Verifiable Delay Functions

Luca Bonamino & Boghos Yoseef & Mathias Alibert

Feb 3, 2024

Master in Information and Computer Sciences
Université du Luxembourg
2 Av. de l'Universite, 4365 Esch-Belval
Esch-sur-Alzette , Luxembourg

ABSTRACT

In this report, we compare two Verifiable Delay Functions, [6] (which we call Pietrzak's VDF) and [7](which we call Wesolowski's VDF), based of the time lock puzzle. The comparison is done in terms of the complexity of their evaluation and verification; both functions are expected to have the lowest complexity possible, but by still keeping a minimum number of sequential steps for the evaluation functions.
We show analytically that Pietrzak's VDF has a worse evaluation and verification time complexity than Wesolowski's and we implement and analyse their time complexity evolutions experimentally by computing the macrostate-like plots of their evaluations and verification functions.
In addition, we provide some considerations and alternatives to the procedures presented on the optimization of the generation of the proofs of both VDF's papers. For [6], we provide a generalisation with an analytical proof for their statements, while for [7], we provide an alternative procedure with a better time complexity and a similar space complexity if compared on the binary numerical base, but a worse time complexity when compared on a different basis. We are sure that a better time complexity for our method can be proven, but we are also sure that we cannot do better than what the paper proposes since the choice of a parameterizable base 2^κ, is what enabled the researchers to decrease the time complexity. We nevertheless try our best to provide an appropriate analytical comparison to their method.

1. INTRODUCTION

1.1 Verifiable Delay functions

A Verifiable Delay Function is a function that requires a given number of steps to be computed but takes significantly fewer to verify its output. When computing this output, a proof of computation would be produced with it and such proof will be publicly verifiable.
The properties of soundness and completeness must always be satisfied, meaning that if a fake proof is produced, its verification must not succeed, while if the provided proof is correct, the verification must always succeed.
An additional property the VDF may have is the unicity of its output, and hence, second pre-image resistance; however, this is not a strict condition for it to be valid.

1.2 Applications of Verifiable Delays Functions

1.2.1 Secure generation of random beacon

VDFs are tools that can have many different applications, one of which is the generation of a shared random value.

In such a scenario, each contributor of the random value would openly commit its value and after all contributions are collected they would all be combined with an XOR operation and the result would then run through a VDF.
Let n be the nonce generated by xoring the nonces n_i of each participant and (m, π) be the output and the proof of computation of the VDF on n after T sequential steps.
The proof π can be verified by anyone in less than T sequential steps.

Since commitments of the nonces (n_is) are opened, a malicious participant could manipulate the value of n. However, if a commitment deadline is fixed, they would not know how to manipulate it in a way that could benefit them since they would have to run the VDF on the manipulated value on their own before committing their nonce - having a deadline for the commitment of the nonces is set, they would not have enough time to do it.
Suppose that j is the malicious agent. j waits for all the n_is to be committed and before committing his nonce, he computes $n^{'} = \bigoplus_{i \neq j}^{k} n_i$ and chooses to remove the contribution of the participant 0 by submitting n_0 as his nonce, causing

$$n = n^{'} \oplus n_0 = \bigoplus_{i=1}^{k} n_i$$

He would then have to get the final nonce m, run the VDF on n, and verify if m benefits them, otherwise he has to choose a different nonce.
A deadline on the commitment of the nonces ensures that the malicious participant is not able to compute $m^{(1)}, m^{(2)}, ...$ and then choose a useful $m^{(i)}$ and the corresponding nonce $n_j^{(i)}$.
Moreover, when the malicious participant fakes the proof without computing it, they would be discovered and penalties can be applied to them.

Notice that this process could be adapted to generate the nonce in a decentralized manner.
In fact, instead of committing their nonce to a central board, each agent could send them to each other,

then after receiving the nonces of all other participants, each one would compute n by $\bigoplus_i^k n_i$ and run the VDF on it to get m with the respective proof π and broadcast it. Assuming the receiving of the proof for all the participants requires less time than to compute the proof, no participant would be able to share a valid proof that they didn't compute.

Decentralised distributed nonce used as proof of possession

We can imagine a protocol to prove the possession of some data using the decentralized generation of a nonce using a VDF.
In such an instance, the nonce m would be used as a session nonce to ensure the non-injectivity property of the information provided by each participant.
Suppose a scenario where we would expect that ϕ of the k participants to own some particular data in a certain time slot, then after generation of m within the k agents, the $\phi \leq k$ participants sign their data with the nonce m and broadcast the signature to the network. Then, they would be able to show that during that session, they possess the piece of data they claim to own.

1.2.2 Secure Random beacon using VDF as a proof of stake (PoS) reinforcement

Building on the use of VDFs for random beacon generation described in Section 1.2.1, this idea has been extensively discussed in the context of proof of stake (PoS) systems. For example, in the *Ethereum* blockchain, validator selection depends on a pseudo-random beacon. Currently, this beacon is implemented via the RANDAO commit–reveal scheme, without the use of VDFs, which makes the process susceptible to manipulation (a problem already discussed in Section 1.2.1). The integration of VDFs to strengthen randomness and security was part of the early Ethereum 2.0 research roadmap, but has not yet been deployed on the mainnet [5, 4, 1].

1.2.3 Proof of sequential work (PoSW) instead of proof of work (PoW)

Due to their sequentiality property and the presence of their proof system allowing the production of a proof of having computed the output in a minimal amount of sequential steps, VDFs are used for the concept of Proof of sequential work (PoSW) (applied in the Chia blockchain) replacing the Proof of Work (PoW), which in contrast, does not provide any efficiently verifiable proof of computation.

1.3 Notations and premises

Here we establish some notations that are used in the next sections

- The delay of the Verifiable Delay Functions is defined as t while the security parameter (the bit size of the modulus of the considered group on which the VDFs are defined) is defined as λ
- Unless specified differently, to be more concise in the text, we use the notion of *time complexity* and the number of *group multiplications* interchangeably.

2. VDFS IN GENERAL

A General Verifiable Delay Function VDF is defined as the tuple of algorithms $(Setup, Eval, Verify)$[3], where:

Setup$(\lambda, T) \rightarrow$ *pp*: The setup function takes as input a *security* parameter λ and a *delay* parameter **T** and outputs *public parameters* **pp**. Those public parameters are comprised of an evaluation key ek and a verification key vk, which define the domain and the range of the *Eval* function of the VDF and are also used to verify and compute the VDF.

Eval$(ek, x) \rightarrow (y, \pi)$: Takes an input x and the public parameters (ek) generated by **Setup** and outputs y and a proof π.

Verify$(ev, x, y, \pi) \rightarrow \{True, False\}$**:** a function that takes as input ev, x, y, π(the outputs of both **Eval** and **Setup**) and computes (efficiently) a boolean output that indicates whether y is a valid output for x in the domain and range of the *Eval* function.

