A Practical Verifiable Delay Function and Delay Encryption Scheme

Angelique Faye Loe[1], Liam Medley[1], Christian O'Connell[2], and Elizabeth A. Quaglia[1]

[1] Royal Holloway University of London `angelique.loe.2016, liam.medley.2018, elizabeth.quaglia, @rhul.ac.uk`
[2] Independent `co362@cantab.ac.uk`

Abstract. We present a novel construction for a Verifiable Delay Function (VDF), in which the prover is challenged to produce the factorisation of a special class of RSA modulus. Our approach produces a VDF with a very efficient verification procedure.
The properties of our VDF allow us to establish the design of the first practical Delay Encryption scheme, a primitive introduced at EUROCRYPT 2021. We provide a formal security analysis of our results, as well as an implementation study detailing the practical performance of our VDF.

Keywords: verifiable delay function, proof of work, delay encryption

1 Introduction

Verifiable delay functions (VDFs) were introduced in 2018 by Boneh et al. [5] as a primitive which takes a fixed length of time t to compute, regardless of parallelism, and is fast to verify. VDFs use a computational task, e.g., repeated squaring, to provide a reliable 'proof of elapsed time' in decentralised applications such as randomness beacons and the design of blockchain protocols [8].

Boneh et al. left as an open problem the construction of a theoretically optimal VDF, which they describe as a simple, inherently sequential function requiring low parallelism to compute, and which is very fast to invert. Shortly after, three papers were proposed to address this problem. Wesolowski [31] and Pietrzak [25] each proposed a scheme based on repeated squaring in a group of unknown order, with different protocols to verify the proof of correct computation. Then De Feo et al. [13] proposed a VDF framework based on isogenies of supersingular elliptic curves, which improved upon the evaluator efficiency by having an empty proof, at the cost of requiring a very long setup.

Each of these schemes is an improvement upon the schemes suggested by Boneh et al. in their seminal paper, but none have a verification procedure with a runtime of $O(1)$, and hence are not theoretically optimal.

In this work we present a theoretically optimal VDF candidate, which improves upon the current state of the art, combining the efficiency benefits of an empty proof with a fast setup algorithm, and a verification procedure which runs in $O(1)$. The underlying computational task is the same as that of Wesolowski and Pietrzak's schemes, namely repeated squaring, but the essence of our VDF is different: the challenge is to factorise a particular type of RSA modulus.

In the schemes by Wesolowski and Pietrzak, the challenge given to the evaluator is a random value, which is then raised to the power of 2^t. The evaluator must also compute a proof, which is then used in the verification protocol to check the computation output is correct. By contrast, our VDF challenge is carefully chosen to ensure the computation output is useful: by evaluating the VDF, the prover is able to factorise the RSA modulus. Therefore, instead of computing a proof and running a protocol, our VDF output can be quickly verified by multiplying the factors of the RSA modulus.

An additional benefit of this feature is that, if we allow parties to encrypt messages to this RSA modulus, our VDF can be used to ensure that the decryption key is recovered, and therefore the ciphertexts can be decrypted, only after the prescribed delay. This is the premise of Delay Encryption (DE), a primitive introduced by Burdges and De Feo [7] at EUROCRYPT 2021. In Burdges et al [7], a DE scheme is derived from the isogeny-based VDF, which is based on a key encapsulation mechanism, and outputs a hash function as a key derivation function. However, this scheme comes with concrete implementation issues, including very large evaluator storage. By combining our VDF with RSA encryption in a natural way, we obtain a DE scheme based on a public key encryption scheme instead, avoiding these implementation issues and inheriting the underlying efficiency of our VDF.

We go on to present a practical implementation of our construction across two hardware environments. We first use a cluster of four Raspberry Pi 3 Model B computers (quad-core 1.2 GHz CPU, 1 GB of memory) to demonstrate performance of different modulus sizes in an embedded environment. We complement these results with an implementation on a desktop PC using a consumer grade quad-core 3.2 GHz Intel i5 processor, with 16 GB of memory.

1.1 Contributions

In our work we construct a novel, theoretically optimal VDF candidate, i.e., it is simple, inherently sequential, and very fast to invert. We provide a security and efficiency analysis of our construction, proving that our VDF is cryptographically secure, and theoretically more efficient than the alternative candidates.

We present evidence of the practicality of our scheme by providing an implementation study using Raspberry Pi devices and a desktop PC, showing that our VDF can be run efficiently on consumer grade hardware. We show that running a 2048-bit setup takes approximately one second on a desktop PC and 30 seconds on a Raspberry Pi. Furthermore, the verification on both devices runs in less than one second regardless of the time parameter.

Finally, we build the first practically implementable Delay Encryption scheme, using our VDF scheme and the RSA-OAEP public key cryptographic scheme [20]. We prove the scheme secure and implement it on a desktop PC.

Paper layout In Section 2 we provide a formal, game-based definition of a VDF, following Pietrzak [25]. In Section 3 we introduce our P-VDF construction. We first give a brief overview, before giving the number theoretic preliminaries, and then provide the formal construction. In Section 4 we prove formally that our construction meets the security properties defined in Section 2. In Section 5 we compare the theoretical efficiency and security assumptions of our construction against other VDF candidates. In Section 6 we give details of our implementation, showing the concrete efficiency of our construction. In Section 7 we extend our P-VDF construction to build an efficient delay encryption scheme.

2 Verifiable Delay Functions

We now provide the definition and properties of a generic VDF [5,25]. Note that $\mathcal{V}$ is the verifier, $\mathcal{P}$ is the prover, and $\mathcal{A}$ is the adversary.

2.1 VDF Definition

In this work we follow Pietrzak [25], and define a VDF as a tuple of algorithms: (Setup, Gen, Eval, Verify). In our notation $\leftarrow$ indicates deterministic evaluation, and $\leftarrow_{\text{R}}$ indicates randomised evaluation.

