Efficient Verifiable Delay Function

Laboratory for cryptologic algorithms (LACAL)
Bachelor semester project

Spring 2019

Novak Kaluderovic, Dusan Kostic, Arjen K. Lenstra

Adrian HAMELINK, Alessio ATTANASIO

March 20, 2021

Contents

1 Introduction

Verifiable Delay Functions are a very new area of research in cryptography, with possible applications ranging from random beacons to blockchains. In this paper we will explain how this class of functions works, present a specific construction by Benjamin Wesolowski, and then present our concrete implementation of it. The goal is to better understand how they work, and develop intuition about their security.

We thank the doctoral assistants Novak Kaluderovic and Dusan Kostic as well as the PhD student Aymeric Genet from the LACL laboratory at EPFL for their help in the realization of this report.

We first give a formal but abstract definition proposed in [Boneh et al.(2018)Boneh, Bünz, and Fisch].

1.1 VDF Definition

Definition 1. Verifiable Delay Function (VDF)
A Verifiable Delay Function (VDF) is a function $f : \mathcal{X} \to \mathcal{Y}$ whose evaluation must take a predetermined amount of time, regardless of parallelization. The correctness of the output must then be publicly verifiable in a much shorter amount of time. It is also required that every input $x \in \mathcal{X}$ has a unique and valid output $y \in \mathcal{Y}$.
An implementation of such a function can be defined by a tuple of three algorithms, namely

$\texttt{setup}(k, \Delta) \to \texttt{pp}$
: Outputs some random public parameters $\texttt{pp}$ depending on the timing parameter Δ and security parameter k.

$\texttt{eval}_{\texttt{pp}}(x) \to (y, \pi)$
: Effectively computes and outputs the value $y = f(x)$ for an input $x \in \mathcal{X}$ as well as a proof π with regards to the public parameters.

$\texttt{verify}_{\texttt{pp}}(x, y, \pi) \to \{\texttt{true}, \texttt{false}\}$
: Outputs `true` if y is the correct output of $f(x)$ and `false` otherwise.

We also require that this construction satisfies the three following properties, which we state informally here but detail in a more rigorous way in chapters 2, 3 and 4 respectively.

Soundness
: It is not possible to generate a misleading proof π' for an incorrect output y'.

Sequentiality
: There is no algorithm (even using $poly(k)$ parallel processors) which can output the correct result of $\texttt{eval}_{\texttt{pp}}$ in time less than Δ, for any random $x \in \mathcal{X}$.

ϵ-evaluation time
: The algorithm $\texttt{eval}_{\texttt{pp}}$ must run in at least $(1 + \epsilon)\Delta$ amount of time. The ϵ represents the extra time necessary to compute a proof.

We now introduce the following notations and a definition that will be used throughout the paper, some of them we have already encountered.

Definition 2. Negligible function
We say that a function $\eta : \mathbb{N} \to \mathbb{R}$ is negligible, if for every polynomial f the exists $N_f > 0$ such that for every $x > N_f$ we have $\frac{1}{|\eta(x)|} < \frac{1}{|f(x)|}$.

Expression	**Meaning**
Δ	Timing parameter
$k \in \mathbb{N}^*$	Security level (typically 128,192 or 256)
$(\mathbb{Z}/N\mathbb{Z})^\times$	RSA group, where N is the product of two large prime numbers
Primes(2k)	Set containing the 2^{2k} first prime numbers
a mod b	Will always mean the **least positive** residue of a modulo b

Table 1: Notations

1.2 Group construction

In this paper, we present a construction due to Benjamin Wesolowski (see [Wesolowski(2018)]) and inspired by the Rivest-Shamir-Wagner time-lock puzzle, which is stated in Assumption 2. We refine the above algorithms for this context.

`setup` will generate a tuple formed of a finite abelian group $\mathbf{G}$ of unknown order, a hash function $H_\mathbf{G} : \mathcal{X} \to \mathbf{G}$ and a time parameter Δ, all of which are functions of the security parameter k. The time parameter Δ should represent the prescribed computation time required to evaluate the VDF. It implicitly defines an integer t representing the number of squarings to be performed in order to achieve an execution time at least Δ. We can view $\Delta = \delta t$, where δ is the time required to perform a single squaring operation in the group $\mathbf{G}$.

$\texttt{eval}_\texttt{pp}$ will then compute $y \leftarrow (H_\mathbf{G}(x))^{(2^t)}$ as well as a proof π used by $\texttt{verify}_\texttt{pp}$. The use of H is justified as it removes the group homomorphism property of $g \mapsto g^{(2^t)}$. Otherwise, we would be able to efficiently evaluate $h^{(2^t)}$ using another evaluation $g^{(2^t)}$. Indeed, if $h = g^\alpha$, then $h^{(2^t)} = (g^\alpha)^{(2^t)} = (g^{(2^t)})^\alpha$ and thus evaluating the VDF on the input h requires a single exponentiation.

Before defining the verification and proof generation procedures, we describe the following interactive protocol in which a *Verifier* attests whether a *Prover*'s output is the correct result of the computation $y \leftarrow (H_\mathbf{G}(x))^{(2^t)}$. The *Verifier* is generally seen as a party with limited computing power who does not trust the *Prover*.

1. The *Verifier* samples a prime l uniformly at random from *Primes(2k)*, sends it to the *Prover*.
2. The *Prover* computes $\pi \leftarrow g^{\lfloor 2^t/l \rfloor}$ and sends it to the *Verifier*.
3. The *Verifier* computes $r \leftarrow 2^t \mod l$ and accepts if $\pi^l g^r = y$.

To see why step 3 produces a valid assessment, we develop the expression $\pi^l g^r$:

$$\pi^l g^r = (g^{\lfloor 2^t/l \rfloor})^l \cdot g^{2^t \bmod l} = g^{\lfloor 2^t/l \rfloor l + (2^t \bmod l)} = g^{2^t} = y$$

We can define the $\texttt{verify}_\texttt{pp}$ algorithm by making this protocol non interactive. Using the Fiat-Shamir heuristic, we replace the prime sampling step with a non-interactive random oracle. This oracle can be modeled as cryptographic hash function $H_{prime}(g_1, g_2)$, returning a random element in $Primes(2k)$ for any pair of group elements (g_1, g_2).
In our case, after evaluating $y \leftarrow (H_\mathbf{G}(x))^{(2^t)}$, the *Prover* can compute $\pi \leftarrow g^{\lfloor 2^t/l \rfloor}$ using $l \leftarrow H_{prime}(g, y)$. Because H_{prime} associates the input/output pair to a random prime number l in a deterministic way, l can then be re-obtained independently by the *Verifier* as required in step 3.