2.1 Some Properties of VDFs

According to[3], VDFs can have certain properties such as the following:
Sequentiality
Honest Parties can call Eval(***pp***,x) and produce the pair (y, π) in t-sequential steps or time, such that no malicious participant, with a parallel-machine that has a polynomial number of processors can recognize y from a random string in fewer steps or time than t. Formally, a sequential function $f : X \rightarrow Y$ *is a* (t, ϵ)*-sequential function if for all* $\lambda = \mathcal{O}(log(|X|))$, *if the following conditions hold:*

1. $\exists$ algorithm A such that $\forall x \in X$, *it evaluates f in parallel time t using* $poly(t, \lambda)$ *processors.*
2. for all A that run in parallel time strictly less than $(1 - \epsilon) \cdot t$ with $poly(t, \lambda)$ processors:

$$Pr\left[\ y_A = f(x) \quad y_A \xleftarrow{\$} A(\lambda, x), x \xleftarrow{\$} X\ \right] \leq negl(\lambda)$$

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

$$Pr\left[\begin{array}{c|c} Verify(vk, x, y, \pi) = True & \mathbf{pp} = (ek, vk) \xleftarrow{\$} Setup(\lambda, t) \\ y \neq Eval(ek, x) & (x, y, \pi) \xleftarrow{\$} A(\lambda, \mathbf{pp}, t) \end{array}\right] \leq negl(\lambda)$$

Correctness
A VDF is correct if for all λ, t *parameters* $(ek, vk) \xleftarrow{\$} Setup(\lambda, t)$, *and all* $x \in Domain(Eval)$ *if* $(y, \pi) \xleftarrow{\$} Eval(ek, x)$ *then* $Verify(vk, x, y, \pi) = True$

Unicity/non-injectivity
Injectivity here means simply that for every x there is a unique mapping to y in the domain of $Eval(ek, x)$. Formally:

$$\forall x_1, x_2 \in X, Eval(ek, x_1) = Eval(ek, x_2) \implies x_1 = x_2$$

2.2 Time-lock puzzles

A time-lock puzzle is a cryptographic primitive that provides a delayed encryption property, generating cipher text that can only be decrypted after a predicted amount of time. By constructing the encryption function in such a way that it cannot be computed with less than the provided number of steps, the availability of the cipher text is delayed and hence the decryption of it can only be done after a certain time. For the puzzle to be useful, the decryption of the cipher text should take fewer steps than the encryption.

The two VDFs that are present in this report are constructed on the *RWS time-lock puzzle*, which is one of these puzzles.

Let (N, x, T) such that $N = pq$ is an *RSA modulus* and $x \leftarrow \mathbb{Z}_N^*$ and $t \in \mathbb{N}$, the solution of the *RWS time-lock puzzle* is given by

$$y = x^{2^t} \mod N \tag{2.1}$$

without the knowledge of the order of the proper group of $\mathbb{Z}_N$, the only way of calculating y, is to compute the t modular squares of x

$$x \to x^2 \to \left(x^2\right)^2 = x^{2^2} \to x^{2^3} \to .... \to x^{2^t} \mod N \tag{2.2}$$

However, with the knowledge of the order of the proper group $|\mathbb{Z}_N^*| = \phi(N) = (p-1)(q-1)$, using the *Euler's theorem*, y can be computed as

$$y = x^{2^t \mod \phi(N)} = x^{k\phi(N)+r} = \left(x^{\phi(N)}\right)^k x^r = x^r \mod N \tag{2.3}$$

where $k = \lfloor \frac{2^t}{\phi(N)} \rfloor$ and $r = 2^t \mod \phi(N)$.

Setting the secret key as $sk = \phi(N)$ and the public key $pk = N$, this can be used as a delayed encryption scheme.

This is however not enough to construct a verifiable delay function, since in a VDF it is required that the verification of the output can be done publicly and not only made possible by the possession of the secret key.
The modification into a VDF is achieved by constructing proof systems allowing the generation of verifiable proofs with the generation of the output y.

The two VDFs presented in this report solve this problem by defining one proof system each; we explain their construction and compare their time and space complexities.

3. PIETRZAK'S VDF

The first VDF that we analyse is the Pietrzak's VDF [6], an RWS time-lock based VDF which bases itself on the positive quadratic residue domain QR_N^+ rather than in the more general RSA group. The particularity of this VDF is its halving-style protocol used in the proof system whose strong point is the unconditional soundness; compared with [7] whose soundness relies on the root-finding problem.

3.1 Setup and building blocks

The VDF restricts itself on the set of positive quadratic residues $QR_N^+ = \{z \in \mathbb{Z}_N^* \text{ such that } |z^2 \mod N| \}$ where $N = pq$ is the product of two safe primes: such that $p^* = (p-1)/2$ and $q^* = (q-1)/2$ are also primes and $|x \mod N| = \min(x, N-x)$.
The multiplication of two elements of such space is defined as

$$a \circ b = |a * b \mod N| \tag{3.1}$$

Considering this setup, the VDF is defined by the functions $(setup, gen, eval, verify)$ given below

setup$(\lambda, t) \to (\lambda, N, t) \in \mathbb{N}_{\geq 2}$ x $\mathbb{N}$ x $\mathbb{N}$.
Takes as input the security parameter λ and the delay t, generated the primes p and q to calculate N and return an object (represented here as a tuple) of public parameters.

gen$((\lambda, N, t)) \to x \in QR_N^+$
Takes as input the public parameters and generates a random value in QR_N^+.

eval$((\lambda, N, t), x) \to (y, \pi)$
Takes as input the public parameters an $x \in QR_N^+$ and computes the time-lock puzzle (2.2) and the corresponding proof π. y is in QR_N^+ and we will see in the next sections that π is in $\left(QR_N^+\right)^\psi$, if $t = 2^\psi,\ \psi \in \mathbb{N}_{\geq 1}$.

verify$((\lambda, N, t), x, y, \pi) \to verification \in \{True, False\}$
Takes as input the public parameter, $x \in QR_+^+, y \in \mathbb{Z}$ and the proof π and verifies if π is the proof corresponding to the y or not. Returns $True$ if it is the case and $False$ otherwise. To notice that y does not need to be in QR_N^+ since we allow the input to be an invalid proof, if $y \notin QR_N^+$, the verification function would output $False$.

Remarks
We state here some different choices that we made compared to the paper in the implementation of these functions

- One of the justifications that the paper provides for using QR_N^+ instead of $\mathbb{Z}_N^*$, is the fact that the membership of QR_N^+ can be verified easier by verifying the positivity of it and checking that its Jacoby symbol is equal to 1. We don't see a way of efficiently doing this without having the factorization of N (the check weather $x \in QR_N^+$ is always done publicly and p, q are not part of the public parameters).

- We restrict ourselves to values of the delays as power of 2. The paper also dose it when focusing on the optimization of the computation of the proof.

3.2 Proof System

To produce and verify the proof the following interactive halving-style protocol is proposed.
Let (λ, N, x, t, y) be the common input to a prover and a verifier, the interaction is as follows

1. If $t = 1$, the verifier return $True$ if $y = x^2 \mod N$, otherwise it returns $False$. If $t > 1$, the protocol continues to the next step

2. The prover computes $\mu = x^{2^{t/2}} \mod N$ and sends it to the verifier

3. If $\mu \notin QR_N^+$, the verifer outputs $False$, otherwise it samples $r \xleftarrow{\$} \mathbb{Z}_{2^\lambda}$ and sends it to the prover

4. The pover and verifier compute (N, x^*, t, y), where

$$x^* := x^r \mu \mod N \tag{3.2}$$
$$y^* := \mu^r y \mod N \tag{3.3}$$

   If $y = x^{2^t} \mod N$,

$$y^* := \mu^r y = \left(x^{2^{t/2}}\right)^r x^{2^t} = x^{r2^{t/2}+2^t} = \left(x^{r+2^{t/2}}\right)^{2^{t/2}} = (x^*)^{2^{t/2}} \mod N$$

   If it is the case, the verifier outputs $True$, otherwise it outputs $False$.

