A Survey of Two Verifiable Delay Functions

Dan Boneh Benedikt Bünz Ben Fisch

Abstract

A verifiable delay function (VDF) is an important tool used for adding delay in decentralized applications. This short note briefly surveys and compares two recent beautiful Verifiable Delay Functions (VDFs), one due to Pietrzak and the other due to Wesolowski. We also provide a new computational proof of security for one of them, and compare the complexity assumptions needed for both schemes.

1 What is a Verifiable Delay Function?

A verifiable delay function (VDF) [11, 2] is a function $f : \mathcal{X} \to \mathcal{Y}$ that takes a prescribed time to compute, even on a parallel computer. However once computed, the output can be quickly verified by anyone. Moreover, every input $x \in \mathcal{X}$ must have a unique valid output $y \in \mathcal{Y}$.

In more detail, a VDF that implements a function $\mathcal{X} \to \mathcal{Y}$ is a tuple of three algorithms:

- $\mathit{Setup}(\lambda, T) \to \mathit{pp}$ is a randomized algorithm that takes a security parameter λ and a time bound T, and outputs public parameters pp,
- $\mathit{Eval}(\mathit{pp}, x) \to (y, \pi)$ takes an input $x \in \mathcal{X}$ and outputs a $y \in \mathcal{Y}$ and a proof π.
- $\mathit{Verify}(\mathit{pp}, x, y, \pi) \to \{\mathit{accept}, \mathit{reject}\}$ outputs *accept* if y is the correct evaluation of the VDF on input x.

If $(y, \pi) \leftarrow F(\mathit{pp}, x)$ then $\mathit{Verify}(\mathit{pp}, x, y, \pi) = \mathit{accept}$, for all $x \in \mathcal{X}$ and pp output by $\mathit{Setup}(\lambda, T)$. A VDF must satisfy three properties. We state these properties informally and refer to [2] for a complete definition:

- **ϵ-evaluation time:** algorithm $\mathit{Eval}(\mathit{pp}, x)$ runs in time at most $(1 + \epsilon)T$, for all $x \in \mathcal{X}$ and all pp output by $\mathit{Setup}(\lambda, T)$. We will explain how to measure run time in the next section.
- **Sequentiality:** a parallel algorithm $\mathcal{A}$, using at most $\mathit{poly}(\lambda)$ processors, that runs in time less than T cannot compute the function. Specifically, for a random $x \in \mathcal{X}$ and pp output by $\mathit{Setup}(\lambda, T)$, if $(y, \pi) \leftarrow \mathit{Eval}(\mathit{pp}, x)$ then $\Pr\left[\mathcal{A}(\mathit{pp}, x) = y\right]$ is negligible.
- **Uniqueness:** for an input $x \in \mathcal{X}$, exactly one $y \in \mathcal{Y}$ will be accepted by *Verify*. Specifically, let $\mathcal{A}$ be an efficient algorithm that given pp as input, outputs (x, y, π) such that $\mathit{Verify}(\mathit{pp}, x, y, \pi) = \mathit{accept}$. Then $Pr[\mathit{Eval}(\mathit{pp}, x) \neq y]$ is negligible.

VDFs have many applications. They are useful for constructing a verifiable randomness beacon, and they provide a "proof of elapsed time" for certain blockchain designs [6]. We refer to [2, Sec. 2] for a survey of their applications.

2 Two Verifiable Delay Functions

A VDF is based on a computational task that cannot be sped up by parallelism. Exponentiation in a group of unknown order is believed to have this property, and was previously used by Rivest, Shamir, and Wagner [13] to construct a time-lock puzzle. The two recent VDF proposals, one due to Pietrzak [12] and the other due to Wesolowski [14], similarly make use of the serial nature of this task.

Both VDF constructions operate as follows:

- The setup algorithm $\mathit{Setup}(\lambda, T)$ outputs two objects:
  - A finite abelian group $\mathbb{G}$ of unknown order – we will discuss concrete groups in Section 6;
  - An efficiently computable hash function $H : \mathcal{X} \to \mathbb{G}$ that we model as a random oracle.

  We set the public parameters pp to be $pp := (\mathbb{G}, H, T)$.
- The evaluation algorithm $\mathit{Eval}(pp, x)$ is defined as follows:
  - compute $\boxed{y \leftarrow H(x)^{(2^T)} \in \mathbb{G}}$ by computing T squarings in $\mathbb{G}$ starting with $H(x)$,
  - compute the proof π as described later,
  - output (y, π).

We measure running time in terms of the number of group operations in $\mathbb{G}$ needed to compute the function. It is believed that computing y requires T sequential squarings in $\mathbb{G}$ even on a parallel computer with $\mathit{poly}(\lambda)$ processors, as required for sequentiality. As we will see, computing the proof π increases the running time to $(1 + \epsilon)T$, as needed for ϵ-evaluation time. In practice one might set $T = 2^{30}$ and $\epsilon = 0.01$.

The remaining question is how a public verifier $\mathit{Verify}(pp, x, y, \pi)$ can quickly check that the output y is correct, namely that $y = H(x)^{(2^T)}$. This is where the proposal of Pietrzak and the proposal of Wesolowski differ. They give two different public-coin succinct arguments for proving that the output y is correct. Thanks to the public-coin nature of these arguments they can be made non-interactive using the Fiat-Shamir Heuristic [3, Sec. 19.6.1].

Proving correctness of the output y. To state the problem more abstractly, let us use the following notation:

- let $g := H(x) \in \mathbb{G}$ be the base element given as input to the VDF evaluator;
- let $h := y \in \mathbb{G}$ be the purported output of the VDF, namely $h = g^{(2^T)}$;
- $T > 0$ is a publicly known quantity.

