rivest, Shamir, and Wagner [RSW96]. Their construction can be viewed as an iteratively sequential function that is *privately verifiable* with a trapdoor—unfortunately, this trapdoor not only enables quickly verifying the output of iterations of the function, but in fact also enables quickly computing the iterations. New publicly verifiable time-based primitives have since emerged, including proofs of sequential work (PoSW) [MMV13, CP18,

[1] We use the terminology from [BBBF18]; these have also been referred to as sequential functions [MMV13].

[2] If g is perfectly iteratively sequential, meaning that t iterations cannot be computed in time faster than *exactly* $t \cdot \ell$, then after t steps of g the *next* value would be unpredictable. However, if t iterations cannot be computed in time faster than $(1-\epsilon) \cdot t \cdot \ell$, we can only guarantee that the $(\epsilon \cdot t)$-th value into the future is unpredictable.

[3] Our notion of a cVDF (just like the earlier notion of a "plain" VDF) also allows for the existence of some trusted public parameters.

DLM19] and verifiable delay functions (VDF) [BBBF18, Pie19, Wes19, BBF18, FMPS19]. While these primitives are enough for many applications, they fall short of implementing a public randomness beacon (on their own). In more detail, a PoSW enables generating a publicly verifiable proof of *some* computation (rather than a specific function with a unique output) that is guaranteed to have taken a long time. This issue was overcome through the introduction of VDFs [BBBF18], which are functions that require some "long" time T to compute (where T is a parameter given to the function), yet the answer to the computation can be efficiently verified given a proof that can be jointly generated with the output (with only small overhead). In fact, one of the motivating applications for constructing VDFs was to obtain a public randomness beacon. A natural approach toward this goal is to simply iterate the VDF at fixed intervals. However, this construction does not satisfy our desired efficiency for verifiability. In particular, even though the VDF enables fast verification of each invocation, we still need to store all proofs for the intermediate values to verify the final output of the iterated function, and thus the proof size and verification time grow linearly with the number of invocations t. While a recent construction of Wesolowski [Wes19] enables aggregating these intermediate proofs to obtain a single short proof, the verification time still grows linearly with t (in contrast, a cVDF enables continuously iterating a function such that the output of t iterations can be efficiently verified in time essentially independent of t, for any t). While a VDF does not directly give a public randomness beacon, it does, however, enable turning a "high-entropy beacon" (e.g., continuous monitoring of stock market prices) into an unbiased and unpredictable beacon as described in [BBBF18]. In contrast, using a cVDF enables dispensing altogether with the high-entropy beacon—we simply need a *single* initial seed x.

Continuous VDFs are useful not only for randomness beacons, but also for standard applications of VDFs. Consider a scenario where some entity is offering a \$5M reward for evaluating a single VDF with time parameter 5 years (i.e., it is supposed to take five years to evaluate it). Alice starts evaluating the VDF, but after two years runs out of money and can no longer continue the computation. Ideally, she would like to sell the work she has completed for \$2M. Bob is willing to buy the intermediate state, verify it, and continue the computation. The problem, however, is that there is no way for Bob to verify Alice's internal state. In contrast, had Alice used a cVDF, she would simply be iterating an iteratively sequential function, and we would directly have the guarantee that at any intermediate state of the computation can be verified and Alice can be compensated for her effort. In other words, cVDF enable verifiably outsourcing VDF computation.

Finally, as we show, cVDFs are intriguing also from a complexity-theoretic point of view. The existence of cVDFs imply that PPAD [Pap94] (the class for which the task of finding a Nash equilibrium in a two-party game is complete) is hard—in fact, the existence of cVDFs imply the existence of a relaxed-SVL [CHK$^+$19a, AKV04] instance with *tight* hardness (which yields improved hardness results also for PPAD). Additionally, the existence of cVDFs imply that there is a constant d such that for large enough c, there is a distribution over Nash equilibrium problem instances of size n that can be solved in time n^c but cannot be solved in depth $n^{c/d}$ (and arbitrary polynomial size)—that is, the existence of "easy" Nash equilibrium problem instances that requires high *sequential* running time. In other words, cVDFs imply that it is possible to sample "moderately-hard" Nash equilibrium problem instances that require a large time to solve, even with many parallel processors.

1.1 Our Results

Our main result is the construction of a cVDF based on the repeated squaring assumption in a finite group of unknown order and a variant of the Fiat-Shamir (FS) heuristic for *constant-round proof* systems. Informally, the iteratively sequential property of our construction comes from the repeated squaring assumption which says that squaring in this setting is an iteratively sequential function. We use the Fiat-Shamir assumption to obtain the continuous verifiability property of our construction. More precisely, we apply the Fiat-Shamir heuristic on a constant-round proof system where the verifier may be inefficient. We note that by the classic results of [GK90] this holds in the random oracle model.

Theorem 1.1 (Informal, see Corollary 6.3)**.** *Under the repeated squaring assumption and the Fiat-Shamir assumption for constant-round proof systems with inefficient verifiers, there exists a cVDF.*

We remark that to obtain a plain VDF we only need the "standard" Fiat-Shamir assumption for constant-round proof systems (with efficient verifiers).

A cVDF readily gives a public randomness beacon. As discussed above, the notions of cVDFs and public randomness beacons are closely related. The main difference between the two is that the output of a randomness beacon should not only be unpredictable before a certain time, but should also be indistinguishable from random. Thus, we obtain our public randomness beacon by simply "hashing" the output of the cVDF. We show that this indeed gives a public randomness beacon by performing the hashing using a pseudo-random generators (PRGs) for unpredictable sources (which exist either in the random oracle model or from extremely lossy functions [Zha16]).

Theorem 1.2 (Informal)**.** *Assuming the existence of cVDFs and PRGs for unpredictable sources, there exists a public randomness beacon.*

Comparison with (plain) VDFs. The two most related VDF constructions are that of Pietrzak [Pie19] and that of Wesolowski [Wes19], as these are based on repeated squaring. In terms of assumptions, Pietrzak's protocol [Pie19] assumes the Fiat-Shamir heuristic for a proof system with a *super-constant* number of rounds and Wesolowski's [Wes19] assumes the Fiat-Shamir heuristic for a constant-round *argument system*. It is known that, in general, the Fiat-Shamir heuristic is not true for super-constant round protocols (even in the random oracle model[4]), and not true for constant-round arguments [Bar01, GK03]. As such, both of these constructions rely on somewhat non-standard assumptions. In contrast, our VDF relies only on the Fiat-Shamir heuristic for a constant-round proof system—no counter examples are currently known for such proof systems.

We additionally note that before applying the Fiat-Shamir heuristic (i.e., a VDF in the random oracle model), our VDF satisfies computational uniqueness while Pietrzak's satisfies statistical uniqueness. He achieves this by working over the group of signed quadratic residues. We note that we can get statistical uniqueness in this setting using the same idea. Lastly, we emphasize that the concrete proof length and verification time are polynomially higher in our case than that of both Pietrzak and Wesolowski. For a detailed comparison of the parameters between our (c)VDF and their VDFs, see Section 2.3.

PPAD hardness. PPAD [Pap94] is an important subclass in TFNP [MP91] (the class of total search problems), most notably known for its complete problem of finding a Nash equilibrium in bimatrix games [DGP09, CDT09]. Understanding whether PPAD contains hard problems is a central open problem and the most common approach for proving hardness was pioneered by Abbot, Kane, and Valiant [AKV04]. They introduced a problem, which [BPR15] termed Sink-of-Verifiable-Line (SVL), and showed that it reduces to End-Of-Line (EOL), a complete problem for PPAD. In SVL, one has to present a function f that can be iterated and each intermediate value can be efficiently verified, but the output of T iterations (where T is some super-polynomial value, referred to as the length of the "line") is hard to compute in polynomial time.

[4]Although, [Pie19] shows that it does hold in the random oracle model for his particular protocol.

In a beautiful recent work, Choudhuri et al. [CHK$^+$19a] defined the RELAXED-SINK-OF-VERIFIABLE-LINE (rSVL) problem, and showed that it reduces to EOL, as well. rSVL is a generalization of SVL where one is required to find either the output after many iterations (as in SVL) or an off-chain value that verifies. Choudhuri et al. [CHK$^+$19a] gave a hard rSVL instance assuming the security of the Fiat-Shamir transformation applied to the sum-check protocol [LFKN92] (which is a *polynomial-round* protocol).

The notion of an (r)SVL instance is very related to our notion of a cVDF. The main differences are that a cVDF requires that the gap between the honest computation and the malicious one is tight and that security holds for adversaries that have access to multiple processors running in parallel. As such, the existence of a cVDF (which handles super-polynomially many iterations) directly implies an rSVL instances with "optimal" hardness—namely, one where the number of computational steps required to solve an instance of the problem with a "line" of length T is $(1-\epsilon)\cdot T$.

Theorem 1.3 (Informal)**.** *The existence of a cVDF supporting superpolynomially many iterations implies an optimally-hard rSVL instance (which in turn implies that* PPAD *is hard (on average)).*

Theorem 1.1 readily extends to give a cVDF supporting super-polynomially many iterations by making a Fiat-Shamir assumption for $\omega(1)$-round proof systems. As a consequence, we get an optimally-hard instance of rSVL based on this Fiat-Shamir assumption for $\omega(1)$-round proofs[5] and the repeated squaring assumption. By following the reductions from rSVL to EOL and to finding a Nash equilibrium, we get (based on the same assumptions) hard PPAD and Nash equilibrium instances. We remark that in comparison to the results of Choudhuri et al., we only rely on the Fiat-Shamir assumption for $\omega(1)$-round protocols, whereas they rely on it for a polynomial-round, or at the very least an $\omega(\log n)$-round proof systems (if additionally assuming that #SAT is sub-exponentially hard). On the other hand, we additionally require a computational assumption–namely, the repeated squaring assumption, whereas they do not.[6]

Our method yields PPAD instances satisfying another interesting property: we can generate PPAD (and thus Nash equilibrium problem) instances that can be solved in polynomial time, yet they also require a high sequential running time—that is, they are "depth-robust" moderately-hard instances. As far as we know, this gives the first evidence that PPAD (and thus Nash equilibrium problems) requires high sequential running time to solve (even for easy instances!).

Theorem 1.4 (Informal)**.** *The existence of a cVDF implies a distribution of depth-robust moderately-hard* PPAD *instances. In particular, there exists a constant d such that for all sufficiently large constants c, there is a distribution over Nash equilibrium problem instances of size n that can be solved in time n^c but cannot be solved in depth $n^{c/d}$ and arbitrary polynomial time.*[7]

Combining Theorems 1.1 and 1.4, we get a depth-robust moderately-hard PPAD instance based on the Fiat-Shamir assumption for constant-round proof systems with inefficient verifiers and the repeated squaring assumption.

[5]As mentioned above, in general, the Fiat-Shamir assumption is false for super-constant-round proofs. But we state a restricted form of a Fiat-Shamir assumption for super-constant-round proofs with *exponentially small soundness error* which holds in the random oracle model, due to the classic reduction from [GK90].

[6]We also note that Choudhuri et al. show how to instantiate the hash function in their Fiat-Shamir transformation assuming a class of fully homomorphic encryption schemes has almost-optimal security against quasi-polynomial time adversaries. We leave such instantiations in our context for future work.

[7]If we additionally assume that the repeated squaring assumption is sub-exponentially hard, then the resulting instance cannot be solved in depth $n^{c/d}$ and sub-exponential time.

1.2 Related Work

In addition to the time lock puzzle of [RSW96] mentioned above, an alternative construction is by Bitansky et al. [BGJ+16] assuming a strong form of randomized encodings and the existence of inherently sequential functions. While the time-lock puzzle of [RSW96] is only privately verifiable, Boneh and Naor [BN00] showed a method to prove that the time-lock puzzle has a solution. Jerschow and Mauve [JM11] and Lenstra and Wesolowski [LW17] constructed iteratively sequential functions based on Dwork and Naor's slow function [DN92] (which is based on hardness of modular exponentiations).

PPAD hardness. The complexity class PPAD (standing for *Polynomial Parity Arguments on Directed graphs*), introduced by Papadimitriou [Pap94], is one of the central classes in TFNP. It contains the problems that can be shown to be total by a parity argument. This class is famous most notably since the problem of finding a Nash equilibrium in bimatrix games is complete for it [DGP09, CDT09]. The class is formally defined by one of its complete problems End-Of-Line (EOL).

Bitansky, Paneth, and Rosen [BPR15] introduced the Sink-of-Verifiable-Line (SVL) problem and showed that it reduces to the EOL problem (based on Abbot et al. [AKV04] who adapted the reversible computation idea of Bennet [Ben89]). They additionally gave an SVL instance which is hard assuming sub-exponentially secure indistinguishability obfuscation and one-way functions. These underlying assumptions were somewhat relaxed over the years yet remain in the class of obfuscation-type assumptions which are still considered very strong [GPS16, KS17, KNT18].

Hubáček and Yogev [HY17] observed that the Sink-of-Verifiable-Line actually reduces to a more structured problem, which they termed End-Of-Metered-Line (EOML), which in turn resides in CLS (standing for *Continuous Local Search*), a subclass of PPAD. As a corollary, all of the above hardness results for PPAD actually hold for CLS.

In an exciting recent work, Choudhuri et al. [CHK+19a] introduced a relaxation of SVL, termed relaxed-SVL (rSVL) which still reduces to EOML and therefore can be used to prove hardness of PPAD and CLS. They were able to give a hard rSVL instance based on the sum-check protocol of [LFKN92] assuming soundness of the Fiat-Shamir transformation and that #SAT is hard.

Verifiable delay functions. VDFs were recently introduced and constructed by Boneh, Bonneau, Bünz, and Fisch [BBBF18]. Following that work, additional constructions were given in [Pie19, Wes19, FMPS19]. The constructions of Pietrzak [Pie19] and Wesolowski [Wes19] are based on the repeated squaring assumption plus the Fiat-Shamir heuristic, while the construction of De Feo et al. [FMPS19] relies on elliptic curves and bilinear pairings. We refer to Boneh et al. [BBF18] for a survey.

VDFs have numerous applications to the design of reliable distributed systems; see [BBBF18, Section 2]. Indeed, they are nowadays widely used in the design of reliable and resource efficient blockchains (e.g., in the consensus mechanism of the Chia blockchain [Chi]) and there is a collaboration [VDF] between the Ethereum Foundation [Eth], Protocol Labs [Pro], and various academic institutions to design better and more efficient VDFs.

Proofs of sequential work. Proofs of sequential work, suggested by Mahmoody, Moran, and Vadhan [MMV13], are proof systems where on input a random challenge and time parameter t one can generate a publicly verifiable proof making t sequential computations, yet it is computationally infeasible to find a valid proof in significantly less than t sequential steps. Mahmoody et al. [MMV13] gave the first construction and Cohen and Pietrzak [CP18] gave a simple and practical construction (both in the random oracle model). A recent work of Döttling et al. [DLM19] constructs an *incremental* PoSW based on [CP18]. The techniques underlying Döttling et al's construction are related in spirit to ours though the details are very different. See Section 2 for a comparison. All of the above constructions of PoSWs do not satisfy uniqueness, which is a major downside for many applications (see [BBBF18] for several examples). Indeed, VDFs were introduced exactly to mitigate this issue. Since our construction satisfies (computational) uniqueness, we actually get the first unique incremental PoSW.

Concurrent works. In a concurrent and independent work, Choudhuri et al. [CHK+19b] show PPAD-hardness based on the Fiat-Shamir heuristic and the repeated squaring assumption. Their underlying techniques are related to ours since they use a similar tree-based proof merging technique on top of Pietrzak's protocol [Pie19]. However, since they use a ternary tree (while we use a high arity tree) their construction cannot be used to get a continuous VDF (and its applications). Also, for PPAD-hardness, their construction requires Fiat-Shamir for protocols with $\omega(\log \lambda)$ rounds (where λ is the security parameter) while we need Fiat-Shamir for $\omega(1)$-round protocols.

VDFs were also studied in two recent independent works by Döttling et al. [DGMV19] and Mahmoody et al. [MSW19]. Both works show negative results for black-box constructions of VDFs in certain regimes of parameters in the random oracle model. The work of Döttling et al. [DGMV19] additionally shows that certain VDFs with a somewhat inefficient evaluator can be generically transformed into VDFs where the evaluator has optimal sequential running time. Whether such a transformation exists for cVDFs is left for future work.

2 Technical Overview

We start by informally defining a cVDF. At a high level, a cVDF specifies an iteratively sequential function Eval where each iteration of the function gives a step of computation. Let x_0 be any starting point and $x_t = \mathsf{Eval}^{(t)}(x_0)$ be the tth step or state given by the cVDF. We let B be an upper bound on the total number of steps in the computation, and assume that honest parties have some bounded parallelism polylog(B) while adversarial parties may have parallelism poly(B). For each step $t \leq B$, we require the following properties to hold:

- **Completeness:** x_t can be verified as the tth state in time polylog(t).
- **Adaptive Soundness:** Any value $x'_t \neq x_t$ computed by an adversarial party will not verify as the tth state (even when the starting point x_0 is chosen adaptively). That is, each state is (computationally) unique.
- **Iteratively Sequential:** Given an honestly sampled x_0, adversarial parties cannot compute x_t in time $(1 - \epsilon) \cdot t \cdot \ell$, where ℓ is the time for an honest party to compute a step of the computation.

We require adaptive soundness due to the distributed nature of a cVDF. In particular, suppose a new party starts computing the cVDF after t steps have elapsed. Then, x_t is the effective starting point for that party, and they may compute for t' more steps to obtain a state $x_{t+t'}$. We want to ensure that soundness holds for the computation from x_t to $x_{t+t'}$, so that the next party that starts at $x_{t+t'}$ can trust the validity of $x_{t+t'}$. Note that the above definition does not contain any proofs, but instead the states are verifiable by themselves. In terms of plain VDFs, this verifiability condition is equivalent to the case where the VDF is unique, meaning that the proofs are empty or included implicitly in the output.

To construct a cVDF, we start with a plain VDF. For simplicity in this overview, we assume that this underlying VDF is unique.

A first attempt. The naïve approach for using a VDF to construct a cVDF is to iterate the VDF as a chain of computations. For any "base difficulty" T, which will be the time to compute a single step, we can use a VDF to do the computation from x_0 to x_T with an associated proof of correctness $\pi_{0\to T}$. Then, we can start a new VDF instance starting at x_T and compute until x_{2T} with a proof of correctness $\pi_{T\to 2T}$. At this point, anyone can verify that x_{2T} is correct by verifying both $\pi_{0\to T}$ and $\pi_{T\to 2T}$. We can continue this process indefinitely.

This solution has the property that after t steps, another party can pick up the current value $x_{t\cdot T}$, verify it by checking each of the proofs computed so far, and then continue the VDF chain. In other words, there is no unverified internal state after t steps of the computation. Still, this naïve solution has the following major drawback (violating completeness). The final proof $\pi_{(t-1)\cdot T\to t\cdot T}$ only certifies that computing a step from $x_{(t-1)\cdot T}$ results in $x_{t\cdot T}$ and does not guarantee anything about the computation from x_0 to $x_{(t-1)\cdot T}$. As such, we need to retain and check all proofs $\pi_{0\to T}, \dots, \pi_{(t-1)\cdot T\to t\cdot T}$ computed so far to be able to verify $x_{t\cdot T}$. Therefore, both the proof size and verification time scale linearly with t. We note that this idea is not new (e.g., see [BBBF18]), but nevertheless it does not solve our problem. Wesolowski [Wes19] partially addresses this issue by showing how to aggregate proofs so the proof size does not grow, but the verification time in his protocol still grows.

One possible idea to overcome the blowup mentioned above is to use generic proof merging techniques. These can combine two different proofs into one that certifies both but whose size and verification time are proportional to that of a single one. Such techniques were given by Valiant [Val08] and Chung et al. [CLP13]. However, being generic, they rely on strong assumptions and do not give the properties that we need (for example, efficiency and uniqueness). We next look at a promising—yet failed—attempt to overcome this.

A logarithmic approach. Since we can implement the above iterated strategy for *any* fixed interval T, we can simply run $\log B$ many independent iterated VDF chains in parallel at the intervals $T = 1, 2, 4, \dots, 2^{\log B}$. Now say that we want to prove that x_{11} is the correct value eleven steps from the starting point x_0. We just need to verify the proofs $\pi_{0\to 8}$, $\pi_{8\to 10}$, and $\pi_{10\to 11}$. For any number of steps t, we can now verify x_t by verifying only $\log(t)$ many proofs, so we have resolved the major drawbacks! Furthermore, the prover can maintain a small state at each step of the computation by "forgetting" the smaller proofs. For example, after completing a proof $\pi_{0\to 2T}$ of size $2T$, the prover no longer needs to store the proofs $\pi_{0\to T}$ and $\pi_{T\to 2T}$.

Unfortunately, we have given up the distributed nature of a continuous VDF. Specifically, completeness fails to hold. Each "step" of the computation that the prover does to compute x_t with its associated proofs is no longer an independent instance of a single VDF computation. Rather, upon computing x_t, the current prover has some internal state for all of the computations which have not yet completed at step t. Since a VDF only provides a way to prove that the output of each VDF instance is correct, then a new party who wants to pick up the computation has no way to verify the internal states of the unfinished VDF computations. As a result, this solution only works in the case where there is one trusted party maintaining the state of all the current VDF chains over a long period of time. In contrast, a cVDF ensures that there is no internal state at each step of the computation (or equivalently that the internal state is unique and can be verified as part of the output).

At an extremely high level, our continuous VDF builds off of this failed attempt when applied to the protocol of Pietrzak [Pie19]. We make use of the algebraic structure of the underlying repeated squaring computation to ensure that the internal state of the prover is verifiable at every step and can be efficiently continued.

2.1 Adapting Pietrzak's VDF

We next give a brief overview of Pietrzak's sumcheck-style interactive protocol for repeated squaring and the resulting VDF. Let $N = p \cdot q$ where p and q are safe primes and consider the language

$$\mathcal{L}_{N,B} = \{(x, y, t) \mid x, y \in \mathbb{Z}_N^\star \text{ and } y = x^{2^t} \bmod N \text{ and } t \leq B\}$$

that corresponds to valid repeated squaring instances with at most B exponentiations (where we think of B as smaller than the time to factor N). In order for the prover to prove that $(x, y, t) \in \mathcal{L}_{N,B}$ (corresponding to t steps of the computation), it first computes $u = x^{2^{t/2}}$. It is clearly enough to then prove that $u = x^{2^{t/2}}$ and that $u^{2^{t/2}} = y$. However, recursively proving both statements separately is too expensive. The main observation of Pietrzak is that using a random challenge r from the verifier, one can merge both statements into a single one $u^r y = (x^r u)^{2^{t/2}}$ which is true if and only if the original two statements are true (with high probability over r). We emphasize that proving that $u^r y = (x^r u)^{2^{t/2}}$ has the same form as our original statement, but with difficulty $t/2$. This protocol readily gives a VDF by applying the Fiat-Shamir heuristic [FS86] on the $\log_2 B$ round interactive proof.

From the above, it is clear that the only internal state that the prover needs to maintain in Pietrzak's VDF consists of the midpoint $u = x^{2^{t/2}}$ and the output $y = x^{2^t}$. Thus, if we want another party to be able to pick up the computation at any time, we need to simultaneously prove the correctness of u in addition to y. Note that proving the correctness of u just requires another independent VDF instance of difficulty $t/2$. This results in a natural recursive tree-based structure where each computation of t steps consists of proving three instances of size $t/2$: $u = x^{2^{t/2}}$, $y = u^{2^{t/2}}$, and $u^r y = (x^r u)^{2^{t/2}}$. Consequently, once these three instances are proven, it directly gives a proof for the "parent" instance $x^{2^t} = y$. Note that this parent proof *only* need to consist of u, y, and a proof that $u^r y = (x^r u)^{2^{t/2}}$ (in particular, it does not require proofs of the first two sub-computations, since they are certified by the proof of the third).

This suggests a high-level framework for making the construction continuous: starting at the root where we want to compute x^{2^t}, recursively compute and prove each of the three sub-instances. Specifically, each step of the cVDF will be a step in the traversal of this tree. At any point when all three sub-instances of a node have been proven, merge the proofs into a proof of the parent node and "forget" the proofs of the sub-instances. This has the two desirable properties we want for a cVDF—first, at any point a new party can verify the state before continuing the computation, since the state only contains the nodes that have been completed; second, due to the structure of the proofs, the proof size at any node is bounded roughly by the height of the tree and hence avoids a blowup in verification time.

Proof merging. The above approach heavily relies on the proof merging technique discussed above, namely that proofs of sub-instances of a parent node can be efficiently merged into a proof at that parent node. We obtain this due to the structure of the proofs in Pietrzak's protocol. We note that similar proof merging techniques for specific settings were recently given by Döttling et al. [DLM19] (in the context of incremental PoSW) and Choudhuri et al. [CHK$^+$19a] (in the context of constructing a hard rSVL instance). While their constructions are conceptually similar to ours, our construction for a cVDF introduces many challenges in order to achieve both uniquely verifiable states and a tight gap between honest and malicious evaluation. Döttling et al. [DLM19] build on the Cohen and Pietrzak [CP18] PoSW and use a tree-based construction to make it incremental. At a high level, [CP18] is a PoSW based on a variant of Merkle trees, where the public verification procedure consists of a challenge for opening a random path in the tree and checking consistency. The main idea of Döttling et al. is to traverse the tree in a certain way and remember a small intermediate state which enables them to continue the computation incrementally. Moreover, they provide a proof at each step by creating a random challenge which "merges" previously computed challenges. The resulting construction is only a PoSW (where neither the output nor the proof are unique) and therefore does not suffice for our purpose. Choudhuri et al. [CHK+19a] show how to merge proofs in the context of the #SAT sum-check protocol. There, they modify the #SAT proof system to be incremental by performing many additional recursive sub-computations, which is sufficient for their setting but in ours would cause a large gap between honest and malicious evaluation. We note that our method of combining proofs by proving a related statement is reminiscent of the approach of [CHK+19a].

Before discussing the technical details of our tree-based construction, we first go over modifications we make to Pietrzak's interactive protocol. Specifically, we discuss adaptive soundness, and we show how to achieve tight sequentiality (meaning that for any T, computing the VDF with difficulty T cannot be done significantly faster than T) in order to use it for our cVDF.

Achieving adaptive soundness. In order to show soundness, we requires the verifier to be able to efficiently check that the starting point of any computation is a valid generator of QR_N. To achieve this, we use the fact that there is an efficient way to test if x generates QR_N *given the square root of* x (see Fact A.1). As a result, we work with the square roots of elements in our protocol, which slightly changes the language. Namely, x, y are now square roots and $(x, y, t) \in \mathcal{L}_{N,B}$ if $(x^2)^{2^t} = y^2 \bmod N$.[8] We note that, following [Pie19], working in QR_N^+, the group of signed quadratic residues, would also give adaptive soundness (without including the square roots). This holds as soundness of Pietrzak's protocol can be based on the low order assumption, and QR_N^+ has no low order elements [BBF18].[9]

Bounding the fraction of intermediate proofs. Recall that to compute $y = x^{2^t}$ using the VDF of Pietrzak for our proposed cVDF, the honest party recursively proves three different computations of $t/2$ squarings, so that each step will be verifiable. This results in computing for *at least* time $t^{\log_2 3}$, since it corresponds to computing the leaves of a ternary tree of depth $\log_2(t)$, and each leaf requires a squaring. Note that this does not even consider the overhead of computing each proof, only the squarings. However, an adversary (even without parallelism) can shortcut this method and compute the underlying VDF to prove that $y = x^{2^t}$ by computing roughly t squarings (and then computing the proof, which has relatively low overhead).

We deal with this issue by reducing the fraction of generating the intermediate proofs in Pietrzak's protocol. Our solution is to (somewhat paradoxically) modify Pietrzak's protocol to keep additional state, which we will need to verify. Specifically, we observe that t squarings can be split into k different segments. To prove that $y = x^{2^t}$, the prover splits the computation into k

[8]Giving the square root x is the cause of our computational uniqueness guarantee, since a different square root for x^2 would verify. As mentioned, working over QR_N^+ would prevent this attack and give information theoretic uniqueness, as in [Pie19].