3.2.1 Transformation to a non interactive iterative system

To transform such interaction to a non-interactive system, the nonce r is replaces by an hash function H, applying the $sha256$ and then truncating at the first λ bits. For soundness and completeness reason, the input to H is the tuple $(x, t/2, y, \mu)$.
The procedure is transformed into an iterative procedure, by iteratively halving t and at each step i, calculating $\mu_i = x_i^{2^{t/2^i}} \mod N$ and getting r_i by $r_i \leftarrow H_\lambda(x_i, t/2^{i-1}, y_i, \mu_i)$ to update x_i and y_i for i going from 1 to $\psi = \log_2 t = \log_2(2^\psi)$ with the starting point of $(x_1, y_1) = (x, y)$.
In such way, the prover produces the proof by collecting all the generated values μ_is.
The verifier then used $\pi = (\mu_i)_i$ to find the corresponding r_i and update the x_i and y_i as the prover. After the having processed has ended, the all array $\pi = (\mu_i)_i$, it verifies if $y_{\psi+1} = x_{\psi+1}^2 \mod N$

Algorithm 1: Algorithm for Pietrzak's naive Proof computation

Data: x, y, and public parameters (λ, N, t)
Result: Proof π
$u_x = x$
$u_y = y$
$\mu_{list} = []$
while $t > 1$ **do**
  $t_old \leftarrow t$;
  $t \leftarrow t/2$;
  $\mu \leftarrow u_x^{2^t}$;
  $r \leftarrow H_\lambda(u_x, 2^{t_old}, u_y, \mu)$
  $u_x \leftarrow u_x^r \circ \mu \mod N$;
  $u_y \leftarrow \mu^r \circ u_y \mod N$;
  $\mu_{list}.append(\mu)$;
return μ_{list};

Since the Verification follows an halving algorithm, it computes $\log_2(t)$ loops and at each loop it computes 2 group operations. After the last iteration, it computes a last group operation $u_x^2 = u_x \circ u_x \mod N$ to do the final check, this resulting in a final complexity of

$$T_{Piet,Verif} = 2\log_2(t) + 1 = O(\log_2(t)) \tag{3.4}$$

Algorithm 2: Algorithm for Pietrzak Verification

Data: x, y, π, and modulo public parameters (λ, N, t)
Result: $True$ or $False$
$u_x = x$
$u_y = y$
for μ ***in*** π **do**
  $r \leftarrow H_\lambda(u_x, 2^t, u_y, \mu)$;
  $u_x \leftarrow u_x^r \circ \mu \mod N$;
  $u_y \leftarrow \mu^r \circ u_y \mod N$;
  $t \leftarrow t/2$
if $u_y == u_x^2 \mod N$ **then**
  return $True$;
else
  return $False$;

We notice that calculation recomputes the same group multiplications that are computed while calculating y. See for example that $y = x^{2^t} = \left(x^{2^{t/2}}\right)^{2^{t/2}} = (\mu_1)^{2^{t/2}}$.
Therefore, it is possible to re-use the first $\mu_1 = x^{2^t}$ in the computation of the proof without having to re-calculate it. Such method reduces the complexity of the computation of the proof from t sequential squares to

$$T_{Piet,Proof} = \sum_{i=1}^{\log_2(t/2)} (t/2^i + 2) = 2\log_2(t/2) + t \sum_{i=1}^{\log_2(t/2)} 1/2^i \tag{3.5}$$

It is also possible to re-use all the μ_is computed during the calculation of y to generate the proof π. In the next section we show Pietrzak's proposition for this problem.

3.3 Optimization of proof generation

In addition to the small improvement that we have just mentioned (which is the one that we have implemented), the paper [6] succeeds to give a further improvement by describing a method to re-use multiple μ_is. A general formulation is never the less not given, therefore, since we did not find the procedure trivial, we have established a general formula for it. We however, got stuck on the extraction of the μ_is due to k-product of binomials, which, not having a general useful expansion, suggests us that our formulation is not useful for our problem.

Let $\hat{z}$ be such that $x^{\hat{z}} = z,\ z \in QR_N^+$, the paper constructs the following sequences

$$\hat{\mu}_i = \hat{x}_i 2^{t/2^i} \tag{3.6}$$
$$\hat{x}_{i+1} = r_i \hat{x}_i + \hat{\mu}_i \tag{3.7}$$
$$\hat{y}_{i+1} = r_i \hat{\mu}_i + \hat{y}_i \tag{3.8}$$

with $\hat{x}_1 = 1$ and $\hat{y}_1 = 2^t$.

We claim that for a general k, the non-inductive sequence of $\hat{\mu}_k$ is

$$\hat{\mu}_k = 2^{t/2^k} \prod_{i=1}^{k-1} \left(r_i + 2^{t/2^i}\right) \tag{3.9}$$

(the proof of this statement is found in the appendix's section 7.1)

Using the fact that $t = 2^{\psi}$

$$\hat{\mu}_k = 2^{2^{\psi}/2^k} \prod_{i=1}^{k-1} \left(r_i + 2^{2^{\psi}/2^i} \right) \tag{3.10}$$

$$= 2^{2^{\psi-k}} \prod_{i=1}^{k-1} \left(r_i + 2^{2^{\psi-i}} \right) \tag{3.11}$$

$$= \prod_{i=1}^{k-1} \left(r_i 2^{2^{\psi-k}} + 2^{2^{2\psi-k-i}} \right) \tag{3.12}$$

and μ_k would then be

$$\mu_k = x^{\hat{\mu}_k} = x^{\prod_{i=1}^{k-1} \left(r_i 2^{2^{\psi-k}} + 2^{2^{2\psi-k-i}} \right)} \tag{3.13}$$

This is where we get stuck since we need to extract the indexes final indexes j of x^{2^j} evaluating from the binomial products.
The best that we can do using our formula, is therefore to select an index $s < t$, small and expand by hand all $\mu_i, i \leq s$ using (3.13) to be able to extract the rs and the js.

3.4 Properties of Pietrzak's VDF

Pietrzak's VDF satisfies the completeness, the sequentiality and the soundness properties.

3.4.1 Completeness

For any delay t and security parameter λ, the probability for the $verify$ function to accept a valid proof is 1

$$Pr[verify((\lambda, N, t), x, y, \pi) = True] = 1$$

where $(\lambda, N, t) \leftarrow setup(\lambda), x \leftarrow gen((\lambda, N, t))$ and $(y, \pi) \leftarrow eval((\lambda, N, t), x)$

3.4.2 Sequentiality

The property of sequentiality is expressed in terms of the probability of the verifier to accept a proof even if an adversary was able access some pre-computed values. The VDF is sequential if the *eval* function is defined such that this probability is non-negligible on the security parameter λ. This is to say that, even if an adversary has access to some pre-computed values, it can only be as fast as the given sequential time.

3.4.3 Soundness

The soundness property is expressed in terms of the probability of the $verifier$ function to accept a proof of a wrong statement. The VDF is sound if the $verify$ function is such that such probability is negligible.