The VDF evaluator needs to produce a proof that a given tuple $(\mathbb{G}, g, h, T)$ satisfies $h = g^{(2^T)}$ in $\mathbb{G}$. More precisely, we need a succinct public-coin interactive argument for the language

$$\mathcal{L}_{\text{EXP}} := \Big\{ (\mathbb{G}, g, h, T) \ : \ h = g^{(2^T)} \text{ in } \mathbb{G} \Big\}. \tag{1}$$

2.1 Wesolowski's succinct argument

Wesolowski [14] presents the following succinct public-coin interactive argument for the language $\mathcal{L}_{\text{EXP}}$ defined in (1). Specifically, given a tuple $(\mathbb{G}, g, h, T)$ as input, the prover and verifier engage in the following protocol to prove that $h = g^{(2^T)}$ in $\mathbb{G}$. We let $\mathit{Primes}(\lambda)$ be the set containing the first 2^λ primes, namely $2, 3, 5, 7$, etc.

0. The verifier checks that $g, h \in \mathbb{G}$ and outputs *reject* if not,
1. The verifier sends to the prover a random prime ℓ sampled uniformly from $\mathit{Primes}(\lambda)$,
2. The prover computes $q, r \in \mathbb{Z}$ such that $2^T = q\ell + r$ with $0 \le r < \ell$, and sends $\pi \leftarrow g^q$ to the verifier.
3. The verifier computes $r \leftarrow 2^T \bmod \ell$ and outputs *accept* if $\pi \in \mathbb{G}$ and $h = \pi^\ell g^r$ in $\mathbb{G}$

We note that the protocol works equally well when the exponent 2^T is an arbitrary integer e, not necessarily a power of two. The verifier just needs a quick way to compute $r := e \bmod \ell$.

Non-interactive variant. When the protocol is made non-interactive using the Fiat-Shamir heuristic it is necessary to increase the size of the random prime ℓ generated in step (1), otherwise the protocol is insecure, as explained in Section 3.3. In the non-interactive variant obtained by applying Fiat-Shamir, the prover first generates ℓ by using a hash function that maps the input $(\mathbb{G}, g, h, T)$ to an element of $\mathit{Primes}(2\lambda)$, the set containing the first $2^{2\lambda}$ primes. The analysis will assume that this hash function is a random oracle. The prover computes $\pi \leftarrow g^q$ as in step (2) above, and outputs this $\pi \in \mathbb{G}$ as the proof. The verifier computes ℓ the same way as the prover and decides to accept or reject as in step (3) above. Overall, the proof π is a single element in $\mathbb{G}$.

Verifier efficiency. The verifier needs to compute $r \leftarrow 2^T \bmod \ell$, which only takes $\log_2 T$ multiplications in $\mathbb{Z}/\ell$. Beyond that, the verifier only computes two small exponentiations in $\mathbb{G}$.

Prover efficiency. The prover needs to compute $\pi = g^q \in \mathbb{G}$ where $q = \lfloor 2^T/\ell \rfloor$. Because T is large, we cannot write out q as an explicit integer exponent. Nevertheless, we can compute $\pi = g^q$ in at most $2T$ group operations and constant space using the long-division algorithm, where the quotient is computed in the exponent base g.

$\pi \leftarrow 1 \in \mathbb{G}, \quad r \leftarrow 1 \in \mathbb{Z}$

repeat T times:

  $b \leftarrow \lfloor 2r/\ell \rfloor \in \{0, 1\}$ and $r \leftarrow (2r \bmod \ell) \in \{0, \ldots, \ell - 1\}$

  $\pi \leftarrow \pi^2 g^b \in \mathbb{G}$

output π // *this* π *equals* g^q

The running time can be reduced to about T group operations using a windowing method where we process k bits of 2^T per iteration, for some parameter $k \ge 1$, say $k = 5$.

In Appendix A we describe an extension that lets us speed up the computation of g^q by a factor of s using s processors. Hence, the VDF output and the proof π can be computed in total time

0. The verifier checks that $g, h \in \mathbb{G}$ and outputs *reject* if not,

1. If $T = 1$ the verifier checks that $h = g^2$ in $\mathbb{G}$, outputs *accept* or *reject*, and stops.

2. If $T > 1$ the prover and verifier do:

   (a) The prover computes $v \leftarrow g^{(2^{T/2})} \in \mathbb{G}$ and sends v to the verifier. The verifier checks that $v \in \mathbb{G}$ and outputs *reject* and stops, if not.

   Next, the prover needs to convince the verifier that $h = v^{(2^{T/2})}$ and $v = g^{(2^{T/2})}$, which proves that $h = g^{(2^T)}$. Because the same exponent is used in both equalities, they can be verified *simultaneously* by checking a random linear combination, namely that

   $$v^r h = (g^r v)^{(2^{T/2})} \quad \text{for a random } r \text{ in } \{1, \ldots, 2^\lambda\}.$$

   The verifier and prover do so as follows.

   (b) The verifier sends to the prover a random r in $\{1, \ldots, 2^\lambda\}$.

   (c) Both the prover and verifier compute $g_1 \leftarrow g^r v$ and $h_1 \leftarrow v^r h$ in $\mathbb{G}$.

   (d) The prover and verifier recursively engage in an interactive proof that $(\mathbb{G}, g_1, h_1, T/2) \in \mathcal{L}_{\text{EXP}}$, namely that $h_1 = g_1^{(2^{T/2})}$ in $\mathbb{G}$.

Figure 1: Pietrzak's succinct argument for $(\mathbb{G}, g, h, T) \in \mathcal{L}_{\text{EXP}}$

approximately $(1 + \frac{1}{s})T$ with s processors and space s. Wesolowski [14] shows that with space 2^k one can further speed-up the computation by a factor of k.

2.2 Pietrzak's succinct argument

Pietrzak [12] presents a different succinct public-coin interactive argument for the language $\mathcal{L}_{\text{EXP}}$ defined in (1). Specifically, given a tuple $(\mathbb{G}, g, h, T)$ as input, the prover and verifier engage in a recursive protocol shown in Figure 1 to prove that $h = g^{(2^T)}$ in $\mathbb{G}$. For simplicity, we assume that T is a power of two in which case the protocol takes $\log_2 T$ rounds. The protocol can be adjusted to handle arbitrary T, including a T that is not a power of two [12].