[9]We thank the anonymous EUROCRYPT reviewers for pointing out that Pietrzak's protocol satisfies adaptive soundness using QR_N^+.

segments each with difficulty t/k:

$$x_1 = x^{2^{t/k}},\ x_2 = x^{2^{2t/k}},\ \ldots,\ x_{k-1} = x^{2^{(k-1)t/k}},\ x_k = x^{2^t} = y.$$

Using a random challenge $(r_1, \ldots, r_k)$ from the verifier, we are able to combine these k segments into a single statement $(\prod_{i=1}^{k}(x_{i-1})^{r_i})^{t/k} = \prod_{i=1}^{k}(x_i)^{r_i}$ (where $x_0 = x$) which is true if and only if *all* of the segments are true (with high probability over the challenge). We call the combined statement the *sketch*.[10] Now in the recursive tree-based structure outlined above, a computation of t steps consists of proving $k+1$ instances of size t/k. By choosing k to be proportional to the security parameter λ, the total fraction of extra proofs in the honest computation of t steps is now sublinear in t. As an additional benefit when $k = \lambda$, we note that the interactive protocol has $\log_\lambda B \in O(1)$ rounds if B is a fixed polynomial in λ (as opposed to $O(\log \lambda)$ rounds when $k = 2$ corresponding to Pietrzak's protocol). Applying the Fiat-Shamir heuristic to a constant-round protocol is a more standard assumption.[11]

Bounding the overhead of each step. Even though we have bounded the total fraction of extra nodes that the honest party has to compute, this does not suffice to achieve the tight gap between honest and adversarial computation for our proposed cVDF. Specifically, the honest computation has an additive (fixed) polynomial overhead λ^d—for example, to check validity of the inputs and sometimes compute the sketch node—an adversary does not have to do so at each step. To compensate for this, we make each base step of the cVDF larger: namely, we truncate the tree. The effect of this is that a single step now takes time $\lambda^{d'}$ for $d' > d$.

2.2 Constructing a Continuous VDF

As outlined above, our main insight is designing a cVDF based on a tree structure where each intermediate state of the computation can be verified and proofs of the computation can be efficiently merged. More concretely, the steps of computation correspond to a specific traversal of a $(k+1)$-ary tree of height $h = \log_k B$. Each node in the tree is associated to a statement (x, y, t, π) for the underlying VDF, where $y = x^{2^t}$ and π is the corresponding proof of correctness. We call x the node's input, y its output, π the proof, and t the difficulty. The difficulty is determined by its height in the tree, namely, a node at distance l from the root has difficulty $t = k^{h-l}$ (so nodes closer to the leaves take less time to compute).

In more detail, the tree is defined as follows. Starting at the root with input x_0 and difficulty $t = k^h$, we divide it into k segments $x_1, \ldots, x_k$, analogous to our VDF construction. These form the inputs and outputs of its first k children: its ith child will have input x_{i-1} and output x_i, and requires a proof that $(x_{i-1})^{t/k} = x_i$. Its $(k+1)$-st child corresponds to the sketch, namely a node where the k statements of the siblings are merged into a single statement. Recursively splitting statements this way gives the statement at each node in the tree, until reaching the leaves where squaring can be done directly. Note that with this structure, only the leaves require computation—the statement of nodes at greater heights can be deduced from the statements of their children (which gives us a way to efficiently merge proofs "up" the tree as we described above).

As a result, we would like each step of computation in the cVDF to correspond to computing the statement of a single leaf. Accomplishing this requires being able to compute the input x of the leaf from the previous state (from which y can be computed via squaring). By the structure

[10] The name sketch is inspired by the notion of a sketch in algorithms, which refers to a random linear projection.

[11] We are talking about an instantiation of the VDF in the plain model using a concrete hash function. The resulting VDF is provably secure in the random oracle model for any k.

of our tree, we observe that this only requires knowing a (small) subset of nodes that were *already* computed, which we call the *frontier*. The frontier of a leaf s, denoted $\mathsf{frontier}(s)$, contains all the left siblings of its ancestors, including the left siblings of s itself.[12] Therefore, a state in the computation contains a leaf label s and the statements associated with the nodes in $\mathsf{frontier}(s)$, which contains at most $k \cdot \log_k(B)$ nodes. A single step of our continuous VDF, given a state $v = (s, \mathsf{frontier}(s))$, first verifies v and then computes the next state $v' = (s', \mathsf{frontier}(s'))$ where s' is the next leaf after s. See Figure 1 for an illustration of computing the next state.

This is the basic template for our continuous VDF. Next, we discuss some of the challenges that come up related to efficiency and security.

Ensuring the iteratively sequential property. Recall that we want to obtain a tight gap between honest and malicious evaluation of the continuous VDF for *any* number of steps. A priori, it seems that computing a sketch for each node in the tree adds a significant amount of complexity to the honest evaluation. To illustrate this, suppose a malicious evaluator wants to compute the statement (x, y, t, π) at the root. This can be done by skipping the sketch nodes for intermediate states and only computing a proof for the final output $y = x^{2^t}$, which in total involves t squarings (corresponding to computing the leaves of a k-ary tree of height $\log_k t$) along with the sketch node for the root. However, for an honest evaluator, this requires computing $(k+1)^{\log_k t}$ leaf nodes (corresponding to every leaf in a $(k+1)$-ary tree of height $\log_k t$). Therefore, the ratio is $\alpha = ((k+1)/k)^{\log_k t}$. In order to get the tight gap, we choose k to be proportional to the security parameter so that $\alpha = (1 + o(1)) \cdot t$. This change is crucial (as we eluded towards above), as otherwise if k is a constant, the relative overhead would be significant. Indeed, in Pietrzak's protocol, $k = 2$ and computing the sketch node constitutes a constant fraction of the computation.

2.3 The Efficiency of our Construction

In this section, we briefly compare the efficiency of our constructions to previous ones which are based on repeated squaring. Specifically, we discuss Wesolowki's VDF [Wes19] (denoted WVDF), Pietrzak's VDF [Pie19] (denoted PVDF), in comparison to our cVDF using a tree of arity k (denoted k-cVDF) and the VDF underlying it (denoted k-VDF), which is simply Pietrzak's VDF with arity k.

For proof length corresponding to t squares, the WVDF proof is just a single group element, and the PVDF proof consists of $\log_2(t)$ group elements. For the k-VDF, generalizing Pietrzak's VDF to use a tree with arity k results in a proof with $(k-1) \cdot \log_k(t)$ group elements. Finally, the k-cVDF output consists of a frontier with at most $(k-1)$ proofs for a k-VDF in each of $\log_k(t)$ levels of the tree, resulting in $(k-1)^2(\log_k(t))^2$ group elements. In all cases, verifying a proof with n group elements requires doing $O(n \cdot \lambda)$ squares. For prover efficiency, the honest prover can compute the proof in the time to do $t(1 + o(t))$ squares (when $t \in \mathrm{poly}(\lambda)$ and $k \in \Omega(\log \lambda)$ for the k-cVDF).

In the full cVDF construction, we set k to be equal to λ for simplicity, but as the above shows, different values of k give rise to different efficiency trade-offs.

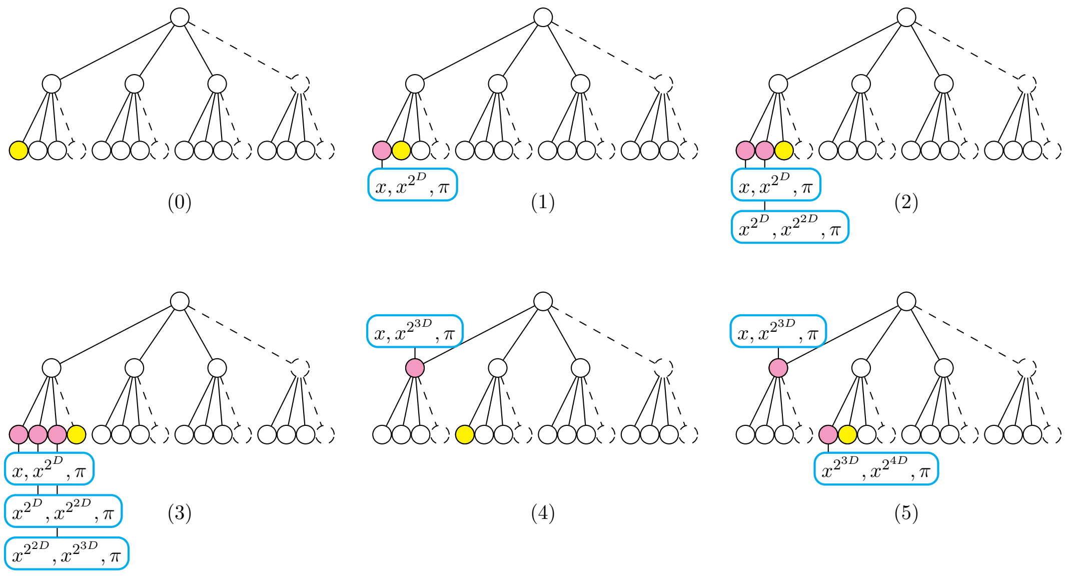
(0)
x, x^{2^D}, π
(1)
x, x^{2^D}, π
$x^{2^D}, x^{2^{2D}}, \pi$
(2)
x, x^{2^D}, π
$x^{2^D}, x^{2^{2D}}, \pi$
$x^{2^{2D}}, x^{2^{3D}}, \pi$
(3)
$x, x^{2^{3D}}, \pi$
(4)
$x, x^{2^{3D}}, \pi$
$x^{2^{3D}}, x^{2^{4D}}, \pi$
(5)

Figure 1: The first six states of our continuous VDF with $k = 3$ and base difficulty $D = k^{d'}$ for a constant d'. In each tree, the segment nodes are given by solid lines and the sketch nodes by dashed lines. The yellow node is the current leaf, and the pink nodes are its frontier. The values in blue are contain (x, y, π) for the corresponding node. The proofs π at leaf nodes with input x and output y correspond to the underlying VDF proof that $x^{2^D} = y$, and the proofs at each higher node consist of its segments (outputs of k first children) and of the proof of the sketch node (the $(k+1)$st child).

3 Preliminaries

Basic notation. For a distribution X we denote by $x \leftarrow X$ the process of sampling a value x from the distribution X. For a set $\mathcal{X}$, we denote by $x \leftarrow \mathcal{X}$ the process of sampling a value x from the uniform distribution on $\mathcal{X}$. Supp(X) denotes the support of the distribution X. For an integer $n \in \mathbb{N}$ we denote by $[n]$ the set $\{1, 2, \ldots, n\}$. We use null to denote the empty string. We use PPT as an acronym for *probabilistic polynomial time*.

A function $\mathsf{negl}\colon \mathbb{N} \to \mathbb{R}$ is *negligible* if it is asymptotically smaller than any inverse-polynomial function, namely, for every constant $c > 0$ there exists an integer N_c such that $\mathsf{negl}(\lambda) \leq \lambda^{-c}$ for all $\lambda > N_c$. Two sequences of random variables $X = \{X_\lambda\}_{\lambda\in\mathbb{N}}$ and $Y = \{Y_\lambda\}_{\lambda\in\mathbb{N}}$ are *computationally indistinguishable* if for any non-uniform PPT algorithm $\mathcal{A}$ there exists a negligible function negl such that $\left|\Pr\left[\mathcal{A}(1^\lambda, X_\lambda) = 1\right] - \Pr\left[\mathcal{A}(1^\lambda, Y_\lambda) = 1\right]\right| \leq \mathsf{negl}(\lambda)$ for all $\lambda \in \mathbb{N}$. We say that an algorithm runs in polynomial time if it runs in time polynomial in the length of its first input. For a language L with relation R_L, we let $R_L(x)$ denote the set of witnesses w such that $(x, w) \in R_L$.

Boolean circuits. Boolean circuits are directed acyclic graphs where each node represents an input gate, output gate, or Boolean gate, and each edge represents a wire in the circuit. The *size* of a circuit is the number of wires in it. The *depth* of a circuit is the length of the longest path from an input gate to an output gate. The *width* of a circuit with depth d is the maximum, over all $i \leq d$, of the number of gates at distance i from an input gate. For a circuit C, we let $\mathsf{size}(C)$,

[12]The term frontier is standard in the algorithms literature. Many other names have been used to describe this notion, such as dangling nodes in [CLP13] and unfinished nodes in [DLM19].

$\mathsf{depth}(C)$, and $\mathsf{width}(C)$ denote the size, depth, and width, respectively.

Number theory. Let $N = p \cdot q$ where p, q are safe primes in $[2^\lambda, 2^{\lambda+1})$ (a prime p is safe if $p = 2p' + 1$ and p' is prime). We recall that $\mathbb{Z}_N^\star$ consists of all integers in $[N]$ that are relatively prime to N (namely, $\mathbb{Z}_N^\star = \{x \in \mathbb{Z}_N \colon \gcd(x, N) = 1\}$). We define the subgroup QR_N to be the set of quadratic residues in $\mathbb{Z}_N^\star$, i.e., μ such that there exists a $x \in \mathbb{Z}_N^\star$ where $\mu = x^2$ (namely, $\mathsf{QR}_N = \{x^2 \colon x \in \mathbb{Z}_N^\star\}$).

By the Chinese Remainder Theorem, each element x in $\mathbb{Z}_N^\star$ can be uniquely represented by two integers (a, b) where $x = a \bmod p$ and $x = b \bmod q$. Let $x, y \in \mathbb{Z}_N^\star$ with representations $(a, b), (c, d)$, respectively. It is straightforward to show that the unique representation of $-x$ is $(-a, -b)$, $x + y$ is $(a + c, b + d)$, and $x \cdot y$ is $(a \cdot c, b \cdot d)$. We define $\langle x \rangle$ to be the cyclic group generated by x, i.e., $\langle x \rangle = \{x, x^2, x^3, \ldots\}$. We say that x is a generator for a (sub-)group G if $\langle x \rangle = G$.

3.1 Verifiable, Sequential, and Iteratively Sequential Functions

In this section, we define different properties of functions which will be useful in subsequent sections when we define unique VDFs (Definition 5.1) and continuous VDFs (Definition 6.1). All of our definitions will be in the public parameter model. We start by defining a verifiable function.

Definition 3.1 (Verifiable Functions)**.** *Let $B \colon \mathbb{N} \to \mathbb{N}$. A B-sound* verifiable function *is a tuple of algorithms* $(\mathsf{Gen}, \mathsf{Eval}, \mathsf{Verify})$ *where* Gen *is PPT,* Eval *is deterministic, and* Verify *is deterministic polynomial-time, satisfying the following property:*

- **Perfect Completeness.** *For every $\lambda \in \mathbb{N}$, $\mathsf{pp} \in \operatorname{Supp}(\mathsf{Gen}(1^\lambda))$, and $x \in \{0,1\}^*$, it holds that*
$$\mathsf{Verify}(1^\lambda, \mathsf{pp}, x, \mathsf{Eval}(1^\lambda, \mathsf{pp}, x)) = 1.$$

- ***B*-Soundness.** *For every non-uniform algorithm $\mathcal{A} = \{\mathcal{A}_\lambda\}_{\lambda \in \mathbb{N}}$ such that $\mathsf{size}(A_\lambda) \in \operatorname{poly}(B(\lambda))$ for all $\lambda \in \mathbb{N}$, there exists a negligible function* negl *such that for every $\lambda \in \mathbb{N}$ it holds that*
$$\Pr\left[\begin{array}{l} \mathsf{pp} \leftarrow \mathsf{Gen}(1^\lambda) \\ (x, y) \leftarrow \mathcal{A}_\lambda(\mathsf{pp}) \end{array} : \mathsf{Verify}(1^\lambda, \mathsf{pp}, x, y) = 1 \wedge \mathsf{Eval}(1^\lambda, \mathsf{pp}, x) \neq y\right] \leq \mathsf{negl}(\lambda).$$

Next, we define a sequential function. At a high level, this is a function f implemented by an algorithm Eval that takes input (x, t), such that computing $f(x, t)$ requires time roughly t, even with parallelism. Our formal definition is inspired by [BBBF18]. Intuitively, it requires that any algorithm $\mathcal{A}_{0,\lambda}$ which first pre-processes the public parameters cannot output a circuit $\mathcal{A}_1$ satisfying the following. Upon receipt of a freshly sampled input x, $\mathcal{A}_1$ outputs a value y and a difficulty t, where y is the output of Eval on x for difficulty t, where t is sufficiently larger than its depth. This captures the notion that $\mathcal{A}_1$ manages to compute y in less than t time, even with large width.

Definition 3.2. *Let $D, B, \ell \colon \mathbb{N} \to \mathbb{N}$ and let $\epsilon \in (0, 1)$. A (D, B, ℓ, ϵ)*-sequential function *is a tuple* $(\mathsf{Gen}, \mathsf{Sample}, \mathsf{Eval})$ *where* Gen *and* Sample *are PPT,* Eval *is deterministic, and the following properties hold:*

- **Honest Evaluation.** *There exists a uniform circuit family $\{C_{\lambda,t}\}_{\lambda,t \in \mathbb{N}}$ such that $C_{\lambda,t}$ computes $\mathsf{Eval}(1^\lambda, \cdot, (\cdot, t))$, and for all sufficiently large $\lambda \in \mathbb{N}$ and $D(\lambda) \leq t \leq B(\lambda)$, it holds that $\mathsf{depth}(C_{\lambda,t}) = t \cdot \ell(\lambda)$ and $\mathsf{width}(C_{\lambda,t}) \in \operatorname{poly}(\lambda)$.*

- **Sequentiality.** *For all non-uniform algorithms* $\mathcal{A}_0 = \{\mathcal{A}_{0,\lambda}\}_{\lambda\in\mathbb{N}}$ *such that* $\mathsf{size}(\mathcal{A}_{0,\lambda}) \in \mathrm{poly}(B(\lambda))$ *for all* $\lambda \in \mathbb{N}$*, there exists a negligible function* negl *such that for every* $\lambda \in \mathbb{N}$*,*

$$\Pr\left[\begin{array}{l} \mathsf{pp} \leftarrow \mathsf{Gen}(1^\lambda) \\ \mathcal{A}_1 \leftarrow \mathcal{A}_{0,\lambda}(\mathsf{pp}) \\ x \leftarrow \mathsf{Sample}(1^\lambda, \mathsf{pp}) \\ (t, y) \leftarrow \mathcal{A}_1(x) \end{array} : \begin{array}{l} \mathsf{Eval}(1^\lambda, \mathsf{pp}, (x,t)) = y \\ \wedge\ \mathsf{depth}(\mathcal{A}_1) \le (1-\epsilon)\cdot t \cdot \ell(\lambda) \\ \wedge\ t \ge D(\lambda) \end{array}\right] \le \mathsf{negl}(\lambda).$$

Next, we define an iteratively sequential function. This is a function f implemented by an algorithm Eval, such that the t-wise composition of f cannot be computed faster than computing f sequentially t times, even using parallelism. We also require that the length of the output of f is bounded, so that it does not grow with the number of compositions.

Definition 3.3 (Iteratively Sequential Function)**.** *Let* $D, B, \ell \colon \mathbb{N} \to \mathbb{N}$ *be functions and let* $\epsilon \in (0,1)$*. A tuple of algorithms* $(\mathsf{Gen}, \mathsf{Sample}, \mathsf{Eval})$ *is a* (D, B, ℓ, ϵ)-iteratively sequential function *if* Gen *and* Sample *are PPT,* Eval *is deterministic, and the following properties hold.*

- **Length Bounded.** *There exists a polynomial* m *such that for every* $\lambda \in \mathbb{N}$ *and* $x \in \{0,1\}^*$*, it holds that* $\left|\mathsf{Eval}(1^\lambda, \mathsf{pp}, x)\right| \le m(\lambda)$*. We define* $\mathsf{Eval}^{(\cdot)}$ *to be the function that takes as input* $1^\lambda, \mathsf{pp}$*, and* (x, T) *and represents the* T*-wise composition given by*

$$\mathsf{Eval}^{(T)}(1^\lambda, \mathsf{pp}, x) \overset{\mathsf{def}}{=} \underbrace{\mathsf{Eval}(1^\lambda, \mathsf{pp}, \cdot) \circ \ldots \circ \mathsf{Eval}(1^\lambda, \mathsf{pp}, \cdot)}_{T \textit{ times}}(x)$$

and note that this function is also length bounded.

- **Iteratively sequential.** *The tuple* $(\mathsf{Gen}, \mathsf{Sample}, \mathsf{Eval}^{(\cdot)})$ *is a* (D, B, ℓ, ϵ)*-sequential function.*

Remark 1 (Decoupling size and depth)**.** *We note that one can also consider a generalization of a* (D, B, ℓ, ϵ)*-sequential function to a* $(D, U, B, \ell, \epsilon)$*-sequential function (and thus iteratively sequential functions), where the size of* $\mathcal{A}_{0,\lambda}$ *remains bounded by* $\mathrm{poly}(B(\lambda))$*, but the parameter* t *output by* $\mathcal{A}_1$ *must be at most* $U(\lambda)$*.*

3.2 Repeated Squaring Assumption

The repeated squaring assumption (henceforth, the RSW assumption[13]) roughly says that there is no parallel algorithm that can perform t squarings modulo an RSA integer N significantly faster than just performing t squarings sequentially. This implicitly assumes that N cannot be factored efficiently. This assumption has been very useful for various applications (e.g., time-lock puzzles [RSW96], reliable benchmarking [CLSY93], and timed commitments [BN00, LPS17] and to date there is no known strategy that beats the naive sequential one.

Define $\mathsf{RSW} = (\mathsf{RSW.Gen}, \mathsf{RSW.Sample}, \mathsf{RSW.Eval})$ as follows.

- $N \leftarrow \mathsf{RSW.Gen}(1^\lambda)$**:**
  Sample random primes p', q' from $[2^\lambda, 2^{\lambda+1})$ such that $p = 2p'+1$ and $q = 2q'+1$ are prime, and output $N = p \cdot q$.

- $x \leftarrow \mathsf{RSW.Sample}(1^\lambda, N)$**:**
  Sample and output a random element $g \leftarrow \mathbb{Z}_N^\star$.

[13]The assumption is usually called the RSW assumption after Rivest, Shamir, and Wagner who used it to construct time-lock puzzles [RSW96].

- $y \leftarrow \mathsf{RSW.Eval}(1^\lambda, N, g)$:
  Output $y = g^2 \bmod N$.

Assumption 3.4 (RSW Assumption)**.** *Let $D, B \colon \mathbb{N} \to \mathbb{N}$. The* (D, B)-RSW assumption *is that there exists a polynomial $\ell \in \mathbb{N} \to \mathbb{N}$ and constant $\epsilon \in (0, 1)$ such that* RSW *is a (D, B, ℓ, ϵ)-iteratively sequential function.*

Note that the RSW assumption implies that factoring is hard. Namely, no adversary can factor an integer $N = p \cdot q$ where p and q are large "safe" primes (a prime p is safe if $p - 1$ has two factors, 2 and p', for some prime number $p' \in [2^\lambda, 2^{\lambda+1})$).

3.3 Interactive Protocols and the Fiat-Shamir Heuristic

We consider interactive Turing machines (ITM) and interactive protocols. Given a pair of ITMs P and V, we denote by $\langle P(x), V(y)\rangle(z)$ the random variable representing the output of V with common input z and private input y, when interacting with P with private input x, when the random tape of each machine is uniformly and independently chosen. We also let $\mathsf{View}_V(P(x), V(y))(z)$ be the random variable representing V's view in such an interaction, which consists of the prover's messages and its own random coins. The round complexity of a protocol $\langle P(x), V(y)\rangle$ is the number of distinct messages sent between P and V. We say that a protocol is non-interactive if it consists of one message from P to V and then V computes its output. Additionally, we say that a protocol is public-coin if all of V's messages to P consist of random coins.

Definition 3.5 (δ-Sound Interactive Proof)**.** *A pair of ITMs (P, V) where V is PPT is called an* interactive proof *with soundness δ for a language L with witness relation R_L if the following two conditions hold:*

- Completeness*: For every $\lambda \in \mathbb{N}$, $x \in L$, and every $w \in R_L(x)$,*

$$\langle P(w), V\rangle(1^\lambda, x) = 1.$$

- Soundness: *For every (possibly inefficient) machine $P^\star = (P_1^\star, P_2^\star)$ and every $\lambda \in \mathbb{N}$,*

$$\Pr[(x, z) \leftarrow P_1^*(1^\lambda) : x \notin L \wedge \langle P_2^*(z), V\rangle(1^\lambda, x) = 1] \leq \delta(\lambda).$$

If the soundness condition only holds against non-uniform PPT *machines $(P_1^\star, P_2^\star)$, the pair (P, V) is called an* interactive argument. *Additionally, (P, V) is an interactive proof or argument* with an inefficient verifier *if V is allowed to be inefficient.*

We omit the prefix *δ-sound* when there exists a negligible function negl such that the interactive protocol is negl-sound. We also consider non-interactive protocols in the public parameters model, where part of the common input is assumed to be generated honestly by a trusted third party and the properties hold where the probabilities are also over the the trusted third party's randomness.

The Fiat-Shamir heuristic. For any public-coin interactive protocol (P, V) and a hash function family $\mathcal{H}$, the Fiat-Shamir (FS) transformation [FS86] relative to $\mathcal{H}$ replaces messages from V with the hash of the transcript so far using a hash function $\mathsf{hash} \leftarrow \mathcal{H}$. Moreover, hash is specified in the public parameters of the final protocol. This results in a non-interactive argument $(P_{\mathsf{FS}}, V_{\mathsf{FS}})$ where P_{FS} sends the final transcript to V_{FS}, and V_{FS} accepts if all of the messages sent by V in the protocol were generated correctly using hash, and if V accepts.

Assumption 3.6 (α-round FS)**.** *There exists a hash function family $\mathcal{H}$ such that for any $\alpha(\lambda)$-round public-coin interactive proof (P, V) with $\mathsf{negl}(\lambda^{\alpha(\lambda)})$-soundness for some negligible function negl, the non-interactive argument $(P_{\mathsf{FS}}, V_{\mathsf{FS}})$ obtained by applying Fiat-Shamir transformation to (P, V) relative to $\mathcal{H}$ has $\mathsf{negl}(\lambda)$-soundness.*

It follows from [GK90] that the α-round FS assumption holds when instantiating hash with a random oracle. In particular, [GK90] shows that if you can break the soundness of the non-interactive FS protocol with probability $1/p(\lambda)$ using $q(\lambda)$ queries to the random oracle, then you can break the interactive protocol with probability $1/(p(\lambda) \cdot q(\lambda)^{\alpha(\lambda)})$. For non-uniform PPT adversaries, $q(\lambda) \in \text{poly}(\lambda)$, so if the interactive protocol satisfies $\mathsf{negl}(\lambda^{\alpha(\lambda)})$-soundness, then the non-interactive FS protocol satisfies $\mathsf{negl}(\lambda)$-soundness.

Recent works presented concrete hash functions (in the plain model) which can be used to instantiate the assumption for all public-coin interactive proof satisfying additional special properties (e.g., [CCRR18, CCH+19]). The n-round assumption FS assumption is known to be false for general arguments, including constant-round ones (e.g., [Bar01, GK03]).

We additionally consider a stronger version of the FS assumption, which assumes that the FS transformation preserves soundness even when the verifier may be inefficient.

Assumption 3.7 (α-round Strong FS)**.** *The α-round FS assumption holds also for any public-coin interactive proof (P, V) with an inefficient verifier.*

The proof of [GK90] also shows that the strong FS assumption holds when instantiating hash with a random oracle.

4 Interactive Proof for Repeated Squaring

In this section, we give an interactive proof for a language representing t repeated squarings. As discussed in Section 2, this protocol is based on that of [Pie19]. We start with an overview. The common input includes an integer t and two values $\widehat{x}_0, \widehat{y} \in \mathbb{Z}_N^\star$, where, for the purpose of this overview, the goal is for the prover to convince the verifier that $\widehat{y} = (\widehat{x}_0)^{2^t} \bmod N$. The protocol is defined recursively.