4. WESOLOWSKI'S VDF

The other VDF that we analyse is Wesolowski's trapdoor VDF [7] which claims to have a total work time (production of the output and the proof) of

$$T_{tot} = T_{ouput} + T_{proof} = t + \frac{t}{\log(t)} = \left(1 + \frac{1}{\log(t)}\right) t \tag{4.1}$$

and a verification of a complexity of two group exponentiations.
It is possible to implement their construction is such a way reduce the T_{tot} allowing to start part of the proof before the computation of the full output finishes. This is done by setting an intermediate output at $t^* < t$ steps and computing $y_{t^*} = x^{2^{t^*}}$ and the proof π_1 for it in parallel of the computation of the rest of the output $y = y_{t^*}^{2^{t-t^*}}$. In such way the computation of y and of π_1 finish more or less at the same, leaving to wait the time to compute the remaining part of the proof π_2 for $y_{t^*}^{2^{t-t^*}}$. π_1 and π_2 would then be both verified.
Let $T_{out:t^*}$ [resp. $T_{out:(t-t^*)}$] be the time needed to compute y_{t^*} [resp. $y_{t^*}^{2^{t-t^*}}$] and $T_{proof:\pi_i}$ the time needed to compute the proof π_i, the total time then becomes

$$T_{out:opt} = T_{out:t^*} + \max\left(T_{proof:\pi_1} + T_{out:(t-t^*)}\right) + T_{proof:\pi_2} \tag{4.2}$$

An additional property of [7] is the presence of the trapdoor, allowing the participants knowing $|\mathbb{Z}_N^*|$ to produce the output in less time, solving the RWS time-lock puzzle faster using (2.3).

4.1 Setup and building-blocks overview

Since the VDF is constructed on the RWS time-lock puzzle, we restrict ourselves to the group $\mathbb{Z}_N^* \setminus \{-1, 1\}$ where $N = p * q$ and $\phi(N) = (p-1)(q-1)$ with $p, q \in \mathbb{Z}$ being two primes smaller or equal to $(2^\lambda - 1)/2$, in such a way that $N \leq 2^\lambda - 1$. We call this group G_N.
The input we wish to evaluate the VDF on, is an integer which may not be in G_{pk}, we therefore use an hash function H_g whose image is G_N. The definition of H_G is given in the algorithm 3.

Algorithm 3: Algorithm to compute H_G

Data: $x \in \mathbb{Z}$, and modulo N
Result: $H_G(x)$
$h \leftarrow int(sha256(\text{"}residue\text{"}||x.encode()) \mod N$;
while $gcd(h, N)! = 1$ **do**
  $h \leftarrow (h + 1) \mod N$;
return h;

Considering this setup, the VDF is defined by the following functions

keygen$(\lambda \in \mathbb{N} \setminus \{0\}) \rightarrow (N, \phi(N))$.
Generates the public key pk and secret key sk both of bit size of $\lambda/2$.

trapdoor$((\lambda, N, \phi(N), t),\ x \in \mathbb{Z}) \rightarrow (y, \pi) \in G_N^2$. Takes the integer x, maps to the space of G_N as $x^* \leftarrow H_G(x)$ and solves the time-lock puzzle on x^* using (2.3) and computes the proof π using $\phi(N)$.

eval$((\lambda, N, t),\ x \in \mathbb{Z}) \rightarrow (y, \pi) \in G_N^2$.
Takes the integer x, maps to the space of G_N as $x^* \leftarrow H_G(x)$ and solves the time-lock puzzle using (2.2), hence by doing the t squares.

verify$((\lambda, N), y \in \mathbb{Z}, \pi \in \mathbb{Z}) \rightarrow$ verification $\in \{True, False\}$
Checks if y is the output associated with x using the proof π. The inputs y and π are elements of $\mathbb{Z}$ and not necessarily of G_N, if these elements are not in G_N the verification will simply not evaluate to $True$.

In addition, the function **setup** is defined, which takes as parameters the security parameter λ, the delay t and a boolean stating if the participant running the setup is allowed to use the trapdoor or not. The function generates the keys with $(N, \phi(N)) \leftarrow keygen(\lambda)$, and it returns $(N, \phi(N), \lambda, t)$ if the participants is allowed to use the trapdoor, otherwise it generates (N, λ, t).

4.2 Proof System

To produce and verify the proof, the following interactive protocol is proposed.
Let $P(2\lambda)$ be the set containing the first $2^{2\lambda}$ prime numbers. Once, y is generated by the prover, the tuple $\left(G, g \leftarrow H_G(x), y = g^{2^t} \mod N, t\right)$ is used as input to the following interaction

1. The verifier samples a prime l from $P(2\lambda)$, $l \xleftarrow{\$} P(2\lambda)$

2. The prover computes the proof as $\pi = g^{\lfloor 2^t/l \rfloor}$ and sends it to the verifier

3. The verifier computes $r = 2^t \mod l$ and $\pi^l g^r \mod N$.
   If $\pi^l g^r$ is such that

$$\begin{aligned}\pi^l g^r &= \left(g^{\lfloor 2^t/l \rfloor}\right)^l g^{2^t} \mod l \\ &= \left(g^{\lfloor \frac{2^t - 2^t \mod l}{l} \rfloor}\right)^l g^{2^t} \mod l \\ &= g^{2^t - 2^t \mod l} g^{2^t} \mod l = g^{2^t} = y \mod N\end{aligned}$$

   the verifier outputs $True$, otherwise it outputs $False$.

4.2.1 Transformation to a non interactive system

To transform such interaction to a non-interactive interaction, a Fiat-Shamir transformation has to be applied, this is done by constructing an hash function H_l mapping from G_N to the set $P(2\lambda)$ taking as inputs g and y to ensure soundness and completeness of the verifier.
H_l is implemented by applying *sha256* to the string $bin(g)||bin(y)$, reducing the output to $2^{2\lambda}$ and then checking if such value is prime or not, if it is the case return it, if not take the next prime (we have used the python sympy's package nextprime method for this). The procedure for H_l is given in algorithm 4.

Algorithm 4: Algorithm to compute H_l

Data: $g, y \in G$ and the security parameter λ
Result: $H_G(x)$
$h \leftarrow int(sha256(bin(g)||bin(y)) \mod 2^{2\lambda}$;
if $isprime(h)$ **then**
  return h
else
  return $nextprime(h)$

4.2.2 Computation of the proof with the trapdoor

In the case in which the prover is in possession of $\phi(N)$, it can generate the proof by computing

$$\pi = g^{\lfloor 2^t/l \rfloor} = g^{(2^t - (2^t \mod l))/l} = g^{\psi} \mod N$$

where $\psi = \frac{2^t - (2^t \mod l)}{l} \mod \phi(N)$

4.2.3 Improvement of the verifier to reduce the allocated storage

The paper [7] shows a possible improvement on the verifier function to reduce the space complexity. The verifier can recover the output y from l and π by

$$y \leftarrow \pi^l g^{2^t \mod l} \mod N$$

This would therefore allow the evaluator to transfer the tuple (l, π) instead of (y, π), which is almost half of the size.

