Capstone:Verifiable Delay Functions

Sona Maharjan

December 21, 2020

1 Introduction

A delay function is an important cryptographic primitive used for adding delay in decentralized applications. Starting with the implementation of cryptocurrency, the last decades have witnessed numerous constructions of delay function involving various mathematical techniques, and a number of exciting applications. An example of such a delay function is Verifiable Delay Function (VDF). This concept was recently introduced by Boneh, Bonneau, Bünz and Fisch [2]. As a part of my capstone project, I went over different constructions of VDFs and implemented them in Javascript. In this paper, I'll be discussing about the various VDF constructions including a recently introduced primitive called Continuous Verifiable Delay Function [3].

2 Verifiable Delay Functions

A Verifiable Delay Function (VDF)[2] is a special type of a delay function based on the notion of repeated squaring in a group of unknown order coined by Rivet Shamir and Wagner in 1996 [8]. It is a slow function that takes a pre-defined amount of time to compute even on a parallel processor, and is exponentially easier to verify by anyone. A VDF consists of three algorithms:

1. **Setup**$(\lambda, \mathcal{T}) \rightarrow pp$ is a randomized algorithm that takes a security parameter λ and a time delay bound $\mathcal{T}$, and outputs public parameters pp.
2. **Eval**$(pp, x) \rightarrow (y, \pi)$ takes an input $x \in \mathcal{X}$ and the public parameters pp, and produces an output $y \in \mathcal{Y}$ and a proof of the computation π.
3. **Verify**$(pp, x, y, \pi) \rightarrow \{accept, reject\}$ is a deterministic algorithm that takes an input $x \in \mathcal{X}$, output $y \in \mathcal{Y}$ and proof π, and outputs *accept* if y is the correct evaluation of the VDF on input x, otherwise outputs $reject$.

2.1 Security Properties

A VDF must satisfy the following three properties:

Sequentiality: A parallel algorithm A, using at most $poly(\lambda)$ processors, that runs in time less than T cannot compute the function. Specifically, for a random $x \in \mathcal{X}$ and pp output by $Setup(\lambda, T)$, if $(y, \pi) \rightarrow Eval(pp, x)$ then $Pr[Eval(pp, x) \neq y]$ is negligible.

Soundness: For an input $x \in \mathcal{X}$, exactly one $y \in \mathcal{Y}$ will be accepted by Verify. Specifically, let A be an efficient algorithm that given pp as input, outputs (x, y, π) such that $Verify(pp, x, y, \pi) = accept$. Then $Pr[Eval(pp, x) \neq y]$ is negligible.

Completeness: The deterministic *Eval* algorithm generates a *unique* output $y \in \mathcal{Y}$ and a proof of the computation π for an input $x \in \mathcal{X}$. The completeness property guarantees that *Verify* always produces an output *accept* for correctly generated proofs and computation.

2.2 Construction

There are two constructions of VDFs that were recently proposed, one by Wesolowski [9] and other by Pietrzak [6]. In this segment, I'll go over each of the constructions and provide a pseudocode to implement them.

2.2.1 Simple VDF- Wesolowski

Setup: The VDF setup requires a security parameter λ (typically between 128 and 256) as an input. It generates an RSA public modulus N of bit length λ and computes a secure hash function $H : \mathcal{X} \rightarrow \mathbb{Z}_N^*$.

Evaluation: The evaluation takes as input a time bound parameter $T \in N$ and computes $x = H(m) \in \mathbb{QR}_N^+$. Then using repeated squaring, it computes $y = (x^{2^{\mathcal{T}}} \ mod \ N) \ \in \mathbb{Z}_N^*$. It is important to know that if the evaluator knows $\phi(N)$, it can cut this computation because $x^{2^{\mathcal{T}}} \ mod \ N = x^{2^{\mathcal{T}} \ mod \ \phi(N)} \ mod \ N$.