Starting with a statement $(\widehat{x}_0, \widehat{y}, t)$, where we assume for simplicity that t is a power of k, the prover splits x_0 into k "segments", where each segment is t/k "steps" of the computation of $(\widehat{x}_0)^{2^t} \bmod N$. The ith segment is recursively defined as the value $(\widehat{x}_{i-1})^{2^{t/k}}$. In other words, $\widehat{x}_i = (\widehat{x}_0)^{2^{i \cdot t/k}}$ for all $i \in \{0, 1, \ldots, k\}$. If one can verify the values of $\widehat{x}_1, \ldots, \widehat{x}_k$, then one can also readily verify that $\widehat{y} = (\widehat{x}_0)^{2^t} \bmod N$. To verify the values of $\widehat{x}_1, \ldots, \widehat{x}_k$ efficiently we rely on interaction and require the prover to convince the verifier that the values $\widehat{x}_1, \ldots, \widehat{x}_k$ are consistent (in some sense) under a random linear relation. To this end, the prover and verifier engage in a second protocol to prove a modified statement $(\widehat{x}'_0, \widehat{y}', t/k)$ which combines all the segments and should only be true if all segments are true (with high probability). The modified statement is proved in the same way, where the exponent t/k is divided by k with each new statement. This process is continued $\log_k t$ times until the statement to verify can be done by direct computation.

For soundness of our protocol, we need to bound the probability of a cheating prover jumping from a false statement in the beginning of the protocol to a true statement in one of the subsequent protocols. One technical point is that to accomplish this, we work in the subgroup QR_N of $\mathbb{Z}_N^\star$ and thus we want the starting point $\widehat{x}_0$ to generate QR_N. To accommodate this, we let the prover provide a square root of every group element as a witness to the fact that it is in QR_N (actually, by Fact A.3, this will imply that all group elements generate QR_N). Therefore, rather than working with $\widehat{x}_0$ and

$\widehat{y}$ directly, we work with their square roots x_0 and y, respectively. Hence, the common input consists of an integer t and (x_0, y), where the goal is actually to prove that $y^2 = (x_0^2)^{2^t} = x_0^{2^{t+1}} \bmod N$.

Note that, in general, the square root x_0 is not unique in $\mathbb{Z}_N^\star$ for a given square x_0^2. Indeed, there are 4 square roots $\pm x_0, \pm x_0'$. In our protocol, the computationally bounded prover can compute only two of them, either $\pm x_0$ or $\pm x_0'$, as otherwise, by Fact A.2 we could use the prover to factor N. Among the two square roots that the prover can compute, we canonically decide that the prover must use the smaller one. This gives rise to our definition of a *valid* element x: $x^2 \bmod N$ generates QR_N and $x = |x|_N$, formally defined in Definition 4.1.

4.1 Protocol

Before presenting the protocol, we first define the language. Toward that goal, we start with the formal definition of a valid element.

Definition 4.1 (Valid element)**.** *For any $N \in \mathbb{N}$ and $x \in \{0,1\}^*$, we say that x is a* valid element *if $x \in \mathbb{Z}_N^\star$, $\langle x^2 \rangle = \mathsf{QR}_N$, and $x = |x|_N$. We say that a sequence of elements $(x_1, \ldots, x_\ell)$ is a* valid sequence *if each element x_i is a valid element.*

By Fact A.1, whenever N is in the support of $\mathsf{RSW.Gen}(1^\lambda)$, validity can be tested in polynomial time by verifying that $x = |x|_N$, and that $\gcd(x \pm 1, N) = 1$ (and outputting 1 if and only if all checks pass). This algorithm naturally extends to one that receives as input a sequence of pairs and verifies each separately.

The language for our interactive proof, $\mathcal{L}_{N,B}$, is parametrized by integers $N \in \operatorname{Supp}\big(\mathsf{RSW.Gen}(1^\lambda)\big)$ and $B = B(\lambda)$, and it is defined as:

$$\mathcal{L}_{N,B} = \left\{ (x_0, y, t) : \begin{array}{l} y^2 = (x_0)^{2^{t+1}} \bmod N \text{ if } x_0 \text{ is valid and } t \le B, \\ y = \bot \text{ otherwise} \end{array} \right\}.$$

Intuitively, $\mathcal{L}_{N,B}$ should be thought of as the language of elements x_0, y where x_0 is valid and $x_0^{2^{t+1}} = y^2 \bmod N$. To be well-defined on any possible statement with $x_0, y \in \mathbb{Z}_N^\star$ and $t \in \mathbb{N}$, we include statements with invalid elements x_0 in the language. Since the verifier can test validity efficiently, this language still enforces that valid elements represent repeated squaring.

Our protocol $\Pi_{\lambda,k,d}$, given in Figure 2, is parametrized by the security parameter λ, as well as integers $k = k(\lambda)$ and $d = d(\lambda)$, where k is the number of segments into which we split each statement and d is a "cut-off" parameter that defines the base of the recursive protocol.

4.2 Proofs

In this section we show that $\Pi_{\lambda,k,d}$ is an interactive proof for the language $\mathcal{L}_{N,B}$ by showing completeness and soundness. Furthermore, we prove an additional property which roughly shows that any cheating prover cannot deviate in a specific way from the honest prover strategy even for statements in the language.

Theorem 4.2. *For any $\lambda \in \mathbb{N}$, $k = k(\lambda)$, $d = d(\lambda)$, $B = B(\lambda)$, and $N \in \operatorname{Supp}\big(\mathsf{RSW.Gen}(1^\lambda)\big)$, the protocol $\Pi_{\lambda,k,d}$ (given in Figure 2) is a $(\log_k(B) - d) \cdot 3/2^\lambda$-sound interactive proof for $\mathcal{L}_{N,B}$.*

We prove Theorem 4.2 by showing completeness in Lemma 4.3 and soundness in Lemma 4.4, respectively.

Lemma 4.3 (Completeness)**.** $\langle P, V \rangle (x_0, y, t) = 1$ *for any* $(x_0, y, t) \in \mathcal{L}_{N,B}$.

INTERACTIVE PROOF $\Pi_{\lambda,k,d} = (P, V)$ ON COMMON INPUT (x_0, y, t)

Prover $P \to$ Verifier V:

1. If x_0 is an invalid element (Definition 4.1), $t \leq k^d$, or $t > B$, send $\mathsf{msg}_P = \perp$ to V.
2. Otherwise, for $i \in [k-1]$, compute $x_i = \left|x_0^{2^{i \cdot t/k}}\right|_N$.
3. Send $\mathsf{msg}_P = (x_1, \ldots, x_{k-1})$ to V.

Verifier $V \to$ Prover P:

1. If x_0 is an invalid element or $t > B$, output 1 if $\mathsf{msg}_P = y = \perp$ and 0 otherwise.
2. If $|y|_N$ is invalid, output 0.
3. If $t \leq k^d$, output 1 if both $y^2 = (x_0)^{2^{t+1}} \bmod N$ and $\mathsf{msg}_P = \perp$ and output 0 otherwise.
4. Output 0 if msg_P is an invalid sequence.
5. Send $\mathsf{msg}_V = (r_1, \ldots, r_k) \leftarrow [2^\lambda]^k$ to P.

Prover $P \leftrightarrow$ Verifier V:

1. Let $x_0' = \left|\prod_{i=1}^k x_{i-1}^{r_i}\right|_N$ and $y' = \left|\prod_{i=1}^k x_i^{r_i}\right|_N$, where $x_k = y$. Note that both P and V can efficiently compute x_0', y' given msg_P, msg_V, and the common inputs. If x_0' is invalid, let $y' = \perp$.
2. Output the result of $\Pi_{\lambda,k,d}$ on common input $(x_0', y', t/k)$.

Figure 2: Interactive Proof $\Pi_{\lambda,k,d}$ for $\mathcal{L}_{N,B}$

Proof. First, note that if x_0 is an invalid element or $t > B$, then $y = \perp$. In this case, the honest prover P sends $\perp$ to V, so V outputs 1 and completeness holds. Therefore, it suffices to show the case where x_0 is valid and $t \leq B$. Since $(x_0, y, t) \in \mathcal{L}_{N,B}$ this implies that $y^2 = (x_0)^{2^{t+1}} \bmod N$, and thus $|y|_N$ is valid by Fact A.3. We proceed to show this case by induction on t.

When $t \leq k^d$, the honest prover P sends $\perp$ to V. V then outputs 1 since $y^2 = (x_0)^{2^{t+1}} \bmod N$, so completeness holds.

For $t > k^d$, we inductively argue that completeness holds assuming it holds for all smaller values of t. The honest prover P sends $\mathsf{msg}_P = (x_1, \ldots, x_{k-1})$ where $x_i = \left|(x_0)^{2^{i \cdot t/k}}\right|_N$ for all $i \in [k-1]$. Since x_0 is valid by assumption, all x_i will be valid by Fact A.3. Then V will accept if and only if it accepts in the recursive step, so it suffices to show that the randomly generated statement $(x_0', y', t/k) \in \mathcal{L}_{N,B}$. First we note that there is some chance that x_0' is invalid, at which case the statement is vacuously in $\mathcal{L}_{N,B}$ since then $y' = \perp$. Otherwise, when x_0' is valid, then

$$(y')^2 = \prod_{i=1}^{k} x_i^{2r_i} = \prod_{i=1}^{k} (x_{i-1})^{r_i \cdot 2^{t/k+1}} = (x_0')^{2^{t/k+1}} \bmod N,$$

so the recursive statement is in $\mathcal{L}_{N,B}$. Therefore completeness holds by our inductive hypothesis. □

Lemma 4.4 (Soundness). *For any unbounded cheating prover $P^\star$ and any $(x_0, y, t) \notin \mathcal{L}_{N,B}$ it holds that*

$$\Pr[\langle P^\star, V\rangle(x_0, y, t) = 1] \leq \max(0, (\log_k(t) - d) \cdot 3/2^\lambda).$$

Proof. Since $(x_0, y, t) \notin \mathcal{L}_{N,B}$, if x_0 is invalid or $t > B$, then $y \neq \bot$. However, in this case V rejects since $y \neq \bot$. Therefore, suppose x_0 is valid and $t \leq B$. Then, it must be the case that $y^2 \neq (x_0)^{2^{t+1}} \bmod N$. We proceed by induction on t.

In the base case when $t \leq k^d$, V outputs 1 only if $y^2 = (x_0)^{2^{t+1}} \bmod N$, so the probability is 0. Now, consider $t > k^d$ and assume soundness holds for all values smaller than t. Let $\mathsf{msg}_{P^\star} = (x_1, \ldots, x_{k-1})$, let $x_k = y$, and let $\mathsf{msg}_V = (r_1, \ldots, r_k)$ be the output of $V(\mathsf{msg}_{P^\star})$.

Since x_0 is valid, then $\langle x_0^2 \rangle = \mathsf{QR}_N$ so there exist unique values $a_i \in [p' \cdot q']$ such that $x_i^2 = (x_0)^{2 \cdot a_i} \bmod N$ for all $i \in \{0, 1, \ldots, k\}$. This implies that

$$(x_0')^2 = \prod_{i=1}^{k} (x_0)^{2 \cdot a_{i-1} \cdot r_i} = (x_0)^{2 \sum_{i=1}^{k} a_{i-1} \cdot r_i} \bmod N$$

and

$$(y')^2 = \prod_{i=1}^{k} (x_0)^{2 \cdot a_i \cdot r_i} = (x_0)^{2 \sum_{i=1}^{k} a_i \cdot r_i} \bmod N.$$

We want to bound the probability that $(x_0', y', t/k) \in \mathcal{L}_{N,B}$. Towards doing so, we split the probability space into two separate events: (1) either x_0' is invalid, or (2) $(y')^2 = (x_0')^{2^{t+1}} \bmod N$. We show in Claim 4.5 that (1) occurs with probability at most $2/2^\lambda$ and in Claim 4.6 that (2) occurs with probability at most $1/2^\lambda$. Given these events don't occur, the probability that V outputs 1 is at most $3 \cdot (\log_k(t/k) - d)/2^\lambda$ by the inductive hypothesis. So assuming these claims, V outputs 1 with probability at most $3/2^\lambda + (\log_k(t/k) - d) \cdot 3/2^\lambda = (\log_k t - d) \cdot 3/2^\lambda$.

Claim 4.5. *For any instance (x_0, y, t) with $t > k^d$ such that x_0 is a valid element, it holds that*

$$\Pr_{r_1,\ldots,r_k}\left[x_0' \text{ is invalid}\right] \leq 2/2^\lambda.$$

Proof. We start by bounding the probability that x_0' is invalid. By definition, $x_0' = |x_0'|_N$, so we bound the probability that $\langle (x_0')^2 \rangle \neq \mathsf{QR}_N$.

Since x_0 is valid, $\langle (x_0)^2 \rangle = \mathsf{QR}_N$, so there exist unique values $a_i \in [p' \cdot q']$ such that $x_i^2 = (x_0^2)^{a_i} \bmod N$ for all $i \in \{0, 1, \ldots, k\}$ (since $x_i^2 \in \mathsf{QR}_N$ by definition of QR_N). This implies that

$$(x_0')^2 = \prod_{i=1}^{k} (x_0^2)^{a_{i-1} \cdot r_i} = (x_0^2)^{\sum_{i=1}^{k} a_{i-1} \cdot r_i} \bmod N.$$

Observe that $\langle (x_0')^2 \rangle \neq \mathsf{QR}_N$ whenever its exponent (with respect to x_0^2) is equal to 0 modulo p' or q', i.e., $\sum_{i=1}^{k} a_{i-1} \cdot r_i = 0 \bmod p'$ or q'. Since $a_0 = 1$, this is equivalent to

$$r_1 = -\sum_{i=2}^{k} a_{i-1} \cdot r_i \bmod p' \text{ or } q'.$$

For any values of $r_2, \ldots, r_k$, this defines at most two unique values for $r_1 \in [2^\lambda]$ satisfying the above equation since $p', q' \geq 2^\lambda$. Thus, the probability over $r_1 \leftarrow [2^\lambda]$ that r_1 satisfies the above is at most $2/2^\lambda$.

Claim 4.6. *For any instance $(x_0, y, t) \notin \mathcal{L}_{N,B}$ where x_0 is a valid element and $t > k^d$, it holds that*

$$\Pr_{r_1,\ldots,r_k}\left[(y')^2 = (x_0')^{2^{t+1}} \bmod N\right] \leq 1/2^\lambda.$$

Proof. Recall that since x_0 is valid then $\langle (x_0)^2 \rangle = \mathsf{QR}_N$. When we view $(y')^2$ and $(x_0')^{2^{t/k+1}}$ as powers of $(x_0)^2$, the event of the claim occurs when the exponents are equivalent modulo the size of the group $|\mathsf{QR}_N| = p' \cdot q'$. Namely, $(y')^2 = (x_0')^{2^{t/k+1}} \bmod N$ if and only if

$$2\sum_{i=1}^{k} a_i \cdot r_i = 2^{t/k+1} \sum_{i=1}^{k} a_{i-1} \cdot r_i \bmod p' \cdot q'.$$

We are given that $y^2 \neq (x_0)^{2^{t+1}} \bmod N$ and $y = x_k$, so we know that $2a_k \neq 2^{t+1} \bmod p' \cdot q'$. This implies that there is some index $j \in [k-1]$ such that $2a_j \neq 2^{t/k+1} \cdot a_{j-1} \bmod p' \cdot q'$ (otherwise we would have that $2a_k = 2^{t+1} \bmod p' \cdot q'$). We separate this term out in the above equation to get that if $(y')^2 = (x_0')^{2^{t/k+1}} \bmod N$, then

$$r_j \cdot (2a_j - 2^{t/k+1} \cdot a_{j-1}) = -2 \sum_{i \neq j} r_i \cdot (a_i - 2^{t/k} a_{i-1}) \bmod p' \cdot q'.$$

We fix the values of r_i for $i \neq j$. Let

$$S = \{ r_j \cdot (2a_j - 2^{t/k+1} a_{j-1}) \bmod p' \cdot q' : r_j \in [2^\lambda] \}$$

be all possible values of the left-hand side. Given $r_j \leq 2^\lambda \leq p', q'$ and $(2a_j - 2^{t/k+1} \cdot a_{j-1}) \neq 0 \bmod p' \cdot q'$, we know that $|S| = 2^\lambda$. If the right-hand side is in S, then the probability over the choice of r_j that the equation holds is at most $1/2^\lambda$ and otherwise the probability is 0.

This completes the proof of Lemma 4.4. □

Let $\mathcal{L}_\perp$ be the empty language. Let $\widetilde{V}$ be an inefficient verifier for $\mathcal{L}_\perp$ that interacts with P (from $\Pi_{\lambda,k,d}$ for $\mathcal{L}_{N,B}$). On common input (x_0, y, t), the verifier $\widetilde{V}$ outputs 1 if and only if (1) x_0 is valid and $t \leq B$, (2) $\langle P, V \rangle (x_0, y, t) = 1$, (3) the first message from the prover satisfies $\mathsf{msg}_P \neq \perp$, and (4) when $\mathsf{msg}_P = (x_1, \ldots, x_{k-1})$ it holds that $x_i^2 \neq x_0^{2^{i \cdot t/k+1}} \bmod N$ for some $i \in [k-1]$. Intuitively, for any cheating prover $P^\star$, it holds that $\widetilde{V}$ accepts if $P^\star$ convinces V that a statement is in $\mathcal{L}_{N,B}$ even when $P^\star$ deviates from the protocol in a specific way. In particular, this holds even for true statements in $\mathcal{L}_{N,B}$. We note that this can be viewed as a relaxation of the notion of *unambiguous soundness* of Reingold, Rothblum, and Rothblum [RRR16].

We now show that $(P, \widetilde{V})$ is a interactive proof for $\mathcal{L}_\perp$. Completeness vacuously holds. We now show soundness.

Lemma 4.7 (Soundness of $(P, \widetilde{V})$)**.** *For any unbounded cheating prover $P^\star$ for $\mathcal{L}_\perp$ and statement (x_0, y, t), it holds that*

$$\Pr\left[\langle P^\star, \widetilde{V} \rangle (x_0, y, t) = 1 \right] \leq (\log_k(t) - d) \cdot 3/2^\lambda.$$

Proof. Let $\mathsf{msg}_{P^\star}$ be the message from the prover $P^\star$ on common input (x_0, y, t). First, note that if $\mathsf{msg}_{P^\star} = \perp$, then $\widetilde{V}$ rejects. Thus, $\mathsf{msg}_{P^\star} \neq \perp$. Then, note that if x_0 is an invalid element or $t > B$, then $\widetilde{V}$ rejects. Henceforth we assume that $\mathsf{msg}_{P^\star} = (x_1, \ldots, x_{k-1})$, x_0 is a valid element, and $t \leq B$.

At this point, $\widetilde{V}$ accepts if and only if V accepts and there exists an index $i \in [k-1]$ such that $x_i^2 \neq x_0^{2^{i \cdot t/k+1}} \bmod N$. Therefore, we show that V rejects when $x_i^2 \neq x_0^{2^{i \cdot t/k+1}} \bmod N$. Let $\mathsf{statement}' = (x_0', y', t/k)$ where $x_0' = \prod_{j=1}^{k} x_{j-1}^{r_j} \bmod N$ and $y' = \prod_{j=1}^{k} x_j^{r_j} \bmod N$. As in Lemma 4.4, it suffices to bound the probability of (1) x_0' is invalid, and (2) $(x_0')^{2^{t/k+1}} = (y')^2 \bmod N$. By Claim 4.5, (1) happens with probability at most $2/2^\lambda$. We show below that (2) occurs with probability at most $1/2^\lambda$, and thus conclude that V outputs 1 with probability at most $3/2^\lambda + 3(\log_k(t/k) - d)/2^\lambda \leq 3(\log_k(t) - d)/2^\lambda$ by Lemma 4.4.

To bound the probability of (2), we observe that a similar argument to Claim 4.6 follows. Specifically, we view each x_j^2 for $j \in \{0, 1, \ldots, k\}$ as a power of x_0^2, i.e., $x_j^2 = x_0^{2a_j}$ for a unique value $a_i \in [p' \cdot q']$. By assumption, there exists some index $i \in [k]$ such that $2a_i \neq 2^{i \cdot t/k+1} \bmod p' \cdot q'$. Note that if $i = k$, then the statement of Claim 4.6 holds, and otherwise, if $i \leq k-1$, then we can apply the same logic as in the proof of Claim 4.6. □

5 Unique Verifiable Delay Function

In this section, we use the Fiat-Shamir heuristic to transform the interactive proof for the language $\mathcal{L}_{N,B}$ corresponding to repeated squaring (given in Section 4) into a unique VDF.

Definition 5.1 (Unique Verifiable Delay Function)**.** *A* (D, B, ℓ, ϵ)-unique verifiable delay function (uVDF) *is a tuple* $(\mathsf{Gen}, \mathsf{Sample}, \mathsf{Eval}, \mathsf{Verify})$ *where* Eval *outputs a value* y *and a proof* π*, such that* $(\mathsf{Gen}, \mathsf{Sample}, \mathsf{Eval})$ *is a* (D, B, ℓ, ϵ)*-sequential function and* $(\mathsf{Gen}, \mathsf{Eval}, \mathsf{Verify})$ *is a* B*-sound verifiable function.*

5.1 Construction

For parameters k, d we define $(P_{\mathsf{FS}}, V_{\mathsf{FS}})$ to be the result of applying the Fiat-Shamir transformation to the protocol $\Pi_{\lambda,k,d}$ for $\mathcal{L}_{N,B}$ relative to some hash family $\mathcal{H}$. At a high level, this construction computes repeated squares and then uses P_{FS} and V_{FS} to prove and verify that this is done correctly.

We start by defining helper algorithms in Figure 3 based on the interactive protocol of Section 4. For notational convenience, we explicitly write algorithms $\mathsf{FS\text{-}Prove}$ and $\mathsf{FS\text{-}Verify}$, which take $\mathsf{pp} = (N, B, k, d, \mathsf{hash})$ as input, as well as $((x_0, t), y)$, where (N, B, k, d) correspond to the parameters of the non-interactive protocol and language, hash is the hash function sampled from the hash family $\mathcal{H}$ when applying the FS transform to $\Pi_{\lambda,k,d}$, and $((x_0, t), y)$ correspond to the statements of the language. We additionally define an efficient algorithm Sketch that outputs the statement for the recursive step in the interactive proof $\Pi_{\lambda,k,d}$.

We emphasize that the algorithms in Figure 3 are a restatement of the interactive protocol from Section 4 after applying the FS transform, given here only for ease of reading.

```
Sketch(pp, $(x_0, t), y$, msg):
    1. Parse msg $= (x_1, \dots, x_{k-1})$ and let $x_k = y$.
    2. Let $(r_1, \dots, r_k) = \mathsf{hash}(\mathsf{pp}, (x_0, t), y, \mathsf{msg})$.
    3. Let $x'_0 = \left|\prod_{i=1}^{k} x_{i-1}^{r_i}\right|_N$ and $y' = \left|\prod_{i=1}^{k} x_i^{r_i}\right|_N$.
    4. If $x'_0$ is invalid, let $y' = \perp$.
    5. Output $(x'_0, y')$.
FS-Prove(pp, $(x_0, t), y$):
    1. If $x_0$ is an invalid element (Definition 4.1), $t \leq k^d$, or $t > B$, output $\perp$.
    2. Let msg $= (x_1, \dots, x_{k-1})$ where $x_i = \left|(x_0)^{2^{i \cdot t/k}}\right|_N$.
    3. Compute $(x'_0, y') = \mathsf{Sketch}(\mathsf{pp}, (x_0, t), y, \mathsf{msg})$.
    4. Output $\pi = (\mathsf{msg}, \pi')$ where $\pi' = \mathsf{FS\text{-}Prove}(\mathsf{pp}, (x'_0, t/k), y')$.
FS-Verify(pp, $(x_0, t), y, \pi$):
    1. If $x_0$ is an invalid element or $t > B$, output 1 if $y = \pi = \perp$ and 0 otherwise.
    2. If $|y|_N$ is an invalid element, output 0.
    3. If $t \leq k^d$, output 1 if both $y^2 = (x_0)^{2^{t+1}} \bmod N$ and $\pi = \perp$ and output 0
       otherwise.
    4. Parse $\pi$ as $(\mathsf{msg}, \pi')$, and output 0 if msg is an invalid sequence.
    5. Compute $(x'_0, y') = \mathsf{Sketch}(\mathsf{pp}, (x_0, t), y, \mathsf{msg})$.
    6. Output $\mathsf{FS\text{-}Verify}(\mathsf{pp}, (x'_0, t/k), y', \pi')$.
```

Figure 3: Helper Algorithms for VDF for $\mathsf{pp} = (N, B, k, d, \mathsf{hash})$.

Next, we give a construction uVDF of a unique VDF consisting of algorithms (uVDF.Gen, uVDF.Sample, uVDF.Eval, uVDF.Verify) relative to a function $B \colon \mathbb{N} \to \mathbb{N}$.

- **pp $\leftarrow$ uVDF.Gen(1^λ):**

  Sample $N \leftarrow \mathsf{RSW.Gen}(1^\lambda)$, $\mathsf{hash} \leftarrow \mathcal{H}$, let $k = \lambda$, $B = B(\lambda)$, and let d be the constant specified in the proof of Lemma 5.12, and output $\mathsf{pp} = (N, B, k, d, \mathsf{hash})$.

- **$x_0 \leftarrow$ uVDF.Sample$(1^\lambda, \mathsf{pp})$:**

  Sample and output a random element $x_0 \leftarrow \mathbb{Z}_N^\star$ such that $\gcd(x_0 \pm 1, N) = 1$ and $x_0 = |x_0|_N$.[14]

- **$(y, \pi) \leftarrow$ uVDF.Eval$(1^\lambda, \mathsf{pp}, (x_0, t))$:**

  If x_0 is an invalid element, output $(\perp, \perp)$. If $t \leq k^d$, compute $y = \left|x_0^{2^t}\right|_N$ and output $(y, \perp)$.

  Otherwise, compute $x_i = \left|(x_0)^{i \cdot t/k}\right|_N$ for $i \in [k]$ and let $\mathsf{msg} = (x_1, \dots, x_{k-1})$ and $y = x_k$. Let $(x'_0, y') = \mathsf{Sketch}(\mathsf{pp}, (x_0, t), y, \mathsf{msg})$. Finally, output (y, π) where $\pi = (\mathsf{msg}, \pi')$ and $\pi' = \mathsf{FS\text{-}Prove}(\mathsf{pp}, (x'_0, t/k), y')$.

[14]We note that x_0 uniformly from $\mathbb{Z}_N^\star$ is sufficient due to the following. By Fact A.1, it holds that uVDF.Sample will succeed whenever $\langle x_0^2 \rangle = \mathsf{QR}_N$. Furthermore, x_0^2 is a random element of QR_N, and therefore is a generator with probability $1 - (p' + q')/(p' \cdot q') \geq 1 - 4/2^\lambda$. Also note that x_0 is distributed according to $\mathsf{RSW.Sample}(1^\lambda, N)$.

- $b \leftarrow \mathsf{uVDF.Verify}(1^\lambda, \mathsf{pp}, (x_0, t), (y, \pi))$:

  If x_0 is an invalid element or $t > B$, output 1 if $y = \pi = \perp$ and 0 if this is not the case. If y is invalid, then output 0. Otherwise, output $\mathsf{FS\text{-}Verify}(\mathsf{pp}, (x_0, t), y, \pi)$.

5.2 Proofs

Theorem 5.2. *Let $D, B, \alpha \colon \mathbb{N} \to \mathbb{N}$ be functions satisfying $D(\lambda) \in \omega(\lambda^2)$, $B(\lambda) \in 2^{O(\lambda)}$, and $\alpha(\lambda) \leq \lceil \log_\lambda(B(\lambda)) \rceil$. Suppose that the α-round strong FS assumption holds and the (D, B)-RSW assumption holds for polynomial $\ell \colon \mathbb{N} \to \mathbb{N}$ and constant $\epsilon \in (0, 1)$. Then, for any constants $\delta > 0$ and $\epsilon' > \frac{\epsilon+\delta}{1+\delta}$ it holds that* uVDF *is a $(D, B, (1+\delta)\cdot\ell, \epsilon')$-unique verifiable delay function.*

We prove Theorem 5.2 by showing verifiability (consisting of completeness in Lemma 5.3 and soundness in Lemma 5.4) and sequentiality (consisting of honest evaluation in Lemma 5.9 and sequentiality in Lemma 5.12).