We suppose that the two methods of verification are equivalent and we will use them interchangeably throughout the paper.

One may wonder if the above condition can also be satisfied given a different pair (y', π') for the same input x. We explore this question in chapter 2 and determine that it would be near impossible without knowledge of the secret key `sk`.

Definition 3. Trapdoor VDF
A trapdoor VDF is a VDF with the additional property, that there exists a secret key $\texttt{sk}$ for the public parameters $\texttt{pp}$ and an algorithm $\texttt{trapdoor}_{\texttt{pp}}(\texttt{sk}, x) \rightarrow (y, \pi)$, whose evaluation takes significantly less time than $\texttt{eval}_{\texttt{pp}}$ (less than Δ). The output (y, π) must be accepted by $\texttt{verify}_{\texttt{pp}}$.

In the context of the group construction, the secret key would represent the order $\mathit{order}(\mathbf{G})$ of the group $\mathbf{G}$ and its utility relies on the following observation:
Let $\texttt{sk} := \mathit{order}(\mathbf{G})$ and $q = \lfloor 2^t/\texttt{sk} \rfloor$

$$g^{2^t} = g^{\texttt{sk}\cdot q + (2^t \bmod \texttt{sk})} = g^{\texttt{sk}\cdot q} \cdot g^{2^t \bmod \texttt{sk}} = e_{\mathbf{G}} \cdot g^{2^t \bmod \texttt{sk}} = g^{2^t \bmod \texttt{sk}}$$

The computation $(H_{\mathbf{G}}(x))^{(2^t)}$ could then be replaced by $(H_{\mathbf{G}}(x))^{(2^t \bmod \texttt{sk})}$. This reduces the number of operations to perform in $\mathbf{G}$ remarkably. We will need $log_2(t)$ operations in $\mathbb{Z}/\texttt{sk}\mathbb{Z}$ to calculate $e = 2^t \bmod \texttt{sk}$ and then $\mathcal{O}(\log_2 \texttt{sk})$ operations in $\mathbf{G}$ to compute the exponentiation by e. The speed up becomes apparent when $t >> \log_2 \texttt{sk}$.

1.3 Algorithms

We now present the following realizations of the above mentioned algorithms.

Algorithm 1 $\texttt{eval}_{\texttt{pp}} : (x, t) \rightarrow (y, \pi)$

```
1: $g \leftarrow H_{\mathbf{G}}(x) \in \mathbf{G}$
2: $y \leftarrow g^{2^t}$        // by computing $t$ sequential squarings
3: $l \leftarrow H_{prime}(g, y)$
4: $\pi \leftarrow g^{\lfloor 2^t/l \rfloor}$        // Using Algorithm 6, explained in section 4.2.4
5: return $(y, \pi)$
```

Algorithm 2 $\texttt{trapdoor}_{\texttt{pp}} : (\texttt{sk}, x, t) \rightarrow (y, \pi)$

```
1: $g \leftarrow H_{\mathbf{G}}(x) \in \mathbf{G}$
2: $e \leftarrow 2^t \bmod \texttt{sk}$
3: $y \leftarrow g^e$
4: $l \leftarrow H_{prime}(g, y)$
5: $r \leftarrow 2^t \bmod l$
6: $q \leftarrow (2^t - r)/l \bmod \texttt{sk}$
7: $\pi \leftarrow g^q$
8: return $(y, \pi)$
```

Algorithm 3 $\texttt{verify}_{\texttt{pp}} : (x, y, \pi) \rightarrow \{\texttt{true}, \texttt{false}\}$

```
1: $g \leftarrow H_{\mathbf{G}}(x) \in \mathbf{G}$
2: $l \leftarrow H_{prime}(g, y)$
3: $r \leftarrow 2^t \mod l$
4: if $\pi^l g^r = y$
5:     return true
6: else
7:     return false
```

1.4 Realization of the VDF in an RSA group

The above construction works for any arbitrary finite abelian group $\mathbf{G}$ of unknown order. Two such groups are the ideal class groups and the well known RSA groups. Our implementation will focus on the latter.

For any integer N, we define the ring of integers modulo N as $\mathbb{Z}/N\mathbb{Z}$. We can then look at its multiplicative subgroup $(\mathbb{Z}/N\mathbb{Z})^\times$. This group's order is equal to $\phi(N)$, where ϕ is Euler's totient function counting the positive integers smaller than N that are relatively prime to it.

The motivation to use the group $(\mathbb{Z}/N\mathbb{Z})^\times$ comes from the computationally hard problem of prime factorization. Indeed, if $N = p \cdot q$ for two prime numbers p, q, then $\phi(N) = (p-1)(q-1)$ and we can only determine the group order if we know p and q. Moreover, knowing the order of $(\mathbb{Z}/N\mathbb{Z})^\times$ is actually equivalent to knowing the factorization of N.
In the context of verifiable delay functions, we can generate such a group in the `setup` algorithm by running a `keygen` : $(k) \mapsto (\texttt{pk}, \texttt{sk})$ procedure which generates two prime numbers p, q and output $\texttt{pk} = p \cdot q$ and $\texttt{sk} = (p-1)(q-1)$. The size of `pk` depends on the security parameter k and should be large enough such that the factorization of `pk` remains hard.
The existence of `sk` implies that a VDF over such a group is a trapdoor VDF. Of course, by "forgetting" the key we would fall back to a regular VDF, but it is not clear how that could be achieved in practice. It might be possible to generate a large N as product of many different prime numbers generated by multiple parties but again is more difficult in practice.

We note that the computational time complexity of performing operations in $(\mathbb{Z}/N\mathbb{Z})^\times$ is $\mathcal{O}((\log_2 N)^2)$ (see page 33 of [Shoup(2008)]).

Hereinafter we shall define the group $\mathbf{G}$ as $(\mathbb{Z}/N\mathbb{Z})^\times/\{\pm 1\}$ where $N = \texttt{pk}$. We name this group RSA-VDF group and explain why we use it in chapter 2.2.

The following chapters 2 and 3 will address the security properties of a VDF over the above mentioned group. The first is soundness, which determines whether it is possible to falsify outputs such that they are still valid. The second is sequentiality, which assures us that the run time of the `eval` algorithm will take the prescribed amount of time. We will see that both of these properties fail when the secret key `sk` is known.

2 Soundness of the Trapdoor VDF

In this chapter we will discuss the soundness of our VDF, i.e. determine whether a polynomially bounded algorithm can produce a false proof and a false output for a given input x. We first introduce the definition of soundness with respect to our VDF. We then explain how someone could generate false proofs for an incorrect output of the VDF, followed by a discussion about the use of the RSA-VDF group. Finally, we prove the soundness of the VDF with the help of some notions that can be found in [Wesolowski(2018)].

