Verifiable Delay Functions

Author: Mitul Patel (B18CSE041)
Instructor: Dr Somitra Sanadhya
IIT Jodhpur

May, 2021

1 Abstract

The goal of the project was to get introduction to the concept of Verifiable Delay Functions(VDFs). The report is oriented around, what are VDFs? and Why are the needed?, The Idea behind Wesolowski VDF scheme & Pietrzak VDF scheme, Some what level of comparison between them.

2 Introduction

What if you are to host a lottery, How do you generate random numbers? You cannot just come up with a number and announce it as a random number, because then you can be easily accused of biasing the lottery towards your personal gain. You need to show that the number is random. Another solution is that you use a bowl filled with some balls with a number written on it and you blindly or somehow randomly draw a ball from it and you announce the number written on the face of the ball, but this method can not be trusted either, the notion of "blindly" or "randomly" drawing a ball to show that the number drawn is random can be biased using magnets & injecting iron inside the balls. In an episode of "Better Call Soul", a story of an interesting character Jimmy from the famous TV Series "Breaking Bad", Jimmy uses another trick in a game of Bingo for some personal gain. Here is the link to the recap of that episode.. So what you want is a publically verifiable source of randomness. Another idea is to have a Pseudo Random Generator(PRG) seeded with a number derived from a huge source of randomness. A stock market can be considered a huge source of randomness. Now you can choose some number of last-minute transactions happened for a stock and hash them and then apply an extractor function to extract a number and seed the extracted number to the PRG & for the verifiability you can make the taken transactions public. The system flow is shown in the figure 1. But since all the information is public, an active adversary in the stock market can influence seed by making some last minute transactions or not making them. Now for the adversary to gain something, he needs to simulate some transactions and find out the transactions he need to make in order to gain something from the generated seed.

You can affect the ability of the adversary to simulate quickly by injecting a delay in the computation of the seed. The system with delay is shown in the figure 2. The delayed computation unit takes the extracted number as

input and runs some computation over it (which results in delay) and produces another number, which seeded to the PRG. With some reasonable delay the adversary can't possibly simulate the effects of some transactions on the seed generation. Now to ensure the verifiability of the system you need to make ensure the verifiability of the delayed computation. Some other things we want from the delayed computation unit,

- The verification process must not be taking much time, otherwise the verification would be tedious and no one would probably engage in it.(In other words fast computation)
- The delay must be controllable.
- No one should be able to generate the same output, with reasonably less time than the time taken by the computation unit, even given some parallel processors.

The delay unit with these desired properties is called a Verifiable Delay Function. (in short a VDF)

3 Formal Definition of VDF

A VDF can be defined as a 3-algorithm set: $(\text{Setup}, \text{Eval}, \text{Verify})$.

- $\text{Setup}(\lambda, T) \rightarrow pp$: Here λ is the security parameter, T is the delay parameter & pp is short for public-parameter, which is later used for Evaluation & Verification.
- $\text{Eval}(pp, x) \rightarrow (y, \pi)$: Here pp is the public-parameter calculated by Setup, x is the input to the delayed computation, y is the result of delayed computation & π is called proof, which is optional(The designer may or may not choose to use it).
- $\text{Verify}(pp, x, y, \pi) \rightarrow \{True, False\}$: Here all the inputs correspond to either product or input of Setup or Eval. $True$ & $False$ corresponds to whether or not y is a delayed computation product of x under the given pp & π.

Additionally we want the following properties from it:

- It must take T sequential steps to compute y. (PRAM Runtime T with $\text{polylog}(T)$ Processors)
- No Adversary with a machine with a polynomial number of processor can distinguish y from random.
- y must be efficiently verifiable. $\rightarrow O(\text{polylog}(T))$
- For all x it is difficuilt to find a y for which $\text{Verify}(pp, x, y, \pi) \rightarrow True$ but $\text{Eval}(pp, x) \nrightarrow y$.
- The VDF must remain secure in presence of an adversary with ability to perform polynomially bounded pre-computation.