Non-interactive variant. When the protocol is made non-interactive using Fiat-Shamir the prover generates the challenge r in every level of the recursion by hashing the quantities $(\mathbb{G}, g, h, T, v)$ at that level, and appends v to the overall proof π. Hence, the overall proof π contains $\log_2 T$ elements in $\mathbb{G}$.

Verifier efficiency. At every level of the recursion the verifier does two small exponentiations in $\mathbb{G}$ to compute g_1 and h_1 for the next level. Hence, verifying the proof takes about $2 \log_2 T$ small exponentiations in $\mathbb{G}$.

Prover efficiency. The prover needs to compute the quantity v at every level of the recursion. We let v_1, r_1 be the values of v and r at the top level of the recursion, v_2, r_2 the values at the next level, and so on. Unwinding the recursion shows that these quantities are:

$$v_1 = g^{(2^{T/2})}$$
$$v_2 = g_1^{(2^{T/4})} = (g^{r_1} v_1)^{(2^{T/4})} = \left(g^{(2^{T/4})}\right)^{r_1} g^{(2^{3T/4})}$$
$$v_3 = g_2^{(2^{T/8})} = (g_1^{r_2} v_2)^{(2^{T/8})} = (g^{r_1 r_2} v_1^{r_2} v_2)^{(2^{T/8})} = \left(g^{(2^{T/8})}\right)^{r_1 r_2} \left(g^{(2^{3T/8})}\right)^{r_1} \left(g^{(2^{5T/8})}\right)^{r_2} g^{(2^{7T/8})}$$
$$v_4 = g_3^{(2^{T/16})} = \text{a power product of eight elements } g^{(2^{T/16})},\ g^{(2^{3T/16})},\ g^{(2^{5T/16})},\ \ldots,\ g^{(2^{15T/16})}.$$

The pattern that emerges suggests an efficient way to construct the proof π. When the VDF evaluator first computes the VDF output $h = g^{(2^T)}$ it stores 2^d group elements $g^{(2^{(i \cdot T/2^d)})}$ for $i = 0, \ldots, 2^d - 1$ as they are encountered along the way. Later, as it constructs the proof π, these 2^d stored values let it compute the group elements $v_1, \ldots, v_d$ needed for the proof using a total of about 2^d small exponentiations in $\mathbb{G}$. The prover computes the remaining elements $v_{d+1}, v_{d+2}, \ldots, v_{\log T}$ from scratch by raising $g_{d+1}, g_{d+2}, \ldots, g_{\log T}$ to the appropriate exponents. This step takes a total of $T/2^d$ multiplications in $\mathbb{G}$. Hence, the total time to compute the proof is about $2^d + T/2^d$, which suggests that $d = \frac{1}{2}\log_2 T$ is optimal. Hence, the VDF output and the proof π can be computed in total time approximately $(1 + \frac{2}{\sqrt{T}})T$.

3 Security assumptions needed to prove soundness

To analyze security of these interactive arguments for $\mathcal{L}_{\text{EXP}}$ we rely on two complexity assumptions: the low order assumption and the adaptive root assumption. We prove security of Pietrzak's argument in groups where the low order assumption holds. We prove security of Wesolowski's argument in groups where the adaptive root assumption holds. We discuss the relation between these assumptions in Section 4.

Notation. In what follows we use $x \xleftarrow{R} S$ to denote an independent uniform random variable over the set S, and use $y \xleftarrow{R} \mathcal{A}(x)$ to denote the random variable that is the output of a randomized algorithm $\mathcal{A}$ on input x. We say that a function $f : \mathbb{Z} \to \mathbb{R}$ is a negligible function of λ if $|f(\lambda)| = o(1/\lambda^d)$ for all $d > 0$.

3.1 Security of Pietrzak's succinct argument

Let $GGen(\lambda)$ be a randomized algorithm that outputs the description of a group $\mathbb{G}$ of unknown order. The low order assumption says that it is hard to find a low order element in a random group output by $GGen(\lambda)$.

Definition 1. *We say that the **low order assumption** holds for GGen if there is no efficient algorithm $\mathcal{A}$ that takes as input the description of a group $\mathbb{G}$ generated by $GGen(\lambda)$, and outputs a pair (μ, d) where $\mu^d = 1$ for $1 \neq \mu \in \mathbb{G}$ and $1 < d < 2^\lambda$. We say that $\mathcal{A}$ outputs a low order element μ in $\mathbb{G}$. More precisely, the advantage*

$$\mathsf{LOadv}_{\mathcal{A},GGen}(\lambda) := \Pr\left[\ \mu^d = 1,\ \ 1 \neq \mu \in \mathbb{G},\ \ 1 < d < 2^\lambda,\ \ : \begin{array}{c} \mathbb{G} \xleftarrow{R} GGen(\lambda), \\ (\mu, d) \xleftarrow{R} \mathcal{A}(\mathbb{G}) \end{array} \right]$$

is a negligible function of λ.

The following theorem proves soundness of Pietrzak's succinct argument using the low order assumption. The proof is given in Section 5.