Definition 4. Trapdoor VDF soundness-breaking game
Let $\mathscr{A}$ be a party playing the game which receives the public parameters $\texttt{pp}$ as input. The player $\mathscr{A}$ must then output a message x, a value y' and a proof π'. The game is won if the output is such that $y' \neq \texttt{eval}_{\texttt{pp}}(x, \Delta)$ and $\texttt{verify}_{\texttt{pp}}(x, y', \pi, \Delta) = \texttt{true}$.

Definition 5. Soundness
A trapdoor VDF is sound if no polynomially bounded algorithm wins the soundness-breaking game (Definition 4) with non-negligible probability in k.

2.1 Generating false proofs

Recall the verification condition $\pi^l g^r = y$, and note that r can be written as $r = 2^t - l\lfloor 2^t/l \rfloor$. Therefore by multiplying the verification condition by g^{-2^t} on both sides we obtain:

$$yg^{-2^t} = \left(\pi g^{-\lfloor 2^t/l \rfloor}\right)^l \tag{1}$$

Suppose an adversary $\mathscr{E}$ wants to generate a false proof π' that any verifier must accept. For an input x, the correct output of the VDF must be y, so $\mathscr{E}$ must convince a verifier to accept $y' \neq y$. Let $g = H_{\mathbf{G}}(x)$ be the hash result obtained during the evaluation procedure. The generation of a false proof π' goes as follows:

- $\mathscr{E}$ computes $\alpha' = y' g^{-2^t}$
- As soon as the prime l is published, $\mathscr{E}$ can extract the l-th root of α' (which we denote β')
- $\mathscr{E}$ multiplies β' by $g^{\lfloor 2^t/l \rfloor}$ (the result will be named π')

From (1), we know that the calculated false proof $\pi' = (y'g^{-2^t})^{1/l} g^{\lfloor 2^t/l \rfloor}$ will trick the verifier into accepting y' as correct output. Fortunately, any adversary $\mathscr{E}$ will have a very hard time in finding the l-th root in an RSA group of unknown order. Root extraction is considered a hard problem and the soundness of our VDF rests on this assumption. This assumption will be addressed in more detail in subchapter 2.3. Note that if $\alpha' = 1_{\mathbf{G}}$ it is always easy to find an l-th root of α'. This leads to the honest output $y = y'$ and $\pi' = \pi$.

Unfortunately, knowledge of the secret $\texttt{sk}$ allows for faster root extractions. If for example we want to extract the l-th root of $g \in (\mathbb{Z}/N\mathbb{Z})^\times$, it is enough to compute the inverse of l, which is seen as an element of $(\mathbb{Z}/\phi(N)\mathbb{Z})^\times$. In order to find this inverse, l must be coprime to $\phi(N)$. We then make use of Bézout's identity: there exist $x, y \in \mathbb{Z}$ such that $lx + \phi(N)y = gcd(l, \phi(N))$. In our case, $gcd(l, \phi(N)) = 1$, so we obtain $lx \equiv 1 \mod \phi(N)$.
We can find x and y using the Extended Euclidian Algorithm, which has a complexity of $\mathcal{O}(log(\phi(N))^2)$ (see page 80 of [Shoup(2008)]). The l-th root of g will then be g^x. The following verifies this claim:

$$(g^x)^l = g^{xl} = g^{1-y\phi(N)} = g^1 * g^{-y\phi(N)} = g^1 = g$$

Therefore the holder of the secret key can easily generate false proofs.

2.2 The RSA-VDF group

Right now, the construction of our VDF leaves one little liberty to an adversary $\mathscr{E}$. In equation (1) we see that after having obtained the output y from the VDF evaluation on x, $\mathscr{E}$ can still decide whether the output should be $-y$ or y. Since l is an odd prime number (we can suppose $l \neq 2$), modifying the sign of the proof π will result in a verifier also accepting $-y$ as the correct output. We can represent $y \in (\mathbb{Z}/N\mathbb{Z})^\times$ by an element of $\{0, 1, \dots N-1\}$ and therefore suppose $y \in \{0, 1, \dots N-1\}$. The element $-y$ of $(\mathbb{Z}/N\mathbb{Z})^\times$ can then be represented as $N - y$, which will have the opposite parity of y because N is odd.
We remove the possibility of choosing between y or $-y$ by slightly modifying the underlying RSA group. Working in the RSA-VDF group contributes to the VDF soundness by removing the possibility of choosing to output y or $-y$, accompanied by its misleading proof.

Definition 6. The RSA-VDF group
Let $(\mathbb{Z}/N\mathbb{Z})^\times$ be an RSA group. The RSA-VDF group will be defined as $(\mathbb{Z}/N\mathbb{Z})^\times/\{\pm 1\}$. If for any two elements $g, h \in (\mathbb{Z}/N\mathbb{Z})^\times$ we have $g \cdot h^{-1} \in \{\pm 1\}$, we consider x and y as the same element when viewed in the RSA-VDF group.

Using the RSA-VDF group we remove the possibility of choosing y or $-y$ as output, because they represent the same element in the RSA-VDF group.

2.3 Proof of the VDF's soundness

In this section we prove the soundness of our VDF.
In [Wesolowski(2018)] this is done very formally by defining a root finding game and working with the following assumption:

Assumption 1.
Let $N = pq$ be a RSA modulus, where p and q are unknown.
It is hard to find $u \in (\mathbb{Z}/N\mathbb{Z})^\times \setminus \{\pm 1\}$ for which l-th roots can be extracted in $(\mathbb{Z}/N\mathbb{Z})^\times$, for arbitrary values l sampled uniformly from *Primes(2k)*.

Instead of reproducing the same soundness proof, we prove the following proposition which explains the mathematical intuition behind using Assumption 1.

Proposition 1.
Our VDF is sound if and only if Assumption 1 holds.

Proof of Proposition 1: Suppose that our VDF is not sound, and that one could generate false proofs in a polynomially bounded amount of time. We show that this breaks Assumption 1. Let x be the input of the VDF, y the correct output alongside its correct proof π. Let $y' \neq y$ be a false output with false proof π'. Let us define $u = \pi'/\pi$. By (1), we have that y'/y is an l-th root of u. This breaks Assumption 1 because we have just found an element $u \neq 0, \pm 1$, from which we could easily extract an l-th root.