– $\mathsf{pp} \leftarrow_{\text{R}} \mathsf{Setup}(1^\kappa)$. Setup is an algorithm run by $\mathcal{V}$ that takes security parameter 1^κ and outputs the public parameter pp. Setup runs in time $\mathsf{poly}(\kappa)$.
– $(\mathsf{pp}, C, t) \leftarrow_{\text{R}} \mathsf{Gen}(\mathsf{pp}, t)$. Gen is an algorithm run by $\mathcal{V}$ that takes the public parameter pp and a time parameter t and outputs a random challenge C. The parameter t indicates the number of sequential steps required to evaluate C. Gen runs in time $\mathsf{poly}(\kappa)$.
– $y \leftarrow \mathsf{Eval}(\mathsf{pp}, C, t)$. Eval is an algorithm run by $\mathcal{P}$ that takes the public parameter pp, a challenge C and evaluates a solution y in t sequential steps. To qualify as a VDF, the solution y must be unique.
– $\{\text{accept}, \text{reject}\} \leftarrow \mathsf{Verify}(\mathsf{pp}, C, t, y)$. Verify is an algorithm run by $\mathcal{V}$ that confirms the soundness of solution y against the challenge C and time parameter t. If y is sound Verify outputs accept. Verify runs in time $\mathsf{polylog}(t)$ and $\mathsf{poly}(\kappa)$.

2.2 VDF Properties

In this section we provide the properties of a VDF. A VDF must be: correct, sound, sequential, efficiently verifiable, and unique [5, 25].

– A VDF is *correct* if a solution output by Eval is accepted by Verify with overwhelming probability. This is made precise in the correctness game, Game 2.1.
– A VDF is *sound* if any solution y' output by any algorithm $\mathcal{E}$, where $y' \neq y \leftarrow \mathsf{Eval}$, has a negligible probability of being accepted by Verify. This is made precise in the soundness game, Game 2.2.

```
1 $\mathcal{V}$ generates random public parameter, challenge, and time parameter:
    $\mathsf{pp} \leftarrow_{\mathrm{R}} \mathsf{Setup}(1^\kappa)$, $(\mathsf{pp}, C, t) \leftarrow_{\mathrm{R}} \mathsf{Gen}(\mathsf{pp}, t)$.;
2 $\mathcal{P}$ selects the legitimate algorithm Eval to generate solution:
    $y \leftarrow \mathsf{Eval}(\mathsf{pp}, C, t)$.;
3 $\mathcal{V}$ verifies the solution: $\{\text{accept}, \text{reject}\} \leftarrow \mathsf{Verify}(\mathsf{pp}, C, t, y)$.;
  $\mathcal{P}$ wins if Verify outputs accept.;
  A VDF is correct if $\mathcal{P}$ wins with probability $1 - \mathsf{negl}(\kappa)$.;
```

Game 2.1: VDF Correctness Game

```
1 $\mathcal{V}$ generates random public parameter, challenge, and time parameter:
    $\mathsf{pp} \leftarrow_{\mathrm{R}} \mathsf{Setup}(1^\kappa)$, $(\mathsf{pp}, C, t) \leftarrow_{\mathrm{R}} \mathsf{Gen}(\mathsf{pp}, t)$.;
2 $\mathcal{A}$ selects a PPT algorithm $\mathcal{E}$, to generate solution: $y' \leftarrow \mathcal{E}(\mathsf{pp}, C, t)$, where
    $y' \neq y \leftarrow \mathsf{Eval}(\mathsf{pp}, C, t)$.;
3 $\mathcal{V}$ verifies the solution: $\{\text{accept}, \text{reject}\} \leftarrow \mathsf{Verify}(\mathsf{pp}, C, t, y')$.;
  $\mathcal{A}$ wins if Verify outputs accept.;
  A VDF is sound if $\mathcal{A}$ wins with probability $\mathsf{negl}(\kappa)$.;
```

Game 2.2: VDF Soundness Game

– A VDF is *sequential* if the following is true. Suppose $\mathcal{A}$ is provided with a public parameter pp and is given a polynomially bounded time to precompute on this public parameter prior to being provided with a random challenge C. The property of sequentiality follows if $\mathcal{A}$ is unable to compute an output y' by an algorithm $\mathcal{E}_{<t}$ which takes less that t sequential steps to calculate, such that $y' = y \leftarrow \mathsf{Eval}(\mathsf{pp}, C, t)$. This is made precise in the sequentiality game, Game 2.3.

```
1 $\mathcal{V}$ generates random public parameter: $\mathsf{pp} \leftarrow_{\mathrm{R}} \mathsf{Setup}(1^\kappa)$.;
2 $\mathcal{A}$ selects a PPT algorithm $\mathcal{E}_p$ to pre-process $\mathcal{L} \leftarrow \mathcal{E}_p(\mathsf{pp}, t)$, where $\mathcal{E}_p$ runs in
    time $O(\mathsf{poly}(t, \kappa))$.;
3 $\mathcal{V}$ generates random challenge, $C \leftarrow_{\mathrm{R}} \mathsf{Gen}(\mathsf{pp}, t)$.;
4 $\mathcal{A}$ selects a PPT algorithm $\mathcal{E}_{<t}$, where $\mathcal{E}_{<t}$ runs in less than $t$ sequential steps
    to generate solution: $y' \leftarrow \mathcal{E}_{<t}(\mathcal{L}, \mathsf{pp}, C, t)$.;
  $\mathcal{A}$ wins if $y' = y \leftarrow \mathsf{Eval}(\mathsf{pp}, C, t)$.;
  A VDF is sequential if $\mathcal{A}$ wins with probability $\mathsf{negl}(\kappa)$.;
```

Game 2.3: VDF Sequentiality Game

– *Efficient verifiability* is achieved if Verify runs in time $\mathsf{polylog(t)}$ and $\mathsf{poly}(\kappa)$. Verify must be notably faster than Eval.
– *Uniqueness* is achieved if for any tuple (pp, C, t) output by Setup and Gen, only a single value of the solution y will be accepted by Verify. That is, the solution y can only be equal to a single value.

3 Practical VDF

We begin this section by providing an overview of our Practical VDF (P-VDF).