Theorem 1. *Suppose the low order assumption holds for GGen. Then Pietrzak's succinct argument has negligible soundness error.*

> *Concretely, let $\mathcal{A}$ be an algorithm that succeeds with probability ϵ in the following task: $\mathcal{A}$ takes a description of $\mathbb{G} \xleftarrow{R} GGen(\lambda)$ as input, outputs a tuple $(\mathbb{G}, g, h, T) \notin \mathcal{L}_{EXP}$ where $1 \leq T < 2^t$ is a power of two, and convinces the verifier to incorrectly accept this tuple. Then there is an algorithm $\mathcal{B}$, whose running time is about twice that of $\mathcal{A}$, that breaks the low order assumption for GGen with advantage at least $\epsilon' = (\epsilon^2/t) - (\epsilon/2^\lambda)$. Hence if ϵ' is negligible then so must be ϵ.*

Necessity of the low order assumption. The low order assumption is necessary for soundness of the protocol – if the assumption does not hold for *GGen* then the protocol becomes insecure. To see why, let $\mathbb{G} \xleftarrow{R} GGen(\lambda)$ and let $\mu \in \mathbb{G}$ be a known element of order $d > 1$ (i.e., low order is broken). Let $(\mathbb{G}, g, h, T) \in \mathcal{L}_{\text{EXP}}$. Then the tuple $(\mathbb{G},\ g,\ h\mu,\ T) \notin \mathcal{L}_{\text{EXP}}$ will be incorrectly accepted by the verifier with probability $1/d$. To do so the prover sends $v \leftarrow g^{(2^{T/2})}\mu \in \mathbb{G}$ which causes the tuple $(\mathbb{G},\ g,\ h\mu,\ T)$ to be incorrectly accepted whenever the verifier chooses an r satisfying $r + 1 \equiv 2^{T/2} \pmod{d}$. This happens with probability $1/d$, which is non-negligible when d is small. Note that when $r + 1 \equiv 2^{T/2} \pmod{d}$ we have that $(\mathbb{G},\ g^r v,\ v^r(h\mu),\ T/2) \in \mathcal{L}_{\text{EXP}}$, which is why the tuple $(\mathbb{G},\ g,\ h\mu,\ T)$ is incorrectly accepted.

Note that if the group $\mathbb{G}$ contains no low order elements other than the identity, then the low order assumption holds unconditionally, as does soundness of Pietrzak's succinct argument. We discuss this further in Section 6.

3.2 Security of Wesolowski's succinct argument

For the next assumption recall that $Primes(\lambda)$ denotes the set of first 2^λ positive integer primes.

Definition 2. *We say that the **adaptive root assumption** holds for GGen if there is no efficient adversary $(\mathcal{A}_1, \mathcal{A}_2)$ that succeeds in the following task. First, $\mathcal{A}_1$ outputs an element $w \in \mathbb{G}$ and some state. Then, a random prime ℓ in $Primes(\lambda)$ is chosen and $\mathcal{A}_2(\ell, \mathsf{state})$ outputs $w^{1/\ell} \in \mathbb{G}$. More precisely, the advantage*

$$\text{ARadv}_{(\mathcal{A}_1,\mathcal{A}_2),GGen}(\lambda) := \Pr\left[u^\ell = w \neq 1 \ : \begin{array}{l} \mathbb{G} \xleftarrow{R} GGen(\lambda), \\ (w, \mathsf{state}) \xleftarrow{R} \mathcal{A}_1(\mathbb{G}), \\ \ell \xleftarrow{R} Primes(\lambda), \\ u \xleftarrow{R} \mathcal{A}_2(\ell, \mathsf{state}) \end{array} \right]$$

is a negligible function of λ.

The advantage is always at least $1/|Primes(\lambda)|$. Indeed, if the adversary $(\mathcal{A}_1, \mathcal{A}_2)$ correctly guesses $\ell \in Primes(\lambda)$ ahead of time, then $\mathcal{A}_1$ would output $w \leftarrow u^\ell$, for some $u \in \mathbb{G}$, and $\mathcal{A}_2$ would output this u. This is why we must choose the set $Primes(\lambda)$ to be sufficiently large. The reason we cannot choose ℓ uniformly in some interval, but must choose it from $Primes(\lambda)$, is because a random ℓ in $\{1, \ldots, 2^\lambda\}$ has a reasonable chance of being a smooth integer. The adversary can then

win by having $\mathcal{A}_1$ output $w \leftarrow u^B$ where B is a product of small prime powers up to some bound k, and having $\mathcal{A}_2$ output $u^{B/\ell}$. This works whenever ℓ is a k-smooth integer. Choosing ℓ as a prime number eliminates this attack.

The following theorem proves soundness of Wesolowski's succinct argument using the low order assumption. The proof is given in Section 5.

Theorem 2 (Wesolowski [14])**.** *Suppose the adaptive root assumption holds for GGen. Then Wesolowski's succinct argument has negligible soundness error.*

> *Concretely, let $\mathcal{A}$ be an algorithm that succeeds with probability ϵ in the following task: $\mathcal{A}$ takes $\mathbb{G} \xleftarrow{R} GGen(\lambda)$ as input, outputs a tuple $(\mathbb{G}, g, h, T) \notin \mathcal{L}_{EXP}$, and convinces the verifier to incorrectly accept this tuple. Then there is an adversary $(\mathcal{B}_1, \mathcal{B}_2)$ whose combined running time is about the same as the running time of $\mathcal{A}$ plus the time to compute T squarings in $\mathbb{G}$. This $(\mathcal{B}_1, \mathcal{B}_2)$ breaks the adaptive root assumption for GGen with the same advantage ϵ that $\mathcal{A}$ breaks soundness.*

Necessity of the adaptive root assumption. The adaptive root assumption is necessary for soundness of the protocol – if the assumption does not hold for *GGen* then the protocol becomes insecure. To see why, let $(\mathcal{A}_1, \mathcal{A}_2)$ be an adaptive root adversary and let $\mathbb{G} \xleftarrow{R} GGen(\lambda)$. To break the protocol using $(\mathcal{A}_1, \mathcal{A}_2)$ choose an arbitrary $g \in \mathbb{G}$, fix some T, and run $(w, \mathsf{state}) \leftarrow \mathcal{A}_1(\mathbb{G})$, where $w \neq 1$. Let $h \leftarrow g^{(2^T)}$. Now, let's see how to convince the verifier to incorrectly accept the tuple $(\mathbb{G},\ g,\ wh,\ T) \notin \mathcal{L}_{\text{EXP}}$. The verifier outputs a random $\ell \in Primes(\lambda)$ and we need to produce a π such that $wh = \pi^\ell g^r$ where $2^T = q\ell + r$ and $0 \leq r < \ell$. To do so, we run $\mathcal{A}_2(\ell, \mathsf{state})$ to a get $u \in \mathbb{G}$ such that $u^\ell = w$. Then $\pi := ug^q$ is a valid proof because

$$\pi^\ell g^r = (ug^q)^\ell g^r = u^\ell g^{q\ell+r} = u^\ell g^{(2^T)} = wh,$$

as required.