Proof: The VDF evaluator needs to compute a proof (π) that satisfies $y = (x^{2^{\mathcal{T}}} \ mod \ N) \ \in \mathbb{Z}_N^*$. The proof begins by computing l which is a nextprime (H_{prime}) of x and y i.e. $l = H_{prime}(x + y)$. Then, it computes $\pi = x^{\lfloor 2^{\mathcal{T}} \rfloor} \ mod \ N$.

Verification: A verifier takes as an input $(m, \mathcal{T}, l, \pi)$ and computes $r = 2^{\mathcal{T}} \ mod \ l$ and $y' = \pi^l . x^r \ mod \ N$. Then it checks whether $H_{prime}(x + y') = l$.

2.2.2 Implementation of Simple VDF:

Wesolowski's construction has two protocols- interactive and non-interactive [9]. The non-interactive protocol is based on Fiat-Shamir Heuristic [4]. In this paper, I will only show the implementation of non-interactive protocol. This implementation requires us to find the

next-prime. I've used the concept of 8/30 wheel followed by a primality test to compute the next-prime.

Algorithm 1 Wesolowski's VDF

Evaluation and Proof

Input: $\mathcal{T} \in N$, $m \in \{0,1\}^*$
Proof:
$x \leftarrow H(m)$
$y \leftarrow x$
for $i \leftarrow 1$ *to* $\mathcal{T}$ **do**
    $y \leftarrow y^2 \ mod \ N$ ▷ Repeated Squaring
end for
$l \leftarrow H_{prime}(x+y)$ ▷ Choose a prime $l \in \{1,..,2^\lambda\}$ to make this interactive
$\pi \leftarrow x^{\lfloor 2^\mathcal{T}/l \rfloor} \ mod \ N$
Output: (π, l)

Verification

Input: x, $\mathcal{T}$, π, l
Process:
$r \leftarrow 2^\mathcal{T} \ mod \ l$
$y \leftarrow \pi^l.x^r \ mod \ N$
if $l = H_{prime}(x+y)$ **then**
    accept
else
    reject
end if
Output: (*accept*, *reject*)

Algorithm 2 Finding Next Prime

Input: num
Process:
if $num = even$ **then**
    $num \leftarrow num + 1$
end if
for $i \leftarrow num$ *to* $2 \times num$ **do**
    if $i \ mod \ 3 \neq 0$ *and* $i \ mod \ 5 \neq 0$ **then**
        if $num = prime$ **then** ▷ Miller Rabin Primality Test
            $pp \leftarrow i$
            break
        end if
    end if
    Increment i by 2
end for
Output: (pp)

2.2.3 Efficient VDF- Pietrzak

The Setup and Evaluation part of the Pietrzak's VDF are the same as Wesolowski's VDF.

Proof: The proof for Pietrzak's VDF is computed recursively. It uses the variables defined below. For $i \in [1, t]$ where $\mathcal{T} = 2^t$

$$u_i = x^{2^{\mathcal{T}/2^i}}$$

$$r_i = H(x_i + y_i + u_i)$$

$$x_{i+1} = x_i^{r_i}.u_i \ mod \ N; x_1 = x$$

$$y_{i+1} = u_i^{r_i}.y_i \ mod \ N; y_1 = y$$

Here, the proof is $\pi = \{u_i\}_{i \in [1,t]}$.

Verification: For verification, the verifier computes x_{t+1} and y_{t+1} the same way as in the proof section for each value of u_i received.

$$r_i = H(x_i + y_i + u_i)$$

$$x_{i+1} = x_i^{r_i}.u_i \ mod \ N; x_1 = x$$

$$y_{i+1} = u_i^{r_i}.y_i \ mod \ N; y_1 = y$$

Finally, it checks whether $y_{t+1} = x_{t+1}^2$.

2.2.4 Implementation of Efficient Simple VDF:

Pietrzak's construction also has two protocols- interactive and non-interactive [6]. The non-interactive protocol is based on Fiat-Shamir Heuristic. In this paper, I will only show the implementation of non-interactive protocol.