4.3 Optimization of Evaluation function Function

Algorithm 5: Algorithm to compute $g^{\lfloor 2^t/l \rfloor}$ [2]

Data: $g \in G$, a prime number l and a delay t
Result: $g^{\lfloor 2^t/l \rfloor}$
$x \leftarrow 1_G \in G$;
$r \leftarrow 1 \in \mathbf{Z}$;
for $i \leftarrow 0$ **to** $t-1$ **do**
  $b \leftarrow \lfloor 2r/l \rfloor \in \{0,1\}$;
  $r \leftarrow 2r \bmod l$;
  $x \leftarrow x^2 g^b$;
return x

4.3.1 Wesolowski solution

Referring to the algorithm 5, it is claimed in [7] that fixing a parameter κ, $g^{\lfloor 2^t/l \rfloor}$, can be described as

$$g^{\lfloor 2^t/l \rfloor} = \prod_i g^{b_i 2^{\kappa i}} = \prod_{b=0}^{2^\kappa - 1} \left(\prod_{i \in I_b} g^{2^{\kappa i}} \right)^b \tag{4.3}$$

where $I_b = \{i | b_i = b\}$ and

$$b_i = \frac{2^\kappa \left(2^{t - \kappa(i+1)} \bmod l \right)}{l}$$

Since the values $g^{2^{\kappa i}}$ can be memorised when computing the output y, they do not have to be re-computed while calculating $g^{\lfloor 2^t/l \rfloor}$, therefore each component $\prod_{i \in I_b} g^{2^{\kappa i}}$ can be computed in $|I_b|$ group multiplications and the total product (4.3) with a total of $\sum_b |I_b| = t/\kappa$.
Taking a parameter γ and defining $I_{b,j} = \{i \in I_b | i \equiv j \mod \gamma\}$ for each j, equation 4.3 can be re-written as

$$g^{\lfloor 2^t/l \rfloor} = \prod_{b=0}^{2^\kappa - 1} \left(\prod_{j=0}^{\gamma - 1} \prod_{i \in I_{b,j}} g^{2^{\kappa i}} \right)^b = \prod_{b=0}^{2^\kappa - 1} \left(\prod_{j=0}^{\gamma - 1} \prod_{i \in I_{b,j}} \left(g^{2^{\kappa i}} \right)^{2^{\kappa j} 2^{-\kappa j}} \right)^b \tag{4.4}$$

$$= \prod_{j=0}^{\gamma - 1} \left(\prod_{b=0}^{2^\kappa - 1} \left(\prod_{i \in I_{b,j}} g^{2^{\kappa(i-j)}} \right)^b \right)^{2^{\kappa j}} \tag{4.5}$$

From (4.5), the article claims a time complexity

$$T_{\kappa\geq 1}(t,\kappa) = \frac{t}{\kappa} + \gamma\kappa 2^{\kappa} \tag{4.6}$$

It then claim that improvements are possible for $\kappa \geq 2$, defining κ_0 and κ_1 such that $\kappa_1 = \lfloor \kappa/2 \rfloor$ and $\kappa_0 = \kappa - \kappa_1$. Therefore

$$2^{\kappa} - 1 = 2^{\kappa_0+\kappa_1} - 1 = 2^{\kappa_0}2^{\kappa_1} - 1 = (2^{\kappa_1} - 1 + 1)\,2^{\kappa_0} - 1 = (2^{\kappa_1} - 1)\,2^{\kappa_0} + 2^{\kappa_0} - 1$$

and hence, defining $y_{b,j}$ as $y_{b,j} = \prod_{i\in I_{b,j}} g^{\kappa(i-j)}$

$$\prod_{b=0}^{2^{\kappa}-1} y_{b,j}^{b} = \prod_{b_1=0}^{(2^{\kappa_1}-1)2^{\kappa_0}} y_{b_1,j}^{b_1} \prod_{b_0=0}^{2^{\kappa_0}-1} y_{b_0,j}^{b_0} \tag{4.7}$$

Using the fact that for any ψ and $x, z \in \mathbf{N}$

$$\prod_{b=0}^{xz} \psi_b = \prod_{b_x=0}^{x-1} \prod_{b_z=0}^{z-1} \psi_{b_x+b_z*z} \tag{4.8}$$

$$Y_j^{(1)} = \prod_{b_1=0}^{(2^{\kappa_1}-1)2^{\kappa_0}} y_{b_1,j}^{b_1} = \prod_{b_1=0}^{(2^{\kappa_1}-1)-1} \prod_{b_0=0}^{2^{\kappa_0}-1} y_{b_0-1+b_1 2^{\kappa_0},j}^{b_1} \tag{4.9}$$

and

$$\prod_{b=0}^{2^{\kappa}-1} y_{b,j}^{b} = \prod_{b_1=0}^{(2^{\kappa_1}-1)} \left(\prod_{b_0=0}^{2^{\kappa_0}-1} y_{b_1 2^{\kappa_0}+b_0,j} \right)^{b_1 2^{\kappa_0}} \prod_{b_0=0}^{2^{\kappa_0}-1} \left(\prod_{b_1=0}^{2^{\kappa_1}-1} y_{b_1 2^{\kappa_0}+b_0,j} \right)^{b_0} \tag{4.10}$$

From (4.10), the article claims a time complexity of

$$T(t,\kappa)_{\kappa\geq 2} = t/\kappa + \gamma 2^{\kappa+1} \leq T(t,\kappa) \tag{4.11}$$

and a space complexity of

$$C(t,\kappa)_{\kappa\geq 2} = \frac{t}{\kappa\gamma} + 2^{k} \tag{4.12}$$

In particular, by selecting $\kappa = \log_2(t)/3$ and $\gamma = \sqrt{t}$ (with the implicit restrictions of $\log_2(t) = 3 * m$ for $m \in \mathbf{N},\ m \geq 2$ and $t = w^2,\ w \in \mathbf{N}$), the following complexities are claimed

$$T_{log} = 3\frac{t}{\log(t)} \tag{4.13}$$

$$C_{log} = \frac{t}{\sqrt{t}\log_2(t)/3} + \left(2^{\log_2(t)}\right)^{1/3} = \frac{3\sqrt{t}}{\log_2(t)} + \sqrt[3]{t} \tag{4.14}$$

4.3.2 Our solution

Referring to algorithm 5, we claim that

$$g^{\lfloor 2^t/l \rfloor} = g^{\sum_{i=0}^{t-1} b_i 2^{t-1-i}} = \prod_{i=0}^{t-1} g^{b_i 2^{t-1-i}} = \prod_{b=0}^{2^\kappa - 1} \left(\prod_{i \in I_b} g^{2^{t-1-i}} \right)^b \tag{4.15}$$

The proof of (4.15) is given in section 7.2.