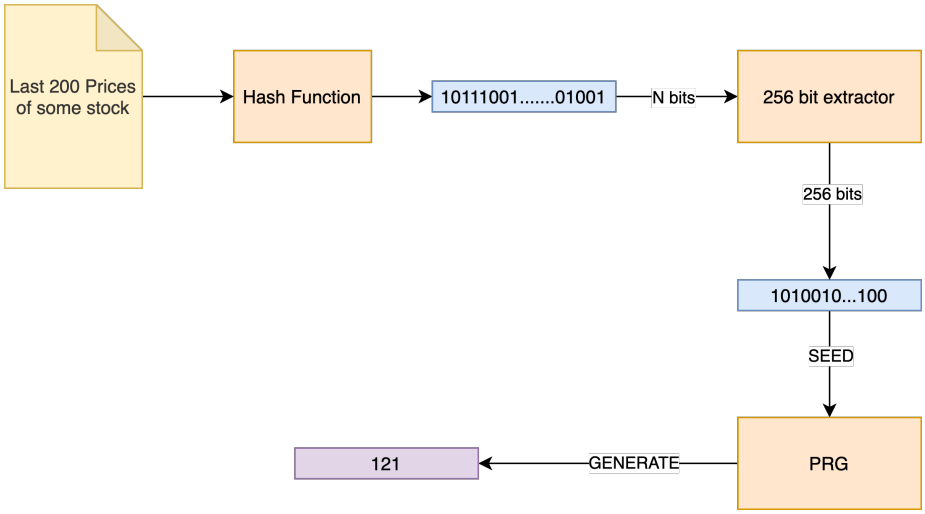

Figure 1: Basic system

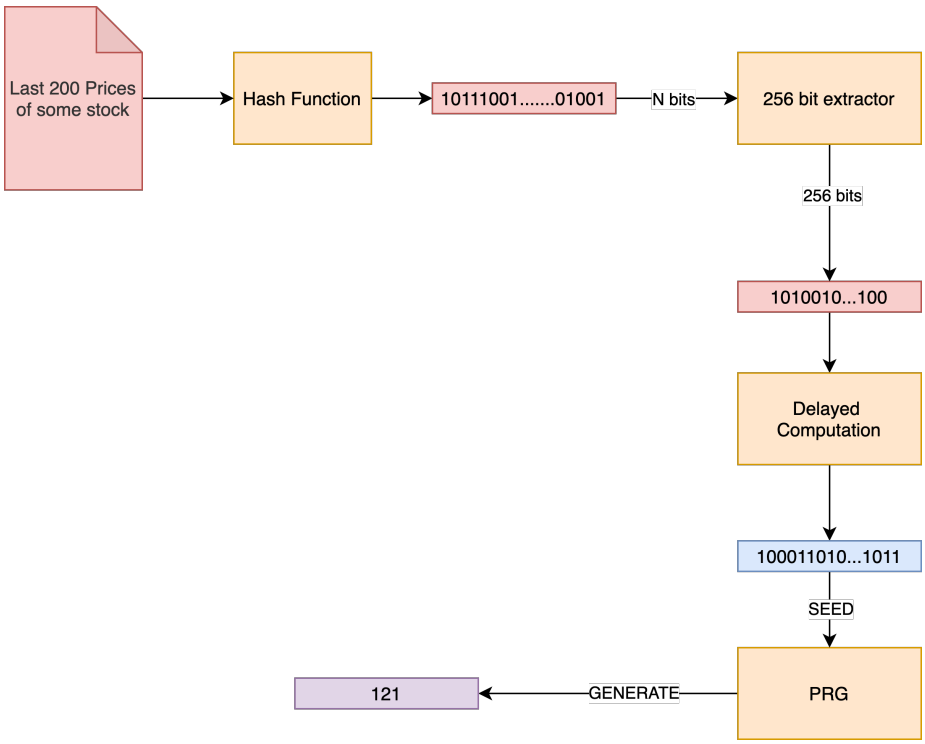

Figure 2: System with delayed computation

4 RSW time-lock puzzle

RSW time-lock puzzle stands for Rivest-Shamir-Wagner time-lock puzzle. The Wesolowski & Pietrzak VDF schemes uses the idea behind RSW time-lock puzzle. The goal behind RSW time-lock puzzle was to "Send information into the future", As in, to have some information to be revealed in the future (or in other words after a fixed amount of time). The idea behind RSW time-lock puzzle is as follows:

- Generate: $n = pq$ where p & q are large primes.
- $\phi(n) = (p-1)(q-1)$
- Compute: $t = TS$ where S is the amount of sequential squarings mod n performable(just for simplicity let's assume with the fastest CPU existing), T is the time parameter.
- Compute: $C_m = Enc(k, m)$, where Enc is a encryption function with some key-size such that it is infeasible for a brute-force attack.
- Compute: $C_k \equiv k + x^{2^t} \mod n$, where x is a random number $\mod n$. Now to compute x^{2^t} easily, compute:

  $$e \equiv 2^t \mod \phi(n)$$

  and then using Fermat's Little Theorem,

  $$x^{2^t} \equiv x^{(k\phi(n))} x^e \mod n$$

  $$x^{2^t} \equiv x^e \mod n$$

- Then output (n, x, t, C_k, C_m) & Throw away other things.
- For simplicity, let's say there is a trusted party which reveals the key k after time T.