P-VDF overview A summary of the inputs and outputs for the Setup, Gen, Eval, and Verify algorithms of our P-VDF construction are as follows:

$$\begin{aligned} \mathsf{pp} := N &\xleftarrow{\text{R}} \mathsf{Setup}(1^\kappa) \\ (C := (x, x_0, x_{-t}), t) &\xleftarrow{\text{R}} \mathsf{Gen}(\mathsf{pp}, t) \\ y := (p', q') &\leftarrow \mathsf{Eval}(\mathsf{pp}, C, t) \\ \{\text{accept}, \text{reject}\} &\leftarrow \mathsf{Verify}(\mathsf{pp}, C, t, y) \end{aligned}$$

In our P-VDF Setup computes the public parameter $\mathsf{pp} := N$ which is a special class of RSA modulus known as a Blum integer [4]. Gen then generates the challenge $C := (x, x_0, x_{-t})$. To generate C the element x is efficiently sampled such that $\mathcal{J}_N(x) = -1$, where $\mathcal{J}_N(x)$ is the Jacobi symbol [20]. Next, the seed is calculated as $x_0 \equiv x^2 \bmod N$. Crucial to our P-VDF is the term x_{-t}, where $x_{-t}^{2^t} \equiv x_0 \bmod N$. We will provide the details of the efficient calculation of the term x_{-t} with the use of the Chinese Remainder Theorem in Section 3.2.

Eval then sequentially calculates the term $x_{-1} := x' \equiv \sqrt{x_0}$ by repeated squaring. The term x' has the property $\mathcal{J}_N(x') = +1$. Therefore, Eval obtains $x^2 \equiv x'^2 \equiv x_0 \bmod N$, where $x \neq x' \bmod N$. Thus Eval obtains all four square roots of x_0 once x' is sequentially calculated. Therefore, Eval can recover the non-trivial factors of N in polynomial time using the result from Rabin [26]. Once Eval recovers the factors of N it sets the solution $y := (p', q')$. Finally, Verify outputs accept if $p'q' = N$. The insight of our P-VDF construction is how the term x_{-t} is calculated by the Gen algorithm, and how the Eval algorithm is able to calculate the non-trivial factors of N.

In the subsequent subsections we review the relevant time-lock assumption and number theory which are required to understand the specifics of our construction. These theorems will be relevant to the formal security analysis of our construction in Section 4. We also provide the detail of each algorithm in our P-VDF construction.

3.1 Time-Lock Assumption and Number Theory

We now review the time-lock assumption and number theory required to construct our P-VDF. For well-known theorems we refer to the relevant sources and we prove the other theorems in Section 4 and Appendix A.1.

The RSW time-lock assumption [27] is core to a number of notable VDFs in the latest literature [5,12,25,31]. Like RSW-based VDFs our P-VDF also relies on this assumption.

Definition 1. ***RSW time-lock assumption:*** *Let $N = pq$ where p and q are distinct odd primes. Uniformly select $x \in \mathbb{Z}_N^*$, where $\mathbb{Z}_N^* = \{x \,|\, x \in (0, N) \wedge$*

$\mathsf{gcd}(x, N) = 1\}$. *Then set the seed term as* $x_0 := x^2 \bmod N$. *If a probabilistic polynomial time (PPT) adversary* $\mathcal{A}$ *does not know the factorisation of* N *or group order* $\phi(N)$ *then calculating* $x_t \equiv x_0^{2^t} \bmod N$ *is a non-parallelizable calculation that will require* t *sequential modular exponentiations calculated with the Algorithm 3.1 Square and Multiply [27].*

Algorithm 3.1: Square and Multiply [10]

```
  input : $(a, b, N)$, // $a, b, N \in \mathbb{N}$, $a^b \bmod N$
1 $d := 1$
2 $B := \mathsf{bin}(b)$ // $b$ in binary
3 for $j \in B$ do
4     $d := d^2 \bmod N$
5     if $j = 1$ then
6         $d := da \bmod N$
7     end
8 end
  output: $d$
```

Secondly we note that the modulus used in our P-VDF will be a Blum integer [4]. A Blum integer $N = pq$, is the product of two Gaussian primes. A Gaussian prime has the property $p \equiv 3 \bmod 4$.

Next, we provide the definition of quadratic residues.

Definition 2. ***Quadratic Residues*** *in* $\mathbb{Z}_N^*$ *are numbers* r *that satisfy congruences of the form:*

$$x^2 \equiv r \mod N \tag{1}$$

If an integer x *exists such that the preceding congruence is satisfied, we say that* r *is a quadratic residue of* N. *If no such* x *exists we say that* r *is a quadratic non-residue of* N.

The Jacobi symbol, denoted $\mathcal{J}_N(r)$, is a function which defines the quadratic character of r in Equation 1. The Jacobi Symbol can be calculated in polynomial time using Euler's Criterion.

Theorem 1. *Euler's Criterion can be used to calculate the Jacobi Symbol of the number* r *in Equation 1 for a prime modulus* p. *If* $\mathtt{gcd}(r, p) = 1$, *then:*

$$\mathcal{J}_p(r) = r^{\frac{p-1}{2}} = \begin{cases} +1, \text{if} r \in \mathcal{QR}_p \\ -1, \text{if} r \in \mathcal{QNR}_p \end{cases} \tag{2}$$

Where $r \in \mathcal{QR}_p$ *indicates that* r *is a quadratic residue of* p *and* $r \in \mathcal{QNR}_p$ *indicates that* r *is a quadratic non-residue of* p.

When the modulus is a prime number if the Jacobi symbol evaluates to $+1$ then r is always a quadratic residue and if the Jacobi symbol evaluates to -1 then r is always a quadratic non-residue. The Jacobi symbol is more complex when the modulus is a composite number $N = pq$.

Corollary 1. *(Of Theorem 1). Euler's Criterion can be used to calculate the Jacobi Symbol of the number r in Equation 1 for a composite modulus N if the factorisation of N is known.*

Algorithm 3.2 shows how to determine the quadratic character of r for composite N using Theorem 1 and Corollary 1. When N is composite the quadratic character of r can take three formats. If the Jacobi symbol evaluates to -1 then r is always a quadratic non-residue, denoted $\mathcal{QNR}_N^{-1}$. However, if the Jacobi symbol evaluates to $+1$ then r can either be a quadratic residue, denoted $\mathcal{QR}_N$ or a quadratic non-residue denoted $\mathcal{QNR}_N^{+1}$.

Algorithm 3.2: Calculating $\mathcal{J}_N(r)$ for composite N.

```
  input : $(r, p, q)$
1 $\mathcal{J}_p(r) := r^{\frac{p-1}{2}} \bmod p$
2 $\mathcal{J}_q(r) := r^{\frac{q-1}{2}} \bmod q$
3 if $\mathcal{J}_p(r) = 1 \wedge \mathcal{J}_q(r) = 1$ then
4     $x := \mathcal{QR}_N$
5 else if $\mathcal{J}_p(r) = -1 \wedge \mathcal{J}_q(r) = -1$ then
6     $x := \mathcal{QNR}_N^{+1}$
7 else
8     $x := \mathcal{QNR}_N^{-1}$
9 end
  output: $x$
```

Quadratic residues and quadratic non-residues for composite N have a distinct distribution in $\mathbb{Z}_N^*$.