3.3 Security of the non-interactive variants

While Theorems 1 and 2 analyze the interactive variants of the protocols, the non-interactive variants obtained by applying the Fiat-Shamir heuristic can be similarly shown to be secure by an analysis in the random oracle model.

Interestingly, for Wesolowski's succinct argument, there is a loss in soundness between the interactive and non-interactive variants of the protocol, as already mentioned in Section 2.1. This forces us to generate the challenge prime ℓ from the set $Primes(2\lambda)$ instead of $Primes(\lambda)$; otherwise there is a $\tilde{O}(2^{\lambda/2})$ time attack on the non-interactive succinct argument. Sampling ℓ from $Primes(2\lambda)$ makes the attack time $\tilde{O}(2^\lambda)$, which is infeasible. It also makes it possible to prove soundness in the random oracle model [14]. This is an example where the Fiat-Shamir heuristic harms the soundness of an argument, and forces the verifier to choose a larger challenge to compensate.

Let's see the $\tilde{O}(2^{\lambda/2})$ time attack. Suppose ℓ is generated using a hash function H that maps an input $(\mathbb{G}, g, h, T)$ to a prime ℓ in $Primes(\lambda)$.

- Let $S = \{\ell_1, \ldots, \ell_d\} \subset Primes(\lambda)$ be a subset of $d = 2^{\lambda/2}$ primes.
- Set $\quad L := \prod_{i=1}^d \ell_i \in \mathbb{Z}, \quad w := g^L \in \mathbb{G}, \quad$ and $\quad h := g^{(2^T)} \in \mathbb{G}.$

The attacker finds the smallest $k \geq 1$ such that

$$H(\mathbb{G},\ g,\ hw^k,\ T) = \ell \quad \text{and} \quad \ell \in S. \tag{2}$$

If we model H as a random oracle, then each candidate $k = 1, 2, \ldots$ satisfies (2) with probability $d/2^\lambda = 2^{-\lambda/2}$. Therefore, finding k takes an expected $O(2^{\lambda/2})$ tries. Once k is found, observe that

$$\pi := g^{\lfloor 2^T/\ell \rfloor} \cdot g^{k(L/\ell)} \qquad \text{satisfies} \qquad \pi^\ell = \left(h \cdot g^{-(2^T \bmod \ell)}\right) \cdot w^k,$$

and the attacker can compute this π because the exponent (L/ℓ) is an integer. This π incorrectly convinces the verifier that $h' := hw^k$ satisfies $h' = g^{(2^T)}$. Indeed, the verification condition

$$\pi^\ell \cdot g^{(2^T \bmod \ell)} = hw^k = h'$$

is satisfied, thus breaking soundness. The complete attack runs in expected time $\tilde{O}(2^{\lambda/2})$, which is needed to compute w and to find k, as promised.

4 Comparison of the two protocols

Each proof system has its own strengths and no one dominates the other. The proof system of Wesolowski [14] produces shorter proofs (one group element versus $\log_2 T$ elements) and proof verification is faster (two exponentiations versus $2\log_2 T$). However, the proof of Pietrzak [12] has two advantages discussed below.

Prover efficiency. For the VDF application, Pietrzak's prover is more efficient. It takes $O(\sqrt{T})$ group operations to construct the proof, where as for Wesolowski it takes $O(T)$. Both provers parallelize well and can be sped up by a factor of s using s processors, for a moderate value of s.

Comparison of the assumptions. If Wesolowski's protocol is secure then so is Pietrzak's, but the converse is not known to be true. The reason is that if the adaptive root assumption holds then so must the low order assumption. In other words, adaptive root is potentially a stronger assumption than low order.

To show that the adaptive root assumption implies the low order assumption we show the converse – if low order is broken then so is adaptive root. Let $\mathbb{G} \xleftarrow{R} GGen(\lambda)$ and let $1 \neq \mu \in \mathbb{G}$ be a public element satisfying $\mu^d = 1$ for a known $d > 1$ (i.e., low order is broken). To break the adaptive root assumption, the adversary $\mathcal{A}_1$ outputs μ, and when given a random prime number $\ell \in Primes(\lambda)$, adversary $\mathcal{A}_2$ computes $\mu^{1/\ell}$ as $\mu^{(\ell^{-1} \bmod d)}$. This works as long as d is not a multiple of ℓ, which only happens with negligible probability.

5 Security proofs

Proof of Theorem 2. We construct an adaptive root adversary $(\mathcal{B}_1, \mathcal{B}_2)$ that uses $\mathcal{A}$. When $\mathcal{B}_1$ is initialized with input $\mathbb{G}$, it runs $\mathcal{A}(\mathbb{G})$ and gets back $(\mathbb{G}, g, h, T) \notin \mathcal{L}_{\text{EXP}}$. Algorithm $\mathcal{B}_1$ then outputs $w \leftarrow h/g^{(2^T)} \in \mathbb{G}$, $\mathsf{state} \leftarrow (\mathbb{G}, g, h, T, w)$ and exits. Note that because $h \neq g^{(2^T)}$ we have that $w \neq 1$, as required of an adaptive root adversary.