Lemma 5.3 (Completeness)**.** *For every $\lambda \in \mathbb{N}$, $\mathsf{pp} \in \mathrm{Supp}\big(\mathsf{uVDF.Gen}(1^\lambda)\big)$, $x_0 \in \{0,1\}^*$, and $t \in \mathbb{N}$,*

$$\mathsf{uVDF.Verify}(1^\lambda, \mathsf{pp}, (x_0, t), \mathsf{uVDF.Eval}(1^\lambda, \mathsf{pp}, (x_0, t)) = 1.$$

Proof. Let $\mathsf{pp} = (N, B, k, d, \mathsf{hash})$ and $(y, \pi) = \mathsf{uVDF.Eval}(1^\lambda, \mathsf{pp}, (x_0, t))$. First, if x_0 is invalid or $t > B$, then $y = \pi = \perp$ and $\mathsf{uVDF.Verify}(1^\lambda, \mathsf{pp}, (x_0, t), (y, \pi))$ outputs 1 as expected. Therefore, suppose x_0 is valid and $t \leq B$. Then, since $y = \left|(x_0)^{2^t}\right|_N$, it holds that y is valid by Fact A.3. Additionally, $y^2 = (x_0)^{2^{t+1}} \bmod N$, so $(x_0, y, t) \in \mathcal{L}_{N,B}$. By Lemma 4.3, $\mathsf{FS\text{-}Verify}$ succeed since π is computed using the honest prover's strategy and the FS transformation preserves perfect completeness. □

Lemma 5.4 (Soundness)**.** *Assuming the (D, B, ϵ)-RSW assumption, then for every non-uniform algorithm $\mathcal{A} = \{\mathcal{A}_\lambda\}_{\lambda \in \mathbb{N}}$ where $\mathcal{A}_\lambda$ runs in time $\mathrm{poly}(B(\lambda))$ for every $\lambda \in \mathbb{N}$, there exists a negligible function* negl *such that for every $\lambda \in \mathbb{N}$ it holds that*

$$\Pr\left[\begin{array}{l} \mathsf{pp} \leftarrow \mathsf{uVDF.Gen}(1^\lambda) \\ (x_0, t), (\hat{y}, \hat{\pi}) \leftarrow \mathcal{A}_\lambda(\mathsf{pp}) \end{array} : \begin{array}{l} \mathsf{uVDF.Verify}(1^\lambda, \mathsf{pp}, (x_0, t), (\hat{y}, \hat{\pi})) = 1 \\ \wedge\ (\hat{y}, \hat{\pi}) \neq \mathsf{uVDF.Eval}(1^\lambda, \mathsf{pp}, (x_0, t)) \end{array}\right] \leq \mathsf{negl}(\lambda).$$

Proof. Let $\mathsf{pp} = (N, B, k, d, \mathsf{hash})$ and $(y, \pi) = \mathsf{uVDF.Eval}(1^\lambda, \mathsf{pp}, (x_0, t))$. Note that if x_0 is an invalid element or $t > B$, then $(\hat{y}, \hat{\pi}) = (\perp, \perp) = (y, \pi)$ since this is the only accepting value of $(\hat{y}, \hat{\pi})$. Therefore, x_0 is valid and $t \leq B$. In this case, verification only succeeds when $\hat{y}$ is valid, and thus $\hat{y} \neq \perp$.

Next, we define four events Event_1, Event_2, Event_3, Event_4 as follows. Event_1 is the event that verification succeeds when $y^2 \neq \hat{y}^2 \bmod N$, and Event_2 is the analogous event for $y \neq \hat{y}$ given $y^2 = \hat{y}^2 \bmod N$. We recall that the proof π is either equal to $\perp$ when $t \leq k^d$ or $((x_1, \ldots, x_{k-1}), \pi')$ when $t > k^d$ where each segment $x_i \in \mathbb{Z}_N^\star$ and π' is a proof for t/k. For any proof π, we define π^2 where we square each element of $\mathbb{Z}_N^\star$ from π (and $\pi^2 = \perp$ if $\pi = \perp$). Given this notation, we define Event_3 as the event that verification succeeds for $\pi^2 \neq \hat{\pi}^2$ given $y = \hat{y}$, and Event_4 for $\pi \neq \hat{\pi}$ given $y = \hat{y}$ and $\pi^2 = \hat{\pi}^2$. In the following claims, we show that, for each $i \in [4]$, there exists a negligible function negl_i such that for all $\lambda \in \mathbb{N}$, $\Pr[\mathsf{Event}_i] \leq \mathsf{negl}_i(\lambda)$.

Claim 5.5. *Assuming $(\log_\lambda(t) - d)$-round FS,* $\Pr[\mathsf{Event}_1] \leq \mathsf{negl}_1(\lambda)$.

Proof. Suppose there exists a non-uniform PPT algorithm $\mathcal{A}_\lambda$ such that $\mathcal{A}_\lambda(\mathsf{pp})$ outputs $\hat{y}$ where $\hat{y}^2 \neq y^2$ and $\mathsf{uVDF.Verify}(1^\lambda, \mathsf{pp}, (x_0, t), (\hat{y}, \hat{\pi})) = 1$. This means that $\mathsf{FS\text{-}Verify}(\mathsf{pp}, (x_0, t), \hat{y}, \hat{\pi}) = 1$ for $\hat{y}^2 \neq y^2 = (x_0)^{2^{t+1}} \bmod N$, which can occur with at most negligible probability $\mathsf{negl}_1(\lambda)$ by the soundness of the non-interactive argument $(P_{\mathsf{FS}}, V_{\mathsf{FS}})$ assuming $(\log_\lambda(t) - d)$-round FS.

Claim 5.6. *Assuming the (D, B, ϵ)-RSW assumption,* $\Pr[\mathsf{Event}_2] \leq \mathsf{negl}_2(\lambda)$.

Proof. Suppose there exists a non-uniform PPT algorithm $\mathcal{A}$ such that $\mathcal{A}_\lambda(\mathsf{pp})$ outputs $\hat{y} \neq y$ where $\hat{y}^2 = y^2$ and $\mathsf{uVDF.Verify}(1^\lambda, \mathsf{pp}, (x_0, t), (\hat{y}, \hat{\pi})) = 1$. Since $y = \left|x_0^{2^t}\right|_N$, then $y = |y|_N$. Furthermore, since $\mathsf{uVDF.Verify}(1^\lambda, \mathsf{pp}, (x_0, t), (\hat{y}, \hat{\pi}))$ accepts, then $\hat{y} = |\hat{y}|_N$, and hence $\hat{y} \neq -y$. Therefore by Fact A.2 we can efficiently factor N given $(y^2, y, \hat{y})$, which can occur with at most negligible probability given the $(D, B(\lambda), \epsilon)$-RSW assumption.

Claim 5.7. *Assuming* $(\log_\lambda(t) - d)$*-round strong FS,* $\Pr[\mathsf{Event}_3] \leq \mathsf{negl}_3(\lambda)$.

Proof. Suppose there exists a non-uniform PPT algorithm $\mathcal{A}$ such that $\mathcal{A}_\lambda(\mathsf{pp})$ outputs $\hat{\pi}$ such that $\hat{\pi}^2 \neq \pi^2$ where $\hat{y} = y$ and $\mathsf{uVDF.Verify}(1^\lambda, \mathsf{pp}, (x_0, t), (\hat{y}, \hat{\pi}))$ accepts. First note that if $t \leq k^d$, $\pi = \perp$ is the only accepting proof, so $\hat{\pi}^2$ and π^2 must both be $\perp$. As a result, we consider $t > k^d$ where π and $\hat{\pi}$ are non-trivial. Since $\hat{y} = y = \left|(x_0)^{2^t}\right|_N$, the adversary's output $(x_0, \hat{y})$ is in the language $\mathcal{L}_{N,B}$ defined for the interactive proof of Figure 2. By Lemma 4.7, any value for $\hat{\pi}^2 \neq \pi^2$ will be accepted with negligible probability assuming $(\log_\lambda(t) - d)$-round strong FS.

Claim 5.8. *Assuming the (D, B, ϵ)-RSW assumption,* $\Pr[\mathsf{Event}_4] \leq \mathsf{negl}_4(\lambda)$.

Proof. Suppose there exists a non-uniform PPT algorithm $\mathcal{A}$ such that $\mathcal{A}_\lambda(\mathsf{pp})$ outputs $\hat{\pi}$ such that $\hat{\pi} \neq \pi$ where $\hat{\pi}^2 = \pi^2$, $\hat{y} = y$, and $\mathsf{uVDF.Verify}(1^\lambda, \mathsf{pp}, (x_0, t), (\hat{y}, \hat{\pi})) = 1$. Again we note that if $t \leq k^d$, $\pi = \perp$ is the only accepting proof, so $\hat{\pi}$ must also be $\perp$. Otherwise, suppose $t > k^d$. Consider the first element $x, \hat{x} \in \mathbb{Z}_N^\star$ that differ in $\pi, \hat{\pi}$, respectively. Since verification accepts and x_0 is valid, this implies that x and $\hat{x}$ must both be valid, unless both are $\perp$, which cannot occur by assumption. This implies that $\hat{x} \neq -x$, and by assumption we know that $x^2 = \hat{x}^2$. Then by Fact A.2, we can efficiently factor N given $(x^2, x, \hat{x})$, which can occur with at most negligible probability given the (D, B, ϵ)-RSW assumption.

The lemma then follows by taking a union bound over the four events, which yields the negligible function $\mathsf{negl} = \sum_{i=1}^{4} \mathsf{negl}_i$ as required. ☐

Lemma 5.9 (Honest Evaluation)**.** *Suppose that the (D, B)-RSW assumption holds for polynomial $\ell \colon \mathbb{N} \to \mathbb{N}$ and constant $\epsilon \in (0, 1)$. Then, for any constant $\delta > 0$, sufficiently large $\lambda \in \mathbb{N}$, and $t \geq D(\lambda)$, it holds that* $\mathsf{uVDF.Eval}(1^\lambda, \cdot, (\cdot, t))$ *can be computed in time* $(1 + \delta) \cdot t \cdot \ell(\lambda)$.

Proof. Fix any $\lambda, t \in \mathbb{N}$, $\mathsf{pp} \in \mathrm{Supp}\left(\mathsf{uVDF.Gen}(1^\lambda)\right)$ and x_0 in the support of $\mathsf{uVDF.Sample}(1^\lambda, \mathsf{pp})$. To analyze the time to compute $\mathsf{uVDF.Eval}(1^\lambda, \mathsf{pp}, (x_0, t))$, we first analyze the running time of Sketch and $\mathsf{FS\text{-}Prove}$ in Claims 5.10 and 5.11, respectively.

Claim 5.10. *There exists a constant c_1 such that the time to compute* $\mathsf{Sketch}(\mathsf{pp}, \cdot, \cdot, \cdot)$ *for* $\mathsf{pp} \in \mathrm{Supp}\left(\mathsf{Gen}(1^\lambda)\right)$ *is bounded by* λ^{c_1}.

Proof. Recall that Sketch first hashes its input to obtain the challenge values r_i, which takes fixed polynomial time in λ and k. Then, it raises elements $x_{i-1}, x_i \in \mathbb{Z}_N^\star$ to the power r_i for $i \in [k]$, and then takes two products over k of the resulting values. Since the values r_i are in $[2^\lambda]$, each exponentiation takes time at most $O(\lambda^3)$, and the k-products can be done in time $O(k\lambda^2)$. Putting everything together and recalling that $k = \lambda$ as defined by $\mathsf{uVDF.Gen}$, we have that Sketch for security parameter λ runs in time λ^{c_1} for some constant c_1 which depends on the time of hash.

Claim 5.11. *There exists a constant c such that $\mathsf{PT}(t) \leq \lambda^c + 2t \cdot \ell(\lambda)$, where $\mathsf{PT}(t)$ is the time to compute $\mathsf{FS\text{-}Prove}(\cdot, (\cdot, t))$.*

Proof. If $t \leq \lambda^d$, then $\mathsf{PT}(t) \in O(\lambda^2)$ since it is dominated by checking validity of x_0, y. Otherwise, $\mathsf{FS\text{-}Prove}$ computes the x_i values for $i \in [k]$ by performing t sequential squares in $\mathbb{Z}_N^\star$, which takes $t \cdot \ell$ time. Next it computes Sketch, which takes time λ^{c_1} by Claim 5.10. Lastly, $\mathsf{FS\text{-}Prove}$ takes $\mathsf{PT}(t/k)$ time to compute the recursive evaluation of $\mathsf{FS\text{-}Prove}$ on input t/k. Let c' be the constant such computing Sketch and checking validity of x_0 takes time at most $\lambda^{c'}$. Then, we have that $\mathsf{PT}(t) \leq \mathsf{PT}(t/k) + t \cdot \ell(\lambda) + \lambda^{c'}$ if $t > k^d$ and $\mathsf{PT}(t) \leq \lambda^{c'}$ if $t \leq k^d$. Solving for the recurrence, we get

$$\mathsf{PT}(t) \leq (\log_k(t) - d) \cdot \lambda^{c'} + t\ell(\lambda)\left(1 + \frac{1}{k} + \cdots + \frac{1}{k^{\log_k(t)-d}}\right).$$

Recalling that $t \leq B(\lambda) \in 2^{O(\lambda)}$ gives the desired bound for some constant c.

Putting this together, we have that the time to compute $\mathsf{uVDF.Eval}$ when $t \leq k^d$ is $O(\lambda^2)$ to check validity and $t \cdot \ell(\lambda)$ to compute y, and is therefore in $(1+o(1)) \cdot t \cdot \ell(\lambda)$ since $t \geq D(\lambda) \in \omega(\lambda^2)$.

When $t > k^d$, we take time $t \cdot \ell(\lambda)$ to compute msg and y, λ^{c_1} to compute the sketch, and $\mathsf{PT}(t/k)$ to compute π'.[15] In total, this takes time $2\lambda^c + t \cdot \ell(\lambda) + \mathsf{PT}(t/k) \leq 3\lambda^c + t \cdot \ell(\lambda) + 2(t/k) \cdot \ell(\lambda)$. Since $t > k^d$, we can choose the base case depth $d \geq c + 1$ so that $2/k + (3\lambda^c)/(t \cdot \ell(\lambda)) \in o(1)$. It follows that the time to compute $\mathsf{uVDF.Eval}$ is again in $(1 + o(1)) \cdot t \cdot \ell(\lambda)$.

In either case ($t \leq k^d$ or $t > k^d$), it holds that for any constant $\delta > 0$ and sufficiently large $\lambda \in \mathbb{N}$, $\mathsf{uVDF.Eval}$ runs in time $(1 + \delta) \cdot t \cdot \ell(\lambda)$.

□

Lemma 5.12 (Sequentiality)**.** *Suppose that the (D, B)-RSW assumption holds for polynomial $\ell \colon \mathbb{N} \to \mathbb{N}$ and constant $\epsilon \in (0, 1)$. Then, for any constants $\delta > 0$ and $\epsilon' > \frac{\epsilon+\delta}{1+\delta}$, uVDF satisfies $(D, B, (1 + \delta) \cdot \ell, \epsilon')$-sequentiality.*

Proof. Let $\delta > 0$ be the constant from honest evaluation and let ϵ' be any constant greater than $\frac{\epsilon+\delta}{1+\delta}$. By way of contradiction, suppose there exists a non-uniform algorithm $\mathcal{A}_0 = \{\mathcal{A}_{0,\lambda}\}_{\lambda \in \mathbb{N}}$ satisfying $\mathsf{size}(\mathcal{A}_{0,\lambda}) \in \mathrm{poly}(B(\lambda))$ for all $\lambda \in \mathbb{N}$ and a polynomial p such that for infinitely many $\lambda \in \mathbb{N}$,

$$\Pr\left[\begin{array}{l} \mathsf{pp} \leftarrow \mathsf{uVDF.Gen}(1^\lambda) \\ \mathcal{A}_1 \leftarrow \mathcal{A}_{0,\lambda}(\mathsf{pp}) \\ x \leftarrow \mathsf{uVDF.Sample}(1^\lambda, \mathsf{pp}) \\ (t, y) \leftarrow \mathcal{A}_1(x) \end{array} : \begin{array}{l} \mathsf{uVDF.Eval}(1^\lambda, \mathsf{pp}, (x, t)) = y \\ \wedge\ \mathsf{depth}(\mathcal{A}_1) \\ \quad \leq (1 - \epsilon') \cdot t \cdot (1 + \delta) \cdot \ell(\lambda) \\ \wedge\ t \geq D(\lambda) \end{array}\right] \geq \frac{1}{p(\lambda)}.$$

Using $\mathcal{A}_0$, we can create a non-uniform algorithm $\mathcal{B}_0 = \{\mathcal{B}_{0,\lambda}\}_{\lambda \in \mathbb{N}}$ that contradicts the assumption that RSW is a (D, B, ℓ, ϵ)-iteratively sequential function.

For every $\lambda \in \mathbb{N}$, the algorithm $\mathcal{B}_{0,\lambda}$ on input pp runs $\mathcal{A}_{0,\lambda}(\mathsf{pp})$ to obtain $\mathcal{A}_1$ and outputs $\mathcal{B}_1$ defined as follows. $\mathcal{B}_1(x)$ computes $(t, y) \leftarrow \mathcal{A}_1(x)$ and outputs (t, y') where y' equals y or $(N - y)$ with equal probability.

For correctness, we observe that $\mathsf{size}(\mathcal{B}_{0,\lambda}) \in \mathrm{poly}(\mathsf{size}(\mathcal{A}_{0,\lambda}))$, so $\mathsf{size}(\mathcal{B}_{0,\lambda}) \in \mathrm{poly}(B(\lambda))$. Furthermore, whenever $\mathcal{A}_{0,\lambda}$ succeeds, then $y' = (x_0)^{2^t} \bmod N$ with probability $1/2$, so $\mathcal{B}_1$ outputs the

[15] We note that computing msg before running $\mathsf{FS\text{-}Prove}$ takes space $O(k\lambda)$. We can improve this time/space tradeoff smoothly as done in [Pie19] and compute the proof in approximately $\sqrt{t}$ time and space.

correct value with probability at least $1/(2p(\lambda))$. To analyze the depth of $\mathcal{B}_1$, note that computing $N - y$ can be done in depth $O(\lambda)$, so there exists a constant c such that

$$\mathsf{depth}(\mathcal{B}_1) \leq \mathsf{depth}(\mathcal{A}_1) + c\lambda \leq (1-\epsilon') \cdot (1+\delta) \cdot t \cdot \ell(\lambda) + c\lambda.$$

Since $t \in \omega(\lambda)$, $\epsilon' > \frac{\epsilon+\delta}{1+\delta} + \frac{c\lambda}{(1+\delta)\cdot t \cdot \ell}$ for sufficeintly large $\lambda \in \mathbb{N}$, and $\mathsf{depth}(\mathcal{B}_1) \leq (1-\epsilon) \cdot t \cdot \ell(\lambda)$. Thus, it holds that for infinitely many $\lambda \in \mathbb{N}$,

$$\Pr\left[\begin{array}{l} \mathsf{pp} \leftarrow \mathsf{RSW.Gen}(1^\lambda) \\ (\mathcal{B}_1) \leftarrow \mathcal{B}_{0,\lambda}(\mathsf{pp}) \\ x \leftarrow \mathsf{RSW.Sample}(1^\lambda, \mathsf{pp}) \\ (t, y') \leftarrow \mathcal{B}_1(x) \end{array} : \begin{array}{l} \mathsf{RSW.Eval}(1^\lambda, \mathsf{pp}, (x,t)) = y \\ \wedge\ \mathsf{depth}(\mathcal{B}_1) \leq (1-\epsilon) \cdot t \cdot \ell(\lambda) \\ \wedge\ t \geq D(\lambda) \end{array}\right] \geq 1/(2p(\lambda)),$$

contradicting that RSW is a (D, B, ℓ, ϵ)-iteratively sequential function.

□

6 Continuous Verifiable Delay Function

In this section, we construct a cVDF. Intuitively, this is an iteratively sequential function where every intermediate state is verifiable. Throughout this section, we denote by $\mathsf{Eval}^{(\cdot)}$ the composed function which takes as input 1^λ, pp, and (x, T), and runs the T-wise composition of $\mathsf{Eval}(1^\lambda, \mathsf{pp}, \cdot)$ on input x.

We first give the formal definition of a cVDF. In the following definition, the completeness requirement says that if v_0 is an honestly generated starting state, then the Verify will accept the state given by $\mathsf{Eval}^{(T)}(1^\lambda, \mathsf{pp}, v_0)$ for any T. Note that when coupled with soundness, this implies that completeness holds with high probability for any intermediate state generated by a computationally bounded adversary.

Definition 6.1 (Continuous Verifiable Delay Function). *Let $B, \ell \colon \mathbb{N} \to \mathbb{N}$ and $\epsilon \in (0,1)$. A (B, ℓ, ϵ)*-continuous verifiable delay function (cVDF) *is a tuple* $(\mathsf{Gen}, \mathsf{Sample}, \mathsf{Eval}, \mathsf{Verify})$ *such that* $(\mathsf{Gen}, \mathsf{Sample}, \mathsf{Eval})$ *is a $(1, B, \ell, \epsilon)$-iteratively sequential function,* $(\mathsf{Gen}, \mathsf{Eval}^{(\cdot)}, \mathsf{Verify})$ *is a B-sound function, and it satisfies the following completeness property:*

- **Completeness from Honest Start.** *For every $\lambda \in \mathbb{N}$, pp in the support of $\mathsf{Gen}(1^\lambda)$, v_0 in the support of $\mathsf{Sample}(1^\lambda, \mathsf{pp})$, and $T \in \mathbb{N}$, it holds that* $\mathsf{Verify}(1^\lambda, \mathsf{pp}, (v_0, T), \mathsf{Eval}^{(T)}(1^\lambda, \mathsf{pp}, v_0)) = 1$.

The main result of this section is stated next.

Theorem 6.2 (Continuous VDF). *Let $D, B, \alpha \colon \mathbb{N} \to \mathbb{N}$ be functions satisfying $B(\lambda) \leq 2^{\lambda^{1/3}}$, $\alpha(\lambda) = \lceil \log_\lambda(B(\lambda)) \rceil$, and $D(\lambda) \geq \lambda^{d'}$ for all $\lambda \in \mathbb{N}$ and for a specific constant d'. Suppose that the α-round strong FS assumption holds and the (D, B)-RSW assumption holds for a polynomial $\ell \colon \mathbb{N} \to \mathbb{N}$ and constant $\epsilon \in (0,1)$. Then, for any constant $\delta > 0$ and $\epsilon' > \frac{\epsilon+\delta}{1+\delta}$, it holds that* cVDF *is a $(B, (1+\delta) \cdot D(\lambda) \cdot \ell, \epsilon')$-cVDF.*

In the case where we want to have a fixed polynomial bound on the number of iterations, we obtain the following corollary.

Corollary 6.3 (Restatement of Theorem 1.1). *For any polynomials B, D where $D(\lambda) \geq \lambda^{d'}$ for a specific constant d', suppose the $O(1)$-round strong FS assumption holds and the (D, B)-RSW assumption holds for a polynomial $\ell \colon \mathbb{N} \to \mathbb{N}$ and constant $\epsilon \in (0,1)$. Then, for any constant $\delta > 0$ and $\epsilon' > \frac{\epsilon+\delta}{1+\delta}$, it holds that* cVDF *is a $(B, (1+\delta) \cdot D(\lambda) \cdot \ell, \epsilon')$-cVDF.*

Remark 2 (Decoupling size and depth)**.** *The definition of a (B, ℓ, ϵ)-cVDF naturally extends to a (U, B, ℓ, ϵ)-cVDF, where we require* $(\mathsf{Gen}, \mathsf{Sample}, \mathsf{Eval})$ *to be a $(1, U, B, \ell, \epsilon)$-iteratively sequential function; see Remark 1. Our construction will satisfy this for all functions U such that $U(\lambda) \leq B(\lambda)$ for all $\lambda \in \mathbb{N}$which will be used in Section 7.Moreover, in this case, the above corollary can be based on the strong Fiat-Shamir assumption for $\lceil \log_\lambda(U(\lambda)) \rceil$ rounds (rather than for $\lceil \log_\lambda(B(\lambda)) \rceil$ rounds).*

We prove Theorem 6.2 by using the unique VDF uVDF from Section 5 as a building block. We start with some definitions which will be helpful in the construction.

Definition 6.4 (Puzzle tree)**.** *A* $(\mathsf{pp}_{\mathsf{uVDF}}, d', g)$-puzzle tree *for* $\mathsf{pp}_{\mathsf{uVDF}} = (N, B, k, d, \mathsf{hash})$ *is a $(k+1)$-ary tree that has the following syntax.*

- *Each node is labeled by a string $s \in \{0, 1, \ldots, k\}^*$, where the root is labeled with the empty string* null*, and for a node labeled s, its ith child is labeled $s||i$ for $i \in \{0, 1, \ldots, k\}$. We let $[s]_i$ denote the ith character of s for $i \in \mathbb{N}$.*[16]
- *We define the height of the tree as $h = \lceil \log_k(B) \rceil - d'$ which determines difficulty at each node. Specifically, each node s is associated with the difficulty $t = k^{h+d'-|s|}$.*[17]
- *Each node s has a value $\mathsf{val}(s) = (x, y, \pi)$, where we call x the input, y the output, and π the proof.*

The inputs, outputs, and proofs of each node are defined as follows:

- *The root has input g. In general, for a node s with input x and difficulty t, its first k children are called* segment *nodes and its last child is called a* sketch *node. Each segment node $s||i$ has input $x_i = \left|x^{2^{i \cdot t/k}}\right|_N$ and the sketch node $s||k$ has input x' where $(x', *) =$* $\mathsf{Sketch}(\mathsf{pp}_{\mathsf{uVDF}}, (x, t), x_k, (x_1, \ldots, x_{k-1}))$ *(given in Figure 3).*
- *For a node s with input x and difficulty t, its output and proof are given by $(y, \pi) =$* $\mathsf{uVDF}.\mathsf{Eval}(\mathsf{pp}_{\mathsf{uVDF}}, (x, t))$.

We note that whenever we refer to a node s, we mean the node labeled by s, and when we refer to a pair (s', value), this corresponds to a node and associated value (where value may not necessarily be equal to the true value $\mathsf{val}(s)$).

Definition 6.5 (Left/Middle/Right Nodes)**.** *For a node with label s in a $(\mathsf{pp}_{\mathsf{uVDF}}, d', g)$-puzzle tree with $s = s'||i$ for $i \in \{0, 1, \ldots, k\}$, we call s a* leftmost *child if $i = 0$, a* rightmost *child if $i = k$, and a* middle *child otherwise. Additionally, we define the* left *(resp.* right*) siblings of s to be the set of nodes $s'||j$ for $0 \leq j < i$ (resp. $i < j \leq k$).*

Next, we define a frontier. At a high level, for a leaf s, the frontier of s will correspond to the state of the continuous VDF upon reaching s. Specifically, it will contain all nodes whose values have been computed at that point, but whose parents' values have not yet been computed.

Definition 6.6 (Frontier)**.** *For a node s in a $(\mathsf{pp}_{\mathsf{uVDF}}, d', g)$-puzzle tree, the* frontier *of s, denoted* $\mathsf{frontier}(s)$*, is the set of pairs $(s', \mathsf{val}(s'))$ for nodes s' that are left siblings of any of the ancestors of s. We note that s is included as one of its ancestors.*[18]

[16]For ease of notation, we store s as a $(k+1)$-ary string and when doing integer operations, they are implicitly done in base $(k+1)$.

[17]Note that since the tree has height h, this implies that each leaf has difficulty $t = k^{d'}$.