Algorithm 3 Pietrzak VDF

Evaluation and Proof

Input: $\mathcal{T} \in N$, $m \in \{0,1\}^*$
Process:
$x \leftarrow H(m)$
$y \leftarrow x$
for $i \leftarrow 1$ *to* $\mathcal{T}$ **do**
    $y \leftarrow y^2 \ mod \ N$ ▷ Repeated Squaring
end for
$t \leftarrow \lfloor \log_2(\mathcal{T}) \rfloor$
$(x_1, y_1) = (x, y)$
for $i \leftarrow 1$ *to* t **do**
    if $\mathcal{T} = Odd$ **then**
        $\mathcal{T} \leftarrow (\mathcal{T} + 1)/2$
        $u_i \leftarrow x_i^{2^\mathcal{T}}$
        $r_i \leftarrow H(x_i + y_i + u_i)$ ▷ Choose any $r \in \{1, 2, .., 2^\lambda\}$ to make this interactive
        $x_{i+1} \leftarrow x_i^{r_i}.u_i \ mod \ N$
        $y'_{i+1} \leftarrow u_i^{r_i}.y_i \ mod \ N$
        $y_{i+1} \leftarrow (y'_{i+1})^2$
    else
        $\mathcal{T} \leftarrow \mathcal{T}/2$
        $u_i \leftarrow x_i^{2^\mathcal{T}}$
        $r_i \leftarrow H(x_i + y_i + u_i)$
        $x_{i+1} \leftarrow x_i^{r_i}.u_i \ mod \ N$
        $y_{i+1} \leftarrow u_i^{r_i}.y_i \ mod \ N$
    end if
end for
Output: $\pi \leftarrow \{u_i\}_{i \in [1,t]}$

Verification

Input: $x, \mathcal{T}, pi$
Process:
$t \leftarrow \lfloor \log_2(\mathcal{T}) \rfloor$
$(x_1, y_1) = (x, y)$
for $i \leftarrow 1$ *to* t **do**
    $r_i \leftarrow H(x_i + y_i + u_i)$
    $x_{i+1} \leftarrow x_i^{r_i}.u_i \ mod \ N$
    $y_{i+1} \leftarrow u_i^{r_i}.y_i \ mod \ N$
end for
if $y_{t+1} = x_{t+1}^2$ **then**
    accept
else
    reject
end if
Output: $\{u_i\}_{i \in [1,t]}$

2.3 Performance Graph

After implementing both the constructions (Wesolowski's VDF and Pietrzak's VDF) on javascript, I tested their performance as mentioned in the paper by Attis, Vigneri and Dimitrov [1]. I ran the simulations for values of $\mathcal{T}$ between 2^{20} and 2^{25} and values of RSA modulus bit length λ in $\{256, 512, 1024\}$. These were my results:

Evaluation Time Analysis: The values for both Pietrzak and Wesolowski VDFs are identical here since both the implementations follow the same Evaluation procedure. As shown in Figure 1, as the number of exponentiation increases, the time taken to compute the function also increases. We can also see an impact of the RSA bit length as it yields a great variation.

Proof Time Analysis: From Figure 2, we can see that Wesolowski's proof takes much more time than Pietrzak's. This might be due to the next-prime function I have on my code for Wesolowski's proof. We can reduce the overall proof time of Wesolowski if we implement a more efficient algorithm for next-prime.

Verification Analysis: From Figure 3, we can see that Wesolowski's proof takes much longer to get verified. One reason could again be due to the next-prime function on my code.