Theorem 2. *The cardinality of $\mathcal{QR}_N$, $\mathcal{QNR}_N^{+1}$, and $\mathcal{QNR}_N^{-1}$ for composite $N = pq$, where p and q are distinct primes is as follows:*

$$\begin{aligned} \left|\mathcal{QR}_N\right| &= \frac{\left|\mathbb{Z}_N^*\right|}{4} = \frac{\phi(N)}{4}. \\ \left|\mathcal{QNR}_N^{+1}\right| &= \frac{\left|\mathbb{Z}_N^*\right|}{4} = \frac{\phi(N)}{4}. \\ \left|\mathcal{QNR}_N^{-1}\right| &= \frac{\left|\mathbb{Z}_N^*\right|}{2} = \frac{\phi(N)}{2}. \end{aligned} \tag{3}$$

Where, $\left|\mathbb{Z}_N^*\right| = \phi(N) = (p-1)(q-1)$, and $\phi(N)$ is Euler's totient function.

Next, we discuss how to calculate preceding terms of the seed term $x_0 \in \mathcal{QR}_N$ in an RSW time-lock sequence. To calculate the subsequent term of x_0 in the sequence evaluate $x_1 \equiv x_0^{2^1} \bmod N$ by inputting $(x_0, 2^1, N)$ into Algorithm 3.1.

If the factorisation of N is known Theorem 1 can be used in conjunction with the Chinese Remainder Theorem (CRT) to calculate the term x_{-1} in polynomial time. The CRT can be found in our Auxiliary material.

Theorem 3. *Let p be a Gaussian prime. For any $r \in \mathbb{Z}_p^*$, if $\mathcal{J}_p(r) = +1$, then finding α such that $\alpha \equiv \sqrt{r} \bmod p$ can be found by calculating $\alpha \equiv r^{\frac{p+1}{4}} \bmod p$.*

Example 1. Let $N = 67 \cdot 139 = pq = 9313$. Given the seed $x_0 = 776 \in \mathcal{QR}_N$, the square root of $x_0 \bmod N$, denoted by $x_{-1} = \sqrt{x_0}$, can be found as follows:

- calculate $\alpha \equiv x_0^{\frac{p+1}{4}} \equiv x_0^{17} \equiv 21 \bmod p$
- calculate $\beta \equiv x_0^{\frac{q+1}{4}} \equiv x_0^{35} \equiv 9 \bmod q$
- calculate $x_{-1} = \alpha q(q^{-1} \bmod p) + \beta p(p^{-1} \bmod q) = 128862$

Then α and β are calculated using Theorem 3 and x_{-1} is calculated using the CRT. Note that $(q^{-1} \bmod p)$ and $(p^{-1} \bmod q)$ are calculated using Euclid's Extended Algorithm. To verify correctness, note that $128862^2 \equiv 776 \equiv x_0 \bmod N$. We provide formal analysis of this in Section 4.

If $r \in \mathcal{QR}_N$ then the CRT implies that there are four distinct solutions to Equation 1.

Theorem 4. *For all $N = pq$, where p and q are distinct odd primes, each $r \in \mathcal{QR}_N$ has four distinct solutions.*

If N is a Blum integer, then the four square roots of each $r \in \mathcal{QR}_N$ has specific properties. That is, two of the square roots of r are quadratic non-residues with Jacobi symbol -1, one square root is a quadratic non-residue with Jacobi symbol $+1$, and one square root is a quadratic residue.

Theorem 5. *Let N be a Blum integer. Then for all $r \in \mathcal{QR}_N$, if $x^2 \equiv x'^2 \equiv r \bmod N$, where $x \neq \pm x'$, then without loss of generality $\mathcal{J}_N(\pm x) = -1$, and $\mathcal{J}_N(\pm x') = +1$. That is $\pm x \in \mathcal{QNR}_N^{-1}$, $x' \in \mathcal{QR}_N$ and $-x' \in \mathcal{QNR}_N^{+1}$. We refer to $x' \in \mathcal{QR}_N$ as the principal square root of $r \bmod N$.*

Finally, we discuss a method to factor a Blum integer N in polynomial time if specific information is provided.

Fermat's factorisation method is a technique to factor an odd composite number $N = pq$ in exponential time [11]. The method requires finding x and x' such that $x^2 - x'^2 = N$ is satisfied. Then the left-hand side can be expressed as a difference of squares $(x - x')(x + x') = N$.

Fermat's method can be extended to finding x and x' to satisfy the following weaker congruence of squares condition $x^2 \equiv x'^2 \bmod N$, where $x \not\equiv \pm x'$. This congruence can be expressed as $(x-x')(x+x') \equiv 0 \bmod N$. Finding a congruence of squares forms the basis for several sub-exponential sieving-based factorisation algorithms [11]. However, if x and x' in a congruence of squares are known, then factoring N can be done in polynomial time.

Theorem 6. *Let N be a Blum integer. If x and x' are known such that $x^2 \equiv {x'}^2 \bmod N$, where $x \not\equiv \pm x' \bmod N$, then the non-trivial factors of N can be recovered in polynomial time.*

Proof. Proofs for Theorems 1, 2, 4, and Corollary 1 can be found in [20]. Proofs for Theorems 3 and 6 can be found in Section 4. The proof for Theorem 5 can be found in Appendix A.1.

3.2 P-VDF Construction

We now provide the details of our P-VDF construction. We begin by presenting the notation for the pseudo code of our implementation, and then we give the full details of our construction.

Notation In the pseudo code $:=$ indicates assignment, $=$ indicates equality, $\neq$ indicates inequality, () indicates a tuple, and $\mathcal{U}(a, b)$ indicates the uniform selection of an integer that is between $a, b \in \mathbb{Z}$, where $a < b$ and a, b are inclusive. The symbols $\wedge$, $\vee$, $\neg$ indicate logical conjunction (and), logical (inclusive) disjunction (or), and negation (not), respectively.

Construction of the P-VDF We now provide the full details of the P-VDF.