By restricting ourselves on the base 2, or $\kappa = 1$, we get

$$g^{\lfloor 2^t/l \rfloor} = \prod_{i=0}^{t-1} g^{b_i 2^{t-1-i}} = \prod_{b=0}^{1} \left(\prod_{i \in I_b} g^{2^{t-1-i}} \right)^b = \prod_{i \in I_1} g^{2^{t-1-i}} \tag{4.16}$$

From (4.16) we see that the number of group multiplications that have to be done to compute $g^{\lfloor 2^t/l \rfloor}$ are $|I_1|$ and since from algorithm 5, $b_0 = 0$, $|I_1| \leq t - 1$.
Looking at the sequence $(r_k)_k$ in (7.3), it can be seen that $|I_0|$ is given by the number of times $2 * r_k \mod l < l$, this occurs for at least the fist $\lfloor \frac{\log_2(l)}{\log_2(2)} \rfloor$ times (a proof of this is found in the subsection below).
This allow to reduce the complexity to $t - \lfloor \frac{\log_2(l)}{\log_2(2)} \rfloor$ group operations, therefore

$$T_{our}(t, \kappa = 1) = T_{our}(t) = t - \lfloor \frac{\log_2(l)}{\log_2(2)} \rfloor = t - \lfloor \log_2(l) \rfloor \tag{4.17}$$

Intuitively, we see that the number of zeros in the series of b is larger than $\lfloor \log_2(l) \rfloor$ but at the moment, we are not able to find a proof for it. If it was possible to prove that $\forall i, m \in [0, t-1], i < m$, if $b_i = 1 = b_j = b_k \forall i < j < k$ and $b_{k+1} = 0$, then $r_{k+1} = 1$, we would have a periodicity in the presence of the sequence of the $\lfloor \log_2(l) \rfloor$ zeros and determining the frequency of the periodicity (let it be for example m), we could reduce the complexity to $t - (m-1)\lfloor \log_2(l) \rfloor$ and if we would be able to determine that $t - (m-1)\lfloor \log_2(l) \rfloor > t/w$ for $w > 2$ we would get a nicer complexity of $t(1 - 1/w)$.
But of course, this it more what we would like it to be rather than what it is, so we stick with T_{our} from (4.17).

Improvement of space complexity

To be able to reach T_{our} time complexity, we have to store all the t components of (4.16), resulting in space complexity of t.
Similarly to how it is done in the paper [7], we can also define the set $I_{1,j} = \{i \in I_1 | i \equiv j \mod \gamma\}$ and re-write (4.16) as

$$g^{\lfloor 2^t/l \rfloor} = \prod_{j=0}^{\gamma-1} \prod_{i \in I_{1,j}} \left(g^{2^{t-1-i} 2^{-t+1+j}} \right)^{2^{t-1-j}} = \prod_{j=0}^{\gamma-1} \prod_{i \in I_{1,j}} \left(g^{2^{i-j}} \right)^{2^{t-1-j}} \tag{4.18}$$

which would drop the complexity to

$$C_{our} = t/\gamma \tag{4.19}$$

However, as for (4.5), we don't see much gain because, since to calculate l, y is needed, before knowing l it is not possible to know the values that you need to keep, you would only know it once you establish the equation of $g^{\lfloor 2^t/l \rfloor}$ and only then free the space of the values which are needed.

4.3.3 Comparison between paper's solution and ours

The paper claims that with their model, a correct proof can be generated in (4.11) group multiplications with $\kappa \geq 2$.
We compare the our solution with the paper [7] when considering the same base (taking $\kappa = 1$ in equation (4.6)) and when considering different basis (taking (4.11) for $\kappa \geq 2$)

Same base: $\kappa = 1$

Taking $\kappa = 1$ in (4.6)

$$T_{\kappa \geq 1}(t, \kappa = 1) = t + \gamma 2 > t - 1 = T_{our}(t)$$

We can therefore conclude that the time complexity of our solution is better compared to the one proposed by [7] for the case in which $\kappa = 1$.

Different bases: $\kappa \geq 2$

We recall the time complexity of [7] for $\kappa \geq 2$ as

$$T(t, \kappa)_{\kappa \geq 2} = t/\kappa + \gamma 2^{\kappa+1}$$

We aim to compare the two complexities identifying the values of t for which our method has a better complexity. Those values are the ones such that

$$t - \lfloor \log_2(l) \rfloor < t/\kappa + \gamma 2^{\kappa+1} \tag{4.20}$$

$\gamma \geq 1 \Rightarrow \min\left(t/\kappa + \gamma 2^{\kappa+1}\right) = \frac{\kappa}{\kappa-1}(2^{\kappa+1} + \lfloor \log_2(l) \rfloor)$, therefore solving (4.20) is equivalent to solve

$$t - \lfloor \log_2(l) \rfloor < t/\kappa + 2^{\kappa+1}$$

with the restriction of $\kappa \geq 2$ and $\kappa i \leq t - 1, \forall i \in [1, t-1]$

For $\kappa = 2$, we have our method is more efficient for

$$t < 16 + 2\lfloor \log_2(l) \rfloor$$

Comparing the complexities for the maximum value of $\kappa = t - 1$ would not make sense since it would mean that $t = 2$: $(t-1) * 0 = 0$ and $(t-1) * 1 = t - 1 \implies$ only one square would be calculated.

Comparing the solutions for a general κ would mean minimizing $t/\kappa + 2^{k+1}$ and hence solving

$$\frac{\partial}{\partial \kappa}(t/\kappa + 2^{k+1}) = \frac{-t}{\kappa^2} + 2^{k+1} = 0$$

which would over complicate and go out of the scope of the report.

Comparison with their logarithmic complexity
Comparing with (4.13), we see that our solution is better only for insignificant values of the delay t. In fact, assuming $\log_2(t) = 3m,\ m \in \mathbf{N},\ m > 1$,

$$t - \lfloor \log_2(l) \rfloor < 3\frac{t}{3m} = \frac{t}{m}, \forall m > 1 \tag{4.21}$$

$$\implies t < \lfloor \log_2(l) \rfloor \frac{m}{m-1} \tag{4.22}$$

Since the delay is a number of steps and hence it must be an integer, we consider the floor $\lfloor m/(m-1) \rfloor = 1 \forall m > 1$, transforming (4.22) in

$$t < \lfloor \log_2(l) \rfloor$$

The prime l does not have a minimal value but only a maximal value of $2^{2\lambda}$.

4.4 Properties or Wesolowski's VDF

Differently that the Pietrzak VDF, the properties of Wesolowski are defined in a game-based way under the assumption of unfeasible problems to break. In particular the soundness property is defined under the assumption of hardness of the root finding problem in the RSA setup.
However, rather than explaining the definition and lemmas of the paper, we think that it makes more sense to give an argument for which the H_G and H_l are valid and secure cryptographic hash functions.

- First of all we notice that they are functions since they are deterministic.
- Secondly, even if both function map to a space smaller than 256 bits, we can make a valid argument that if λ is not too small and if the underlying function is a resistant hash function (pre-image, second image and collision resistance), then these cryptographic properties are transmitted to H_G and H_l. The reason for this is that if an hash function is resistance on all its k-bits output is in general also resistance on a subset of its output (if the subset is not too small).

4.5 Some remarks about the implementation

There are a few things which are work saying about our implementation of Wesolowski's VDF

- **Usage of H_G**: We chose to move the H_G hash function out of the *eval* and the *trapdoor* functions but instead use it in a *gen* function generating random values in $G_{\lambda,N}$. The *gen* function is therefore generating a random value and then using H_G to map it in $G_{\lambda,N}$. The *trapdoor* and the *eval* functions would then be called with inputs in $G_{\lambda,N}$ instead than in $\mathbb{Z}$ as in [7]. We took this decision to keep some consistency with the *Pietrzak* VDF's implementation.
- **Computation of the proof**: We have left two versions of our method of the computation of the proof because we think that there is trade-off in terms of Python execution speed of the two implementations. In the version *alg_4_revisited*, we compute the values of the gs in an open loop appending the each time the needed value, while in the version *alg_4_revisited_comprehension*, we use (7.3) to detach the in-loop dependency of b to r allowing us to easily compress the computation in a list comprehension. The reason for which we think that there is a trade-off of execution speed is that in general Python optimizes a lot on list comprehensions which makes the second choice a better choice, however, using (7.3) also requires doing multiple squares, compared to the first choice where we do integer divisions and modular reductions.
- **Invalid proofs of the trapdoor**: For some reasons, our trapdoor function does not always produce a valid proof. This is unexpected, so there must be a problem in our function. We have identified from our unit tests that the problem is in fact the generation of the proof.