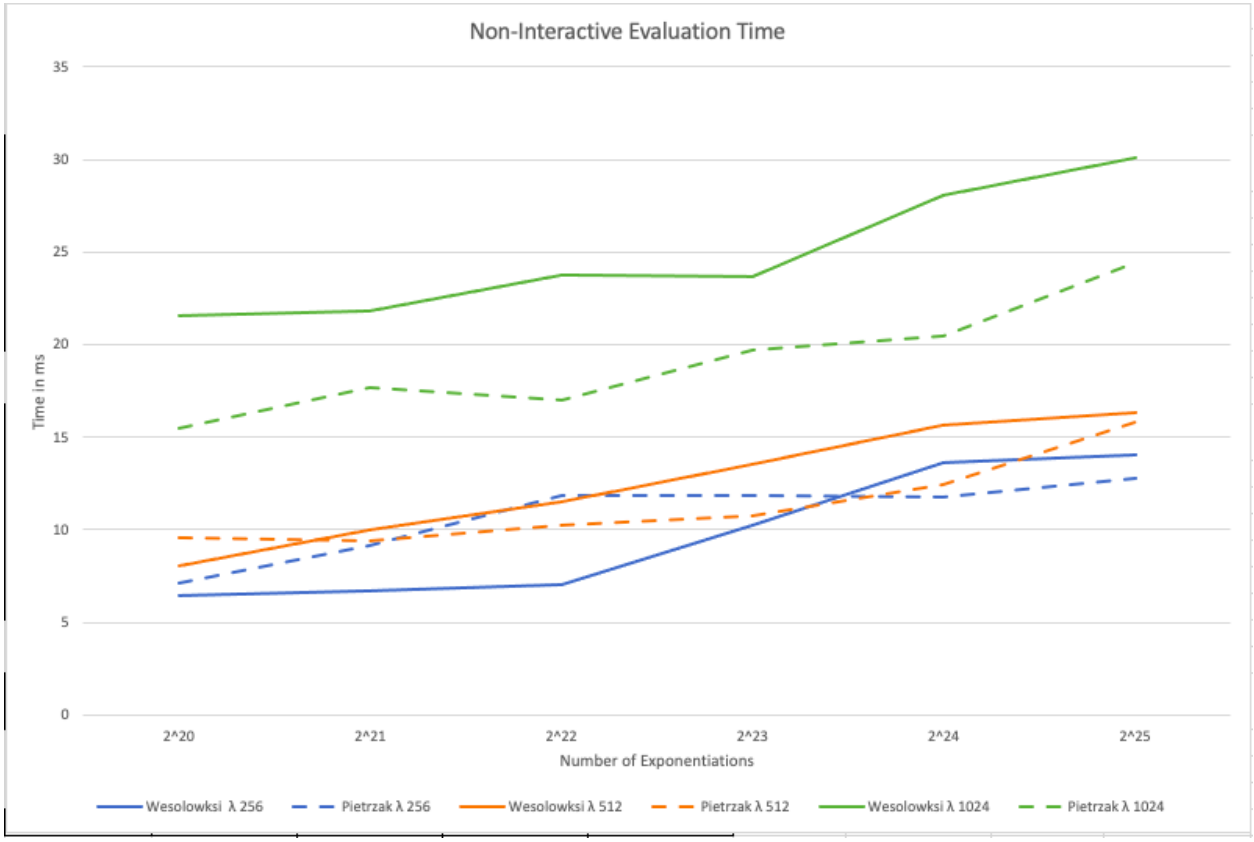

Figure 1: Evaluation Time

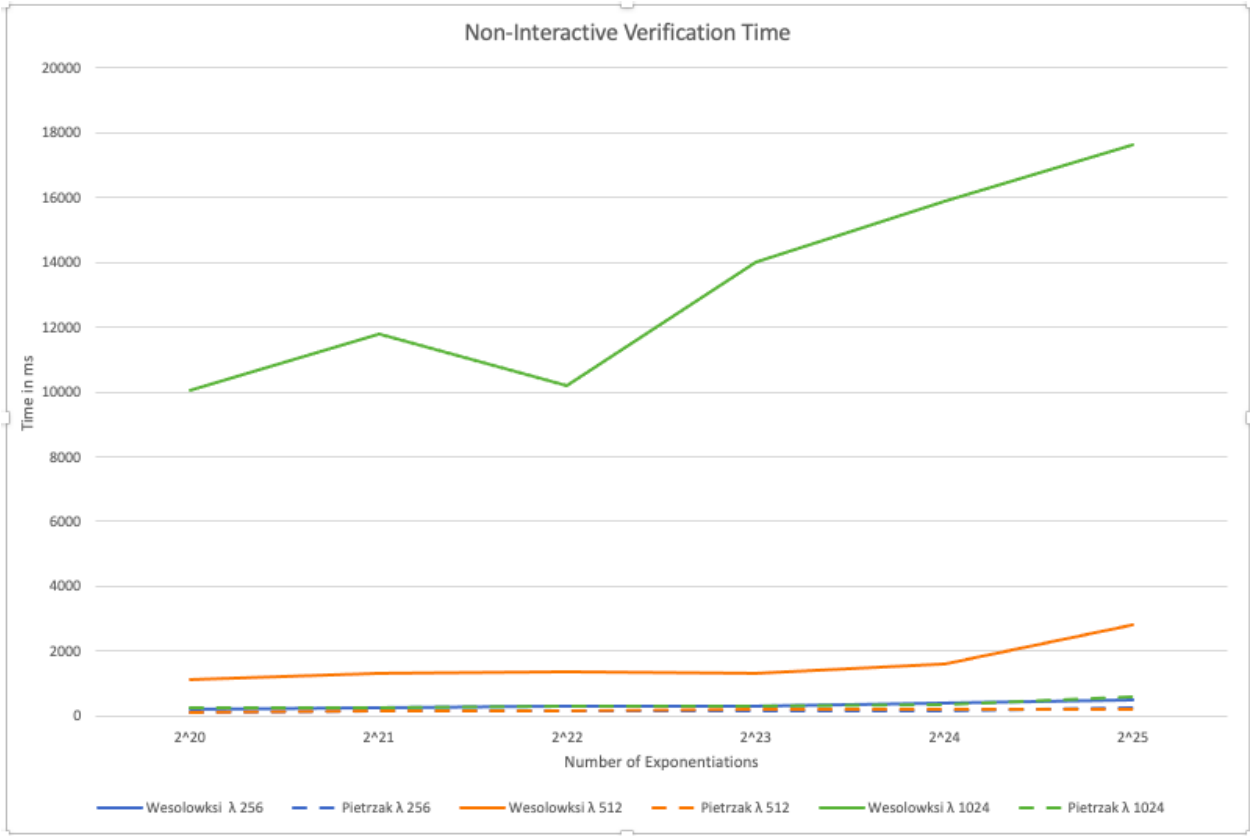

Figure 2: Proof Time

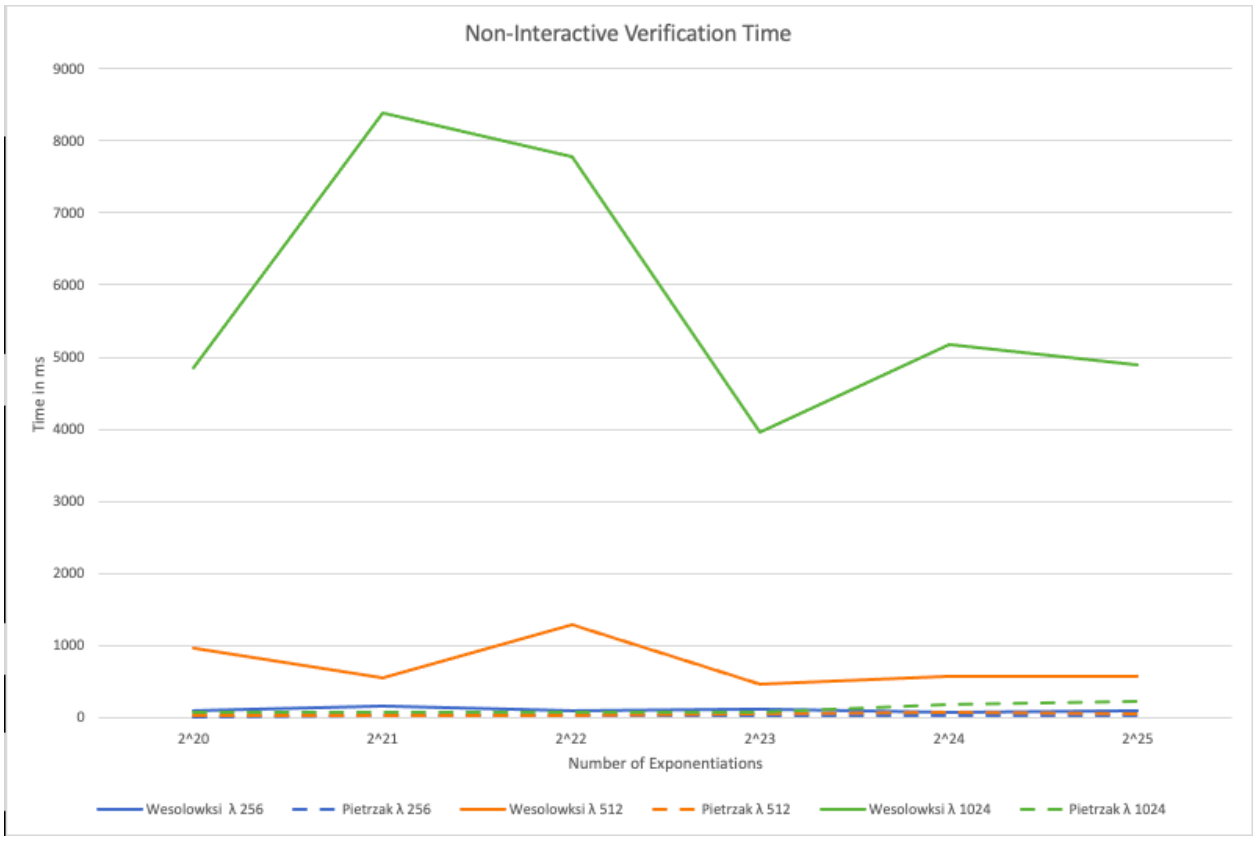

Figure 3: Verification Time

2.4 Application of VDF

Randomness Beacon: Randomness Beacon is a notion first suggested by Rabin in 1983 [7]as an ideal service that regularly publishes unpredictable and independent random values which no party can predict or manipulate. This concept has received a substantial amount of attention since its introduction, and even more so in recent years due to its many applications to more efficient and reliable consensus protocols in the context of blockchain technologies . VDFs are useful for constructing randomness beacons from source such as stock prices or proof-of-work blockchains [2]. It's sequentiality property ensures that it takes at least time $\mathcal{T}$ to compute the function even with