Suppose that we can break Assumption 1 and thus have found an element u from which we can extract easily l-th roots. Denote by ξ the l-th root of u, i. e. $\xi^l = u$. Let again y and π be the correct output and proof on the input x of the VDF. We can generate a valid proof π' for a false output $y' \neq y$ by setting $y' = uy$ and $\pi' = \xi\pi$. This pair is accepted by the verifier:

$$(\pi')^l x^r = (\xi\pi)^l x^r = u\pi^l x^r = uy = y' \qquad \text{where } y = x^{2^t}, \text{ and } r = 2^t \bmod l$$

Therefore our VDF is not sound.

□

3 Sequentiality of the Trapdoor VDF

In this chapter we will discuss the sequentiality of our VDF depending by the classic time-lock assumption of Rivest, Shamir and Wagner introduced in [Rivest et al.(1996)Rivest, Shamir, and Wagner]. Under this assumption, Wesolowski proves in [Wesolowski(2018)] that no polynomially bounded player can evaluate the VDF correctly without the secret key `sk` and in time less than $\Delta = \delta t$.

We first introduce some notations.
Let $\mathcal{M}$ be a model of computation, that is a set of allowable operations along with their respective cost. For an algorithm $\mathscr{A}$, we define $\mathrm{T}(\mathscr{A}, x)$ as the amount of sequential work $\mathscr{A}$ has to perform in $\mathcal{M}$ given x as input, i.e. the time-cost. The cost $\mathrm{C}(\mathscr{A}, x)$ will represent the sum of the cost of all the elementary operations performed by $\mathscr{A}$ in $\mathcal{M}$ on the input x. Therefore the cost C does not take into account parallelization whereas the time-cost T does. Note that $\mathrm{T}(\mathscr{A}, x)$ is an abstraction of the time $\mathscr{A}$ takes to produce the output on input x and does not correspond to real life wall-clock time. Recall the security parameter k from Table 1. We define $\delta : \mathbb{N}^* \to \mathbb{R}_{\geq 0}$ to be a function of k, which encodes the time-cost of computing a single modular squaring (i.e. a single squaring in the RSA-VDF group). We now introduce the (δ, t)-time-lock game and the associated time-lock assumption of Rivest, Shamir and Wagner.

Definition 7. (δ, t)-time-lock game
Let $\mathscr{A}$ be an algorithm the following game:

- An RSA modulus N is generated at random by an RSA key-generation precedure for the security parameter k (e.g. `keygen`)
- $\mathscr{A}$ outputs an algorithm $\mathscr{B}$
- An element $g \in (\mathbb{Z}/N\mathbb{Z})^\times$ is generated uniformly at random
- $\mathscr{B}(g)$ outputs $h \in (\mathbb{Z}/N\mathbb{Z})^\times$

$\mathscr{A}$ wins the game if $h = g^{2^t} \mod N$ and $\mathrm{T}(\mathscr{B}, g) < t\delta(k)$.

Assumption 2. Time-lock

- There is no algorithm $\mathscr{A}$ such that for any modulus N generated by an RSA key-generation procedure with security parameter k, and any element $g \in (\mathbb{Z}/N\mathbb{Z})^\times$, the output of $\mathscr{A}(N, g)$ is the square of g and $\mathrm{T}(\mathscr{A}, (N, g)) < \delta(k)$
- For any $t \in \mathbb{N}^*$, no algorithm $\mathscr{A}$ of polynomial cost ($\mathrm{C}(\mathscr{A}, g) = \mathcal{O}(f(log_2(g))$ for a polynomial f) wins the (δ, t)-time-lock game with non-negligible probability with respect to k

The time-lock assumption is crucially important for the security of our VDF. The assumption states that without knowing the factorization of N, there is no reliable (it succeed with non-negligible probability) faster way to compute g^{2^t} than by performing the t squarings sequentially Note that there will always be the possibility to precompute the results for some values and therefore the probability of winning the time-lock game will never be zero. But by increasing k (thus the size of the RSA group), this probability will get very close to zero and therefore be negligible.

It is also worth noting that the running time of our VDF will depend on the hardware on which it is performed. Since we assume that the computation is not parallelisable, there is no use in using multiple CPU's. What our analysis so far didn't take into account is specialized hardware built to perform specific computations. Modular squarings could then be performed much more efficiently than on general purpose hardwares. Wesolowski points out in his paper [Wesolowski(2018)], that the Ethereum Foundation and Protocol Labs hope to construct a piece of hardware, close enough to today's technological limits, to perform multiplications in RSA groups.

4 Computation of the proof without the secret key

In this section we will discuss how to efficiently compute the proof $\pi = g^{\lfloor 2^t/l \rfloor}$. First we present a method that computes it in $\mathcal{O}(t)$ using on-the-fly long division. This method will serve as a building block in the construction of our final algorithm. In the end, given some specific intermediary results from t sequential squarings, the proof can be calculated in $\mathcal{O}(t/log(t))$ group operations.
In a last optimization attempt, we reduce the overhead induced by the proof calculation in $\texttt{eval}_{\texttt{pp}}$, by splitting the computation into multiple segments. By computing these segments in parallel, we can finish the procedure slightly after the squarings have been performed, at the expense of larger proof sizes. All the derivations and the algorithms come from [Wesolowski(2018)].

Our final algorithm produces a proof in time less than t, verifying the ϵ-evaluation time property mentionned in the introduction.

4.1 Computing the proof using on-the-fly long division in $\mathcal{O}(t)$

The first problem we encounter in calculating $g^{\lfloor 2^t/l \rfloor}$ is that $\lfloor 2^t/l \rfloor$ is too big of a number and we can't exponentiate g directly by it. Therefore we compute it on-the-fly using long division. Consider first Algorithm 4 which, given two numbers t and l, returns the quotient of the division $2^t/l$, i.e. $\lfloor 2^t/l \rfloor$.

Algorithm 4 Long division for powers of 2: $\mathcal{O}(t)$

```
1: $q := 0$
2: $r := 1$
3: for $i = 0 \to t - 1$
4:     $b = \lfloor 2r/l \rfloor \in \{0, 1\} \subset \mathbb{Z}$
5:     $r = 2r \mod l$
6:     $q = 2q + b$
7: return $q$
```

Proof of Algorithm 4: We can prove it very easily by induction:
For $t = 1$ we get $b = \lfloor 2r/l \rfloor = \lfloor 2/l \rfloor$. The algorithm then returns $2q + \lfloor 2/l \rfloor = \lfloor 2/l \rfloor$ which proves the base case as $q = 0$. Furthermore, $r = 2 \mod l$ is the correct remainder in the base case.
Suppose now that that Algorithm 4 outputs the right result, i.e. $\lfloor 2^s/l \rfloor$ for all s strictly smaller than $t > 1$. Therefore after the $(t-1)$-th iteration, we have $q = \lfloor 2^{t-1}/l \rfloor$ and $r = 2^{t-1} \bmod l$, which leads us to

$$2^t = 2(ql + r) = (2q + \lfloor 2r/l \rfloor)l + (2r \mod l)$$

Which means that

$$\lfloor 2^t/l \rfloor = 2q + \lfloor 2r/l \rfloor$$

Note that this is exactly what Algorithm 4 would output after the t-th iteration. This proves its correctness.