[18]It may be helpful to observe that for a leaf node $s = [s]_1||[s]_2|| \cdots ||[s]_h$, the frontier contains $[s]_i$ nodes at level i for $i \in [h]$.

Next, we define what it means for a set to be consistent. At a high level, for a set of nodes and values, consistency ensures that the relationship of their given inputs and outputs across different nodes is in accordance with the definition of a puzzle tree. If a set is consistent, it does not imply that the input-output pairs are correct, but it implies that they "fit" together logically. Note that consistency does not check proofs.

Definition 6.7 (Consistency)**.** *Let S be a set of pairs (s, value) for nodes s and values* value *in a $((N, B, k, d, \mathsf{hash}), d', g)$-puzzle tree. We say that $(s', (x, y))$ is* consistent *with S if the following hold:*

1. *The input x of s' is (a) the output given for its left sibling if its left sibling is in S and s' is a middle child, (b) given by the sketch of its left siblings' values if all of its left siblings are in S and s' is a rightmost child, or (c) defined recursively as its parent's input if s' is a leftmost child (where the base of the recursion is the root with input g).*

2. *The output y of s' is (a) given by the sketch of its left siblings' values if all of its left siblings are in S and s' is a rightmost child, or (b) given recursively by its parent's output if s' is a kth child (where, upon reaching the root recursively, we then accept any output for s').*

We say that S is a consistent set *if every node in S is consistent with S.*

6.1 Construction

Before giving the cVDF construction, we give a detailed overview. At a high level, the cVDF will iteratively compute each leaf node in a $(\mathsf{pp}_{\mathsf{uVDF}}, d', g)$-puzzle tree, where $\mathsf{pp}_{\mathsf{uVDF}} = (N, B, k, d, \mathsf{hash})$ are the public parameters of the underlying uVDF and g is the starting point of the tree given by $\mathsf{uVDF.Sample}$.

The heart of our construction is the $\mathsf{cVDF.Eval}$ functionality which takes a state v corresponding to a leaf s in the tree and computes the next state v' corresponding to the next leaf. Each state v will be of the form (g, s, F), where s is the current leaf in the tree and F is the frontier of s. Then, $\mathsf{cVDF.Eval}(1^\lambda, \mathsf{pp}, (g, s, \mathsf{frontier}(s))$ will output $(g, s+1, \mathsf{frontier}(s+1))$. There are three phases of the algorithm $\mathsf{cVDF.Eval}$. First, it checks that its input is well-formed. It then computes $\mathsf{val}(s)$ using $\mathsf{frontier}(s)$, and then computes $\mathsf{frontier}(s+1)$ using both $\mathsf{frontier}(s)$ and $\mathsf{val}(s)$. These are discussed next.

Checking that v is well-formed. Recall that $v = (g, s, F)$ corresponds to the node s in the tree. This state v is correct if running $\mathsf{cVDF.Eval}$ for s steps (where s is interpreted as an integer in base $(k+1)$) starting at the leaf 0^h results in $(g, s, \mathsf{frontier}(s))$. Therefore, before computing the next state, $\mathsf{cVDF.Eval}$ needs to verify that the state it was given is correct. To do this, we run $\mathsf{cVDF.Verify}$ with input state $(g, 0^h, \perp)$ and output state (g, s, F), and check that this is s steps of computation.

Computing the value of s. To compute $\mathsf{val}(s)$, we have the following observation: for every node, its input is a function of the input of its parent and the outputs of its left siblings. Indeed, if s is a middle child, its input is the output of the sibling to its left (given in F). If s is a rightmost child, its input is the sketch of the values of its left siblings (also given in F). If s is a leftmost child, its input is input of its parent, defined recursively. Therefore, we compute its input based on F in this manner. Then, we compute its output by running $\mathsf{uVDF.Eval}$ on its input.

Computing the frontier of $s+1$. The final phase of $\mathsf{cVDF.Eval}$ is to compute the next frontier using $\mathsf{val}(s)$ and $\mathsf{frontier}(s)$. To do this, we consider the closest common ancestor a of s and $s+1$ and note that by definition, $\mathsf{frontier}(a) \subset \mathsf{frontier}(s+1)$. Moreover, its straightforward to see that

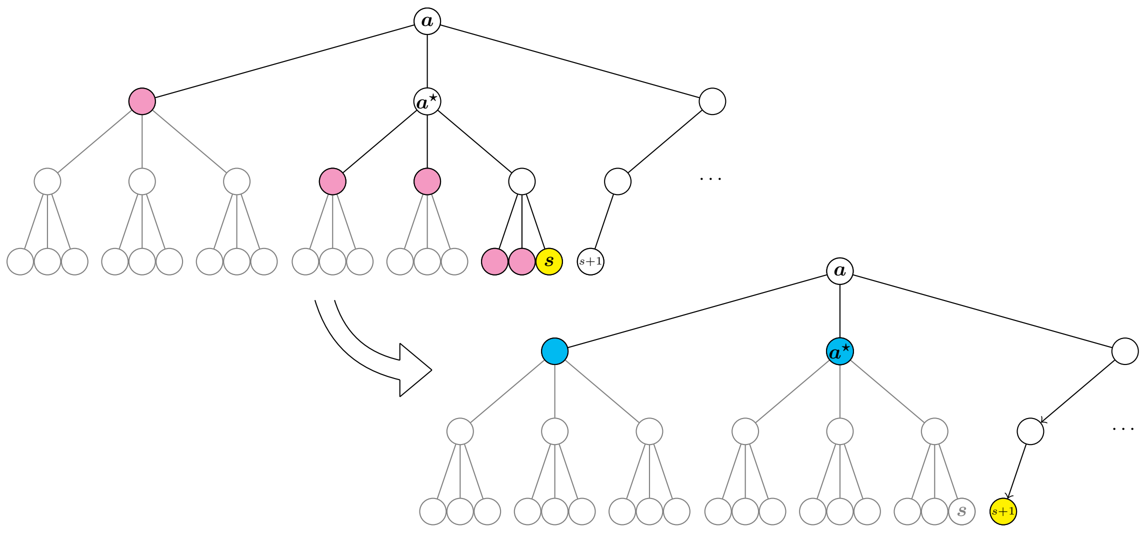

Figure 4: An example of computing $\mathsf{frontier}(s+1)$ from $\mathsf{frontier}(s)$ for $k=2$ with nodes s, $s+1$, $a^\star$, and a given. In both graphs, the yellow node is the current node at that point in the computation, and the nodes in gray are those whose proofs have already been merged to proofs at their parents. In the left graph, the frontier of s is shown in pink. The right graph is the result of merging values to obtain the frontier of s', which is shown in blue.

$\mathsf{frontier}(s+1) \setminus \mathsf{frontier}(a)$ only contains a node $a^\star$ and its left siblings, where $a^\star$ is the child of a along the path to s. Note that when s and $(s+1)$ are siblings, then $a^\star = s$, and otherwise, it can be shown that $a^\star$ is the closest ancestor of s that is not a rightmost child.

Therefore, to compute $\mathsf{frontier}(s+1)$, we start by computing the value of node $a^\star$. If $a^\star = s$, then we have already computed it, and otherwise it's input and output are known from its children's values in F. Specifically, its input is the input of its first child, and its output is the output of its kth child. These are in F because of the definition of $a^\star$, which implies that each of its descendants along the path to s must be rightmost children. To compute its proof, observe that the values of s and its siblings are all known, so they can be efficiently merged into a proof of its parent. If the parent is $a^\star$, then we are done. If not, we can similarly merge values into a proof of the grandparent of s. We can continue this process until we reach $a^\star$. We show how to do this by traversing the path from s up to $a^\star$ and by iteratively "merging" values up the tree. An example depicting $s, s+1, a, a^\star$ is given in Section 6.1.

Formal construction. Next, we give the formal details of our construction $\mathsf{cVDF} = (\mathsf{cVDF.Gen}, \mathsf{cVDF.Sample}, \mathsf{cVDF.Eval}, \mathsf{cVDF.Verify})$.

- $\mathsf{pp} \leftarrow \mathsf{cVDF.Gen}(1^\lambda)$:

  Sample $\mathsf{pp}_{\mathsf{uVDF}} \leftarrow \mathsf{uVDF.Gen}(1^\lambda)$ where $\mathsf{pp}_{\mathsf{uVDF}} = (N, B, k, d, \mathsf{hash})$. Let d' be the constant specified in Lemma 6.22, and set tree height $h = \lceil \log_k(B) \rceil - d'$. Output $\mathsf{pp} = (\mathsf{pp}_{\mathsf{uVDF}}, d', h)$.

- $v \leftarrow \mathsf{cVDF.Sample}(1^\lambda, \mathsf{pp})$:

  Sample $g \leftarrow \mathsf{uVDF.Sample}(1^\lambda, \mathsf{pp}_{\mathsf{uVDF}})$ and output $v = (g, 0^h, \emptyset)$.

- $v' \leftarrow \mathsf{cVDF.Eval}(1^\lambda, \mathsf{pp}, v)$:

  Check that v is well-formed:

1. Parse v as (g, s, F), where s is a leaf label in a $(\mathsf{pp}_{\mathsf{uVDF}}, g)$-puzzle tree and F is a frontier. Output $\bot$ if v cannot be parsed this way.
2. Run $\mathsf{cVDF.Verify}(1^\lambda, \mathsf{pp}, ((g, 0^h, \emptyset), s), (g, s, F))$ to verify v. Output $\bot$ if it rejects.

Compute the value of s:

1. Compute the input x of node s as the output of the sibling to its left (given in F) if s is a middle child, a sketch of its left siblings' values (given in F) if s is a rightmost child, or recursively as its parent's input if s is a leftmost child.
2. Compute its output and proof as $(y, \pi) = \mathsf{uVDF.Eval}(1^\lambda, \mathsf{pp}_{\mathsf{uVDF}}, (x, k^{d'}))$.

Compute the frontier of $s + 1$:

1. Let a be the closest common ancestor of s and $s+1$, and let $a^\star$ be the ancestor of s that is a child a.
2. If $a^\star = s$, compute its value as $(x^\star, y^\star, \pi^\star) = (x, y, \pi)$.
3. If $a^\star$ is a strict ancestor of s, let $x^\star$ be the input of its leftmost child in F, let $y^\star$ be the output of its kth child in F, and let $\pi^\star$ be $\bot$ if $x^\star$ is invalid and otherwise the outputs of its first $k-1$ children in F along with the proof, computed recursively, of its child along the path to s.
4. Form the next frontier F' by removing all descendants of $a^\star$ from F, and adding $(a^\star, (x^\star, y^\star, \pi^\star))$.

Finally, output $(g, s+1, F')$.

- $b \leftarrow \mathsf{cVDF.Verify}(1^\lambda, \mathsf{pp}, (v, T), v')$:

  Check that v is well-formed:

  Parse v as (g, s, F) where $g \in \mathbb{Z}_N^\star$, s is a leaf node, and F is a frontier. If v cannot be parsed this way, then output 1 if $v' = \bot$ and 0 otherwise.

  If $(g, s, F) \neq (g, 0^h, \emptyset)$, then verify the state v by recursively running this verification algorithm, i.e., $\mathsf{cVDF.Verify}(1^\lambda, \mathsf{pp}, ((g, 0^h, \emptyset), s), (g, s, F))$. If this rejects, then output 1 if $v' = \bot$ and 0 otherwise.

  Check that v' is correct:

  Output 1 if the following checks succeed, and 0 otherwise:

  1. Parse v' as $(g, s+T, F')$ where F' is a frontier.
  2. Check that the set of nodes in F' is the set of nodes in $\mathsf{frontier}(s')$ (considering only node labels and not values).
  3. Check that F' is a consistent set.[19]
  4. For each element $(s', (x, y, \pi)) \in F'$, check that $\mathsf{uVDF.Verify}(1^\lambda, \mathsf{pp}_{\mathsf{uVDF}}, (x, t), (y, \pi))$ accepts, where $t = k^{h+d'-|s'|}$.

[19]This can be done efficiently, since consistency of every element in F' can be checked by looking at k nodes in each of the h levels of the tree and performing at most one sketch.

6.2 Proofs

The main result of this section is stated next.

Theorem 6.8. *Let $D, B\colon \mathbb{N} \to \mathbb{N}$ where $B(\lambda) \le 2^{\lambda^{1/3}}$, $D(\lambda) = \lambda^{d'}$ for all $\lambda \in \mathbb{N}$ and specific constant d'. Assume that (1) the (D,B)-RSW assumption holds for an $\epsilon \in (0,1)$ and a polynomial ℓ, and (2) for any constants $\epsilon', \delta \in (0,1)$, uVDF (given in Section 5) is a $(D, B, (1+\delta)\cdot\ell, \epsilon')$-unique VDF. Then cVDF is a $(B, (1+\delta')\cdot D\cdot\ell, \epsilon'')$-cVDF for any $\epsilon'' > \frac{\epsilon+\delta'}{1+\delta'}$ and $\delta' > \delta$.*

As a corollary, by combining Theorem 5.2 with Theorem 6.8, we obtain Theorem 6.2: a continuous VDF under the Fiat-Shamir and the repeated squaring assumptions.

To show Theorem 6.8, we show that cVDF satisfies completeness from honest start in Lemma 6.9 and that $(\mathsf{cVDF.Gen}, \mathsf{cVDF.Eval}^{(\cdot)}, \mathsf{cVDF.Verify})$ satisfies B-soundness in Lemma 6.16. We then show that $(\mathsf{cVDF.Gen}, \mathsf{cVDF.Sample}, \mathsf{cVDF.Eval})$ is a (D, B, ℓ, ϵ'')-iteratively sequential function by showing length bounded in Lemma 6.18, honest evaluation in Lemma 6.19, and sequentiality in Lemma 6.22. Together, these complete the proof of Theorem 6.8.

Lemma 6.9 (Completeness from Honest Start)**.** *For all $\lambda \in \mathbb{N}$, pp in the support of $\mathsf{cVDF.Gen}(1^\lambda)$, v_0 in the support of $\mathsf{cVDF.Sample}(1^\lambda, \mathsf{pp})$, and $T \in \mathbb{N}$, it holds that*

$$\mathsf{cVDF.Verify}(1^\lambda, \mathsf{pp}, (v_0, T), \mathsf{cVDF.Eval}^{(T)}(1^\lambda, \mathsf{pp}, v_0)) = 1.$$

Proof. Fix any $\mathsf{pp} = (\mathsf{pp}_{\mathsf{uVDF}}, d', h)$ in the support of $\mathsf{cVDF.Gen}(1^\lambda)$ where $\mathsf{pp}_{\mathsf{uVDF}} = (N, B, k, d, \mathsf{hash})$, any $v_0 = (g, 0^h, \emptyset)$ in the support of $\mathsf{cVDF.Sample}(1^\lambda, \mathsf{pp})$, and $T \in \mathbb{N}$. The lemma follows from the following two claims:

Claim 6.10. *For any leaf s, $\mathsf{cVDF.Verify}(1^\lambda, \mathsf{pp}, ((g, 0^h, \emptyset), s), (g, s, \mathsf{frontier}(s)))$ accepts.*

Claim 6.11. *For any leaf s, $\mathsf{cVDF.Eval}(1^\lambda, \mathsf{pp}, (g, s, \mathsf{frontier}(s))) = (g, s+1, \mathsf{frontier}(s+1))$.*

We first show how to use these claims to complete the proof of the lemma. Since $v_0 = (g, 0^h, \mathsf{frontier}(0^h))$, it holds that T applications of Claim 6.11 imply that $\mathsf{cVDF.Eval}^{(T)}(1^\lambda, \mathsf{pp}, v_0) = (g, T, \mathsf{frontier}(T))$ and thus by Claim 6.10, $\mathsf{cVDF.Verify}(1^\lambda, \mathsf{pp}, (v_0, T), (g, T, \mathsf{frontier}(T)))$ will accept, which will complete the proof of the lemma.

It remains to prove Claims 6.10 and 6.11. Towards this goal, we define the following three invariants relative a tuple (g, s, F):

(a) The set of nodes in F is the set of nodes in $\mathsf{frontier}(s)$ (ignoring the values of the nodes).

(b) F is a consistent set.

(c) Each element $(s', (x, y, \pi))$ in F is correct, that is, $(y, \pi) = \mathsf{uVDF.Eval}(1^\lambda, \mathsf{pp}_{\mathsf{uVDF}}, (x, k^{h+d'-|s'|}))$.

We next prove Claim 6.10, i.e., $\mathsf{cVDF.Verify}(1^\lambda, \mathsf{pp}, ((g, 0^h, \emptyset), s), (g, s, \mathsf{frontier}(s)))$ accepts for any leaf s.

Proof of Claim 6.10. Recall that the algorithm $\mathsf{cVDF.Verify}(1^\lambda, \mathsf{pp}, ((g, 0^h, \emptyset), s), (g, s, \mathsf{frontier}(s)))$ first checks that $(g, 0^h, \emptyset)$ is well-formed, which passes by definition of $(g, 0^h, \emptyset)$. Then, $\mathsf{cVDF.Verify}$ checks that the first two invariants hold relative to $(g, s, \mathsf{frontier}(s))$ and that $\mathsf{uVDF.Verify}$ accepts on every element of $\mathsf{frontier}(s)$. We have that invariant (a) trivially holds. Invariant (b) follows because the values of nodes in a puzzle tree are consistent (Definitions 6.4 and 6.7) and every node s' in $\mathsf{frontier}(s)$ has its real value $\mathsf{val}(s')$ given. To show that $\mathsf{uVDF.Verify}$ accepts on every element of $\mathsf{frontier}(s)$, we note that invariant (c) also holds by definition of $\mathsf{val}(s)$, namely, $\mathsf{val}(s) = (x, y, \pi)$ implies that $(y, \pi) = \mathsf{uVDF.Eval}(1^\lambda, \mathsf{pp}_{\mathsf{uVDF}}, (x, t))$ where t is the difficulty of s. Therefore, $\mathsf{uVDF.Verify}$ accepts by perfect correctness of uVDF, concluding the proof of claim.

Now we prove Claim 6.11, i.e., $\mathsf{cVDF.Eval}(1^\lambda, \mathsf{pp}, (g, s, \mathsf{frontier}(s))) = (g, s+1, \mathsf{frontier}(s+1))$ for any leaf s.

Proof of Claim 6.11. Let $v = (g, s, \mathsf{frontier}(s))$ and let $v' = \mathsf{cVDF.Eval}(1^\lambda, \mathsf{pp}, v)$ (so our goal is to prove $v' = (g, s+1, \mathsf{frontier}(s+1))$). Recall that $\mathsf{cVDF.Eval}(1^\lambda, \mathsf{pp}, v)$ first checks that v is well-formed, meaning that v can be parsed appropriately and that $\mathsf{cVDF.Verify}(1^\lambda, \mathsf{pp}, ((g, 0^h, \emptyset), s), v)$ accepts. This holds by Claim 6.10, which also means the three invariants hold relative to $(g, s, \mathsf{frontier}(s))$. By definition of $\mathsf{cVDF.Eval}$, this implies that its output v' is of the form $(g, s+1, F)$ where F is a frontier. We first show that $(g, s+1, F)$ satisfies the above three invariants, and then we conclude the claim by showing that this implies that $F = \mathsf{frontier}(s+1)$.

Proposition 6.12. *The set of nodes in F is the set of nodes in* $\mathsf{frontier}(s+1)$.

Proof. Let a be closest common ancestor of s and $s+1$, and let $a^\star$ be the child of a that is an ancestor of s. The nodes in $\mathsf{frontier}(s+1)$ can be partitioned as follows. All nodes in $\mathsf{frontier}(a)$ are in both $\mathsf{frontier}(s)$ and $\mathsf{frontier}(s+1)$ since a is a common ancestor. At the level of the tree containing $a^\star$, $\mathsf{frontier}(s+1)$ contains $a^\star$ and its left siblings, since the sibling to the right of $a^\star$ must be an ancestor of $s+1$. Finally, below this level, there are no left siblings of the ancestors of $s+1$, since s and $s+1$ are adjacent leaves. Therefore, $\mathsf{frontier}(s+1) = \mathsf{frontier}(a^\star) \cup \{a^\star, \mathsf{val}(a^\star)\}$. Since $\mathsf{cVDF.Eval}$ forms F by removing the elements corresponding to descendants of $a^\star$ from $\mathsf{frontier}(s)$ and then adding $a^\star$, the claim follows.

Proposition 6.13. *F is a consistent set.*

Proof. We start with the following observations. For any node s' and input-output pair (x, y), (1) the checks required by consistency for s' only involve nodes that are lexicographically smaller than s', and (2) if $(s', (x, y))$ is consistent with some set S, then it is consistent with any $S' \subseteq S$.

Note that every element in F is either in the intersection $F \cap \mathsf{frontier}(s)$, or is $(a^\star, (x^\star, y^\star, \pi^\star))$ (where $a^\star$ is the ancestor of s that is a child of the closest common ancestor of s and $s+1$). This because $\mathsf{cVDF.Eval}$ does not make any changes to elements corresponding to nodes that are in $F \cap \mathsf{frontier}(s)$. We proceed to show that every element $(s', (x, y)) \in F$ is consistent with F.

First, let s' be a node with input-output pair (x, y) in $F \cap \mathsf{frontier}(s)$. By observation (2), since $(s', (x, y))$ is consistent with $\mathsf{frontier}(s)$, it is consistent with $F \cap \mathsf{frontier}(s)$. Additionally, $F \setminus \mathsf{frontier}(s)$ does not contain nodes lexicographically smaller than s', so by observation (1) it holds that $(s', (x, y))$ is consistent with F.

Next, consider $a^\star$ with input-output pair $(x^\star, y^\star)$ in F. Note that $a^\star$ is the closest ancestor of s that is not a rightmost child because the sibling to its right is an ancestor of $s+1$. We first consider the case where $a^\star$ is a strict ancestor of s, and then the case where $a^\star = s$. If $a^\star$ is a strict ancestor of s, then its input $x^\star$ is the input of its leftmost child in $\mathsf{frontier}(s)$ and its output $y^\star$ is the output of its kth child in $\mathsf{frontier}(s)$. Both of these nodes appear in $\mathsf{frontier}(s)$ by definition of $a^\star$. Therefore consistency of $a^\star$ with $\mathsf{frontier}(s)$ follows from consistency of its leftmost and kth children with $\mathsf{frontier}(s)$. Therefore $a^\star$ is consistent with $F \cap \mathsf{frontier}(s)$ by observation (2), and hence is consistent with F by observation (1).

If $a^\star = s$, then its input $x^\star$ is computed exactly to satisfy consistency. Its output $y^\star$ is given by $\mathsf{uVDF.Eval}(1^\lambda, \mathsf{pp}, (x^\star, k^{d'}))$. Observe that consistency only enforces a requirement on the output of s if each ancestor of s is a kth child up until some ancestor a' which is a rightmost child. By definition of $\mathsf{frontier}(s)$, all the left siblings of these ancestors are in $\mathsf{frontier}(s)$. Let x', y' be the sketch of the values of the left siblings of a'. We want to show that $y^\star = y'$.

Toward this goal, suppose first that x' and y' are valid. We have that $a'||0$ has input x' since $a'||0$ is consistent with F. This implies that the output of $a'||(k-2)$ corresponds to $t(k-1)/k$ squarings of x' for $t = k^{h+d'-|a'|}$ because $\mathsf{frontier}(s)$ is consistent and all elements of $\mathsf{frontier}(s)$ are correct. By continuing to apply this argument and noting that each level i out of the $h-|a'|$ levels we have done $(k-1)\cdot t/k^i$ squarings, it holds that after doing $k^{d'}$ squarings of $x^\star$, we obtain $y' = y^\star$ by definition of Sketch.

To handle the case where x' or y' are not valid, note that it must hold that x' is invalid and $y' = \bot$. This follows since the left siblings of a' are consistent with $\mathsf{frontier}(s)$ and are correct. In this case, $\bot$ will propagate down the tree so $y^\star = \bot = y'$, as desired.

Proposition 6.14. *For every element $(s', (x, y, \pi))$ in F, it holds that $(y, \pi) = \mathsf{uVDF.Eval}(1^\lambda, \mathsf{pp}_{\mathsf{uVDF}}, (x, k^{h+d'-|s'|}))$.*

Proof. Consider the elements of F'. Recall from Proposition 6.12 that every element of F is either in $\mathsf{frontier}(s)$ or is $\{(a^\star, (x^\star, y^\star, \pi^\star))\}$, where $(x^\star, y^\star, \pi^\star)$ is the value computed by $\mathsf{cVDF.Eval}(1^\lambda, \mathsf{pp}, v)$ for $a^\star$. Therefore, we need to show that $(y^\star, \pi^\star) = \mathsf{uVDF.Eval}(1^\lambda, \mathsf{pp}_{\mathsf{uVDF}}, (x^\star, k^{h+d'-|a^\star|}))$.

The proof $\pi^\star$ is computed by traversing the path from s to $a^\star$ and iteratively computing the proof of each node along the way. Let b be an ancestor of s along this path and let (x, y, π) be the value where:

1. If $b = s$, then x is the input computed for s by $\mathsf{cVDF.Eval}(1^\lambda, \mathsf{pp}, v)$ and (y, π) is computed as $\mathsf{uVDF.Eval}(1^\lambda, \mathsf{pp}_{\mathsf{uVDF}}, (x, k^{d'}))$.
2. If b is a strict ancestor of s, then x is the input of $b||0$ in $\mathsf{frontier}(s)$, y is the output of $b||(k-1)$ in F, and π is the proof computed for b by $\mathsf{cVDF.Eval}(1^\lambda, \mathsf{pp}, (g, s, \mathsf{frontier}(s)))$.

Note that when $b = a^\star$, then $(x^\star, y^\star, \pi^\star) = (x, y, \pi)$. We show by induction on the length of the path from s to b that (y, π) is equal to $\mathsf{uVDF.Eval}(1^\lambda, \mathsf{pp}_{\mathsf{uVDF}}, (x, t))$ where $t = k^{h+d'-|b|}$ is the difficulty of b. Note that whenever $b \neq s$, its rightmost child is an ancestor of s, so its first k children always appear in F.

For the base case when the path has length 0, then $b = s$ so this holds by definition of (y, π). Suppose that the claim holds when the path has length $j-1$ for $j-1 \geq 0$ consider the case when the path has length j. Let

$$(y_{\mathsf{real}}, \pi_{\mathsf{real}}) = \mathsf{uVDF.Eval}\left(1^\lambda, \mathsf{pp}_{\mathsf{uVDF}}, (x, t)\right).$$

We want to show that $(y, \pi) = (y_{\mathsf{real}}, \pi_{\mathsf{real}})$. For the output y, since $\mathsf{frontier}(s)$ contains correct elements corresponding to the first k children of b, it holds that $y = y_{\mathsf{real}}$.

For the proof π, recall that $\pi = (\mathsf{msg}, \pi')$. Let $\pi_{\mathsf{real}} = (\mathsf{msg}_{\mathsf{real}}, \pi'_{\mathsf{real}})$. To show that $\mathsf{msg} = \mathsf{msg}_{\mathsf{real}}$, recall that msg consists of the outputs of the first $k-1$ children of b from F and $\mathsf{msg}_{\mathsf{real}}$ contains the segments the $(x_1, \ldots, x_{k-1})$ of x. Since $\mathsf{frontier}(s)$ is consistent and contains correct elements, it holds that these segments correspond to the outputs of the first $k-1$ children of b given by F, so $\mathsf{msg} = \mathsf{msg}_{\mathsf{real}}$.