parallel machines. Using a VDF, we can construct a randomness beacon where the output at interval $\mathcal{T}$ is computed as the $\mathcal{T}$-times repeated squaring of the seed x, i.e., $f(x, \mathcal{T}) = x^{2^{\mathcal{T}}}$. After time $\mathcal{T}$, the beacon's output should be unpredictable sufficiently far in the future.

3 Continuous Verifiable Delay Functions

One of the problems with VDF is that it can't be outsourced. This means that only its final state can be verified. So if anybody wants to verify it's current state, they would have to compute the entire function i.e. perform a repeated squaring for time $\mathcal{T}$ which is time consuming by definition. To solve this problem, Ephraim et. al [3] introduced the concept of Continuous Verifiable Delay functions (CVDF). A CVDF is essentially a VDF in which the intermediate steps of the VDF computation can be continuously and publicly verified. The existing CVDF is constructed based on Pietrzak's protocol. However, the existing CVDF construction scheme is not cost efficient in terms of proof size ($\mathcal{O}(b^2, h^2)$) where b is the branching factor of the tree and $h = \log_b(t)$, and verification time.

As a part of my capstone project, I also worked on the implementation of a new CVDF construction that is based on Wesolowski's protocol and the soundness of the Fiat-Shamir Heuristic [4]. This new construct is more cost efficient in terms of proof size $\mathcal{O}(b)$ and the verification time.

3.1 Implementation

There are two constructions of CVDF using Wesolowski's protocol [9]. One is a basic implementation where every state of Wesolowski's VDF is sent for verification. The other one is a recursive implementation called Stronger CVDF that uses a branching factor b as mentioned in the existing Peitrzak based CVDF [3].