Algorithm 3.3: Setup run on security parameter 1^κ to create the public parameter pp.

```
input : $1^\kappa$
1 $p, q := 1$
2 while $p = q$ do
3     $p := \texttt{prime}(\frac{\kappa}{2})$
4     $q := \texttt{prime}(\frac{\kappa}{2})$
5 end
6 $N := pq$
```

1) $\mathcal{V}$ runs $\mathsf{pp} := N \xleftarrow{R} \mathsf{Setup}(1^\kappa)$ to generate the public parameter as seen on Algorithm 3.3 Setup. The function $\texttt{prime}(j)$ on lines 3 and 4 is the Miller-Rabin Monte Carlo algorithm [22] which generates j bit Gaussian primes. That is, $p \xleftarrow{R} \texttt{prime}(j)$. This guarantees that N, which is calculated on line 6, is a Blum integer. $\mathcal{V}$ then runs $(C := (x, x_0, x_{-t}), t) \xleftarrow{R} \mathsf{Gen}(\mathsf{pp}, t)$ to generate the challenge and the time parameter as seen in Algorithm 3.4 Gen. Gen first enters a while loop. The purpose of the while loop is to find an x such that $x \in \mathcal{QNR}_N^{-1}$. The logic statement on line 1 condenses the conditional statements in lines 3, 5 and 7 of Algorithm 3.2 using De Morgan's laws [17]. Once a suitable x is found x_0 is set to $x^2 \bmod N$.

Algorithm 3.4: Gen run on public parameter pp and time parameter t to create the public challenge C.

```
   input  : $\mathsf{pp}, t$
1  $\mathcal{J}_p(x), \mathcal{J}_q(x) := 1$
2  while $\neg(\mathcal{J}_p(x) = 1 \land \mathcal{J}_q(x) \neq 1) \land \neg(\mathcal{J}_p(x) \neq 1 \land \mathcal{J}_q(x) = 1)$ do
3      $x := \mathcal{U}(2, N)$
4      $\mathcal{J}_p(x) := x^{\frac{p-1}{2}} \bmod p$
5      $\mathcal{J}_q(x) := x^{\frac{q-1}{2}} \bmod q$
6  end
7  $x_0 := x^2 \bmod N$
8  $\alpha_t := x_0^{{\frac{p+1}{4}}^t \bmod p-1} \bmod p$
9  $\beta_t := x_0^{{\frac{q+1}{4}}^t \bmod q-1} \bmod q$
10 $x_{-t} := \alpha_t q(q^{-1} \bmod p) + \beta_t p(p^{-1} \bmod q) \bmod N$
11 $C := (x, x_0, x_{-t})$
   output: $(\mathsf{pp} = N, C, t)$
```

We note that it is also possible to efficiently sample parameter x without knowledge of p and q using algorithm A.2 Samplex, which we present in Appendix A.2. Samplex allows any party to select the randomised challenge for $\mathcal{P}$, whilst still ensuring that $x \in \mathcal{QNR}_N^{-1}$, and it can be used in lieu of lines 1–6 of the Algorithm 3.4 Gen. This use of Samplex moderates the trust assumption of Gen by allowing a party that does not run Setup to choose the input.

Once x is sampled and x_0 is computed the term x_{-t} is calculated, where $x_{-t}^{2^t} \equiv x_0 \bmod N$. To calculate x_{-t} in polynomial time, Euler's Criterion, the Fermat-Euler Theorem and the CRT must be applied. A toy example of this calculation is noted in Example 1 and the Fermat-Euler Theorem can be found in Appendix A.1, Theorem 14.

In Algorithm 3.4 Gen, we let $\omega = \frac{p+1}{4}$, then Theorem 3 tells us that $\alpha \equiv \sqrt{x_0} \equiv x_0^{\omega} \bmod p$. Let α_t be the t^{th} square root of $x_0 \bmod p$. For example, if $t = 2$, then $\alpha_2 \equiv \sqrt{\sqrt{x_0}} \equiv (x_0^{\omega})^{\omega} \equiv x_0^{\omega^2}$. Therefore, $\alpha_t \equiv x_0^{\omega^t} \bmod p$. Note that the exponent ω^t, for large t will make calculating $x_0^{\omega^t} \bmod p$ computationally infeasible. Therefore, the Fermat-Euler Theorem is used so the exponent ω^t can be reduced $\bmod (p-1)$.

Next, α_t is calculated, where α_t is the t^{th} square root of $x_0 \bmod p$. Next, β_t is calculated, where β_t is the t^{th} square root of $x_0 \bmod q$. To complete the calculation of the term x_{-t}, the CRT is used on line 10, where the terms $(q^{-1} \bmod p)$ and $(p^{-1} \bmod q)$ are calculated using Euclid's Extended Algorithm.

The challenge C is set to the tuple (x, x_0, x_{-t}) and then $\mathcal{V}$ passes (pp, C, t) to $\mathcal{P}$ who must solve:

$$\text{Given } (\mathsf{pp}, C, t), \text{ find the factors of } \mathsf{pp} := N.$$

2) $\mathcal{P}$ runs $y \leftarrow \mathsf{Eval}(\mathsf{pp}, C, t)$ to evaluate the challenge, as seen on Algorithm 3.5 Eval. First Eval calculates the term x' in $t-1$ sequential steps by evaluating

$x_{-t}^{2^{t-1}}$ mod N. This is where the sequential calculation takes place using Algorithm 3.1 with inputs $(x_{-t}, 2^{t-1}, N)$. The term x' is guaranteed to be in $\mathcal{QR}_N$ by Definition 2. $\mathcal{P}$ now has $x \in \mathcal{QNR}_N^{-1}$ and $x' \in \mathcal{QR}_N$. Therefore, x must be distinct from x', and we have $x^2 \equiv x'^2 \equiv x_0 \bmod N$. Finally, using the result from Theorem 6, Eval calculates $\mathsf{gcd}(x - x', N)$ to recover one factor p' of N using Euclid's Extended Algorithm. Next, $\frac{N}{\mathsf{gcd}(x-x',N)}$ is calculated to recover the other factor q'. Finally, Eval passes the solution $y := (p', q')$ to $\mathcal{V}$.