Now since The information about $\phi(n)$ is not available publically & computation of x^{2^t} cannot be parallelised $\Rightarrow$ it will take at least T time to compute: $(x^{2^t} \mod n) \to k \to m$ unless the key k is revealed by the Trusted Party(which will happen after time T). Eitherway the message can only be revealed after time T.

5 The idea behind Wesolowski VDF Scheme

We are going to study the Wesolowski VDF scheme, following the definition structure of VDFs, as defined in Section 3.

5.1 Scheme

The scheme is given as follows, which is basically the 3-algorithm set (Setup, Eval, Verify).

5.1.1 Setup

- A RSA Group with $n = p * q$

$$Z_n^* = \{x | (1 \leq x < n) \ \& \ \gcd(x, n) = 1\}$$

- The delay parameter: T
- The info about input domain $\mathcal{X}$ and the output domain $\mathcal{Y}$.
- Security parameter λ.
- The public parameter will be : $pp \leftarrow (n, T, \mathcal{X}, \mathcal{Y}, \lambda)$

5.1.2 Eval

- Calculate: $y = x^{2^T} \mod n$.
- Since, $\phi(n)$ is unknown, the evaluation of y cannot be made fast using Fermat's little theorem.
- $2^T \xrightarrow{binary}$ 100000...00(1 followed by T zeroes), so the square-and-multiply algorithm will take almost T steps to calculate $x^{2^T} \mod n$.
- So, The actual delay will be $O(T)$.

5.1.3 Verify

Verification is shown as an protocol between a Verifier and the Prover:

1. The Verifier generates a large prime number L. (L is chosen uniformly randomly from the set containing first $2^{2\lambda}$ primes)
2. The Prover computes $2^T = bL + r$.
3. The Prover computes $\pi \leftarrow x^b$.
4. The Prover sends (y, π) to the Verifier.
5. The Verifier computes $r = 2^T \mod L$.
6. The Verifier matches if $y == \pi^L x^r$

5.2 Analysis

5.2.1 Setup

We need not necessarily take the RSA group, as we can see the idea revolves around the computation of the power in modulo n arithmetic & gaining sequentiality of T. Also the knowledge of the group size must be kept secret otherwise the scheme will lose the necessary sequentiality. So the production of **Group of unknown size** is necessary.

5.2.2 Verification

The verification process here is shown as an interactive protocol between the prover and the verifier, which can be transformed into a non-interactive version using the Fiat-Shamir heuristic.

why does the $y == \pi^L x^r$ leads to verification:

- $\pi^L x^r = (x^b)^L x^r$ (since $\pi \leftarrow x^b$).
- $(x^b)^L x^r = x^{bL+r}$
- $x^{bL+r} = x^{2^T}$ (since $2^T = bL + r$).
- Hence, $(y == \pi^L x^r) \equiv (y == x^{2^T})$.

Time taken for proof construction: Proof construction basically requires $\pi \leftarrow x^b$, where $2^T = bL + r$, now using the basic Long division method we can compute the prooof π in 2T group operations. But using a method specified by wesolowski, the total time for Eval & proof construction can be reduced to $(1 + \frac{1}{sk})T$ with s processors, $O(2^k)$ storage space[4][9].

Time taken for verification: The verification basically requires computation of $(r = 2^T \mod L) \rightarrow (\pi^L x^r \mod N)$, the computation of r can be done by $\log_2(T)$ multiplications using square-and-multiply algorithm & additionally 2 exponentiations depending upon L. So the total time is polylog(T).

6 The Idea behind Pietrzak VDF Scheme

6.1 Scheme

The scheme is given as follows, which is basically the 3-algorithm set (Setup, Eval, Verify).