Next, a random $\ell \in \mathit{Primes}(\lambda)$ is chosen and $\mathcal{B}_2(\ell, \mathsf{state})$ is activated. Let $2^T = q\ell + r$ with $0 \leq r < \ell$. Algorithm $\mathcal{B}_2$ sends the ℓ it was given to $\mathcal{A}$, and $\mathcal{A}$ outputs $\pi \in \mathbb{G}$. Now, $\mathcal{B}_2$ outputs $u \leftarrow \pi / g^q \in \mathbb{G}$ and exits. If $\mathcal{A}$ outputs a valid proof, namely π satisfies $h = \pi^\ell g^r$, then

$$u^\ell = (\pi/g^q)^\ell = \pi^\ell g^r / g^{q\ell+r} = h/g^{(2^T)} = w.$$

Hence, $(\mathcal{B}_1, \mathcal{B}_2)$ succeeds in breaking the adaptive root assumption with the same advantage as $\mathcal{A}$ succeeds in breaking soundness, as required. □

Proof of Theorem 1. We use a forking argument to construct an adversary $\mathcal{B}$ that breaks the low order assumption using $\mathcal{A}$.

Recall that 2^t is an upper bound on the value T output by $\mathcal{A}$. Let $\mathcal{A}(\mathbb{G}, r_0, \ldots, r_{t-1}; R)$ denote an execution of $\mathcal{A}$ with random tape R, where $r_0, \ldots, r_{t-1}$ are the verifier's challenges at each level of the recursion. The adversary $\mathcal{A}$ outputs the protocol transcript which is a sequence of $t+1$ tuples:

$$(P_0, v_0), \ldots, (P_t, v_t)$$

where $P_i = (\mathbb{G}, g_i, h_i, T/2^i)$ is the input to the recursion at level i, and v_i is the prover's message at level i. Recall that $g_i \leftarrow g_{i-1}^{r_{i-1}} v_{i-1}$ and $h_i \leftarrow v_{i-1}^{r_{i-1}} h_{i-1}$ for $i = 1, \ldots, t$. Here we assume $T = 2^t$, but if $T < 2^t$ then we replicate the last pair $(P_{\log_2 T}, v_{\log_2 T})$ to get a full transcript of $t+1$ tuples.

Next, define the following probabilistic experiment EXP:

1. choose a random tape R for $\mathcal{A}$.
2. choose uniform $r_0, \ldots, r_{t-1}$ in $\{1, \ldots, 2^\lambda\}$.
3. run $\mathcal{A}(\mathbb{G}, r_0, \ldots, r_{t-1}; R)$ to get $(P_0, v_0), \ldots, (P_t, v_t)$.
4. if $P_0 \notin \mathcal{L}_{\text{EXP}}$ but $P_t \in \mathcal{L}_{\text{EXP}}$ (i.e., the verifier incorrectly accepts P_0) then:
   - let j be the lowest index for which $P_j \notin \mathcal{L}_{\text{EXP}}$ but $P_{j+1} \in \mathcal{L}_{\text{EXP}}$.
   - choose fresh uniform $r'_j, \ldots, r'_{t-1}$ in $\{1, \ldots, 2^\lambda\}$.
   - run $\mathcal{A}(\mathbb{G}, r_0, \ldots, r_{j-1}, r'_j, \ldots, r'_{t-1}; R)$ to get (P_0, v_0), …, (P_j, v_j), (P'_{j+1}, v'_{j+1}), …, (P'_t, v'_t).
   - if $P'_{j+1} \in \mathcal{L}_{\text{EXP}}$ and $r_j \neq r'_j$, output $(g_j, h_j, T/2^{j+1}, v_j, r_j, r'_j)$ and stop.
5. in all other cases output *fail*.

Let $\mathcal{E}$ be the event that EXP does not output *fail*. When $\mathcal{E}$ happens we have $P_j \notin \mathcal{L}_{\text{EXP}}$ and $P_{j+1}, P'_{j+1} \in \mathcal{L}_{\text{EXP}}$. Therefore, if EXP outputs $\left(g, h, \hat{T}, v, r, r'\right)$ we have that

$$h \neq g^{(2^{2\hat{T}})} \qquad \text{and} \qquad (g^r v)^{(2^{\hat{T}})} = v^r h \qquad \text{and} \qquad (g^{r'} v)^{(2^{\hat{T}})} = v^{r'} h. \tag{3}$$

Re-arranging terms of the two equalities on the right we get

$$\left(g^{(2^{\hat{T}})} / v\right)^r = h / v^{(2^{\hat{T}})} \qquad \text{and} \qquad \left(g^{(2^{\hat{T}})} / v\right)^{r'} = h / v^{(2^{\hat{T}})}. \tag{4}$$

Dividing the left equality by the right we obtain

$$\left(g^{(2^{\hat{T}})} / v\right)^{r-r'} = 1.$$

Hence $\mu := g^{(2^{\hat{T}})}/v$ is an element of order at most $0 < |r - r'| < 2^\lambda$ in $\mathbb{G}$.

Let's see why $\mu \neq 1$. By (3) we have $h \neq g^{(2^{2^{\hat{T}}})}$ and therefore either $h \neq v^{(2^{\hat{T}})}$ or $v \neq g^{(2^{\hat{T}})}$. But then by (4) it must be that $v \neq g^{(2^{\hat{T}})}$. Hence $\mu \neq 1$ and μ is of order at most $d := |r - r'|$.