Algorithm 4 Basic CVDF using Wesolowski

Evaluation and Proof

Input: $\mathcal{T} \in N$, $m \in \{0,1\}^*$
Process:
$x \leftarrow H(m)$
for $i \leftarrow 1$ *to* $\mathcal{T}$ **do**
    $y \leftarrow x^{2^i} \ mod \ N$ ▷ Repeated Squaring
    $l_i \leftarrow H_{prime}(x + y)$
    $\pi_i \leftarrow x^{\lfloor 2^i / l \rfloor} \ mod \ N$
end for
Output: $(\pi \leftarrow \{\pi_i\}, l \leftarrow \{l_i\} \ where \ i \in [1, \mathcal{T}])$

Verification

Input: x, $\mathcal{T}$, π, l
Process:
for $i \leftarrow 1$ *to* $\mathcal{T}$ **do**
    $r_i \leftarrow 2^i \ mod \ l_i$
    $y \leftarrow \pi_i^{l_i}.x^{r_i} \ mod \ N$
    if $l_i = H_{prime}(x + y)$ **then**
        accept
    else
        reject
        break
    end if
end for
Output: (*accept*, *reject*)

Algorithm 5 Stronger CVDF

Evaluation and Proof

procedure EVAL($(pp, m, \mathcal{T}, b)$)
  $g \leftarrow H(m)$
  if $\mathcal{T} < b$ **then**
    $y \leftarrow g^{2^{\mathcal{T}}} \ mod \ N$
    $l \leftarrow H_{prime}(x + y)$
    $\pi' \leftarrow g^{\lfloor 2^{\mathcal{T}}/l \rfloor} \ mod \ N$
  else
    for $i \leftarrow 1$ *to* b **do**
      if $i = 1$ **then**
        $x_i \leftarrow g$
        $\pi \leftarrow (x_i, y_i, \pi')$, where $(y_i, \pi') \leftarrow$EVAL$(pp, x_i, \mathcal{T}/b, b)$
      else
        $x_i \leftarrow g^{(i-1).\mathcal{T}/b} \ mod \ N$
        $\pi \leftarrow (x_i, y_i, \pi')$, where $(y_i, \pi') \leftarrow$EVAL$(pp, x_i, \mathcal{T}/b, b)$
      end if
    end for
  end if
  return (y, π)
end procedure

Verification

procedure VERIFY$(pp, m, y, \pi, b, \mathcal{T})$
  $g \leftarrow H(m)$
  if $\mathcal{T} < b$ **then**
    $l \leftarrow H_{prime}(x + y)$
    $r \leftarrow 2^{\mathcal{T}} \ mod \ l$
    if $\pi^l.g^r = y$ **then**
      accept
    else
      reject
    end if
  else
    for $i \leftarrow 1$ *to* b **do**
      VERIFY$(pp, g, y, \pi, b, \mathcal{T}/\lfloor)$
    end for
  end if
  return $(accept, reject)$
end procedure

3.2 Outsourcing with CVDF

We can show the outsourcing of CVDF using Multiplicative Homomorphic Encryption. For this project, I've used El-Gamal cryptosystem to show the outsourcing of CVDF. Proposed by Taher Elgamal [5], El-Gamal cryptosystem can be defined as follows:

1. **Key Generation:** The key generation requires a cyclic group G with order n using a generator g. Then, it computes $h = g^y$ for $y \in \mathcal{Z}_n^*$ and outputs the public key (G, n, g, h) and the secret key x.

2. **Encryption:** The message m is encrypted using g and x, where $x \in \{1, 2, ..., n-1\}$. The output is a ciphertext pair $(c = c_1, c_2)$:

$$c = E(m) = (g^x, mh^x) = (g^x, mg^{xy}) = (c_1, c_2)$$

3. **Decryption:** The decryption algorithm works as follows:

$$c_2.s^{-1} = mg^{xy}.g^{-xy} = m$$

4. **Homomorphic Property:** The homomorphic multiplicative property works as follows:

$$E(m_1).E(m_2) = (g^{x_1}, m_1h^{x_1}) \times (g^{x_2}, m_2h^{x_2}) = g^{x_1+x_2}, m_1 \times m_2h^{x_1+x_2} = E(m_1 \times m_2)$$

3.2.1 Limitations on the coding segment:

1. While I managed to implement a basic version of outsourceable CVDF, my code does not work for larger values of x or time bound $\mathcal{T}$. The code works until the output generated i.e. the final decrypted value is less than the RSA modulus N. I tried applying the modulus ($mod\ N$) on the final decrypted value. While this approach seemed intuitive and correct to me, it did not give me a correct output for larger values.