To show that $\pi' = \pi'_{\mathsf{real}}$, recall that π' is the proof computed at the previous iteration of the induction. Let b' be the rightmost child of b. If $b' = s$, then $\pi' = \pi'_{\mathsf{real}}$ by definition. Otherwise, by the inductive hypothesis,

$$(*, \pi') = \mathsf{uVDF.Eval}\left(1^\lambda, \mathsf{pp}_{\mathsf{uVDF}}, (x', t/k)\right),$$

where x' is the input of $b'||0$ from $\mathsf{frontier}(s)$. Let x'_{real} be the true input of the sketch child of b, i.e.,

$$(x'_{\mathsf{real}}, y'_{\mathsf{real}}) = \mathsf{Sketch}(\mathsf{pp}_{\mathsf{uVDF}}, (x,t), y_{\mathsf{real}}, \mathsf{msg}_{\mathsf{real}}) = \mathsf{Sketch}(\mathsf{pp}_{\mathsf{uVDF}}, (x,t), y, \mathsf{msg}).$$

By definition of $\mathsf{uVDF.Eval}$, we have that

$$(*, \pi'_{\mathsf{real}}) = \mathsf{uVDF.Eval}(1^\lambda, \mathsf{pp}_{\mathsf{uVDF}}, (x'_{\mathsf{real}}, t/k)).$$

It follows that $x' = x'_{\mathsf{real}}$ by consistency of $b'||0$ with $\mathsf{frontier}(s)$, so the claim follows.

Proposition 6.15. *If $(g, s+1, F)$ satisfies the three invariants defined above, then $F = \mathsf{frontier}(s+1)$.*

Proof. Suppose the three invariants hold for $(g, s+1, F)$. By invariant (a), the set of nodes in F and $\mathsf{frontier}(s+1)$ are the same. Order the elements of F and $\mathsf{frontier}(s+1)$ lexicographically by node. We show by induction on the ordering that for each node s' in $\mathsf{frontier}(s+1)$, its value (x, y, π) in F is equal to its value $(x_{\mathsf{real}}, y_{\mathsf{real}}, \pi_{\mathsf{real}})$ in $\mathsf{frontier}(s+1)$. Let t be the difficulty of s'. For the base case, we have that $x = g = x_{\mathsf{real}}$ by invariant (2), and $(y, \pi) = \mathsf{uVDF.Eval}(1^\lambda, \mathsf{pp}_{\mathsf{uVDF}}, (x,t)) = (y_{\mathsf{real}}, \pi_{\mathsf{real}})$ by invariant (3). Suppose that this holds for all nodes up until some node s'. By invariant (2), it must be that $x = x_{\mathsf{real}}$, since these can always be computed from the previous elements and $\mathsf{frontier}(s+1)$, which agree by assumption. Therefore, just like in the base case, we have that $(y, \pi) = \mathsf{uVDF.Eval}(1^\lambda, \mathsf{pp}_{\mathsf{uVDF}}, (x,t)) = (y_{\mathsf{real}}, \pi_{\mathsf{real}})$.

This gives Claim 6.11, which in turn completes the proof of Lemma 6.9.

□

Lemma 6.16 (Soundness)**.** *The tuple* $(\mathsf{cVDF.Gen}, \mathsf{cVDF.Eval}^{(\cdot)}, \mathsf{cVDF.Verify})$ *satisfies B-soundness.*

Proof. Suppose for contradiction that there exists a non-uniform algorithm $\mathcal{A} = \{\mathcal{A}_\lambda\}_{\lambda\in\mathbb{N}}$ with $\mathsf{size}(\mathcal{A}_\lambda) \in \mathrm{poly}(B(\lambda))$ for all $\lambda \in \mathbb{N}$ and a polynomial p such that for infinitely many $\lambda \in \mathbb{N}$,

$$\Pr\left[\begin{array}{l} \mathsf{pp} \leftarrow \mathsf{cVDF.Gen}(1^\lambda) \\ ((v,T), v') \leftarrow \mathcal{A}(\mathsf{pp}) \end{array} : \begin{array}{l} \mathsf{cVDF.Verify}(1^\lambda, \mathsf{pp}, (v,T), v') = 1 \\ \wedge\, v' \neq \mathsf{cVDF.Eval}^{(T)}(1^\lambda, \mathsf{pp}, v) \end{array}\right] \geq \frac{1}{p(\lambda)}.$$

Then, for every $\lambda \in \mathbb{N}$ we construct an adversary $\mathcal{B}_\lambda$ against the soundness of uVDF, as follows. $\mathcal{B}_\lambda$ receives $\mathsf{pp}_{\mathsf{uVDF}}$ from the challenger for uVDF, computes $\mathsf{pp} = (\mathsf{pp}_{\mathsf{uVDF}}, d', h)$ as done by $\mathsf{cVDF.Gen}(1^\lambda)$, and runs $((v,T), v') \leftarrow \mathcal{A}_\lambda(\mathsf{pp})$.

Whenever $\mathcal{A}_\lambda$ succeeds, v must be well-formed. Specifically, v should be able to be parsed as (g, s, F) where $\mathsf{cVDF.Verify}(1^\lambda, \mathsf{pp}, ((g, 0^h, \emptyset), s), v) = 1$, because otherwise verification only accepts the unique value $\perp$ for v'.

Therefore it must be that v, v' can be parsed appropriately as $v = (g, s, F)$ and $v' = (g, s+T, F')$. Note that either $F = \mathsf{frontier}(s)$ or not. We proceed to describe the attack of $\mathcal{B}_\lambda$ assuming that $F = \mathsf{frontier}(s)$ and show how to handle the case where this does not hold at the end of the claim.

Claim 6.17. *Suppose $F = \mathsf{frontier}(s)$. Then, there exists an algorithm $\mathcal{B}_\lambda$ that breaks the soundness of* uVDF *with probability $1/2p(\lambda)$.*

Proof. $\mathcal{B}_\lambda$ first calculates $(g, s+T, \widetilde{F}) = \mathsf{cVDF.Eval}^{(T)}(1^\lambda, \mathsf{pp}, v)$. He then sorts the elements in $\widetilde{F}, F'$ lexicographically by node and finds the first elements $(s', (x, y, \pi))$, $(s', (\widetilde{x}, \widetilde{y}, \widetilde{\pi}))$ that differ (which must exist by assumption). Then, $\mathcal{B}_\lambda$ and outputs $((x,t), y, \pi)$ and $((\widetilde{x}, t), \widetilde{y}, \widetilde{\pi})$ with equal probability, where $t = k^{h+d'-|s'|}$ is the difficulty of s'.

Before analyzing the success probability of $\mathcal{B}_\lambda$, we note that the size of $\mathcal{B}_\lambda$ is dominated by $\mathsf{size}(\mathcal{A}_\lambda) \in \text{poly}(B(\lambda))$ and the time to run $\mathsf{cVDF.Eval}^{(T)}$. Since v' can be parsed appropriately, then T is a valid node in the tree so $T \leq B^2$, and each iteration of $\mathsf{cVDF.Eval}$ takes $\text{poly}(\lambda)$ time. Therefore in total $\mathsf{size}(\mathcal{B}_\lambda) \in \text{poly}(B(\lambda))$.

To analyze the success probability of $\mathcal{B}_\lambda$, note that by Claims 6.11 and 6.10, since $F = \mathsf{frontier}(s)$ by assumption it holds that $\widetilde{F} = \mathsf{frontier}(s+T)$, $\widetilde{F}$ is consistent, and $\mathsf{uVDF.Verify}$ accepts on the value of every node in $\widetilde{F}$. Similarly, since $\mathsf{cVDF.Verify}(1^\lambda, \mathsf{pp}, (v,t), v')$ accepts, then the set of nodes in F' is the set of nodes in $\mathsf{frontier}(s+T)$, F' is a consistent set, and $\mathsf{uVDF.Verify}$ accepts on the value of every element of F'. We proceed by comparing $(s', (x,y,\pi))$ and $(s', (\widetilde{x}, \widetilde{y}, \widetilde{\pi}))$, and note that they correspond to the same node s' since the sets of nodes in F' and $\widetilde{F}$ are the same. Let $S = F' \cap \widetilde{F}$ i.e., S consists of all elements in F' and $\widetilde{F}'$ with equal nodes and values. Note that S contains all elements in $F' \cup \widetilde{F}$ with nodes that are lexicographically smaller than s' by assumption.

By consistency, we claim that $x = \widetilde{x}$. To see this, if s' is a middle child, then $x, \widetilde{x}$ are both the output of the sibling to its left, which is in S. If s' is a rightmost child, then $x, \widetilde{x}$ both correspond to the sketch of their left siblings' values, again in S. If s' is a leftmost child, then $x, \widetilde{x}$ recursively must be equal to their parent's input. Since the base case of this recursion is at the root with input g, then whichever type of node s' is (leftmost, middle, or rightmost) it follows that its input is found in S, so $x = \widetilde{x}$. Therefore, we have that for $(y,\pi) \neq (\widetilde{y}, \widetilde{\pi})$

$$\mathsf{uVDF.Verify}(1^\lambda, \mathsf{pp}_{\mathsf{uVDF}}, (x,t), (y,\pi)) = \mathsf{uVDF.Verify}(1^\lambda, \mathsf{pp}_{\mathsf{uVDF}}, (x,t), (\widetilde{y}, \widetilde{\pi})).$$

Since the output of $\mathsf{uVDF.Eval}(1^\lambda, \mathsf{pp}_{\mathsf{uVDF}}, (x,t))$ is unique, at least one of these must be different to that value. Therefore, when $\mathcal{A}_\lambda$ succeeds, $\mathcal{B}_\lambda$ succeeds with probability $\frac{1}{2}$, so $\mathcal{B}_\lambda$ breaks the soundness of uVDF with probability $\frac{1}{2p(\lambda)}$, in contradiction.

It remains to analyze the case where $F \neq \mathsf{frontier}(s)$. We extend $\mathcal{B}_\lambda$ to also calculate $(g, s, \widehat{F}) = \mathsf{cVDF.Eval}^{(s)}(1^\lambda, \mathsf{pp}, (g, 0^h, \emptyset))$ and compare $\widehat{F}$ to F. Note that this only adds $\text{poly}(B(\lambda))$ time to the running time of $\mathcal{B}_\lambda$. Then, when $\widehat{F} \neq F$, $\mathcal{B}_\lambda$ finds the first element that differs between $\widehat{F}$ and F, rather than $\widetilde{F}$ and F' as done above. Since $\widehat{F} = \mathsf{frontier}(s)$ by Claim 6.11, then the same argument as in the above claim follows for this case.

□

Lemma 6.18 (Length Bounded)**.** *There exists a polynomial $p_{\mathsf{len}}(\lambda) \in O(\lambda^5)$ such that for $\lambda \in \mathbb{N}$, $(\mathsf{pp}_{\mathsf{uVDF}}, d', h) \in \text{Supp}(\mathsf{cVDF.Gen}(1^\lambda))$ and $v \in \{0,1\}^*$, it holds that $|\mathsf{cVDF.Eval}(1^\lambda, \mathsf{pp}, v)| \leq p_{\mathsf{len}}(\lambda)$.*

Proof. First, observe that if v is not well-formed, then $\mathsf{cVDF.Eval}$ outputs $\perp$. Therefore, we focus on the case where v is well formed, so $v = (g, s, F)$. Let $v' = \mathsf{cVDF.Eval}(1^\lambda, \mathsf{pp}, v)$. By definition of Eval, $v' = (g, s+1, F')$ where F' has length corresponding to a frontier.

To bound the length of v', we have that $|g| \leq 2\lambda$ and $|s+1| \leq h$, so we only need to bound the length of F. Any frontier contains at most k nodes in each of the h levels. Moreover, each entry $(s', (x,y,\pi))$ satisfies that $|s| \leq h$, $x, y \in \mathbb{Z}_N^\star$. For π, it is straightforward to see that it contains at most $k \cdot h$ elements of $\mathbb{Z}_N^\star$. Using the fact that $k, h \leq \lambda$, we obtain the desired bound. □

Lemma 6.19 (Honest Evaluation)**.** *For all sufficiently large $\lambda \in \mathbb{N}$, any $\mathsf{pp} = (\mathsf{pp}_{\mathsf{uVDF}}, d', h)$ in the support of $\mathsf{cVDF.Gen}(1^\lambda)$, $v \in \{0,1\}^*$, and $T \in [D(\lambda), B(\lambda)]$, it holds that $\mathsf{cVDF.Eval}^{(T)}(1^\lambda, \mathsf{pp}, v)$ can be computed in time $T \cdot (1+\delta') \cdot k^{d'} \cdot \ell(\lambda)$.*

Proof. Fix $\lambda \in \mathbb{N}$, $\mathsf{pp} = (\mathsf{pp}_{\mathsf{uVDF}}, d', h)$ in the support of $\mathsf{cVDF.Gen}(1^\lambda)$ where $\mathsf{pp}_{\mathsf{uVDF}} = (N, B, k, d, \mathsf{hash})$, $v \in \{0,1\}^*$, and $T \in \mathbb{N}$. Recall that $\mathsf{Sketch}(\mathsf{pp}, \cdot, \cdot, \cdot)$ runs in time at most λ^{c_1} for some constant c_1 by Claim 5.10. In the following claims, we analyze the running time to compute $\mathsf{cVDF.Eval}(1^\lambda, \mathsf{pp}, v)$.

Claim 6.20. *There exists a polynomial p_1 such that the running time of* $\mathsf{cVDF.Eval}(1^\lambda, \mathsf{pp}, v)$ *to check that v is well-formed is at most $p_1(\lambda) \cdot \lambda^d \cdot \ell(\lambda)$.*

Proof. The running time of this phase is dominated by the running time of $\mathsf{cVDF.Verify}(1^\lambda, \mathsf{pp}, ((g, 0^h, \emptyset), s), v)$ after parsing $v = (g, s, F)$. It is easy to see that $\mathsf{cVDF.Verify}$ runs in time polynomial in k, h, λ, and in the running time of $\mathsf{uVDF.Verify}(1^\lambda, \mathsf{pp}_{\mathsf{uVDF}}, (x, t), (y, \pi))$ for elements $(s', (x, y, \pi))$ in F and $t = k^{h+d'-|s'|}$. In particular, each call to $\mathsf{uVDF.Verify}$ checks the validity of its inputs in time polynomial in λ, does k^d squarings at the base case, and otherwise runs Sketch and calls itself recursively at most $\log_k(t)$ times. Therefore $\mathsf{uVDF.Verify}$ takes time $\mathrm{poly}(\lambda) \cdot \log_k(t) + k^d \cdot \ell(\lambda)$ for a fixed polynomial independent of d. Recalling that $k = \lambda$, $t \le B(\lambda) \le 2^\lambda$, and that $\mathsf{uVDF.Verify}$ is run at most $k \cdot h \le \lambda^2$ times gives the claim.

If v is not well-formed, then $\mathsf{cVDF.Eval}$ simply outputs $\perp$, otherwise it continues to the next phase, where we can assume that $v = (g, s, F)$.

Claim 6.21. *There exists a polynomial p_2 such that to compute* $\mathsf{frontier}(s+1)$ *and* $\mathsf{val}(s)$*, the running time of* $\mathsf{cVDF.Eval}(1^\lambda, \mathsf{pp}, v)$ *is at most $p_2(\lambda) + (1+\delta) \cdot k^{d'} \cdot \ell(\lambda)$.*

Proof. Computing $\mathsf{frontier}(s+1)$ consists of computing the input, output, and proof of s and $a^\star$, where $a^\star$ the child of the closest common ancestor of s and $s+1$ along the path to s.

The input of s is computed either as a sketch of its left siblings values in F, as the output of the sibling to its left in F, or recursively as its parent's input. Therefore, this takes time at most $O(k \cdot h + h \cdot \lambda^{c_1})$ where λ^{c_1} is the time to compute a sketch. Computing the output of s requires running $\mathsf{uVDF.Eval}$ its input, which takes time $(1+\delta) \cdot k^{d'} \cdot \ell(\lambda)$ by assumption. Finally, the proof for s is $\perp$.

If $a^\star = s$, then computing the input-output pair of $a^\star$ does not take additional time. Otherwise, the input and output are computed by looking them up in F, which takes time $O(k \cdot h)$. Finally, computing its proof requires taking the outputs of the left siblings of $a^\star$ along with the proof of its rightmost child, computed recursively. Therefore, computing the proof given F takes time $O(k \cdot h)$.

Therefore, computing the values of s and $a^\star$ takes time $p_2(\lambda) + (1+\delta) \cdot k^{d'} \cdot \ell(\lambda)$ where $p_2(\lambda)$ is a polynomial in $O(\lambda^2 + \lambda^{c_1+1})$.

Let c be a constant such that $p_1(\lambda) + p_2(\lambda) \le \lambda^c$ for sufficiently large $\lambda \in \mathbb{N}$. By Claims 6.20 and 6.21, the T-wise composition of $\mathsf{cVDF.Eval}$ can be done in time $T \cdot \ell(\lambda) \cdot (\lambda^{c+d} + (1+\delta) \cdot k^{d'})$. Since $k = \lambda$, we can set $d' \ge c + d + 1$, so $\lambda^{c+d}/k^{d'} \le 1/\lambda \in o(1)$. It follows that the total running time is at most $T \cdot (1 + \delta + 1/\lambda) \cdot k^{d'} \cdot \ell(\lambda)$, so in particular for any constant $\delta' > \delta$ and sufficiently large $\lambda \in \mathbb{N}$, the total running time is at most $T \cdot (1+\delta') \cdot k^{d'} \cdot \ell(\lambda)$. □

Lemma 6.22 (Sequentiality)**.** $(\mathsf{cVDF.Gen}, \mathsf{cVDF.Sample}, \mathsf{cVDF.Eval}^{(\cdot)})$ *satisfies sequentiality.*

Proof. Let $\delta' > \delta$ be the constant from honest evaluation and let $\epsilon'' > \frac{\epsilon+\delta'}{1+\delta'}$. By way of contradiction, suppose there exists a non-uniform algorithm $\mathcal{A}_0 = \{\mathcal{A}_{0,\lambda}\}_{\lambda \in \mathbb{N}}$ where $\mathsf{size}(\mathcal{A}_{0,\lambda}) \in \mathrm{poly}(B(\lambda))$ for all $\lambda \in \mathbb{N}$ and a polynomial p such that for infinitely many $\lambda \in \mathbb{N}$ it holds that

$$\Pr\left[\begin{array}{l} \mathsf{pp} \leftarrow \mathsf{cVDF.Gen}(1^\lambda) \\ \mathcal{A}_1 \leftarrow \mathcal{A}_{0,\lambda}(\mathsf{pp}) \\ (g, 0^h, \emptyset) \leftarrow \mathsf{cVDF.Sample}(1^\lambda, \mathsf{pp}) \\ (T, v) \leftarrow \mathcal{A}_1((g, 0^h, \emptyset)) \end{array} : \begin{array}{l} \mathsf{cVDF.Eval}^{(T)}(1^\lambda, \mathsf{pp}, (g, 0^h, \emptyset)) = v \\ \wedge\ \mathsf{depth}(\mathcal{A}_1) \\ \quad \leq (1-\epsilon'') \cdot T \cdot (1+\delta') \cdot k^{d'} \cdot \ell(\lambda) \end{array}\right]$$
$$\geq \frac{1}{p(\lambda)}$$

We construct a non-uniform algorithm $\mathcal{B}_0 = \{\mathcal{B}_{0,\lambda}\}_{\lambda \in \mathbb{N}}$ to break the sequential property of $\mathsf{RSW} = (\mathsf{RSW.Gen}, \mathsf{RSW.Sample}, \mathsf{RSW.Eval}^{(\cdot)})$ as follows. For every $\lambda \in \mathbb{N}$, $\mathcal{B}_{0,\lambda}(N)$ first computes $\mathsf{pp} = (N, B, k, d, \mathsf{hash}, d', h)$ distributed according to $\mathsf{cVDF.Gen}(1^\lambda)$. It then runs $\mathcal{A}_{0,\lambda}(\mathsf{pp})$ to receive $\mathcal{A}_1$ and constructs a circuit $\mathcal{B}_1$ that expects an input g from $\mathsf{RSW.Sample}(1^\lambda, N)$.

Overview of $\mathcal{B}_1$. At a high level, $\mathcal{B}_1(g)$ first runs $\mathcal{A}_1((g, 0^h, \emptyset))$ to get some intermediate state $v = (g, T, F)$ (note that g is distributed identically in both $\mathsf{RSW.Sample}$ and $\mathsf{uVDF.Sample}$). When $\mathcal{A}_1$ succeeds, then $v = \mathsf{cVDF.Eval}^{(T)}(1^\lambda, \mathsf{pp}, (g, 0^h, \emptyset))$. $\mathcal{B}_1$ will use the partial computation done by $\mathcal{A}_1$ on input g in order to efficiently compute an output y and an integer t such that $g^{2^t} = y$. Consider the nodes in the frontier given by $\mathcal{A}_1$. If we look at the tree and remove all sketch nodes and their sub-trees, then we are left with a k-ary tree, where the last node in this tree that appears in F will have the output y such that $y = g^{2^t}$ for some t (which is related to T). We will use this to break the sequential property of RSW. Just as done in the proof of sequentiality for the uVDF, we show that the sketch nodes that are computed by $\mathcal{A}_1$ do not introduce too much overhead for $\mathcal{B}_1$.

Formal definition of $\mathcal{B}_1$. The algorithm $\mathcal{B}_1(g)$ does the following:

1. Compute $(T, v) \leftarrow \mathcal{A}_1((g, 0^h, \emptyset))$ and parse v as (g, T, F).
2. Let $S \subseteq F$ be the set of elements corresponding to nodes that do not have any ancestors that are rightmost children (i.e., remove all nodes in the sub-tree rooted at sketch nodes). Let m_i be the number of nodes in S at height i for $i \in \{0, \dots, h\}$.
3. Compute $t = \sum_{i=0}^{h} m_i \cdot k^{i+d'}$ (and note that $k^{i+d'}$ is the difficulty at height i by definition).[20]
4. Let y be the output of the rightmost node in S.
5. Output (t, y') where y' is equal to y or $(N-y)$ with equal probability.

It remains to show that for all $\lambda \in \mathbb{N}$, $\mathcal{B}_{0,\lambda}$ as defined above succeeds at breaking the sequentiality property of RSW whenever $\mathcal{A}_1$ succeeds as above. Namely, we will show that for infinitely many $\lambda \in \mathbb{N}$,

$$\Pr\left[\begin{array}{l} N \leftarrow \mathsf{RSW.Gen}(1^\lambda) \\ \mathcal{B}_1 \leftarrow \mathcal{B}_{0,\lambda}(N) \\ g \leftarrow \mathsf{RSW.Sample}(1^\lambda, N) \\ (t, y) \leftarrow \mathcal{B}_1(g) \end{array} : \begin{array}{l} \mathsf{RSW.Eval}^{(t)}(1^\lambda, N, g) = y \\ \wedge\ \mathsf{depth}(\mathcal{B}_1) \leq (1-\epsilon) \cdot t \cdot \ell(\lambda) \\ \wedge\ t \geq D(\lambda) \end{array}\right] \geq \frac{1}{2p(\lambda)}$$

Let H be the largest height of the tree with nodes contained in F (and therefore in S by definition), i.e., H is the largest value such that $T \geq (k+1)^H + 1$. First, we observe that the

[20] In general, when we refer to the height of a node, this is the height of the subtree rooted at that node.

difficulty t that $\mathcal{B}_1$ outputs satisfies $t \geq m_H \cdot k^{H+d'}$ where $m_H \geq 1$ and $H \geq 0$, so $t \geq D(\lambda) = \lambda^{d'}$ always holds since $k = \lambda$. In Claim 6.23, we show that either y or $N - y$ is equal to x^{2^t} whenever $\mathcal{A}_1$ computes the correct output v of $\mathsf{cVDF.Eval}^{(T)}(1^\lambda, \mathsf{pp}, (g, 0^h, \emptyset))$, and in Claim 6.24, we show that the circuit $\mathcal{B}_1$ computed by $\mathcal{B}_{0,\lambda}$ has the correct depth whenever $\mathcal{A}_1$ as computed by $\mathcal{A}_{0,\lambda}$ has the correct depth.

Claim 6.23. *Let $(T, v) \leftarrow \mathcal{A}_1((g, 0^h, \emptyset))$ and $(t, y) \leftarrow \mathcal{B}_1(g)$. If $\mathsf{cVDF.Eval}^{(T)}(1^\lambda, \mathsf{pp}, (g, 0^h, \emptyset)) = v$ then $\mathsf{RSW.Eval}^{(t)}(1^\lambda, N, g) \in \{y, N - y\}$.*

Proof. Let $v = (g, T, F)$ and H, S, and m_i be as defined above for all i. By completeness (specifically Lemma 6.9 and Claim 6.11), it holds that $F = \mathsf{frontier}(T)$.

Order the elements of S lexicographically by node, i.e., from left to right in the tree. Let $x_{(i,j)}, y_{(i,j)}$ be the input and output, respectively, of the jth node at height i in S under this ordering, where for a pair indexed by (i, j) it holds that $j \in \{1, \ldots, m_i\}$. Let $t_i = k^{i+d'}$ be the difficulty at height i. We want to show that the rightmost node under this ordering has output $y = \left|g^{2^t}\right|_N$ or $N - y$, where $t = \sum_{i=0}^{H} m_i \cdot k^{i+d'}$.

Toward this goal, we first show by induction on $j \in \{1, \ldots, m_i\}$ that for every fixed height $i \in \{1, \ldots, H\}$, it holds that $y_{(i,j)} = \left|(x_{i,1})^{2^{j \cdot t_i}}\right|_N$. For the base case, it follows by correctness of the elements of F that $y_{(i,1)} = \left|(x_{i,1})^{2^{t_i}}\right|_N$. Suppose this holds for the jth node at height i for some $j \in [1, m_i)$. The $(j+1)$st node must be a middle child (since F does not contain rightmost children) and hence this follows by the consistency and correctness of elements of F.

Next, we show by induction on the height i (decreasing from the node at height H down to the leaf nodes) that $y_{(i,m_i)} = \left|g^{2^{L_i}}\right|_N$ where $L_i = \sum_{i'=i}^{H} m_{i'} \cdot t_{i'}$ is the sum of the difficulties so far, which will give the claim. For the base case, when $i = H$ it follows from the consistency of F that the input $x_{(H,1)}$ of the first node is g, which by the above induction implies that $y_{(H,m_H)} = \left|g^{2^{m_H \cdot t_H}}\right|_N$. Suppose this holds for all nodes at heights from H through some height $i \in (0, H]$, and consider the node with input $x_{(i-1,1)}$, meaning the first node at height $i - 1$. By consistency of F, it holds that $x_{(i-1,1)} = y_{(i',m_{i'})}$, where i' is the closest level higher than i where $m_{i'} \neq 0$ (and note that the $m_{i'}$th node at height i' must be in S if there is a node in S at height $i + 1$). Therefore, it follows by the first induction that

$$\begin{aligned} y_{(i-1,m_{i-1})} &= \left| \left(x_{(i-1,1)}\right)^{2^{m_{i-1} \cdot t_{i-1}}} \right|_N = \left| \left(y_{(i',m_{i'})}\right)^{2^{m_{i-1} \cdot t_{i-1}}} \right|_N \\ &= \left|g^{2^{L_{i'} + m_{i-1} \cdot t_{i-1}}}\right|_N = \left|g^{2^{L_{i-1}}}\right|_N, \end{aligned}$$

which completes the proof of the claim.

Claim 6.24. *If $\epsilon'' > \frac{\epsilon+\delta}{1+\delta'}$ and $\mathsf{depth}(\mathcal{A}_1) \leq (1 - \epsilon'') \cdot T \cdot k^{d'} \cdot (1 + \delta') \cdot \ell(\lambda)$, then for sufficiently large $\lambda \in \mathbb{N}$,*

$$\mathsf{depth}(\mathcal{B}_1) \leq (1 - \epsilon) \cdot t \cdot \ell(\lambda).$$

Proof. $\mathcal{B}_1$ first runs $\mathcal{A}_1$, which by assumption has depth at most $(1 - \epsilon'') \cdot T \cdot k^{d'} \cdot (1 + \delta') \cdot \ell(\lambda)$. Because the depth of $\mathcal{B}_1$ must be bounded by its output t and the depth of $\mathcal{A}_1$ depends on its output T, we first relate the outputs of $\mathcal{B}_1$ and $\mathcal{A}_1$, t and T, respectively.

Toward that goal, we show that $T \cdot k^{d'} \in t \cdot (1 + o(1))$, meaning that t and the number of squares done by $\mathcal{A}_1$ are almost the same. We split this into two cases, depending on the number of nodes m_H at height H.