6.1.1 Setup

- A Finite abelian multiplicative group of unknown order: G, with mod n.
- The delay parameter: T
- The info about input domain $\mathcal{X}$ and the output domain $\mathcal{Y}$.
- Security Parameter: λ
- The public parameter will be : $pp \leftarrow (G, T, \mathcal{X}, \mathcal{Y}, \lambda)$

6.1.2 Eval

- Calculate: $y = x^{2^T} \mod n$.
- Since, $\phi(n)$ is unknown, the evaluation of y cannot be made fast using Fermat's little theorem.
- $2^T \xrightarrow{binary}$ 100000...00(1 followed by T zeroes), so the square-and-multiply algorithm will take almost T steps to calculate $x^{2^T} \mod n$.
- So, The actual delay will be $O(T)$.

6.1.3 Verify

The verification protocol between the Verifier & Prover is shown below:

Protocol 1: Verify(n, x, T, y)

if $T == 1$ **then**
  if $y == x^2$ **then** Verifier **Accepts** ;
  else Verifier **Rejects** ;
else
  Prover Sends $\mu \leftarrow x^{2^{T/2}}$ to the Verifier.
  if $\mu \notin G$ **then**
    Verifier **Rejects** ;
  else
    Verifier samples $r \xleftarrow{\$} \mathbb{Z}_{2^\lambda}$ and sends it to the Prover
    The Prover & Verifier computes: $x' \leftarrow x^r \mu$
    The Prover & Verifier computes: $y' \leftarrow \mu^r y$
    if $T/2$ *is even* **then**
      The Prover & Verifier engages in: Verify$(n, x', T/2, y')$
    else
      The Prover & Verifier engages in: Verify$(n, x', \lceil T/2 \rceil, y'^2)$
    end
  end
end

6.2 Analysis

The verification process shown in protocol 1 is recursive in nature, which will have $\log_2(T)$ recursions. The protocol can be converted to a non-interactive version using the Fiat-Shamir heuristic.

What does each recursive iteration do: In each interactive iteration the verifier basically asks for the proof of $T/2$ amount of work, so sequentially: $T/2 \to T/4 \to T/8 \to T/16 \to T/32 \to \cdots$ which sums up as shown below:

$$\begin{aligned} T/2 + T/4 &= 3T/4 \\ 3T/4 + T/8 &= 7T/8 \\ 7T/8 + T/16 &= 15T/16 \\ 15T/16 + T/32 &= 31T/33 \\ &\vdots \end{aligned}$$

At the end when recursively $T \to 1$, it becomes easy for the verifier to assert the given proof, by simply checking $y == x^{2^T} (\equiv y == x^2)$. If we observe the the growth of terms μ, x', y' in each iteration: In i^{th} recursive step: $(i \in [1 \dots d]_{d=\log_2(T)})$

$$\begin{aligned} \mu_i &\leftarrow x^{(\prod_{k=0}^{i-1} f(k)) \cdot 2^{T_i/2}} \\ x'_i &\leftarrow x^{(\prod_{k=0}^{i} f(k))} \\ y'_i &\leftarrow x^{(\prod_{k=0}^{i} f(k)) \cdot 2^{T_i/2}} \\ \text{Where: } T_i = T/2^{i-1} \ &\mid \ f(k) = r_k + 2^{T_{k+1}} \ , \ f(0) = 1 \end{aligned}$$

Also at each level the relation $y' = x'^{2^{T_i/2}}$ is maintained & T_i is halved, so recursively $\Rightarrow x' \xrightarrow{sums-to} x^{2^T}$ (if $r_i \leftarrow 0$ just for the sake of this statement).

Time taken for proof construction: The proof is not defined in the interactive version, but it can be defined in the non-interacive as the set of μ computed by the prover at each recursive level. let μ_i define the μ computed by the prover at i^{th} recursive level.

$$\pi \leftarrow \{\mu_1, \mu_2, \mu_3, \cdots, \mu_d\} \ \mid \ d = \log_2(T)$$

If we observe the sequence of μ_i,

$$\begin{aligned} \mu_1 &\leftarrow x^{2^{T/2}} \\ \mu_2 &\leftarrow x^{r_1 2^{T/4} + 2^{3T/4}} \\ \mu_3 &\leftarrow x^{r_1 \cdot r_2 \cdot 2^{T/8} + r_1 \cdot 2^{3T/8} + r_2 \cdot 2^{5T/8} + 2^{7T/8}} \\ &\vdots \end{aligned}$$