□

In order to compute the proof we need to do the exact same operations as in Algorithm 4, but in the exponent of g, which gives birth to Algorithm 5.
The correctness of Algortihm 5 follows directly from that of Algorithm 4. The complexities are both dominated by the for loop, which iterates t times. This gives us complexity $\mathcal{O}(t)$, which means that calculating the proof will take as long as calculating the output g^{2^t} of the VDF itself.

Algorithm 5 On-the-fly computation of the proof: $\mathcal{O}(t)$

```
1: $x := 1_G = g^0 \in \mathbf{G}$
2: $r := 1 \in \mathbb{Z}$
3: for $i = 0 \to t-1$
4:     $b = \lfloor 2r/l \rfloor \in \{0,1\} \subset \mathbb{Z}$
5:     $r = 2r \mod l$
6:     $x = x^2 g^b$
7: return $x$
```

4.2 Computing the proof using precomputed values in $\mathcal{O}(t/log(t))$

The basic ideas behind the following method is expressing $\lfloor 2^t/l \rfloor$ in basis 2^κ for some integer $\kappa \in \mathbb{N}$. Note that all the elements of the form $\{g, g^{2^\kappa}, g^{2^{2\kappa}}, \dots\}$ were previously computed during $\texttt{eval}_{\texttt{pp}}$, by storing every κ-th element when performing the t squarings.
We then can calculate the proof in the following way:

$$\lfloor 2^t/l \rfloor = \sum_{i=0} b_i 2^{\kappa i} \iff g^{\lfloor 2^t/l \rfloor} = \prod_{i=0} (g^{2^{\kappa i}})^{b_i} \tag{2}$$

4.2.1 Calculating the basis coefficients

Let's denote by b_i the coefficients used to write $\lfloor 2^t/l \rfloor$ in basis 2^κ, i.e. like in the LHS of the equivalence in (2). In order to compute the proof like above we need to efficiently calculate the coefficients b_i. We do so using Algorithm 4. The euclidian division of 2^t by κ gives $t = s\kappa + m \Leftrightarrow 2^t = 2^{s\kappa}2^m$ where $0 \leq m < \kappa$. Using the same notations as in Algorithm 4 we develop and obtain

$$\begin{aligned}
2^m &= q_0 l + r_0 \\
2^\kappa 2^m &= q_1 l + r_1 \\
&\vdots \\
2^{i\kappa} 2^m &= q_i l + r_i
\end{aligned}$$

where $q_i = \lfloor 2^{i\kappa}2^m/l \rfloor$ and $r_i = 2^{i\kappa}2^m \mod l$.

By denoting $r'_i = \lfloor 2^\kappa r_i/l \rfloor$ we establish two recursive relations for q_i and r_i:

$$\begin{aligned}
q_i &= 2^\kappa q_{i-1} + r'_{i-1} \\
r_i &= 2^\kappa r_{i-1} \mod l
\end{aligned}$$

By expanding the procedure of Algorithm 4 we notice that

$$\begin{aligned}
q_i &= 2^\kappa q_{i-1} + r'_{i-1} \\
&= 2^\kappa(2^\kappa q_{i-2} + r'_{i-2}) + r'_{i-1} \\
&\vdots \\
&= 2^{\kappa i} q_0 + 2^{\kappa(i-1)} r'_0 + 2^{\kappa(i-2)} r'_1 + \dots 2^\kappa r'_{i-2} + r'_{i-1} \\
&= 2^{\kappa i} q_0 + \sum_{j=0}^{i-1} 2^{\kappa j} r'_{i-1-j} \\
\implies & \quad \lfloor 2^t/l \rfloor = q_s = 2^{\kappa s} q_0 + \textstyle\sum_{j=0}^{s-1} 2^{\kappa i} r'_{s-1-j}
\end{aligned}$$

The searched coefficients are thus given by $b_i = r'_{s-1-j}$ if $i = 0, 1, \dots s-1$ and $b_s = q_0$. By unraveling the definitions, we get the following closed formula for all the coefficients:

$$b_i = \left\lfloor \frac{2^\kappa (2^{t-\kappa(i+1)} \mod l)}{l} \right\rfloor \quad \text{for} \quad i = 0, 1, \dots s$$

4.2.2 Complexity analysis

In this section we will show that the number of group operations required to compute the proof is $\mathcal{O}(t/log(t))$. By denoting $I_b = \{i | b_i = b\}$ for $b \in \{0, 1, \dots 2^\kappa - 1\}$ we can write

$$g^{\lfloor 2^t/l \rfloor} = \prod_{i=0} (g^{2^{\kappa i}})^{b_i} = \prod_{b=0}^{2^\kappa - 1} \Big(\prod_{i \in I_b} g^{2^{\kappa i}} \Big)^b \tag{3}$$

Since all the elements in $\{g, g^{2^\kappa}, g^{2^{\kappa 2}}, \dots\}$ have already been computed, it will take exactly $|I_b|$ group operations to compute the inner product in (3). This is done for every $b \in \{0, 1, \dots 2^\kappa - 1\}$, which means that the total amount of group operations for computing all the inner products is

$$\sum_{b=0}^{2^\kappa - 1} |I_b| = log_{2^\kappa}(2^t/l) = \frac{tlog_2(2) - log_2(l)}{\kappa log_2(2)} = \frac{t - log_2(2)}{\kappa} \approx \frac{t}{\kappa} \tag{4}$$

The amount of group operations for the outer product in (3) is approximately given by

$$\sum_{b=0}^{2^\kappa - 1} log_2(b) + 1 \leq \sum_{b=0}^{2^\kappa - 1} log_2(2^\kappa) + 1 = \kappa 2^\kappa + 2^\kappa \approx \kappa 2^\kappa \tag{5}$$

Henceforth by choosing $\kappa = log_2(t)/2$ we get that the total number of group operations needed to compute (3) is approximately

$$\frac{t}{\kappa} + \kappa 2^\kappa = \frac{2t}{log_2(t)} + \frac{log_2(t)}{2} t^{\frac{1}{2}} = \mathcal{O}(t/log(t))$$

Here we see that it is very important for l to be significantly smaller than N. If l would be as big as N, the calculations for the b_i's will take as long as every other group operation and thus the complexity of calculating the proof would increase.