Algorithm 3.5: Eval runs on public parameter, challenge, and time parameter pp, C, t to produce solution y.

```
  input : pp, $C, t$
  // pp $= N$,  $C = (x, x_0, x_{-t})$
1 $x' := x_{-t}^{2^{t-1}} \bmod N$
2 $p' := \mathsf{gcd}(x - x', N)$
3 $q' := \frac{N}{p'}$
4 $y := (p', q')$
  output: $y$
```

3) $\mathcal{V}$ runs $\mathsf{V} \leftarrow \mathsf{Verify}(\mathsf{pp}, C, t, y)$ to verify if solution y is sound, as seen on Algorithm 3.6 Verify. Verify checks if $p'q' = N$ and also checks if $x_{-t}^{2^t \bmod (p'-1)(q'-1)}$ mod $N = x_0$. The second check validates that the parameter t was honestly distributed from $\mathcal{V}$ to $\mathcal{P}$, and it also validates that p' and q' are not trivial factors of N i.e., $p' = 1$ and $q' = N$. The second check also uses the Fermat-Euler Theorem in Appendix A.1, Theorem 14 to reduce the exponent 2^t mod $(p'-1)(q'-1)$. If both conditions are true, then Verify outputs accept, else it outputs reject.

Algorithm 3.6: Verify checks the soundness of y against the public parameter, public challenge and time parameter pp, C, t.

```
  input : pp, $C, t, y$
  // pp $= N$,  $C = (x, x_0, x_{-t})$,  $y = (p', q')$
1 if $p'q' = N \wedge x_{-t}^{2^t \bmod (p'-1)(q'-1)} \bmod N = x_0$ then
2   V := accept
3 else
4   V := reject
5 end
  output: V
```

4 Security Analysis

In this section we provide a formal security analysis that our P-VDF is correct, sound, sequential, unique, and efficiently verifiable, as discussed in Section 2.2.

We begin by proving the correctness of our P-VDF. The definition of correctness indicates that every output of Eval must be accepted by Verify with overwhelming probability. In our P-VDF Algorithm 3.6 Verify outputs accept if the solution $y := (p', q')$ are the non-trivial factors of N.

Theorem 7. *The P-VDF is correct.*

The correctness proof must show that the construction of our P-VDF allows an honest prover to factor the public parameter N during the execution of Algorithm 3.5 Eval. The proof of correctness will require a sequence of arguments based on the Theorems outlined in Section 3.1.

First, we must prove that Algorithm 3.4 Gen correctly selects the term x such that $x \in \mathcal{QNR}_N^{-1}$.

Corollary 2. *(Of Theorem 2). The while loop in Algorithm 3.4 Gen efficiently samples $x \in \mathcal{QNR}_N^{-1}$ with overwhelming probability.*

Proof. The while loop in Algorithm 3.4 Gen selects a quadratic non-residue with Jacobi Symbol equal to -1 by running a series of Bernoulli trials with probability $\mathrm{P}\left(x = \mathcal{QNR}_N^{-1}\right) = \frac{1}{2}$. This forms a geometric distribution $G \sim \mathrm{Geo}(\frac{1}{2})$. Therefore, we can expect to find $x \in \mathcal{QNR}_N^{-1}$ in $\mathbb{E}\{G\} = 2$ trials.

Second, we must prove that Algorithm 3.4 Gen correctly calculates the term x_{-t}, which is the t^{th} principal square root of x_0. This proof begins by proving Theorem 3, and subsequently uses the CRT for the final proof.

Proof. (Theorem 3). Let $\alpha = r^{\frac{p+1}{4}} \bmod p$. Then $\alpha^2 \equiv (r^{\frac{p+1}{4}})^2 \equiv r^{\frac{2p+2}{4}} \equiv r^{\frac{p+1}{2}} \bmod p$. Next, let $\frac{p+1}{2} = 1 + \frac{p-1}{2}$. Therefore, by Euler's Criterion (Theorem 1) $\alpha^2 \equiv r^1 r^{\frac{p-1}{2}} \equiv r \bmod p$. We refer to α as the principal square root of $r \bmod p$.

Theorem 8. *The Algorithm 3.4 Gen correctly calculates the t^{th} principal square root x_{-t} of the seed x_0.*

Proof. Let $\omega = \frac{p+1}{4}$. If Algorithm 3.4 Gen provides the seed term $x_0 \in \mathcal{QR}_N$, then, by Theorem 3, the t^{th} principal square root of $x_0 \bmod p$ is $\alpha_t := x_0^{\omega^t} \bmod p$ and the t^{th} principal square root of $x_0 \bmod q$ is $\beta_t := x_0^{\omega^t} \bmod q$. Then, the CRT is used to calculate: $x_{-t} := [\alpha_t q(q^{-1} \bmod p) + \beta_t p(p^{-1} \bmod q)] \bmod N$.

Third we must prove that the Algorithm 3.5 Eval correctly calculates the term $x' \in \mathcal{QR}_N$ using Algorithm 3.1.

Theorem 9. *Algorithm 3.1 Square and Multiply correctly calculates the term x_i, where $x_i \equiv x_0^{2^i} \bmod N$.*

Proof. The input to calculate the term x_i in Algorithm 3.1 Square and Multiply is $(x_0, 2^i, N)$, where $x_0 \in \mathcal{QR}_N$ is the seed term, and $N = pq$, where p and q are distinct odd primes. By Definition 2, selecting $x_0 \in \mathcal{QR}_N$ can be done by uniformly selecting $x \in \mathbb{Z}_N^*$ and setting $x_0 \equiv x^2 \bmod N$. Consider the base case when $i := 1$. The algorithm proceeds as follows: d is set to 1 and the exponent $b := 2^1$ is set to the binary string $B = 10$. Next, the algorithm enters the for loop on the first iteration. On the first iteration j is the first digit of B, which is 1. Next $d := 1$ is squared to output 1. Then the first conditional **if** statement is met as $j = 1$, therefore $d := 1 \cdot x_0 = x_0 \bmod N$, and the first iteration of the loop is done. On the second iteration j is the second digit of B, which is 0. Next, as d was set to x_0 on the first iteration d is now set to $x_0^2 \bmod N$ on the second iteration. The first conditional **if** statement is not met, and the loop terminates as the final digit of B was processed. The algorithm then returns $d := x_1 \equiv x_0^2 \equiv x_0^{2^1} \bmod N$, as required. Therefore, the base case is true.