Case 1. In the first case, suppose $m_H < k$ and let H' be the next highest height with nodes that appear in F (note that if there are no nodes below height H, then $T = t$ so this follows trivially). Then, it holds that

$$T \leq m_H \cdot (k+1)^H + (m_{H'} + 1) \cdot (k+1)^{H'},$$

since T steps of cVDF correspond to all the leaf nodes necessary to compute the m_H nodes at height H, the $m_{H'}$ nodes at height H', and then the remaining nodes at lower heights, where the latter is upper bounded by $(k+1)^{H'}$. Additionally, since $m_H < k$, then we have that

$$t \geq m_H \cdot k^{H+d'} + m_{H'} \cdot k^{H'+d'}$$

where here we used the fact that S includes m_H nodes at height H and $m_{H'}$ nodes at height H' (since there cannot be sketch nodes among these since $m_H < k$). Therefore, noting that $H' \leq H-1$, we have that

$$\begin{aligned}
T \cdot k^{d'} &\leq k^{d'} \cdot \left(m_H \cdot (k+1)^H + (m_{H'}+1) \cdot (k+1)^{H-1}\right) \\
&\leq k^{d'} \cdot \left(m_H \cdot (k^H + H^2 \cdot k^{H-1}) + (m_{H'}+1) \cdot (k^{H-1} + H^2 \cdot k^{H-2})\right) \\
&\leq k^{d'} \cdot \big(m_H \cdot k^H + m_H \cdot k^{H-1} \cdot H^2 \\
&\quad + m_{H'} \cdot k^{H-1} + m_{H'} \cdot H^2 \cdot k^{H-2} + k^{H-1} + H^2 \cdot k^{H-2}\big) \\
&\leq t\left(1 + 1/\lambda^{1/3}\right) + k^{H+d'-1} + \lambda^{2/3} \cdot k^{H+d'-2} \\
&\leq t\left(1 + 1/\lambda^{1/3}\right) + 2 \cdot k^{H+d'-1} \leq t\left(1 + 1/\lambda^{1/3} + 1/\lambda\right) \in t(1+o(1)),
\end{aligned}$$

where we used that $k = \lambda$, that $(k+1)^H \leq k^H + H \cdot k^{H-1} + \ldots + H^H \leq k^H + H^2 \cdot k^{H-1}$, and that $H \leq \log_{k+1} T \leq \lambda^{1/3}$ since $T \leq B(\lambda) \leq 2^{\lambda^{1/3}}$.

Case 2. In the second case, suppose that $m_H = k$. Then, we have that

$$T \leq (m_H + 1) \cdot (k+1)^H = (k+1)^{H+1},$$

and

$$t = k^{H+d'+1},$$

since if $m_H = k$ then all nodes in F below height H have an ancestor that is a rightmost child, so they are not included in S. Therefore, we have that

$$\begin{aligned}
T \cdot k^{d'} &\leq k^{d'} \cdot (k+1)^{H+1} \\
&\leq k^{d'} \cdot (k^{H+1} + (H+1)^2 \cdot k^H) \\
&\leq k^{H+d'+1} + (2 \cdot \lambda^{2/3}) \cdot k^{H+d'} \\
&\leq t(1 + 2/\lambda^{1/3}) \in t(1+o(1))
\end{aligned}$$

as above.

Putting it all together. After running $\mathcal{A}_1$, recall that $\mathcal{B}_1$ needs to compute t from the frontier given by $\mathcal{A}$ and needs to find y in the frontier, which has some polynomial overhead $\lambda^{c'}$ for some constant c'. Setting $d' = c' + c + d + 1$, where c, d are the constant specified in Lemma 6.19, we get that this overhead is $\lambda^c \in o(t)$ since $t \geq \lambda^{d'}$.

To conclude, let $\gamma > 0$ be any constant such that $T \cdot k^{d'} + \lambda^c \leq t(1+\gamma)$. We have that for sufficiently large $\lambda \in \mathbb{N}$,

$$\begin{aligned}\mathsf{depth}(B_1) &\leq \mathsf{depth}(\mathcal{A}_1) + \lambda^c \\ &\leq (1-\epsilon'') \cdot T \cdot k^{d'} \cdot (1+\delta') \cdot \ell(\lambda) + \lambda^c \\ &\leq (1-\epsilon'') \cdot (1+\delta') \cdot \ell(\lambda) \cdot (1+\gamma) \cdot t.\end{aligned}$$

This is bounded by $(1-\epsilon) \cdot \ell(\lambda) \cdot t$ as long as $\epsilon'' \geq 1 - \frac{1-\epsilon}{(1+\delta')(1+\gamma)}$.

Furthermore, we can choose γ to be arbitrarily small, so we only require that ϵ' is a constant strictly greater than $\frac{\epsilon+\delta'}{1+\delta}$ which is given by assumption.

Sequentiality then follows since $\mathcal{B}_{0,\lambda}$ succeeds as required with probability $1/2p(\lambda)$ for sufficiently large $\lambda \in \mathbb{N}$. □

7 Applications

We formalize some applications of continuous VDFs below.

7.1 Public Randomness Beacons

A randomness beacon is an ideal functionality proposed by Rabin [Rab83]. In this section, we formally define a computational variant of this notion and show that it can be achieved using a continuous VDF. In the following definition, the algorithm Tick corresponds to the iterated sequential function which computes the states of the beacon, and the algorithm Tock gives the output of the beacon at each time step.

Definition 7.1 ((B,ℓ,Δ)-Public Randomness Beacon)**.** *Let $B, \ell, \Delta \colon \mathbb{N} \to \mathbb{N}$. A (B,ℓ,Δ)-public* randomness beacon *is a tuple of algorithms* (Gen, Init, Tick, Tock, Verify) *with the following syntax:*

- $\mathsf{pp} \leftarrow \mathsf{Gen}(1^\lambda)$**:** *A PPT algorithm that takes as input 1^λ and outputs the public parameters* pp.
- $\mathsf{state}_0 \leftarrow \mathsf{Init}(1^\lambda, \mathsf{pp}, x_0)$**:** *A deterministic algorithm that takes as input 1^λ,* pp*, and a starting point x_0 and outputs the initial state* state_0.
- $\mathsf{state}' \leftarrow \mathsf{Tick}(1^\lambda, \mathsf{pp}, \mathsf{state})$**:** *A deterministic algorithm that takes as input 1^λ,* pp*, and a state* state*, and outputs the next state* state'*. We let* $\mathsf{Tick}^{(\cdot)}$ *denote an algorithm that takes 1^λ,* pp*, and* (state, t) *as input and outputs the t-wise composition* $\mathsf{Tick}^{(t)}(1^\lambda, \mathsf{pp}, \cdot)$ *on input* state.
- $x \leftarrow \mathsf{Tock}(1^\lambda, \mathsf{pp}, \mathsf{state})$**:** *A deterministic algorithm that takes as input 1^λ,* pp*, and a state* state*, and outputs the value x of the beacon corresponding to that state.*
- $b \leftarrow \mathsf{Verify}(1^\lambda, \mathsf{pp}, (x_0, t), (x, \mathsf{state}))$**:** *A deterministic algorithm that takes as input 1^λ,* pp*, (x_0, t) for $x_0 \in \{0,1\}^\lambda$ and $t \in \mathbb{N}$, and (x, state) for a beacon value x and a state* state*, and outputs $b \in \{0,1\}$.*

We require the following properties to hold:

Completeness: *For all $t \leq B(\lambda)$ and $x_0 \in \{0,1\}^\lambda$, let* $\mathsf{state}_0 = \mathsf{Init}(1^\lambda, \mathsf{pp}, x_0)$ *and* $\mathsf{state}_t = \mathsf{Tick}^{(t)}(1^\lambda, \mathsf{pp}, \mathsf{state}_0)$*. Then*

$$\mathsf{Verify}(1^\lambda, \mathsf{pp}, (x_0, t), (\mathsf{Tock}(1^\lambda, \mathsf{state}_t), \mathsf{state}_t)) = 1.$$

Soundness: *For all $t \leq B(\lambda)$ and non-uniform algorithms $\mathcal{A} = \{\mathcal{A}_\lambda\}_{\lambda \in \mathbb{N}}$ such that $\mathsf{size}(\mathcal{A}_\lambda) \in \mathrm{poly}(B(\lambda))$, there exists a negligible function* negl *such that for all $\lambda \in \mathbb{N}$,*

$$\Pr\left[\begin{array}{l} \mathsf{pp} \leftarrow \mathsf{Gen}(1^\lambda) \\ x_0 \leftarrow \{0,1\}^\lambda \\ (x, \mathsf{state}, t) \leftarrow \mathcal{A}_\lambda(\mathsf{pp}, x_0) \\ \mathsf{state}_0 = \mathsf{Init}(1^\lambda, \mathsf{pp}, x_0) \end{array} : \begin{array}{l} x \neq \mathsf{Tock}(1^\lambda, \mathsf{Tick}^{(t)}(1^\lambda, \mathsf{pp}, \mathsf{state}_0)) \\ \mathsf{Verify}(1^\lambda, \mathsf{pp}, (x_0, t), (x, \mathsf{state})) = 1 \end{array}\right]$$
$$\leq \mathsf{negl}(\lambda).$$

Honest Evaluation: *For all $\lambda \in \mathbb{N}$ and $\mathsf{pp} \in \mathrm{Supp}\big(\mathsf{Gen}(1^\lambda)\big)$, $\mathsf{Tick}(1^\lambda, \mathsf{pp}, \cdot)$ runs in time at most $\ell(\lambda)$.*

Indistinguishability: *For every $t \leq B(\lambda)$ and non-uniform algorithm $\mathcal{A}_0 = \{\mathcal{A}_{0,\lambda}\}_{\lambda \in \mathbb{N}}$ with $\mathsf{size}(\mathcal{A}_{0,\lambda}) \in \mathrm{poly}(B(\lambda))$, there exists a negligible function* negl *such that the probability that the following experiment outputs 1 is at most $1/2 + \mathsf{negl}(\lambda)$ for all $\lambda \in \mathbb{N}$ and $t + \Delta(t) \leq t' \leq B(\lambda)$:*

$\mathsf{Exp}_{t'}(\lambda)$*:*

1. $\mathsf{pp} \leftarrow \mathsf{Gen}(1^\lambda)$
2. $\mathcal{A}_1 \leftarrow \mathcal{A}_{0,\lambda}(\mathsf{pp}, t')$
3. $x_0 \leftarrow \{0,1\}^\lambda$, $b \leftarrow \{0,1\}$, $\mathsf{state}_0 \leftarrow \mathsf{Init}(1^\lambda, \mathsf{pp}, x_0)$
4. *If $b = 0$, then $x \leftarrow \{0,1\}^\lambda$. Else, $x = \mathsf{Tock}(1^\lambda, \mathsf{Tick}^{(t')}(1^\lambda, \mathsf{pp}, \mathsf{state}_0))$.*
5. $b' \leftarrow \mathcal{A}_1(x, x_0, \mathsf{state}_0)$
6. *Output 1 if $b = b'$ and $\mathsf{depth}(\mathcal{A}_1) \leq t \cdot \ell$ and 0 otherwise.*

While this notion is very related to that of a cVDF, we remark on a few points about the definition of a public randomness beacon above and how it relates to a cVDF.

First, for an honest algorithm that starts computing as soon as the seed x_0 and start state $\mathsf{state}_0 = \mathsf{RB.Init}(1^\lambda, \mathsf{pp}, x_0)$ is generated, it will be able to output the random value produced by $\mathsf{Tock}(1^\lambda, \mathsf{Tick}^{(t)}(1^\lambda, \mathsf{pp}, \mathsf{state}_0))$ at time $t \cdot \ell(\lambda) + s(\lambda)$ where $s(\lambda)$ is the time to compute $\mathsf{Tock}(1^\lambda, \cdot)$. After computing the initial state, it continuously computes Tick and can compute the beacon at each interval by spawning an extra processor in parallel. At any point in time, the algorithm will need to run at most $s(\lambda)/\ell(\lambda)$ extra processors in parallel, which is independent of the number of steps t so far. Furthermore, any other party that arrives at some time $t \cdot \ell(\lambda)$ and sees $\mathsf{state}_t = \mathsf{Tick}^{(t)}(1^\lambda, \mathsf{pp}, \mathsf{state}_0)$ will be able to output the beacon value at the same time. Thus, ℓ is the interval time corresponding to the randomness beacon, but there will be a slight delay corresponding to the time to compute Tock and Init before the first beacon is output. An illustration of this is given in Figure 5.

Second, a randomness beacon only guarantees completeness, soundness, and indistinguishability with respect to an honestly chosen seed. In contrast, a cVDF specifies a stronger property that it is verifiable from any point in its computation, and we only require that sequentiality holds from an honestly sampled starting point. We could have defined this weaker notion of a cVDF for this application, but we found this stronger notion more natural for a general verifiable iteratively sequential function.

Lastly, a randomness beacon guarantees that up until time $t \cdot \ell(\lambda)$, the random values produced by the beacon $\Delta(t)$ steps into the future are indistinguishable from random. For a cVDF, we only guarantee that the states past time $t + \Delta(t)$ cannot be computed—or, stated differently, are unpredictable—before time $(1 - \epsilon) \cdot (t + \Delta(t))$. To deal with this, we can apply a suitable hash

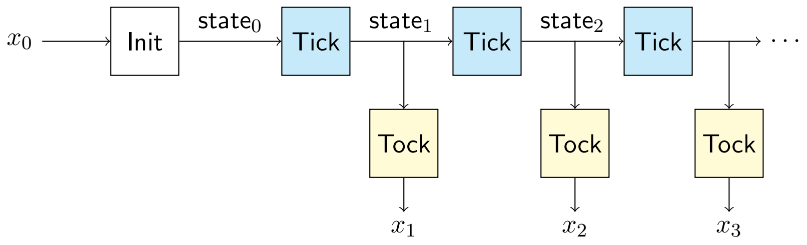

Figure 5: An example of running the randomness beacon. At each time step, running Tick on state$_i$ produces state$_{i+1}$, where all states are publicly verifiable. Then, each state can be used to get the value of the beacon at that time. Note that Tick is iterated sequentially, while Tock is applied to the output at each step.

function to the cVDF's state at each step to obtain such a pseudorandom value. We show that this is secure assuming a pseudo-random generator (PRG) for unpredictable seeds, which are known to exist in the random oracle model. Intuitively, a PRG for unpredictable seeds is a function whose output is indistinguishable from random against a class of algorithms that cannot predict the input to the PRG. We need such a notion of PRGs for unpredictable seeds with respect to classes of algorithms which have bounded depth. We formally define this notion as follows:

Definition 7.2 (Unpredictable distributions)**.** *Let $\mathcal{C} = \{\mathcal{C}_\lambda\}_{\lambda\in\mathbb{N}}$ be a collection of classes of circuits and let $\{S_\lambda\}_{\lambda\in\mathbb{N}}$ be a family of distributions where S_λ is a distribution over $(x,z) \in \{0,1\}^{\mathrm{poly}(\lambda)} \times \{0,1\}^{\mathrm{poly}(\lambda)}$ for all $\lambda \in \mathbb{N}$. We say that S is an* unpredictable distribution against $\mathcal{C}$ *if for every algorithm $\mathcal{A} = \{\mathcal{A}_\lambda\}_{\lambda\in\mathbb{N}}$ where $\mathcal{A}_\lambda \in \mathcal{C}_\lambda$ for all $\lambda \in \mathbb{N}$, there exists a negligible function* negl *such that for all $\lambda \in \mathbb{N}$,*

$$\Pr\left[\begin{array}{c}(x,z) \leftarrow S_\lambda \\ x' \leftarrow \mathcal{A}_\lambda(z)\end{array} : x' = x\right] \leq \mathsf{negl}(\lambda).$$

Definition 7.3 (PRGs for unpredictable seeds)**.** *Let $\mathcal{C} = \{\mathcal{C}_\lambda\}_{\lambda\in\mathbb{N}}$ be a collection of classes of circuits and let $G = \{G_\lambda \colon \{0,1\}^{\mathrm{poly}(\lambda)} \to \{0,1\}^\lambda\}_{\lambda\in\mathbb{N}}$ be a family of polynomial-time computable functions. We say that G is a* PRG for unpredictable seeds against $\mathcal{C}$ *if for any unpredictable distribution $\{S_\lambda\}_{\lambda\in\mathbb{N}}$ against $\mathcal{C}$, for any distinguisher $\mathcal{B} = \{\mathcal{B}_\lambda\}_{\lambda\in\mathbb{N}}$ where $\mathcal{B}_\lambda \in \mathcal{C}_\lambda$ for all $\lambda \in \mathbb{N}$, there exists a negligible function* negl *such that for all $\lambda \in \mathbb{N}$,*

$$\left|\Pr\left[(x,z) \leftarrow S_\lambda : \mathcal{B}_\lambda(G_\lambda(x), z, G_\lambda) = 1\right] - \Pr\left[\begin{array}{c}(x,z) \leftarrow S_\lambda \\ r \leftarrow \{0,1\}^\lambda\end{array} : \mathcal{B}_\lambda(r, z, G_\lambda) = 1\right]\right|$$
$$\leq \mathsf{negl}(\lambda).$$

We now show that the PRGs for unpredictable seeds and cVDFs (with suitable parameters) suffice to construct a public randomness beacon. One subtle point is that for our construction, we need to start with a cVDF where cVDF.Sample includes its randomness in its output. In our cVDF construction, Sample(pp) outputs $(g, 0^h, \perp)$ where g is a uniform group element. Therefore, without loss of generality, g is simply the randomness of sample. We call a cVDF with this property *publicly sampleable.*

Theorem 7.4 (Restatement of Theorem 1.2)**.** *Let $B, \ell \colon \mathbb{N} \to \mathbb{N}$ and $\epsilon \in (0,1)$. Assuming the existence of a publicly sampleable (B,ℓ,ϵ)-cVDF and a PRG for unpredictable seeds against the classes $\mathcal{C}_t = \{\mathcal{C}_{t,\lambda}\}_{\lambda\in\mathbb{N}}$ for all $t \leq B(\lambda)$ where $\mathcal{C}_{t,\lambda}$ is the class of circuits with size* $\mathrm{poly}(B(\lambda))$ *and depth $t \cdot \ell(\lambda)$. Then, there exists a (B,ℓ,Δ)-public randomness beacon for $\Delta(t) = \frac{\epsilon \cdot t}{1-\epsilon}$.*

Proof. Let $\mathsf{cVDF} = (\mathsf{cVDF.Gen}, \mathsf{cVDF.Sample}, \mathsf{cVDF.Eval}, \mathsf{cVDF.Verify})$. Our public randomness beacon $\mathsf{RB} = (\mathsf{RB.Gen}, \mathsf{RB.Init}, \mathsf{RB.Tick}, \mathsf{RB.Tock}, \mathsf{RB.Verify})$ is defined as follows:

- $\mathsf{RB.Gen}(1^\lambda)$ uses the same public parameters as $\mathsf{cVDF.Gen}(1^\lambda)$.
- $\mathsf{RB.Init}(1^\lambda, \mathsf{pp}, x_0)$ computes $\mathsf{state}_0 = \mathsf{cVDF.Sample}(1^\lambda, \mathsf{pp}; x_0)$ using x_0 as randomness.[21]
- $\mathsf{RB.Tick}(1^\lambda, \mathsf{pp}, \mathsf{state})$ computes the next state by outputting $\mathsf{cVDF.Eval}(1^\lambda, \mathsf{pp}, \mathsf{state})$.
- $\mathsf{RB.Tock}(1^\lambda, \mathsf{state})$ outputs a random beacon value computed by $G_\lambda(\mathsf{state})$.
- $\mathsf{RB.Verify}(1^\lambda, \mathsf{pp}, (x_0, t), (x, \mathsf{state}))$ checks that the current state as defined by cVDF verifies, i.e., $\mathsf{cVDF.Verify}(1^\lambda, \mathsf{pp}, (\mathsf{RB.Init}(1^\lambda, \mathsf{pp}, x_0), t), \mathsf{state}) = 1$, and that the beacon value is correct, i.e., $\mathsf{RB.Tock}(1^\lambda, \mathsf{pp}, \mathsf{state}) = x$. Output 1 if both checks pass, and output 0 otherwise.

Completeness follows since cVDF satisfies completeness from honest start. For the rest of the proof, let $\mathsf{state}_0 = \mathsf{RB.Init}(1^\lambda, \mathsf{pp}, x_0)$, which is the starting point of the cVDF. Honest evaluation follows from honest evaluation of cVDF. We now argue that soundness and indistinguishability hold.

Soundness. Suppose that there exists an algorithm $\mathcal{A} = \{\mathcal{A}_\lambda\}_{\lambda\in\mathbb{N}}$ where $\mathsf{size}(\mathcal{A}_\lambda) \in \mathrm{poly}(B(\lambda))$ and a polynomial p such that for infinitely many $\lambda \in \mathbb{N}$, it holds that

$$\Pr\left[\begin{array}{l} \mathsf{pp} \leftarrow \mathsf{RB.Gen}(1^\lambda) \\ x_0 \leftarrow \{0,1\}^\lambda \\ (x, \mathsf{state}, t) \leftarrow \mathcal{A}_\lambda(\mathsf{pp}, x_0) \end{array} : \begin{array}{l} x \neq \mathsf{RB.Tock}(1^\lambda, \mathsf{RB.Tick}^{(t)}(1^\lambda, \mathsf{pp}, \mathsf{state}_0)) \\ \mathsf{RB.Verify}(1^\lambda, \mathsf{pp}, (x_0, t), (x, \mathsf{state})) = 1 \end{array}\right] \geq \frac{1}{p(\lambda)}.$$

We consider two cases: either state is equal to $\mathsf{RB.Tick}^{(t)}(1^\lambda, \mathsf{pp}, \mathsf{state}_0)$ or not. If they are equal and $x \neq \mathsf{RB.Tock}(1^\lambda, \mathsf{state})$, this implies that $\mathsf{RB.Verify}$ will reject by definition. If the are not equal, then we know that $\mathsf{state} \neq \mathsf{cVDF.Eval}^{(t)}(1^\lambda, \mathsf{pp}, \mathsf{state}_0)$ but $\mathsf{cVDF.Verify}(1^\lambda, \mathsf{pp}, (\mathsf{state}_0, t), \mathsf{state})$ accepts since $\mathsf{RB.Verify}$ accepts. This implies that we can use $\mathcal{A}$ to break the soundness of cVDF with probability $1/p(\lambda)$, which is a contradiction.

Indistinguishability. Suppose there exists a $t \leq B(\lambda)$, an algorithm $\mathcal{A}_0 = \{\mathcal{A}_{0,\lambda}\}_{\lambda\in\mathbb{N}}$, and a polynomial p that causes the indistinguishability experiment to output 1 with probability more than $1/2 + 1/p(\lambda)$ for some $t' \in [t + \Delta(t), B(\lambda)]$. In other words, let $\mathcal{A}_1$ be the algorithm output by $\mathcal{A}_{0,\lambda}$. Then the following holds for this experiment:

$$\Pr[b = b' \wedge \mathsf{depth}(\mathcal{A}_1) \leq t \cdot \ell(\lambda)] \geq 1/2 + 1/p(\lambda).$$

We claim that this breaks that security of the PRG for unpredictable seeds G against the class of circuits $\mathcal{C}_t$ with size $B(\cdot)$ and depth at most $t \cdot \ell(\cdot)$.

Specifically, let $\mathsf{state}_{t'} = \mathsf{RB.Tick}^{(t')}(1^\lambda, \mathsf{pp}, \mathsf{state}_0) = \mathsf{cVDF.Eval}^{(t')}(1^\lambda, \mathsf{pp}, \mathsf{state}_0)$. We will show that by the iteratively sequential property of cVDF, we know that $\mathsf{state}_{t'}$ is unpredictable for adversaries with depth at most $(1-\epsilon) \cdot t' \cdot \ell(\lambda)$. Because $\Delta(t) = \frac{\epsilon \cdot t}{1-\epsilon}$, it follows that $\mathsf{state}_{t'}$ is also unpredictable for adversaries with depth at most $t \cdot \ell(\lambda)$. More formally, consider the family of distributions $S = \{S_\lambda\}_{\lambda\in\mathbb{N}}$ where S_λ samples $x_0 \leftarrow \{0,1\}^\lambda$ and outputs $(\mathsf{state}_{t'}, (x_0, \mathsf{state}_0))$ where

[21] If we need more bits of randomness, we can use a standard PRG with polynomial expansion.

$\mathsf{state}_0 = \mathsf{cVDF.Sample}(1^\lambda, \mathsf{pp}; x_0)$ and $\mathsf{state}_{t'} = \mathsf{cVDF.Eval}^{(t')}(1^\lambda, \mathsf{pp}, \mathsf{state}_0)$. For any class of circuits $\mathcal{B} = \{\mathcal{B}_\lambda\}_{\lambda\in\mathbb{N}}$ such that $\mathsf{depth}(\mathcal{B}_\lambda) \leq t \cdot \ell(\lambda)$ for all $\lambda \in \mathbb{N}$, it holds that

$$\Pr\left[\begin{array}{l}(\mathsf{state}_{t'}, (x_0, \mathsf{state}_0)) \leftarrow S_\lambda \\ \mathsf{state} \leftarrow \mathcal{B}_\lambda((x_0, \mathsf{state}_0))\end{array} : \mathsf{state} = \mathsf{state}_{t'}\right] \leq \mathsf{negl}(\lambda)$$

by the sequentiality property of cVDF. Note that this is where we used the fact that $\mathsf{cVDF.Sample}$ includes its randomness as its output (since $\mathcal{B}_\lambda$ expects to see the randomness x_0 of $\mathsf{cVDF.Sample}$ but the adversary in the iteratively sequential experiment of the cVDF only receives the output of $\mathsf{cVDF.Sample}$).

Thus, it suffices to show that using $\mathcal{A}_{0,\lambda}$, we can construct a circuit $\mathcal{D}_\lambda$ with depth at most $t \cdot \ell(\lambda)$ that distinguishes $(x_{t'}, (x_0, \mathsf{state}_0))$ from $(r, (x_0, \mathsf{state}_0))$ where $r \leftarrow \{0,1\}^\lambda$ and $x_{t'} = G_\lambda(\mathsf{state}_{t'})$ (where we assume the description of G_λ is public).

We construct an algorithm $\mathcal{D} = \{\mathcal{D}_\lambda\}_{\lambda\in\mathbb{N}}$ as follows. We sample $\mathsf{pp} \leftarrow \mathsf{Gen}(1^\lambda)$ and compute $\mathcal{A}_1 \leftarrow \mathcal{A}_{0,\lambda}(\mathsf{pp}, t')$. If $\mathsf{depth}(\mathcal{A}_1) \leq t \cdot \ell(\lambda)$, we set $\mathcal{D}_\lambda = \mathcal{A}_1$, and otherwise, we set $\mathcal{D}_\lambda$ to be a dummy circuit that always outputs $\bot$. The distinguishing probability of $\mathcal{D}_\lambda$ for all $\lambda \in \mathbb{N}$ is given by the following set of inequalities:

$$\begin{aligned}&|\Pr[\mathcal{D}_\lambda(x_{t'}, \mathsf{state}_0) = 1] - \Pr[\mathcal{D}_\lambda(r, \mathsf{state}_0) = 1]| \\ &= \big|\Pr[b' = 1 \wedge \mathsf{depth}(\mathcal{A}_1) \leq t \cdot \ell(\lambda) \mid b = 1] \\ &\quad - \Pr[b' = 1 \wedge \mathsf{depth}(\mathcal{A}_1) \leq t \cdot \ell(\lambda) \mid b = 0]\big| \\ &= \big|2\Pr[b' = b \wedge \mathsf{depth}(\mathcal{A}_1) \leq t \cdot \ell(\lambda)] - 1\big| \\ &\geq 2/p(\lambda),\end{aligned}$$

which contradicts the assumption that G is a secure against the class $\mathcal{C}_t$.