2. I tried implementing the outsourcing of CVDF using two clients. However, since the encryted values were big-integers, I was unable to send these values from one client to the other. Therefore, my code does not show a peer-to-peer communication.

4 Future Work

4.1 Double Spending

While the CVDF solves the problem of outsourcing VDF, it can't handle the 'double spending' problem of the outsourceable VDF. The double spending problem arises when the same amount of work is outsourced to more than one person. Eg: Assume Alice tries to compute

a VDF $y = x^{2^t}$. She works on the problem for time $t/2$ and decides to outsource it to Bob. With CVDF, this is possible since each of the states of VDF can be verified. However, there is always a possibility that Alice will outsource her work to another person, say Carol. This leads to a double spending problem. Future work concerns solving the double spending problem and implementing it.

References

[1] Vidal Attias, Luigi Vigneri, and Vassil Dimitrov. "Implementation Study of Two Verifiable DelayFunctions". In: *IACR Cryptol. ePrint Arch.* 2020 (2020), p. 332. URL: https://eprint.iacr.org/2020/332.

[2] Dan Boneh et al. "Verifiable Delay Functions". In: *IACR Cryptol. ePrint Arch.* 2018 (2018), p. 601. URL: https://eprint.iacr.org/2018/601.

[3] Naomi Ephraim et al. "Continuous Verifiable Delay Functions". In: *Advances in Cryptology - EUROCRYPT 2020, 39th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, May 10-14, 2020*. Ed. by Anne Canteaut and Yuval Ishai. Springer, 2020, pp. 125–154. DOI: 10.1007/978-3-030-45727-3_5. URL: https://doi.org/10.1007/978-3-030-45727-3%5C_5.

[4] Amos Fiat and Adi Shamir. "How to Prove Yourself: Practical Solutions to Identification and Signature Problems". In: *Advances in Cryptology - CRYPTO '86, Santa Barbara, California, USA, 1986, Proceedings*. Ed. by Andrew M. Odlyzko. Vol. 263. Lecture Notes in Computer Science. Springer, 1986, pp. 186–194. DOI: 10.1007/3-540-47721-7_12. URL: https://doi.org/10.1007/3-540-47721-7%5C_12.

[5] Taher El Gamal. "A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms". In: *Advances in Cryptology, Proceedings of CRYPTO '84, Santa Barbara, California, USA, August 19-22, 1984, Proceedings*. Ed. by G. R. Blakley and David Chaum. Vol. 196. Lecture Notes in Computer Science. Springer, 1984, pp. 10–18. DOI: 10.1007/3-540-39568-7_2. URL: https://doi.org/10.1007/3-540-39568-7%5C_2.

[6] Krzysztof Pietrzak. "Simple Verifiable Delay Functions". In: *IACR Cryptol. ePrint Arch.* 2018 (2018), p. 627. URL: https://eprint.iacr.org/2018/627.

[7] Michael O. Rabin. "Transaction Protection by Beacons". In: *J. Comput. Syst. Sci.* 27.2 (1983), pp. 256–267. DOI: 10.1016/0022-0000(83)90042-9. URL: https://doi.org/10.1016/0022-0000(83)90042-9.

[8] Ronald L Rivest, Adi Shamir, and David A Wagner. "Time-lock puzzles and timed-release crypto". In: (1996).

[9] 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*. Ed. by Yuval Ishai and Vincent Rijmen. Springer, 2019, pp. 379–407. DOI: 10.1007/978-3-030-17659-4_13. URL: https://doi.org/10.1007/978-3-030-17659-4%5C_13.