We can see that, for computation of μ, we can use already computed values in computation of x^{2^T}, such as for μ_1 we can directly use $x^{2^{T/2}}$. We can use

such values because we approach the x^{2^T} by T sequential squarings. But this becomes costly, since for the computation of μ_i we need to store 2^{i-1} values $\{x^{2^{kT/2^i}}\}_{k\in[1\ldots 2^{i-1}]}$, So in total we need to store 2^d values. But if we make a tradeoff between storage & compute by having a $s \in [1 \ldots d]$, such that we only use storage upto s^{th} recursive level & recompute the rest values, using

$$\frac{T}{2^{s+1}} + \frac{T}{2^{s+2}} + \cdots + \frac{T}{2^d} < \frac{T}{2^s}$$

squaring operations. It is shown that for a **random** λ bit exponentiation we require 1.5λ multiplications[6]. So, with 2^s values stored & rest to be computed with $\frac{T}{2^s}$ multiplications, we need a total of $2^s + \frac{T}{2^s}$ operations, now to minimise that we need $s = \log_2(\sqrt{T})$. So the proof generation time is of the order: $O(\sqrt{T})$.

With 2^s values stored & with $2^p < 2^s$ bounded parallelised computation for the 2^s remaining elements, Pietrzak shows the computation to be of $2^{s-p} \cdot \lambda \cdot (s-1) \cdot \frac{1.5}{2} + 2^{t-s}$ sequential steps with $2^s \cdot \log(n)$ storage capacity [6].

Time taken for verification: Time taken for verification is for the computation of $x' \leftarrow x^r\mu$ & $y' \leftarrow \mu^r y$ for $\log_2(T)$ recursive levels. Since those expressions contain exponentiation with r (of at most λ bits). So the time taken would be of $\log_2(T) \cdot 2 \cdot \lambda$ sequential(multiplicative) steps. So it is of polylog(T) order.

7 Comparison

7.1 Proof Size

The Pietrzak scheme has proof size of $\log_2(T)$ elements:

$$\pi \leftarrow \{\mu_1, \mu_2, \mu_3, \cdots, \mu_d\} \mid d = \log_2(T)$$

While the Wesolowski scheme has proof size of 1 element:

$$\pi \leftarrow x^b$$

So We can say that The proof size in Wesolowski scheme is $\log_2(T)$ time smaller than the proof size in the Pietrzak scheme.

7.2 Verification Speed

The time taken for the verification for the Pietrzak scheme depends on total $2\log_2(T)$ exponentiations in $\log_2(T)$ recursive levels, in computation of:

$$x' \leftarrow x^r \mu$$

$$y' \leftarrow \mu^r y$$

The time taken for the verification in Wesolowski scheme depends on the 2 exponentiations done in:

$$y == \pi^L x^r$$

So we can say that the verification is Wesolowski scheme will be almost $\log_2(T)$ times faster than verification in the Pietrzak scheme.

7.3 Proof Generation Speed

If we don't use parallelization, with the Pietrzak scheme, as discussed in Section 6.2, The proof generation time is of the order:

$$O(\sqrt{T})$$

If we don't use parallelization, with the Wesolowski scheme, as discussed in Section 5.2.2, With $O(2^k)$ storage capacity available, we can reduce the proof generation to $\frac{T}{k}$ operations. So the proof generation time is of the order:

$$O(T)$$

So proof generation in Pietrzak scheme can be considered faster by a factor of $\sqrt{T}$. But both proof-generation schemes are parallelizable, so consideration of that factor will have some effect on the so discussed $\sqrt{T}$ factor.

References

[1] A vdf explainer. https://reading.supply/@whyrusleeping/a-vdf-explainer-5S6Ect.

[2] Introduction to verifiable delay functions (vdfs). https://blog.trailofbits.com/2018/10/12/introduction-to-verifiable-delay-functions-vdfs/, Oct 2018.

[3] Dan Boneh, Joseph Bonneau, Benedikt Bünz, and Ben Fisch. Verifiable delay functions. *IACR Cryptol. ePrint Arch.*, 2018:601, 2018.

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

[5] Joe Netti. Pietrzak verifiable delay functions. https://medium.com/@joenetti/pietrzak-verifiable-delay-functions-f5683131882b, May 2020.

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

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

[8] Alfonso de la Rocha. @adlrocha - a gentle introduction to vdfs. https://adlrocha.substack.com/p/adlrocha-a-gentle-introduction-to, Jun 2020.

[9] Benjamin Wesolowski. Efficient verifiable delay functions. *J. Cryptol.*, 33(4):2113–2147, 2020.