4.2.3 Memory efficiency

One big issue about the method presented above is its consumption of memory space, since we need to store $log_{2^\kappa}(2^t) = \mathcal{O}(\frac{t}{k})$ elements of $\mathcal{B} = \{1, 2^\kappa, 2^{2\kappa}, \dots\}$. In this section we will discuss how to reduce this memory consumption to only $O(\sqrt{t})$ elements. The idea is to memorize only every $\kappa\gamma$-th element of $\mathcal{B}$. If we choose such a γ to be in $\mathcal{O}(\sqrt{t})$, we get the desired memory consumption. In this section we will discuss how to compute the proof, without having all elements of $\mathcal{B}$ in our hands.
Let $\mathcal{B}' = \{g^{2^0}, g^{2^{\kappa\gamma}}, g^{2^{2\kappa\gamma}}, \dots\}$ denote the memorized values, and $I_{b,j} = \{i \in I_b | i \equiv j \mod \gamma\}$ We obtain the following by permuting the terms in (3):

$$g^{\lfloor 2^t/l \rfloor} = \prod_{b=0}^{2^\kappa - 1} \Big(\prod_{i \in I_b} g^{2^{\kappa i}} \Big)^b = \prod_{b=0}^{2^\kappa - 1} \left(\prod_{j=0}^{\gamma - 1} \prod_{i \in I_{b,j}} g^{2^{\kappa i}} \right)^b = \prod_{j=0}^{\gamma - 1} \left(\prod_{b=0}^{2^\kappa - 1} \Big(\prod_{i \in I_{b,j}} g^{2^{\kappa(i-j)}} \Big)^b \right)^{2^{\kappa j}} \tag{6}$$

Note that by definition of $I_{b,j}$, every term in the innermost product of 6 will be precomputed, as $\gamma | (i-j)$. The amount of necessary group operations to calculate the proof will be $\frac{t}{\kappa} + \gamma\kappa 2^\kappa$, which is obtained by the same method shown above.

A further optimization can be achieved by splitting the product in (6) into two products. To do this we define $\kappa_1 = \lfloor \kappa/2 \rfloor$ and $\kappa_0 = \kappa - \kappa_1$, in order to simplify the notation we will denote $y_{b,j} = \prod_{i \in I_{b,j}} g^{2^{\kappa(i-j)}}$. We finally obtain that for each $j \in \{0, 1, \ldots \gamma - 1\}$

$$\prod_{b=0}^{2^\kappa - 1} y_{b,j}^b = \prod_{b_1=0}^{2^{\kappa_1}-1} \left(\prod_{b_0=0}^{2^{\kappa_0}-1} y_{b_1 2^{\kappa_0} + b_0, j} \right)^{b_1 2^{\kappa_0}} \cdot \prod_{b_0=0}^{2^{\kappa_0}-1} \left(\prod_{b_1=0}^{2^{\kappa_1}-1} y_{b_1 2^{\kappa_0} + b_0, j} \right)^{b_0} \tag{7}$$

Computing the product in (7) requires a total of $2(2^\kappa + \kappa 2^{\kappa/2})$ group operations.

4.2.4 The final algorithm for computing the proof

We now present the final algorithm for the computation of the proof, based on the formula (7). It requires $t/\kappa + \gamma 2^{\kappa+1}$ group operations and storage for $t/(\kappa\gamma) + 2^\kappa$ group elements.

Algorithm 6 Efficient computation of the proof: $\mathcal{O}(t/log(t))$

```
1: $\kappa_1 \leftarrow \lfloor \kappa/2 \rfloor$
2: $\kappa_0 \leftarrow \kappa - \kappa_1$
3: $x \leftarrow 1_G \in \mathbf{G}$
4: for $j \leftarrow \gamma - 1 \ldots, 0$ (descending order)
5:     $x \leftarrow x^{2^\kappa}$
6:     for $b \in \{0, \ldots, 2^\kappa - 1\}$
7:         $y_b \leftarrow 1_G \in \mathbf{G}$
8:     $b \leftarrow 1$
9:     $r \leftarrow 2^{(t-(j+1)\kappa) \bmod \kappa\gamma} \mod l$
10:    for $i \leftarrow 0, \ldots, \lfloor \frac{t-(j+1)\kappa}{\kappa\gamma} \rfloor + 1$
11:        $b \leftarrow \lfloor \frac{2^\kappa r}{l} \rfloor$
12:        $r \leftarrow 2^{\kappa\gamma} r \mod l$
13:        $y_b = y_b \cdot g^{2^{(\lfloor \frac{t}{\kappa\gamma} \rfloor - i)\kappa\gamma}}$
14:    for $b_1 \in \{0, \ldots, 2^{\kappa_1} - 1\}$
15:        $z = 1_G \in \mathbf{G}$
16:        for $b_0 \in \{0, \ldots, 2^{\kappa_0} - 1\}$
17:            $z = z \cdot y_{b_1 2^{\kappa_0} + b_0}$
18:        $x = x \cdot z^{b_1 2^{\kappa_0}}$
19:    for $b_0 \in \{0, \ldots, 2^{\kappa_0} - 1\}$
20:        $z = 1_G \in \mathbf{G}$
21:        for $b_1 \in \{0, \ldots, 2^{\kappa_1} - 1\}$
22:            $z = z \cdot y_{b_1 2^{\kappa_0} + b_0}$
23:        $x = x \cdot z^{b_0}$
24: return x
```

If, for example, one chooses $\kappa = \frac{\log t}{3}$ and $\gamma = \sqrt{t}$, this will give us.

$$\text{Operations:} \quad \frac{3t}{log(t)} + t^{1/2} 2^{log(t)/3+1} = \frac{3t}{log(t)} + 2t^{5/6} = \mathcal{O}\Big(\frac{t}{log(t)}\Big)$$

$$\text{Storage:} \quad \frac{3t}{log(t) t^{1/2}} + t^{1/3} = 3\frac{t^{1/2}}{log(t)} + t^{1/3} = \mathcal{O}(\sqrt{t})$$

4.3 Overhead minimization

Recall the evaluation of our VDF through $\texttt{eval}_{\texttt{pp}}$ (Algorithm 1): we first perform the t squarings, followed by the calculation of the corresponding proof. After having performed the t squarings, we know that we already have the correct output, but other parties would not believe us as long as we haven't outputed the proof. The time elapsed between the calculation g^{2^t} and the calculation of the proof $\pi = g^{\lfloor 2^t/l \rfloor}$ is called overhead. Our goal in this section is to explain a method how to minimize the overhead when evaluating the VDF. The drawback of this method is that the length of the proof will be longer and will therefore need more space and take longer to verify. The method is due to [Wesolowski(2018)].