To summarize, algorithm $\mathcal{B}$ runs experiment EXP, and if it does not fail, it outputs (μ, d). This shows that when event $\mathcal{E}$ happens, algorithm $\mathcal{B}$ succeeds in breaking the low order assumption. It remains to determine how likely is event $\mathcal{E}$ to happen. Fortunately this has already been worked out in the generalized forking lemma of Bellare and Neven [1, Lemma 1]. An application of their lemma shows that if $\mathcal{A}$ succeeds in fooling the verifier with probability ϵ, then event $\mathcal{E}$ happens with probability at least $(\epsilon^2/t) - (\epsilon/2^\lambda)$, as required. □

6 Concrete groups

The RSA group. Let *GGen* be an algorithm that outputs an odd integer N with an unknown factorization. Computing the order of the multiplicative group $\mathbb{G} := (\mathbb{Z}/N)^*$ is as hard as factoring N, and therefore $\mathbb{G}$ can be used as a group of unknown order. However, the low order assumption is trivially false in such groups because $(-1) \in \mathbb{Z}/N$ is an element of order two. Fortunately, this is the only impediment and it is easily corrected by instead working in the group $\mathbb{G}^+ := \mathbb{G}/\{\pm 1\}$. Elements in this group are represented as cosets $\{x, -x\}$ for $x \in \mathbb{G}$ and multiplication is defined as $\{x, -x\} \cdot \{y, -y\} = \{xy, -xy\}$. Of course when computing in this group it suffices to represent a coset $\{x, -x\}$ by a single number, either x or $-x$, whichever is in the range $[0, N/2)$. The low order assumption is believed to hold for a group generator *GGen* that generates such groups.

We note that while Pietrzak [12] suggested using integers N that are a product of strong primes, our use of the low order assumption suggests that soundness holds for more general N. Recall that a prime number p is *strong* if $(p-1)/2$ is also a prime number. If $N = p \cdot q$ is a product of distinct strong primes then the group $\mathbb{G}'$ of quadratic residues in $(\mathbb{Z}/N)^*$ (i.e. $\mathbb{G}' := \{z^2 : z \in (\mathbb{Z}/N)^*\}$) contains no elements of low order other than 1. Hence, the low order assumption holds unconditionally in this group. Pietrzak proved unconditional soundness of the protocol when used in this group $\mathbb{G}'$. By relying on the low order assumption we are able to prove soundness even when N is not a product of strong primes. We note that checking membership in $\mathbb{G}'$ is difficult and this complicates the protocol. Checking membership in $\mathbb{G}^+$ is easy so that the protocol in Figure 1 can be used as is.

The difficulty with the group $(\mathbb{Z}/N)^*$ is that for best results the group generator *GGen* must be trusted to not reveal the factorization of N. One can instead make *GGen* use public randomness to choose a sufficiently large N so that factoring N is hard. However the resulting N must be so large as to be impractical.

The class group of an imaginary quadratic number field. To solve the trusted setup problem one can instead use the class group of the number field $\mathbb{Q}(\sqrt{p})$, where p is a negative prime $p \equiv 1 \bmod 4$, as suggested by Wesolowski [14]. This class group has odd order and computing its order is believed to be difficult when $|p|$ is large. See [4] for a discussion on the choice of cryptographic parameters for such groups. Concretely, the group generator $GGen(\lambda)$ outputs a negative prime p from which the class group of $\mathbb{Q}(\sqrt{p})$ is completely specified.

The Cohen-Lenstra heuristics [7] suggest that for imaginary quadratic number fields:

- the frequency of fundamental discriminants for which the odd part of the class group is cyclic is about 97.6%,

- the frequency $f(d)$ of fundamental discriminants for which the order of the class group is divisible by d is approximately:

$$f(3) = 44\%, \qquad f(5) = 24\%, \qquad f(7) = 16\%.$$

These heuristics suggest that the class group is often cyclic, but often contains elements of small odd order. The question is how hard is it to find an element of small odd order, if one exists?

An approach to finding low order elements in class groups. The low order assumption in the class group of an imaginary quadratic extension has not been studied much, and is a fascinating avenue for future work. For example, can we find an element of order three if one exists?

We mention one possible avenue for attack based on the work of Ellenberg and Venkatesh [9]. Let I be an ideal of order 3 in the class group of $\mathbb{Q}(\sqrt{p})$. Then I^3 is principle meaning that $I^3 = \langle a + b\sqrt{p} \rangle$ for some $a, b \in \mathbb{Z}$. Then the ideal norm $N(I)$ satisfies $N(I)^3 = N(I^3) = a^2 + |p|b^2$. Setting $z = N(I)$ we see that the existence of an ideal of order three implies an integral point on the surface

$$z^3 = a^2 + |p|b^2 \tag{5}$$

where

$$|z| \leq \sqrt{|p|}, \qquad |a| \leq |p|^{3/4}, \qquad |b| \leq |p|^{1/4}. \tag{6}$$

The first inequality follows from the fact that we can take I to be a reduced ideal in the class group. The second and third inequalities follow from the first.

If we could find an integral point (x, y, z) satisfying (6) on the surface (5), where z is not a perfect square, then we will likely break the low order assumption in the class group of $\mathbb{Q}(\sqrt{p})$. We want a point (x, y, z) where $|z| \leq \sqrt{|p|}$ is not a perfect square to ensure that z is not the norm of a principal ideal. Fortunately for this paper, the bounds (6) are out of reach for Coppersmith's method for finding low-norm integral points on curves and surfaces [8]. However, perhaps Coppersmith's method can be tuned specifically for this family of surfaces? We leave that for future work.

7 Open problems

Post-quantum security. We conclude by pointing out that the two VDFs surveyed here are insecure against an adversary who has access to a quantum computer – a quantum computer can easily calculate the order of the group $\mathbb{G}$ using Shor's algorithm and break the VDF. It is a wonderful open problem to find a simple VDF that is post-quantum secure. Some of the VDFs studied in [2] are post-quantum secure, but it would be helpful to have a simpler construction. For example, Buterin [5] describes and implements one of the constructions from [2] using a combination of MiMC and a STARK. Sequential composition of isogenies on elliptic curves is a promising area for a post-quantum VDF. An interesting proposal for a VDF from isogenies is reported in [10]. However, that construction is not designed to be post-quantum secure, and requires a trusted setup.

Acknowledgments

We thank Krzysztof Pietrzak, Benjamin Wesolowski, and Justin Drake for their helpful comments about this writeup.