By the inductive hypothesis we claim that for any $i := k$, the loop invariant of Algorithm 3.1 returns the term $x_0^{2^k} \bmod N$ after k iterations. Therefore after k iterations, where b was set to 2^{k+1}, Algorithm 3.1 will have $d := x_0^{2^k} \bmod N$, and j will be the final digit of $B := 10\dots0$. For any k, the variable B will be a binary string starting with the digit 1 followed by a trail of k digits equal to 0. Therefore, after the first iteration of the for loop all remaining $j \in B$ will be 0. Thus, at the $k+1$ iteration of the for loop d will be set to $x_k^2 \bmod N$, and by definition $x_k^2 \equiv x_{k+1} \equiv x_0^{2^{k+1}} \bmod N$. Finally, Algorithm 3.1 will terminate at the $k+1$ iteration as the final digit of B was processed, and the algorithm will return $d := x_0^{2^{k+1}} \bmod N$.

Finally, Theorem 6 is proven to show that Algorithm 3.5 Eval calculates $\mathsf{gcd}(x' - x, N)$ to recover a non-trivial factor of N [26].

Proof. (Theorem 6.) As x and x' are distinct we have $x^2 \equiv x'^2 \bmod N$. This implies that $pq \mid x^2 - x'^2$. As p and q are both prime this indicates that $p \mid (x - x')(x + x')$ and $q \mid (x - x')(x + x')$. Also, because p is prime it must be the case that $p \mid (x - x')$ or $p \mid (x + x')$. Similarly, it must be the case that $q \mid (x - x')$ or $q \mid (x + x')$. Without loss of generality, assume that $p \mid (x - x')$ is true and that $q \mid (x - x')$ is true. This implies that $pq \mid (x - x')$, which indicates that $x \equiv x' \bmod N$. This is a contradiction because x and x' are distinct. Then it must be the case that $p \mid (x - x')$ and $q \nmid (x - x')$. Therefore, one of the factors of N can be recovered by calculating $p' := \mathsf{gcd}(x - x', N)$ using Euclid's Extended Algorithm, and the other factor of N can be recovered by calculating $q' := \frac{N}{\mathsf{gcd}(x - x', N)} = \frac{N}{p'}$.

We now prove Theorem 7, the correctness of our P-VDF.

Proof. (Theorem 7) Suppose $\mathcal{V}$ honestly generates a random public parameter, challenge, and time parameter $\mathsf{pp} := N = pq, C := (x, x_0, x_{-t}), t$ and presents these to honest $\mathcal{P}$. Next, suppose $\mathcal{P}$ selects the legitimate evaluation algorithm Eval to evaluate pp, C, t. Algorithm 3.5 Eval will calculate the term x' by entering the following parameters $(x_{-t}, 2^{t-1}, N)$ into Algorithm 3.1, which will output $x' := x_{-t}^{2^{t-1}} \bmod N$. The term x' is guaranteed to be correct by Theorem 9 and is guaranteed to be in $\mathcal{QR}_N$ by Definition 2. $\mathcal{P}$ now has $x \in \mathcal{QNR}_N^{-1}$ and $x' \in \mathcal{QR}_N$. This guarantees that x must be distinct from x'. Therefore, by Theorem 6, calculating $p' = \mathsf{gcd}(x - x', N)$ will recover one factor of N using Euclid's Extended Algorithm, and the other factor can be recovered by calculating $q' = \frac{N}{\mathsf{gcd}(x-x',N)}$.

Once N is factored $\mathcal{P}$ will pass the solution $y := (p', q')$ to $\mathcal{V}$ to verify. Verify will then confirm if $p'q' = N$ and if $x_{-t}^{2^t \bmod (p'-1)(q'-1)} \bmod N = x_0$ are both true. If p' and q' are non-trivial factors of N, then the second check will be correct by the Fermat-Euler Theorem 14 and Corollary 3 in Appendix A.1. Therefore, Verify will output accept with overwhelming probability.

Theorem 10. *The P-VDF is unique.*

Proof. The fundamental theorem of arithmetic states that every integer greater than 1 has a unique prime number factorisation. By construction, the public parameter of our P-VDF is $\mathsf{pp} := N = pq = qp$, where p and q are distinct odd primes. That is, the prime factorisation of N is unique up to ordering because multiplication in $\mathbb{Z}$ has the commutative property. Therefore, if the solution $y := (p', q')$ output by Eval is accepted by Verify it must be the case that either:

$$p' = p \wedge q' = q \tag{4}$$

$$p' = q \wedge q' = p \tag{5}$$

Theorem 11. *The P-VDF is sequential.*

Proof. Suppose $\mathcal{V}$ honestly generates a random public parameter pp and presents this to a PPT adversary $\mathcal{A}$. Let $\mathcal{A}$ produce a PPT algorithm $\mathcal{E}_p$ to pre-process the parameter pp and produce output $\mathcal{L} \leftarrow \mathcal{E}_p(\mathsf{pp})$.

Next, suppose $\mathcal{V}$ honestly generates a public challenge $C = (x, x_0, x_{-t})$, using Algorithm 3.4 Gen, where $x \in \mathcal{QNR}_N^{-1}$, $x_0 \equiv x^2 \bmod N$, and x_{-t} is the t^{th} square root of x_0.

To win the sequentiality game, $\mathcal{A}$ must compute a solution $y^* := (p^*, q^*)$, such that $y^* = y \leftarrow \mathsf{Eval}(\mathsf{pp}, C, t)$. Namely, $\mathcal{A}$ must recover the non-trivial factors of N in less than t sequential steps.

We split the proof into two parts: i) when $\mathcal{A}$ attempts to compute an x', where $x' \equiv \sqrt{x_0} \bmod N$ and $x' \in \mathcal{QR}_N$, in less than t sequential steps, and ii) when $\mathcal{A}$ attempts to recover the non-trivial factors of N using a method that does not use x'.

We start by proving part (i): that computing x' in time less than t reduces to the RSW time-lock assumption. First note that pre-processing is carried out before the selection of C. Therefore, the probability that $\mathcal{L}$ can compute x' in less than t steps is negligible. Specifically, if Eval is honestly run, then $x' := x_0^{2^{t-1}} \bmod N$ is calculated using Algorithm 3.1 with the input $(x_{-t}, 2^{t-1}, N)$. By the RSW time-lock assumption calculating x' using Algorithm 3.1 requires $t - 1$

sequential steps. Once x' is calculated, Algorithm 3.5 Eval recovers the factors of N by calculating $p' := \mathsf{gcd}(x - x', N)$ and $q' = \frac{N}{p'}$ then sets $y := (p', q')$.