By defining $\Delta = \delta t$ to be the time necessary to perform the t squarings (one squaring takes δ time), we say that the overhead is Δ/ω. This makes sense as the overhead is basically the time needed to compute the proof, which takes $\mathcal{O}(t/log(t))$ group operations compared to the squarings. We will use two different threads and minimize the overhead from Δ/ω to Δ/ω^n, where $n \in \mathbb{N}$. Instead of outputting a proof consisting of a single group element π, the proof will be a tuple of the form $(\pi_1, \pi_2, \dots \pi_n)$ as well as $(n-1)$ small prime numbers. By definition, ω depends on t, since $\omega \in \mathcal{O}(log(t))$. To simplify the following procedure, we assume ω is a constant.

The method consists in proving intermediate squaring results while performing the g^{2^t}. We explain the method for $n = 2$:

- Set $t_1 := t\frac{\omega}{\omega+1}$ and start evaluating the VDF
- As soon as we performed the t_1-th squaring, we store $g_1 := g^{2^{t_1}}$. On the same thread, we continue performing the $t_2 := t - t_1 = t - t\frac{\omega}{\omega+1} = t\frac{1}{\omega+1}$ remaining squarings.
- On another thread, we will calculate the proof $\pi_1 = g^{\lfloor 2^{t_1}/l_1 \rfloor}$, where $l_1 = H_{prime}(g, g_1)$. This proof will prove that g_1 is the correct output to the t_1 squarings of g.
- Note that performing the t_2 remaining squarings and computing the partial proof π_1 will take (approximately) the same amount of time.
  t_2 squarings: $\delta t_2 = \delta t\frac{1}{\omega+1} = \Delta\frac{1}{\omega+1}$ computing π_1: $\delta t_1\frac{1}{\omega} = \delta t\frac{\omega}{\omega+1}\frac{1}{\omega} = \Delta\frac{1}{\omega+1}$
- We now need to prove that the last t_2 squarings produced the correct output, i. e. compute a proof π_2 for the statement $y = g_1^{2^{t_2}} = g^{2^{t_1}2^{t_2}} = g^{2^{t_1+t_2}} = g^{2^t}$.
- The computation of π_2 will take $\delta t_2/\omega = \delta t\frac{1}{\omega(\omega+1)} \leq \Delta/\omega^2$. The overhead has therefore been reduced from Δ/ω to Δ/ω^2.

Note that the proof now consists of (l_1, π_1, π_2). The verification of the final result $y = g^{2^t}$ will take longer than with the conventional method and goes as follows:

- Calculate $g_1 = \pi_1^{l_1} g^{2_1^t \mod l_1}$
- Get $l_2 = H_{prime}(g_1, y)$
- Verify that $y = \pi_2^{l_2} g_1^{2^{t_2} \mod l_2}$

The above described procedure can be generalized for any $n \in \mathbb{N}^*$. In that case we would output a proof of the form $(l_1, \dots l_{n-1}, \pi_1, \dots \pi_n)$. We would then define $t_i = t\omega^{n-i}\frac{\omega-1}{\omega^n-1}$ for $i = 1, \dots n$. Note that we have

$$\sum_{i=1}^{n} t_i = \sum_{i=1}^{n} t\omega^{n-i}\frac{\omega-1}{\omega^n-1} = t\omega^n\frac{\omega-1}{\omega^n-1}\sum_{i=1}^{n}\frac{1}{\omega^i} = t\omega^n\frac{\omega-1}{\omega^n-1}\frac{(\omega^n-1)\omega}{(\omega-1)\omega^{n+1}} = t\omega^n\frac{1}{\omega^n} = t$$

We can define recursively $g_{i+1} = g_i^{t_{i+1}}$ and π_i the proof of it, where $g_0 = g$. Furthermore, every time we are finished with computing g_i we are also finished with computing the proof $\pi_i = g_{i-1}^{\lfloor 2^{t_i}/l_i \rfloor}$. Indeed we have that the time to produce the proof π_i is $t_i/\omega = t\omega^{n-i}\frac{\omega-1}{\omega^n-1}\omega^{-1} = t\omega^{n-(i+1)}\frac{\omega-1}{\omega^n-1} = t_{i+1}$. This is only true because of the assumption that ω is constant, since ω is empirically defined as such that the proof takes Δ/ω time to compute. Therefore saying that computing π_i takes t_i/ω time is an approximation, since ω would depend from t_i and thus change at each step. By the results we have seen so far, choosing $\omega = log(t)$ seems reasonable, as the time we need to compute the proof is in $\mathcal{O}(t/log(t))$.

5 Implementation and results

We can now present details of our practical implementation of the verifiable delay function. Our main focus was speed and expanding our intuition of the different parameters. We therefore took a few liberties in our design, perhaps at the expense of some security. The following are some design choices we have made.

- We chose to implement the VDF in C, mainly because our prior experience with it and the finer control it provides. Additionally we used the GNU Multiprecision Arithmetic Library and OpenSSL libraries both for their efficiency and established reliability.
- For simplicity, we used $k = 128$ as the security parameter but it is only used within the H_{prime} procedure. However, it should also be used to determine a minimum size for the RSA key.
- Because of the many difficulties associated to correctly implementing cryptographically secure hashing, we decided not to implement the initial hashing in $y \leftarrow (H(x))^{(2^t)}$. While weakening the security, for benchmarking purposes it is insignificant as it would only add a constant time operation.
- Rather than arbitrarily take $\kappa = log(t)/3$ as suggested in [Wesolowski(2018)], we used the Newton-Raphson method to minimize the number of group operations $t/\kappa + \gamma 2^{\kappa+1}$ for a given t and $\gamma = \sqrt{t}$. The choice of γ provides a fair middle ground between memory usage and speed.
- Because the construction of the VDF applies to any finite abelian group of unknown order, we can easily change the underlying group by linking to a different group source file.
- As suggested in [Boneh et al.(2018)Boneh, Bünz, and Fisch], it is possible to reduce the overhead proof calculation by splitting it into parallelizable blocks. However, we chose Wesolowski's approach of splitting the squarings into multiple parts and calculating smaller proofs on different threads. This is more practical when less cores are available.
- All the following results were produced using a quad-core Intel Core i7-4970k running at 4.0 GHz without Turboboost.

We now analyze some of the results obtained.