□

7.2 PPAD Hardness

Recall that TFNP, introduced by Megiddo and Papadimitriou [MP91], is the class of NP search problems with a guaranteed solution. Since its introduction, there has been a systematic study that clusters the problems in TFNP into subclasses based on the type of combinatorial argument establishing their totality. We consider two important subclasses of TFNP: PPAD and CLS. The class PPAD (for Polynomial Parity Argument on Directed graphs), introduced by Papadimitriou [Pap94], is important partly due to the fact that one of its complete problems is finding Nash equilibrium in bimatrix games [DGP09, CDT09]. The definition of PPAD is formally given by one of its complete problems called End-Of-Line (EOL). The class CLS (for Continuous Local Search), introduced by Daskalakis and Papadimitriou [DP11], is the smallest non-trivial class among the currently defined subclasses of TFNP (in particular, CLS ⊆ PPAD) and nevertheless it contains many important problems (see [HY17] for references). One problem that lies inside CLS (yet is not known to be complete) is called End-Of-Metered-Line (EOML) [HY17].

We next define recall the definitions of two promise problems (which are not total) which reduce to EOML: (1) The Sink-of-Verifiable-Line (SVL) problem that was introduced by Abbot, Kane and Valiant [AKV04] and further developed by [BPR15], and (2) the relaxed-Sink-of-Verifiable-Line (rSVL) problem that was introduced by Choudhuri et al. [CHK⁺19a]. The definitions are taken from [CHK⁺19a].

Definition 7.5. *A* Sink-of-Verifiable-Line *(SVL) instance is a tuple* (S, V, T, v_0) *where* $T \in [2^\lambda]$, $v_0 \in \{0,1\}^\lambda$, *and* $S\colon \{0,1\}^\lambda \to \{0,1\}^\lambda$, $V\colon \{0,1\}^\lambda \times [T] \to \{0,1\}$ *are circuits with the guarantee* *that for every $(v, i) \in \{0,1\}^\lambda \times [T]$ such that $v = S^i(v_0)$, it holds that $V(v, i) = 1$. The goal is to find a sink: a vertex $v \in \{0,1\}^\lambda$ such that $V(v, T) = 1$.*

The circuit S can be viewed as implementing the successor circuit of a directed graph over $\{0,1\}^\lambda$ that consists of a single line starting at v_0. Using the circuit V one can efficiently test whether a vertex $v^\star$ is at distance t from v_0. The goal is to find the node at distance T. In the relaxed-SVL problem, defined next, the setting is very similar except that there might be off-chain vertices and finding one that verifies also counts as a valid solution.

Definition 7.6. *A* RELAXED-SINK-OF-VERIFIABLE-LINE *(rSVL) instance is a tuple (S, V, T, v_0) where $T \in [2^\lambda]$, $v_0 \in \{0,1\}^\lambda$, and $S\colon \{0,1\}^\lambda \to \{0,1\}^\lambda$, $V\colon \{0,1\}^\lambda \times [T] \to \{0,1\}$ are circuits with the guarantee that for every $(v, i) \in \{0,1\}^\lambda \times [T]$ such that $v = S^{(i)}(v_0)$, it holds that $V(v, i) = 1$. The goal is to find one of the following:*

- **The sink:** *a vertex $v \in \{0,1\}^\lambda$ such that $V(v, T) = 1$, or*
- **A false positive:** *a pair $(v, i) \in \{0,1\}^\lambda \times \{0, \ldots, 2^\lambda\}$ such that $v \neq S^{(i)}(v_0)$ and $V(v, i) = 1$.*

Hardness of rSVL (resp. SVL) is defined in the standard way. Specifically, rSVL is (worst-case) hard if for every non-uniform polynomial-time (in the description size of the instance) algorithm there is an instance (S, V, T, v_0) of rSVL (resp. SVL) for which the algorithm fails. rSVL (resp. SVL) is average-case hard if there exists an efficient instance sampler such that every non-uniform polynomial-time algorithm cannot solve a random instance generated by the sampler. It is known that rSVL and SVL reduce to EOML [AKV04, BPR15, HY17, CHK$^+$19a]. Thus, worst-case (resp. average-case) hardness of rSVL or SVL translates into worst-case (resp. average-case) hardness of EOML which translates into worst-case (resp. average-case) hardness of CLS and PPAD.

In what follows we define a fine-grained version of (r)SVL hardness that measures the required length of the chain to get security for adversaries running in bounded time. Namely, given an adversary $\mathcal{A}$ that runs in time t, we consider the minimal required chain length to guarantee security. The smaller the gap, the tighter the security is. The following definition is stated for worst-case hardness but extends to average-case hardness naturally.

Definition 7.7 (Optimal hardness)**.** *Fix a function s. We say that rSVL is f-*hard *if for any non-uniform algorithm $\mathcal{A} = \{\mathcal{A}_\lambda\}_{\lambda\in\mathbb{N}}$ with $\mathsf{size}(\mathcal{A}_\lambda) \leq s(\lambda)$ for all $\lambda \in \mathbb{N}$, there exists an rSVL instance of length $f(s(\lambda))$ that $\mathcal{A}_\lambda$ cannot solve (for large enough λ). If f is linear then we say that rSVL is* optimally hard*.*

The recent work of Choudhuri et al. [CHK$^+$19a] gave an f-hard rSVL instance with $f(s) = s^c$ for some constant $c \geq 2$ (namely, there is a polynomial gap). Concretely, the length of their chain is $2^{n \cdot \log_2 d}$ for a constant $d \geq 4$ and one can find the label of the last node by solving #SAT on n variables (which can be done in size 2^n). We show that a continuous VDF, on the other hand, implies optimal (average-case) hardness of rSVL.

Theorem 7.8 (rSVL optimal hardness; restatement of Theorem 1.3)**.** *Let $B, \ell, \epsilon\colon \mathbb{N} \to \mathbb{N}$ be functions where $B(\lambda) \geq \lambda^c$ for a sufficiently large constant c. If there exists a (B, ℓ, ϵ)-cVDF, then rSVL is optimally average-case hard for algorithms with size $s(\lambda) \leq (1 - \epsilon) \cdot B(\lambda)$.*

As a corollary, by combining Theorems 6.2 and 7.8, we obtain Theorem 1.3: assuming that the Fiat-Shamir transformation for $\omega(1)$-round proof systems is sound and that the repeated squaring assumption holds, there exists an optimally (average-case) hard rSVL instance.

Corollary 7.9. *Let $D, B, \alpha\colon \mathbb{N} \to \mathbb{N}$ be functions satisfying $\lambda^c \leq B(\lambda) \leq 2^{\lambda^{1/3}}$ for a sufficiently large constant c, $\alpha(\lambda) \leq \lceil \log_\lambda(B(\lambda)) \rceil$, and $D(\lambda) \geq \lambda^{d'}$ for all $\lambda \in \mathbb{N}$ and for a specific constant d'. Suppose that the α-round strong FS assumption holds and the (D, B)-RSW assumption holds for a polynomial $\ell\colon \mathbb{N} \to \mathbb{N}$ and constant $\epsilon \in (0,1)$. Then, rSVL is optimally average-case hard for adversaries with size $s(\lambda) \leq (1-\epsilon) \cdot B(\lambda)$.*

Not only we get an optimally hard rSVL instance, but we also rely on a weaker variant of Fiat-Shamir than [CHK+19a]. Concretely, letting the chain length be $B = B(\lambda)$, our construction relies on Fiat-Shamir for $\log_\lambda B$-round protocols while the construction of [CHK+19a] needs Fiat-Shamir for $\log_2 B$-round protocols. In particular, when B is a polynomial function, we rely on a constant-round Fiat-Shamir transformation while they need Fiat-Shamir for super constant round protocol. A further comparison regarding optimal hardness is given below.

Proof of Theorem 7.8. Let $(\mathsf{cVDF.Gen}, \mathsf{cVDF.Sample}, \mathsf{cVDF.Eval}, \mathsf{cVDF.Verify})$ be a (B, ℓ, ϵ)-cVDF. Fix any function s satisfying $s(\lambda) \leq (1-\epsilon) \cdot B(\lambda)$ for all $\lambda \in \mathbb{N}$. Define the instance sampler $\mathcal{I}$ for average-case hard rSVL instances as follows. First, $\mathcal{I}(1^\lambda)$ samples $\mathsf{pp} \leftarrow \mathsf{Gen}(1^\lambda)$ and $v_0 \leftarrow \mathsf{cVDF.Sample}(\mathsf{pp})$ and sets $T = (1 + \frac{\epsilon}{1-\epsilon}) \cdot s(\lambda)$. Then, it sets S, V to be the circuits where

$$S(v) = \mathsf{cVDF.Eval}(1^\lambda, \mathsf{pp}, v) \text{ and } V(v, i) = \mathsf{cVDF.Verify}(1^\lambda, \mathsf{pp}, (v_0, i), v).$$

The resulting instance is (S, V, T, v_0). Note that for every $\lambda \in \mathbb{N}$, it holds that every (S, V, T, v_0) in the support of $\mathcal{I}(1^\lambda)$ is a valid instance of rSVL by the completeness property of cVDF.

To show optimal hardness of rSVL using this sampler, suppose for contradiction that there exists an algorithm $\mathcal{A}_\lambda = \{A_\lambda\}_{\lambda \in \mathbb{N}}$ and a polynomial p such that $\mathsf{size}(\mathcal{A}_\lambda) = s(\lambda)$ for all $\lambda \in \mathbb{N}$ and for infinitely many $\lambda \in \mathbb{N}$, $\mathcal{A}_\lambda$ can solve the rSVL instance sampled by $\mathcal{I}(1^\lambda)$ with probability $1/p(\lambda)$.[22] Fix $\lambda \in \mathbb{N}$ and an instance $(S, V, T, v_0) \leftarrow \mathcal{I}(1^\lambda)$. Whenever $\mathcal{A}_\lambda$ succeeds, it either finds a false positive or a sink for the rSVL instance.

In the first case, suppose $\mathcal{A}_\lambda$ finds a false positive (v, i), meaning that $v \neq S^{(i)}(v_0)$ and $V(v, i) = 1$. By definition of S and V, this implies that $v \neq \mathsf{cVDF.Eval}^{(i)}(1^\lambda, \mathsf{pp}, v_0)$ yet $\mathsf{cVDF.Verify}(1^\lambda, \mathsf{pp}, (v_0, i), v) = 1$. This directly implies an algorithm of size in $\mathrm{poly}(B(\lambda))$ which on input pp samples $v_0 \leftarrow \mathsf{Sample}(\mathsf{pp})$, runs $\mathcal{A}_\lambda$ on the corresponding rSVL instance to obtain (v, i), and outputs $((v_0, i), v)$ in contradiction with the soundness of the cVDF.

In this second case, suppose $\mathcal{A}_\lambda$ outputs a sink v. Then, $\mathcal{A}_\lambda$ can be used to construct an algorithm $\mathcal{B}_{1,\lambda}$ that breaks the sequentiality property of cVDF as follows. The algorithm $\mathcal{B}_{1,\lambda}(\mathsf{pp})$ for $\mathsf{pp} \leftarrow \mathsf{cVDF.Gen}(1^\lambda)$ outputs $\mathcal{B}_2$, where $\mathcal{B}_2$ is a circuit that has pp hardcoded. On input $v_0 \leftarrow \mathsf{cVDF.Sample}$, $\mathcal{B}_2$ forms the corresponding rSVL instance (S, V, T, v_0), runs $\mathcal{A}_\lambda$ to obtain v, and outputs (v, T). Whenever $\mathcal{A}_\lambda$ succeeds at outputting a sink v, it holds that $V(v, T) = 1$, which implies that $\mathsf{cVDF.Verify}(1^\lambda, \mathsf{pp}, (v_0, T), v)$ accepts. Let c' be the constant such that $\mathcal{I}(1^\lambda)$ runs in time $\lambda^{c'}$. Observe that

$$\mathsf{size}(\mathcal{B}_2) \leq \mathsf{size}(\mathcal{A}_\lambda) + \mathsf{size}(\mathcal{I}(1^\lambda)) \leq (1-\epsilon) \cdot B(\lambda) + \lambda^{c'} \leq (1-\epsilon') \cdot B(\lambda) \cdot \ell(\lambda)$$

for $\epsilon' \leq \epsilon - \frac{\lambda^{c'}}{B(\lambda) \cdot \ell(\lambda)} \in \epsilon - o(1)$ when $B(\lambda)$ is polynomially larger than $\lambda^{c'}$. It follows that $\mathcal{B}_{1,\lambda}$ breaks the sequentiality of cVDF for all sufficiently large λ with probability $1/p(\lambda)$, in contradiction. $\square$

Remark 3 (Hardness of EOL and Nash equilibrium)**.** *The reduction from rSVL (and SVL) to EOL is not tight (and thus so is the Nash equilibrium instance). Namely, even if we start with an optimally hard instance of rSVL, the resulting EOL instance has f-hardness for some small*

[22] In fact, it suffices to assume that $\mathsf{size}(\mathcal{A}_\lambda) \leq B(\lambda)$ and $\mathsf{depth}(\mathcal{A}_\lambda) \leq s(\lambda)$. See below.

polynomial $f(s) \in \text{poly}(s)$ (The non-tightness stems from the fact that the underlying pebbling argument introduces a blow-up in the number of nodes in the instance). Coming up with an optimally hard EOL instance is left as an open problem.

On depth-robustness moderate-hardness. The proof of Theorem 7.8 actually shows another flavor of hard instances—an rSVL instance that can be solved in fixed polynomial-time, yet no adversary with bounded depth (but any polynomial size) can solve. For this application we only require the Fiat-Shamir heuristic for constant-round proofs. For concreteness, consider the regime where T, the chain length, is a fixed polynomial (think of $T(\lambda) = \lambda^{100}$). The proof of Theorem 7.8 shows that *any polynomial-size* algorithm, as long as its depth is at most $(1 - \epsilon) \cdot T(\lambda)$, cannot solve the rSVL instance.

Theorem 7.10 (Moderately-hard depth-robustness)**.** *Let $B, \ell, \epsilon, T \colon \mathbb{N} \to \mathbb{N}$ be functions where $B(\lambda) > \lambda^{c'}$ for a sufficiently large constant c', such that there exists a (T, B, ℓ, ϵ)-cVDF (see Remark 2). Then, there exists an instance of rSVL that can be solved in time $T(\lambda)$ yet is hard for algorithms with depth at most $(1 - \epsilon) \cdot T(\lambda) \cdot \ell(\lambda)$ and size at most $B(\lambda)$.*

As stated in Theorem 1.4, we can apply the reduction from rSVL to EOL (and then to Nash equilibrium) and get a *depth-robust moderately-hard* instance—namely, there is a constant d such that for sufficiently large c, there is a distribution over EOL instances of size n that can be solved in time n^c but is hard for all polynomial-time algorithm of depth $n^{c/d}$.

In contrast, the rSVL instance of [CHK+19a] (and hence Nash equilibrium instance) does not give any guarantee for general polynomial-time algorithms, since breaking their instance would correspond to solving a #SAT instance with $O(\log(\lambda))$ variables, which is solvable in low-depth given parallel processors.

Acknowledgments. We thank Ian Miers for suggesting the name *continuous* VDFs and Eylon Yogev for discussions regarding our PPAD hardness results.

This work was supported in part by NSF Award SATC-1704788, NSF Award RI-1703846, AFOSR Award FA9550-18-1-0267, and by NSF Award DGE-1650441. This research is based upon work supported in part by the Office of the Director of National Intelligence (ODNI), Intelligence Advanced Research Projects Activity (IARPA), via 2019-19-020700006. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of ODNI, IARPA, or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright annotation therein.

References

[AKV04] Tim Abbot, Daniel Kane, and Paul Valiant. On algorithms for Nash equilibria, 2004. Accessed: 2019-09-18.

[Bar01] Boaz Barak. How to go beyond the black-box simulation barrier. In *42nd IEEE Symposium on Foundations of Computer Science, FOCS*, pages 106–115, 2001.

[BBBF18] Dan Boneh, Joseph Bonneau, Benedikt Bünz, and Ben Fisch. Verifiable delay functions. In *Advances in Cryptology - CRYPTO*, pages 757–788, 2018.

[BBF18] Dan Boneh, Benedikt Bünz, and Ben Fisch. A survey of two verifiable delay functions. *IACR Cryptology ePrint Archive*, 2018:712, 2018.

[Ben89] Charles H. Bennett. Time/space trade-offs for reversible computation. *SIAM J. Comput.*, 18(4):766–776, 1989.

[BGJ$^+$16] Nir Bitansky, Shafi Goldwasser, Abhishek Jain, Omer Paneth, Vinod Vaikuntanathan, and Brent Waters. Time-lock puzzles from randomized encodings. In *Innovations in Theoretical Computer Science, ITCS*, pages 345–356, 2016.

[BN00] Dan Boneh and Moni Naor. Timed commitments. In *Advances in Cryptology - CRYPTO*, pages 236–254, 2000.

[BPR15] Nir Bitansky, Omer Paneth, and Alon Rosen. On the cryptographic hardness of finding a Nash equilibrium. In Venkatesan Guruswami, editor, *IEEE 56th Symposium on Foundations of Computer Science, FOCS*, pages 1480–1498, 2015.

[CCH$^+$19] Ran Canetti, Yilei Chen, Justin Holmgren, Alex Lombardi, Guy N. Rothblum, Ron D. Rothblum, and Daniel Wichs. Fiat-Shamir: from practice to theory. In *51st ACM SIGACT Symposium on Theory of Computing, STOC*, pages 1082–1090, 2019.

[CCRR18] Ran Canetti, Yilei Chen, Leonid Reyzin, and Ron D. Rothblum. Fiat-Shamir and correlation intractability from strong kdm-secure encryption. In *Advances in Cryptology - EUROCRYPT*, pages 91–122, 2018.

[CDT09] Xi Chen, Xiaotie Deng, and Shang-Hua Teng. Settling the complexity of computing two-player Nash equilibria. *J. ACM*, 56(3):14:1–14:57, 2009.

[Chi] Chia network. https://chia.net/. Accessed: 2019-05-17.

[CHK$^+$19a] Arka Rai Choudhuri, Pavel Hubáček, Chethan Kamath, Krzysztof Pietrzak, Alon Rosen, and Guy N. Rothblum. Finding a Nash equilibrium is no easier than breaking Fiat-Shamir. In *51st ACM SIGACT Symposium on Theory of Computing, STOC*, pages 1103–1114, 2019.

[CHK$^+$19b] Arka Rai Choudhuri, Pavel Hubáček, Chethan Kamath, Krzysztof Pietrzak, Alon Rosen, and Guy N. Rothblum. PPAD-hardness via iterated squaring modulo a composite. *IACR Cryptology ePrint Archive*, 2019:667, 2019.

[CLP13] Kai-Min Chung, Huijia Lin, and Rafael Pass. Constant-round concurrent zero knowledge from P-certificates. In *54th IEEE Symposium on Foundations of Computer Science, FOCS*, pages 50–59, 2013.

[CLSY93] Jin-yi Cai, Richard J. Lipton, Robert Sedgewick, and Andrew Chi-Chih Yao. Towards uncheatable benchmarks. In *8th Structure in Complexity Theory Conference*, pages 2–11. IEEE Computer Society, 1993.

[CP18] Bram Cohen and Krzysztof Pietrzak. Simple proofs of sequential work. In *Advances in Cryptology - EUROCRYPT*, pages 451–467, 2018.

[DGMV19] Nico Döttling, Sanjam Garg, Giulio Malavolta, and Prashant Nalini Vasudevan. Tight verifiable delay functions. *IACR Cryptology ePrint Archive*, 2019:659, 2019.

[DGP09] Constantinos Daskalakis, Paul W. Goldberg, and Christos H. Papadimitriou. The complexity of computing a Nash equilibrium. *Commun. ACM*, 52(2):89–97, 2009.

[DLM19] Nico Döttling, Russell W. F. Lai, and Giulio Malavolta. Incremental proofs of sequential work. In *Advances in Cryptology - EUROCRYPT*, pages 292–323, 2019.

[DN92] Cynthia Dwork and Moni Naor. Pricing via processing or combatting junk mail. In *Advances in Cryptology - CRYPTO*, pages 139–147, 1992.

[DP11] Constantinos Daskalakis and Christos H. Papadimitriou. Continuous local search. In *22nd ACM-SIAM Symposium on Discrete Algorithms, SODA*, pages 790–804, 2011.

[Eth] Ethereum foundation. https://www.ethereum.org/. Accessed: 2019-05-17.

[FMPS19] Luca De Feo, Simon Masson, Christophe Petit, and Antonio Sanso. Verifiable delay functions from supersingular isogenies and pairings. *IACR Cryptology ePrint Archive*, 2019:166, 2019.

[FS86] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In *Advances in Cryptology - CRYPTO*, pages 186–194, 1986.

[GK90] Oded Goldreich and Hugo Krawczyk. On the composition of zero-knowledge proof systems. In *International Colloquium on Automata, Languages, and Programming, ICALP*, pages 268–282, 1990.

[GK03] Shafi Goldwasser and Yael Tauman Kalai. On the (in)security of the Fiat-Shamir paradigm. In *44th IEEE Symposium on Foundations of Computer Science, FOCS*, pages 102–113, 2003.

[GPS16] Sanjam Garg, Omkant Pandey, and Akshayaram Srinivasan. Revisiting the cryptographic hardness of finding a Nash equilibrium. In *Advances in Cryptology - CRYPTO*, pages 579–604, 2016.

[HY17] Pavel Hubáček and Eylon Yogev. Hardness of continuous local search: Query complexity and cryptographic lower bounds. In *28th ACM-SIAM Symposium on Discrete Algorithms, SODA*, pages 1352–1371, 2017.

[JM11] Yves Igor Jerschow and Martin Mauve. Non-parallelizable and non-interactive client puzzles from modular square roots. In *6th International Conference on Availability, Reliability and Security, ARES1*, pages 135–142. IEEE Computer Society, 2011.

[Kal00] Burt Kaliski. Pkcs #5: Password-based cryptography specification version 2.0, 2000.

[KNT18] Fuyuki Kitagawa, Ryo Nishimaki, and Keisuke Tanaka. Obfustopia built on secret-key functional encryption. In *Advances in Cryptology - EUROCRYPT*, pages 603–648, 2018.

[KS17] Ilan Komargodski and Gil Segev. From Minicrypt to Obfustopia via private-key functional encryption. In *Advances in Cryptology - EUROCRYPT*, pages 122–151, 2017.

[LFKN92] Carsten Lund, Lance Fortnow, Howard J. Karloff, and Noam Nisan. Algebraic methods for interactive proof systems. *J. ACM*, 39(4):859–868, 1992.

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

[LW17] Arjen K. Lenstra and Benjamin Wesolowski. Trustworthy public randomness with sloth, unicorn, and trx. *IJACT*, 3(4):330–343, 2017.

[MMV13] Mohammad Mahmoody, Tal Moran, and Salil P. Vadhan. Publicly verifiable proofs of sequential work. In *Innovations in Theoretical Computer Science, ITCS*, pages 373–388, 2013.

[MP91] Nimrod Megiddo and Christos H. Papadimitriou. On total functions, existence theorems and computational complexity. *Theor. Comput. Sci.*, 81(2):317–324, 1991.

[MSW19] Mohammad Mahmoody, Caleb Smith, and David J. Wu. A note on the (im)possibility of verifiable delay functions in the random oracle model. *ePrint*, page 663, 2019.

[Pap94] Christos H. Papadimitriou. On the complexity of the parity argument and other inefficient proofs of existence. *J. Comput. Syst. Sci.*, 48(3):498–532, 1994.

[Pie19] Krzysztof Pietrzak. Simple verifiable delay functions. In *10th Innovations in Theoretical Computer Science Conference, ITCS*, pages 60:1–60:15, 2019.

[Pro] Protocol labs. https://protocol.ai/. Accessed: 2019-05-17.

[Rab79] Michael O. Rabin. Digitalized signatures and public key functions as intractable as factoring. Technical report, TR-212, LCS, MIT, Cambridge, MA, 1979.

[Rab83] Michael O. Rabin. Transaction protection by beacons. *J. Comput. Syst. Sci.*, 27(2):256–267, 1983.

[RRR16] Omer Reingold, Guy N. Rothblum, and Ron D. Rothblum. Constant-round interactive proofs for delegating computation. In *48th ACM SIGACT Symposium on Theory of Computing, STOC*, pages 49–62, 2016.

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

[Val08] Paul Valiant. Incrementally verifiable computation or proofs of knowledge imply time/space efficiency. In *Theory of Cryptography, TCC*, pages 1–18, 2008.

[VDF] VDF research effort. https://vdfresearch.org/. Accessed: 2019-05-17.

[Wes19] Benjamin Wesolowski. Efficient verifiable delay functions. In *Advances in Cryptology - EUROCRYPT*, pages 379–407, 2019.

[Zha16] Mark Zhandry. The magic of ELFs. In *Advances in Cryptology - CRYPTO*, pages 479–508, 2016.

A Number Theory Facts

For $N \in \mathbb{N}$ and any $x \in \mathbb{Z}_N$, we use the notation $|x|_N$ to denote $\min\{x, N-x\}$. Next, we state three standard useful facts. Full proofs are given for completeness.

Fact A.1. *Let $N \in \mathrm{Supp}\left(\mathsf{RSW.Gen}(1^\lambda)\right)$. Then, for $\mu \in \mathbb{Z}_N^\star$, it holds that $\langle\mu\rangle = \mathsf{QR}_N$ if and only if there exists an $x \in \mathbb{Z}_N^\star$ such that $\mu = x^2$ and $\gcd(x \pm 1, N) = 1$.*

Proof. Let the representation of x be (a, b), i.e., $x = a \bmod p$ and $x = b \bmod q$, and thus $\mu = x^2$ has representation (a^2, b^2). First, suppose that $\langle \mu \rangle = \mathsf{QR}_N$. This implies that $a^2, b^2 \neq 1$, so $a, b \notin \{\pm 1\}$. It follows that $\gcd(x \pm 1, N) = 1$. For the other direction, suppose now that $\gcd(x \pm 1, N) = 1$. Since p and q do not divide $(x \pm 1)$, it holds that $a, b \notin \{\pm 1\}$. This implies that $a^2, b^2 \neq 1$. Because $\mathbb{Z}_p^\star$ has order $2p'$ and a^2 is a quadratic residue in $\mathbb{Z}_p^\star$, a^2 must have order p'. Similarly, b^2 must have order q'. This implies that the order of μ is $p' \cdot q' / \gcd(p', q') = p' \cdot q' = |\mathsf{QR}_N|$, so $\langle \mu \rangle = \mathsf{QR}_N$. $\square$

Fact A.2 ([Rab79])**.** *There exists a polynomial time algorithm $\mathcal{A}$ such that for any $\lambda \in \mathbb{N}$, N in the support of $\mathsf{RSW.Gen}(1^\lambda)$, and $\mu, x, x' \in \mathbb{Z}_N$, if $\mu = x^2 = x'^2$ and $x' \notin \{x, -x\}$, then $\mathcal{A}(1^\lambda, N, (\mu, x, x'))$ outputs (p, q) such that $N = p \cdot q$.*

Proof. $\mathcal{A}(1^\lambda, N, (\mu, x, x'))$ computes $p = \gcd(x + x', N)$ and $q = N/p$, and outputs (p, q). To see that this is correct, let (a, b) be the representation of x, i.e., x is equal to $a \bmod p$ and $b \bmod q$. If $(x')^2 = x^2$, then x' has representation in $\{(\pm a, \pm b)\}$. However, we assume that $x' \notin \{x, -x\}$, so x' must have representation $(a, -b)$ or $(-a, b)$. In the first case, $x + x' = 0 \bmod q$ and in the second case $x + x' = 0 \bmod p$, as needed. $\square$

Fact A.3. *Let $N \in \mathrm{Supp}\big(\mathsf{RSW.Gen}(1^\lambda)\big)$ and let $\langle x \rangle = \mathsf{QR}_N$. Then, for any $i \in \mathbb{N}$, it holds that $\langle x^{2^i} \rangle = \mathsf{QR}_N$.*

Proof. Let $\mu = x^{2^i} \bmod N$. Since $\langle x \rangle = \mathsf{QR}_N$, then μ generates QR_N whenever its exponent with respect to x is not divisible by p' or q'. This trivially holds since 2^i is even and p', q' are odd. $\square$