5. COMPARISON BETWEEN AND PIETRZAK'S AND WESOLOWSKI'S VDF

We compare the two VDF in terms of their time complexity

Evaluation With the first optimization of the proof, Pietrzak's VDF has a total time complexity of

$$T_{Piet,Eval} = t + T_{Piet,Proof} = t + 2\log_2(t/2) + t \sum_{i=1}^{\log_2(t/2)} 1/2^i \tag{5.1}$$

$$= t\left(1 + \sum_{i=1}^{\log_2(t/2)} 1/2^i\right) + 2\log_2(t/2) \tag{5.2}$$

while Wesolowski's VDF has a total complexity of

$$T_{Wes,Eval} = t + T_{log} = t + 3t/\log_2(t) = t\left(1 + \frac{3}{\log_2(t)}\right) \tag{5.3}$$

We have that for $t >> 0$

$$T_{Piet,Eval} > T_{Wes,Eval} \tag{5.4}$$

(proof of this statement is found in appendix)

Moreover, as seen in section 4, Wesolowski's construction also allows to reduce the overall complexity of the eval function by allowing the production of part of proof of part of the output before having to wait the full output y.
We therefore conclude that the time complexity of the evaluation of Pietrzak is bigger than the one of Wesolowski.

Verification Pietrzak's VDF has a logarithmic time complexity verification $T_{Piet,Verif}$, while the Wesolowski's VDF's verification complexity $T_{Wes,Verif}$ is constant time. $T_{Wes,Verif}$ still increases with t due to the calculation of $r = 2^t \mod l$, however, this does not impact the time complexity as much as the group multiplication $\pi^r g^l \mod N$; moreover, since l is in general small compared to x and y, $r = 2^t \mod l$ is still reduced to relatively small number.
Therefore we conclude that Wesolowski's VDF has also a better verification time complexity

5.1 Complexity plots

We produce some complexity plots to verify time evolution of the two VDFs by measuring the execution time of their *eval* and $verify$ functions of different delays. Since the execution time may be effected by some stochastic oscillations, we consider a macrostate-like measurements by running the functions multiple times for the same delay and then taking the mean over each each measurement for each delay. At each execution, the input to the *eval* function is generated at random by the respective *gen* functions. We moreover, also plot the confidence interval of each delay taking a confidence of 95% and see how our macrostate-like values place themselves on it.

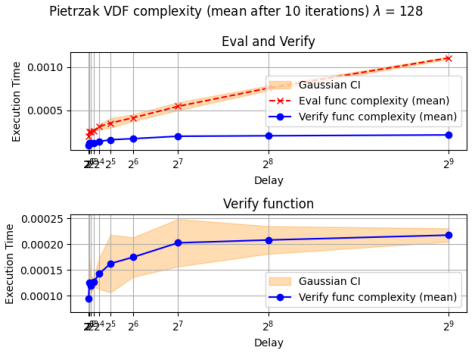

Fig. 5.1: Time complexity plots for *eval* and *verify* functions of Pietrzak's VDF.

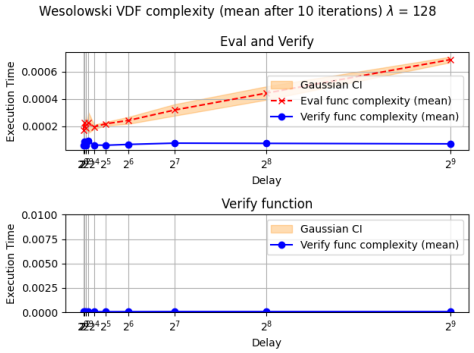

Fig. 5.2: Time complexity plots for *eval* and *verify* functions of Wesolowski's VDF.

Fig. 5.3: Complexity plots of *eval* and $verify$ functions.

We compute both plots with a security parameter $\lambda = 128$bits and going from a delay of 2^0 to a delay of 2^9, where each delay is ran 10 times.

As we can see from the plots below, in both VDF's the evaluation function follows a linear patter on the delay, while the the verify function is logarithmic on the delay for Pietrzak and constant on the delay for Wesolowski. In both cases, we see that the confidence interval also follows the pattern of the function, suggesting that 10 iterations are enough to have a result respecting the theory; the more the number of iterations are increased, the more the confidence interval will approach the macrostate-like plots.

6. CONCLUSION

In conclusion, we have shown that Pietrzak's VDF [6] has a worse evaluation and verification time complexity than Wesolowski's VDF [7]. But, Pietrzak's is unconditionally sound compared to the Wesolowski's which is sound under the root-finding problem - finding v such that $v^l = u \neq 1 \mod pk$ with l being a prime.
Moreover, we have produced plots to verify the time complexity evolution of the two VDFs and we saw that, in fact, we have what we expect: linear time complexity for the evaluations of both VDFs, and logarithmic time complexity for Pietrzak's verify function and constant time complexity for Wesolowski's.
In addition, we have established our own formulation for the optimization of the Wesolowski's evaluation function. Even if our solution has a worse time complexity for $\kappa > 1$, we believe it be more clear since it is presented by a formal proof, differently than what is done by the paper [7] which we believe to contain too many implicit statements.

6.1 Source code

The source code can be cloned from `https://github.com/LucaBonamino/crypto-VDF.git`.
Once cloned the package can be locally installed by running *make setup*, this will install the CLI command *cryptoVDF* from which the VDF can be ran, and the plot can be generated. The usage of the CLI tool is not mandatory, however, we strongly suggest to use it since it drastically speed up the executions of the scripts.

To see how to use the the CLI tool, run *cryptoVDF –help*.
We will anyway provide some examples in the readMe file of the Github repository.

REFERENCES

[1] M. Bégel et al. "The Ethereum 2.0 Beacon Chain". In: *Proceedings of the 4th International Conference on Blockchain Technology (ICBCT)*. 2021. DOI: 10.1145/3459104.3459112.

[2] Dan Boneh, Benedikt Bünz, and Ben Fisch. "A survey of two verifiable delay functions". In: *Cryptology ePrint Archive* (2018).

[3] Dan Boneh et al. "Verifiable delay functions". In: *Annual international cryptology conference*. Springer. 2018, pp. 757–788.

[4] Ethereum Foundation. *The Great Eth2 Renaming*. https://blog.ethereum.org/2022/01/24/the-great-eth2-renaming/. 2022.

[5] Ethereum Foundation. *The Merge*. https://blog.ethereum.org/2022/08/24/the-merge-mainnet/. 2022.

[6] Krzysztof Pietrzak. "Simple verifiable delay functions". In: *10th innovations in theoretical computer science conference (itcs 2019)*. Schloss-Dagstuhl-Leibniz Zentrum für Informatik. 2019.

[7] Benjamin Wesolowski. "Efficient verifiable delay functions". In: *Advances in Cryptology–EUROCRYPT 2019: 38th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Darmstadt, Germany, May 19–23, 2019, Proceedings, Part III 38*. Springer. 2019, pp. 379–407.