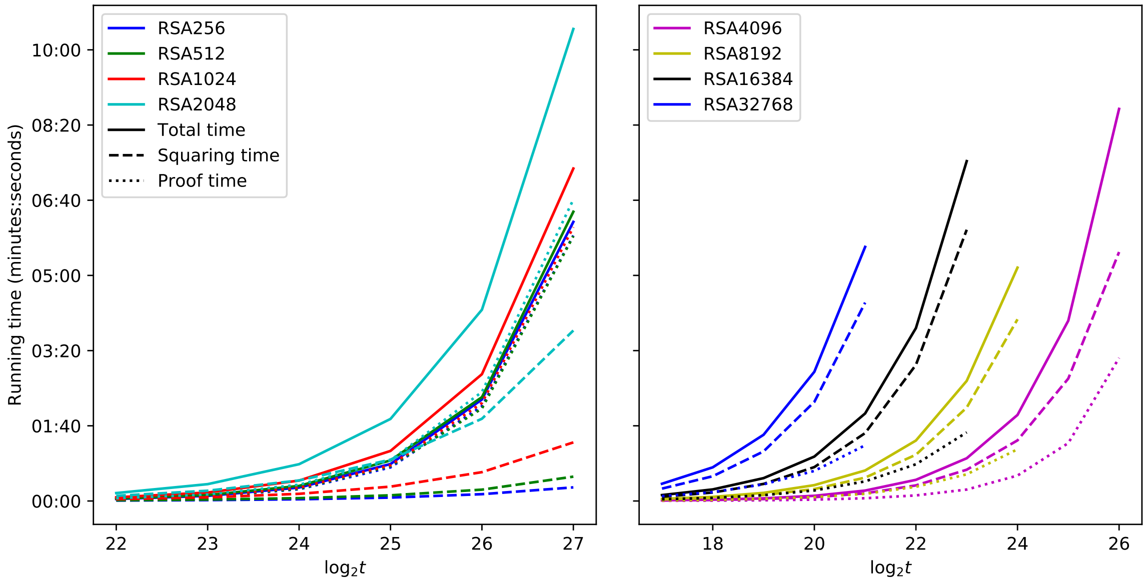

Figure 1: Running time of the VDF with different RSA moduli

Figure 1 explores the amount of time to evaluate the VDF using different RSA key sizes. The left graph shows that keys up to 2048 bits are not useful since the proof generation will take up more time than the squaring procedure. The right graph
We also can determine the desired t so that the computation of the VDF takes the desired amount of time. The next graphs shows the effect of splitting the computation into multiple segments in order to minimize the overhead induced by the proof calculation.

The next figure depicts the speed up obtained when using the overhead minimization technique detailed at the end of Chapter 4.

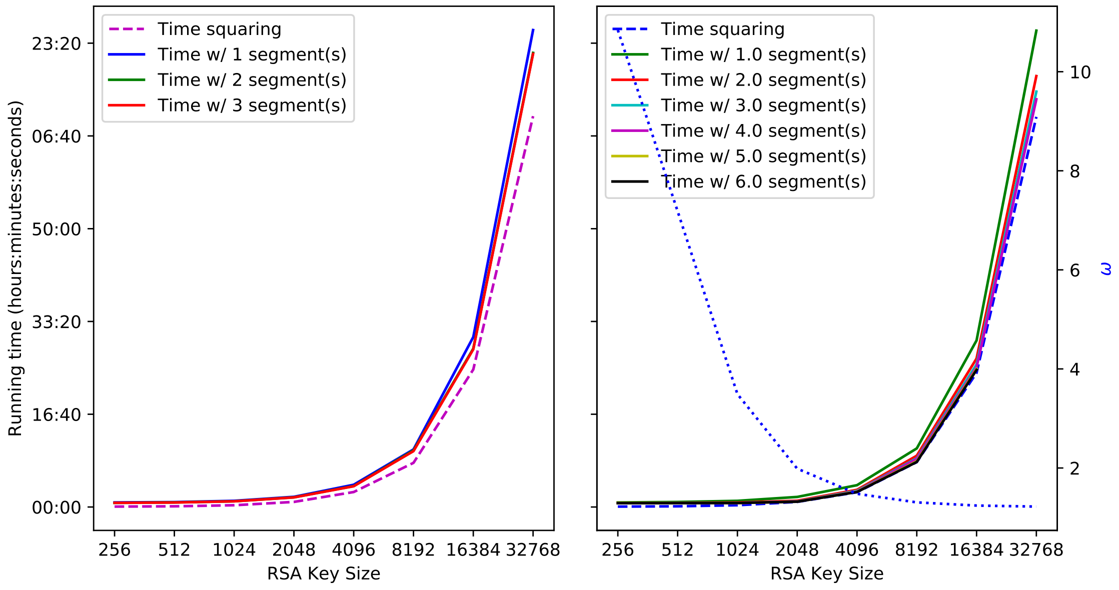

Figure 2: Runtime of the VDF for $t = 2^{25}$ with various segments and ω values.

In the left graph, we chose $\omega = \log t \approx 7.5$ as suggested at the end of Chapter 4. Even when using multiple segments however, `eval` must wait for the first proof to finish because the running time does not improve with more than two segments. This shows that careful considerations must be made when choosing ω, and that it depends on t as well as the RSA key size. We deduce that ω must be smaller in order to give more time to compute the first proof and effectively use a larger n.

The graph on the right shows the improvements in run time when setting $\omega = \frac{\text{time for squaring}}{\text{time for proof}}$. This is determined after having first computed the VDF sequentially. Because ω is small, the last segments are still quite large and we must now wait for the last proof to finish. This shows the inversely proportional relationship between ω and the number of segments and means that we must increase n when we use smaller ω. We would obviously increase the proof size though.

6 References

[Boneh et al.(2018)Boneh, Bünz, and Fisch] Dan Boneh, Benedikt Bünz, and Ben Fisch. A survey of two verifiable delay functions. *IACR Cryptology ePrint Archive*, 2018:712, 2018. URL `https://eprint.iacr.org/2018/712`.

[Rivest et al.(1996)Rivest, Shamir, and Wagner] R. L. Rivest, A. Shamir, and D. A. Wagner. Time-lock puzzles and timed-release crypto. Technical report, Cambridge, MA, USA, 1996.

[Shoup(2008)] V. Shoup. *A computational introduction to Number Theory and Algebra.* `https://www.shoup.net/ntb/ntb-v2.pdf`, 2008.

[Wesolowski(2018)] Benjamin Wesolowski. Efficient verifiable delay functions. *IACR Cryptology ePrint Archive*, 2018:623, 2018. URL `https://eprint.iacr.org/2018/623`.