A A parallel algorithm to compute quotients in the exponent

We conclude with a brief description of how to compute $\pi = g^{\lfloor 2^T/\ell \rfloor} \in \mathbb{G}$ in parallel, as needed to speed up Wesolowski's succinct argument. We can accelerate the prover's time to compute π by a factor of s using s processors. We do so by storing s group elements as the prover evaluates the VDF. Taking $s = 100$ seems reasonable in practice.

So, let $b := \lfloor T/(s-1) \rfloor$. As the prover computes the VDF it stores the following s group elements as they are encountered along the way:

$$u_0 = g, \quad u_1 = g^{(2^b)}, \quad u_2 = g^{(2^{2b})}, \quad \ldots, \quad u_{s-1} = g^{(2^{(s-1)b})}.$$

Next, our algorithm to compute π uses the following subroutine exp, which is essentially the same as the algorithm from Section 2.1. Here $0 \le d < \ell$ is an additional input parameter.

$\exp(h, t, d, \ell)$: // *output* $a = h^{\lfloor d2^t/\ell \rfloor} \in \mathbb{G}$
- $a \leftarrow 1 \in \mathbb{G}, \quad r \leftarrow d \in \{0, \ldots, \ell - 1\}$
- repeat t times:
  - $q \leftarrow \lfloor 2r/\ell \rfloor \in \{0, 1\}, \quad r \leftarrow (2r \bmod \ell) \in \{0, \ldots, \ell - 1\}$
  - $a \leftarrow a^2 \cdot h^q \in \mathbb{G}$
- output a // *this a is equal to* $h^{\lfloor d2^t/\ell \rfloor} \in \mathbb{G}$

Using subroutine exp we can compute $\pi = g^{\lfloor 2^T/\ell \rfloor} \in \mathbb{G}$ in time $O(T/s)$ as follows. The algorithm starts by quickly computing all the remainders needed for the s steps of long division, and then runs these s steps in parallel.

input: g, T, ℓ, s as well as $u_i = g^{(2^{ib})} \in \mathbb{G}$ for $i = 0, \ldots, s-1$ $\quad \big($need $s > 1, \quad T > s(s-2)\big)$

output: $\pi := g^{\lfloor 2^T/\ell \rfloor} \in \mathbb{G}$ computed with s-way parallelism in time $O(T/s)$

- $b \leftarrow \lfloor T/(s-1) \rfloor$ // *batch size*
- // *compute remainders by a quick sequential process*
- $r_0 \leftarrow (2^{(T \bmod b)} \bmod \ell) \in \{0, \ldots, \ell - 1\}$
- for $i = 1, \ldots, s-1$:
  - $r_i \leftarrow \big(2^b \cdot r_{i-1} \bmod \ell\big) \in \{0, \ldots, \ell - 1\}$
- // *compute π in parallel*
- (1) $\pi_0 \leftarrow \exp\big(u_{s-1},\ (T \bmod b),\ 1,\ \ell\big)$ // *compute* $\pi_0 \leftarrow (u_{s-1})^{\lfloor 2^{(T \bmod b)}/\ell \rfloor} \in \mathbb{G}$
- for $i = 1, \ldots, s-1$:
- (2) $\quad \pi_i \leftarrow \exp\big(u_{s-1-i},\ b,\ r_{i-1},\ \ell\big)$ // *compute* $\pi_i \leftarrow (u_{s-1-i})^{\lfloor r_{i-1} \cdot 2^b/\ell \rfloor} \in \mathbb{G}$
- output $\pi \leftarrow \prod_{i=0}^{s-1} \pi_i$

The bulk of the work happens on lines (1) and (2), where each call to the function exp requires b sequential squarings. The point is that all the calls to exp can be processed in parallel. The algorithm needs enough memory to store only s group elements.

References

[1] M. Bellare and G. Neven. Multi-signatures in the plain public-key model and a general forking lemma. In *Proceedings of the 13th ACM conference on Computer and communications security*, pages 390–399. ACM, 2006.

[2] D. Boneh, J. Bonneau, B. Bünz, and B. Fisch. Verifiable delay functions. In *CRYPTO 2018*, pages 757–788, 2018. `https://eprint.iacr.org/2018/601`.

[3] D. Boneh and V. Shoup. *A graduate course in applied cryptography.* Cambridge, 2018.

[4] J. Buchmann and S. Hamdy. A survey on iq cryptography. In *Public-Key Cryptography and Computational Number Theory*, pages 1–15, 2001.

[5] V. Buterin. STARKs, part 3: Into the weeds, 2018. `https://vitalik.ca/general/2018/07/21/starks_part_3.html`.

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

[7] H. Cohen and H. W. Lenstra. Heuristics on class groups of number fields. In *Number Theory Noordwijkerhout 1983*, pages 33–62. Springer, 1984.

[8] D. Coppersmith. Small solutions to polynomial equations, and low exponent rsa vulnerabilities. *Journal of Cryptology*, 10(4):233–260, 1997.

[9] J. S. Ellenberg and A. Venkatesh. Reflection principles and bounds for class group torsion. *International Mathematics Research Notices*, 2007, 2007.

[10] L. D. Feo, S. Masson, C. Petit, and A. Sanso. Verifiable delay functions from supersingular isogenies and pairings. In *ASIACRYPT 2019*, pages 248–277, 2019.

[11] A. K. Lenstra and B. Wesolowski. Trustworthy public randomness with sloth, unicorn, and trx. *International Journal of Applied Cryptography*, 3(4):330–343, 2017.

[12] K. Pietrzak. Simple verifiable delay functions. In *Innovations in Theoretical Computer Science Conference, ITCS*, pages 60:1–60:15, 2019. `https://eprint.iacr.org/2018/627`.

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

[14] B. Wesolowski. Efficient verifiable delay functions. In *EUROCRYPT 2019*, pages 379–407, 2019. `https://eprint.iacr.org/2018/623`.