7. APPENDIX

7.1 Proof of (3.9)

$$\hat{\mu}_k = 2^{T/2^k} \prod_{i=1}^{k-1} \left(r_i + 2^{T/2^i} \right) = 2^{T/2^k} \hat{x}_k = 2^{T/2^k} \hat{x}_{i-1} \left(r_{k-1} + 2^{T/2^{k-1}} \right) \tag{7.1}$$

Suppose

$$\hat{\mu}_k = 2^{T/2^k} \prod_{i=1}^{k-1} \left(r_i + 2^{T/2^i} \right) = 2^{T/2^k} \hat{x}_k = 2^{T/2^k} \hat{x}_{i-1} \left(r_{k-1} + 2^{T/2^{k-1}} \right)$$

then, we prove it for $k+1$ steps

$$\begin{aligned}
\hat{\mu}_{k+1} &= 2^{T/2^{k+1}} \prod_{i=1}^{k} (r_i + 2^{T/2^i}) \\
&= 2^{T/(2^{k+1})} (r_k + 2^{T/2^k}) \prod_{i=1}^{k-1} (r_i + 2^{T/2^i}) \\
&= 2^{T/(2^{k+1})} \left(r_k \prod_{i=1}^{k-1} (r_i + 2^{T/2^i}) + 2^{T/2^k} \prod_{i=1}^{k-1} (r_i + 2^{T/2^i}) \right) \\
&= 2^{T/(2^{k+1})} \left(r_k \prod_{i=1}^{k-1} (r_i + 2^{T/2^i}) + \hat{\mu}_k \right)
\end{aligned}$$

and

$$\hat{\mu}_{k+1} = 2^{T/(2^{k+1})} \left(r_k * x_k + \hat{\mu}_k \right)$$

with

$$\hat{x}_k = \prod_{i=1}^{k-1} (r_i + 2^{T/2^i})$$

7.2 Proof of of (4.15)

Analysing the algorithm 5, it can be seen that it can be described by an inductive equation

$$g^{\lfloor 2^k/l \rfloor} = f(x_k) = x_k = \begin{cases} x_{k-1}^2 g^{b_k} \text{ if } k \geq 1 \\ 1 \text{ if } k = 0 \end{cases} \tag{7.2}$$

with

$$\left.\begin{aligned} b_k &= \lfloor 2r_k/l \rfloor \\ r_k &= \begin{cases} 2r_{k-1} \bmod l & \text{if } k \geq 1 \\ 1 & \text{if } k = 0 \end{cases} \end{aligned}\right\} \Rightarrow b_k = \lfloor \frac{2\left(2^k \bmod l\right)}{l} \rfloor \tag{7.3}$$

Therefore, proving (4.15) is equivalent to proving

$$g^{\lfloor 2^t/l \rfloor} = f(x_t) = x_{t-1}^2 g^{b_{t-1}} = \begin{cases} \prod_{i=0}^{t-1} \left(g^{2^{t-1-i}}\right)^{b_i} & \text{if } t \geq 2 \\ g^{b_0} \text{ if } t = 1 \end{cases}$$

Taking $k = 1$, we see that

$$x_1 = x_0^2 g^{b_0} = g^{b_0}$$

since $x_0 = 1$
Taking $k = 2$, we see that

$$x_2 = x_1^2 g^{b_1} = g^{2b_0} g^{b_1} = \prod_{i=0}^{2-1} \left(g^{2^{2-1-i}}\right)^{b_i}$$

now, we suppose

$$x_k = x_{k-1}^2 g^{b_{k-1}} = \prod_{i=0}^{k-1} \left(g^{2^{k-1-i}}\right)^{b_i}$$

and show

$$x_{k+1} = x_k^2 g^{b_k} = \prod_{i=0}^{(k+1)-1} \left(g^{2^{(k+1)-1-i}}\right)^{b_i}$$

$$\begin{aligned} x_{k+1} = x_k^2 g^{b_k} &= \left(\prod_{i=0}^{k-1} \left(g^{2^{k-1-i}}\right)^{b_i}\right)^2 g^{b_k} \\ &= g^{2^0 b_k} \prod_{i=0}^{k-1} \left(g^{2*2^{k-1-i}}\right)^{b_i} \\ &= g^{2^{(k+1)-1-k} b_k} \prod_{i=0}^{k-1} \left(g^{2^{(k+1)-1-i}}\right)^{b_i} \\ &= \prod_{i=0}^{(k+1)-1} \left(g^{2^{(k+1)-1-i}}\right)^{b_i} \end{aligned}$$

Hence

$$g^{\lfloor 2^{k+1}/l \rfloor} = x_{k+1} = \prod_{i=0}^{(k+1)-1} \left(g^{2^{(k+1)-1-i}}\right)^{b_i}$$

and therefore

$$g^{\lfloor 2^t/l \rfloor} = x_t = \prod_{i=0}^{t-1} \left(g^{2^{t-1-i}}\right)^{b_i}, \quad \forall t \geq 2 \tag{7.4}$$

7.3 Proof of (4.17)

To define see the sequence $(y_k)_k$ defined as

$$y_k = \begin{cases} 2y_{k-1} & \text{if } k! = 0 \\ 1 & \text{if } k = 0 \end{cases} = 2^k$$

from a given value l, we want to determine $|S|$ such that

$$S = \{y \in (y_k)_k | y < l\}$$

Therefore, we get the k such that $2^k < l \iff k \log_2(2) < \log_2(l)$, we want the maximum k such that the conditions is satisfied

$$k = \lfloor \frac{\log_2(l)}{\log_2(2)} \rfloor$$

7.4 Proof of (5.4)

We want to prove that from a certain t

$$T_{Piet,Eval} = t\left(1 + \sum_{i=1}^{\log_2(t/2)} 1/2^i\right) + 2\log_2(t/2) > t\left(1 + \frac{3}{\log_2(t)}\right) = T_{Wes,Eval}$$

We do this by taking the showing that

$$t\left(1 + \frac{3}{\log_2(t)}\right) < t\left(1 + \sum_{i=1}^{\log_2(t/2)} 1/2^i\right) \leq t\left(1 + \sum_{i=1}^{\log_2(t/2)} 1/2^i\right) + 2\log_2(t/2)$$

which is equivalent to show

$$1 + \frac{3}{\log_2(t)} < 1 + \sum_{i=1}^{\log_2(t/2)} 1/2^i \iff \frac{3}{\log_2(t)} < \sum_{i=1}^{\log_2(t/2)} 1/2^i \tag{7.5}$$

From here we could directly see that the statement is valid for $t >> 0$, since

$$\sum_{i=1}^{\log_2(t/2)} 1/2^i \xrightarrow{t>>0} 1 > 0 \xleftarrow{t>>0} \frac{3}{\log_2(t)}$$

However we want to see a smaller t for which this is the case.
Taking $t = 2^6$, $\frac{2}{\log_2 t} = 1/2$ and

$$\sum_{i=1}^{\log_2(t/2)} 1/2^i = \sum_{i=1}^{\log_2(2^5)} 1/2^i = \frac{\sum_{i=1}^{4} 2^i}{2^5} = 30/32 > 1/2$$

Since $1 + \frac{3}{\log_2(t)}$ is a descreasing function and $\sum_{i=1}^{\log_2(t/2)} 1/2^i$ is an increasing function, we can be sure that for all $t > 2^6$, (7.5) holds and hence also the initial inequality (5.4) holds.