Next, suppose $\mathcal{A}$ selects a PPT algorithm $\mathcal{E}_{<t}$ to evaluate x' in less than $t - 1$ sequential steps. Finding such an x' using $\mathcal{E}_{<t}$ reduces to the RSW time-lock assumption and we obtain a contradiction. Therefore, $\mathcal{A}$ will not be able to recover $p^* := \mathsf{gcd}(x - x', N)$ without sequentially evaluating x'.

Next, we prove part (ii): that factoring N faster than sequential squaring reduces to an open problem. First note that N is a Blum integer, which is an RSA modulus that is the product of Gaussian primes. Therefore, we assume N cannot be factored by any PPT algorithm with more than negligible probability.

Next, giving $\mathcal{A}$ either (N, x, x_0, t) or (N, x_{-t}, t) also reduces to a standard factoring assumption, as seen in Section 4 of Rabin [26]. What remains is to show that giving an adversary all of the challenge C does not allow them to factorise N. To see this, note that x_0 can be trivially obtained from x, and that by construction x_{-t} and x_0 are terms in a BBS_CSPRNG sequence [4].

Knowledge of these terms does not allow factorisation of N faster than sequential squaring unless $x_{-t}^{2^{\lambda(\lambda(N))}}$ mod N is calculated efficiently. This is an open problem given by Theorem 9 of Blum et al. [4, 14, 18].

Therefore, the only way a PPT algorithm could produce a solution $y^* = y \leftarrow \mathsf{Eval}$ given $(\mathcal{L}, \mathsf{pp}, C, t)$ with non-negligible probability is to sequentially evaluate x' and subsequently recover the factors by calculating $p' := \mathsf{gcd}(x - x', N)$ and $q' = \frac{N}{p'}$.

Theorem 12. *The P-VDF is sound.*

Proof. Suppose $\mathcal{V}$ honestly generates a random public parameter, challenge and time parameter (pp, C, t) and presents these to an adversary $\mathcal{A}$. The public parameter $N = pq$ is a Blum integer. Next, suppose $\mathcal{A}$ selects an algorithm $\mathcal{E}$ to evaluate $y^* \leftarrow \mathcal{E}(\mathsf{pp}, C, t)$, where $y^* := (p^*, q^*)$, $y^* \neq y \leftarrow \mathsf{Eval}(\mathsf{pp}, C, t)$, and $y := (p', q')$. Further, suppose $\mathcal{A}$ wins the soundness game, i.e., Verify accepts the solution y^* with non-negligible probability. Algorithm 3.6 Verify will output accept if conditions i) $p^*q^* = N$ and if condition ii) $x_{-t}^{2^t \bmod (p^*-1)(q^*-1)} \bmod N = x_0$ are both true.

First consider condition i) $p^*q^* = N$. By the fundamental theorem of arithmetic, the factors of N are the set $\{1, p, q, N\}$. Also, by Theorem 10 the solution $y := (p', q')$ is unique up to ordering. Therefore, for the condition i) to evaluate as true, it must be the case that either case (1) or case (2) is true:

1. $(p^* = p \wedge q^* = q)$
2. $(p^* = 1 \wedge q^* = N)$

If (1) is true, then it must be the case that $y^* := (p^*, q^*) = y := (p', q')$. Therefore case (1) can be excluded as it implies that $y^* = y \leftarrow \mathsf{Eval}(\mathsf{pp}, C, t)$.

Finally, assume case (2) is true and Verify accepts the solution y^* with overwhelming probability. If case (2) is true, then condition i) $p^*q^* = N$ will be true. However, if $p^* = 1$, then $(p^* - 1)(q^* - 1) = 0$ and reducing $2^t \bmod 0$ is equivalent to dividing by 0. This is not defined and implies that condition ii) will not evaluate to true. Therefore, we have a contradiction, as Algorithm 3.6 Verify will output reject with overwhelming probability.

Theorem 13. *The P-VDF is efficiently verifiable.*

Proof. The Algorithm 3.6 Verify must confirm if both statements in Equation 6 are true.

$$p'q' = N \wedge x_{-t}^{2^t \bmod (p'-1)(q'-1) \bmod N} = x_0 \tag{6}$$

Firstly, the calculation of $p'q'$ is a multiplication operation which takes $O(\kappa^{\log_2 3}) = \mathsf{poly}(\kappa)$ time [19]. Secondly, the calculation of $x_{-t}^{2^t \bmod (p'-1)(q'-1)} \bmod N$ requires a single modular exponentiation operation. Therefore, a constant number of modular exponentiations is needed, i.e., $O(1) < \mathsf{polylog}(t)$. Furthermore, using Algorithm 3.1, calculating $d \equiv a^b \bmod N$ takes $O((\mathsf{log}(N))^2 \mathsf{log}(b))$ time [28]. Observe that the exponent 2^t is reduced $\bmod (p'-1)(q'-1)$, where $(p'-1)(q'-1) < N$ Therefore, the number the modular exponentiation required will take $O((\mathsf{log}(N))^2 \mathsf{log}(N)) = O(\kappa^3) = \mathsf{poly}(\kappa)$ time.

5 A Comparison of VDF Candidates

In this section we compare the current VDF candidates. We assess the efficiency of the Setup, Eval, and Verify algorithms and review the trust and security assumptions of each candidate. To provide a meaningful comparison we normalise the constructions by unifying the Gen algorithm into the discussion of the Setup algorithm. This follows with the constructions of the Wesowlowski and De Feo VDFs [13, 31]. Furthermore, our empirical tests in Section 6 reveal that the run time of Gen is dominated by Setup.

Currently, there are two classes of VDF, the RSW-based and isogeny-based. The RSW-based VDFs consist of schemes defined by Wesolowski [31] and Pietrzak [25]. These VDFs are both based upon exponentiation in a group of unknown order. We shall refer to these VDF candidates and our P-VDF as the RSW-based VDFs. The De Feo et al. [13] VDF is the isogeny-based VDF candidate. This VDF is based on supersingular isogenies and pairings in elliptic curve cryptography.

5.1 Performance

In this section we compare the efficiency of the Setup, Eval and Verify algorithms of our P-VDF against the other RSW-based VDF candidates and the isogeny-based VDF. We also compare the size of the solution and optional proof parameter output by Eval of our P-VDF against the other VDF candidates.

In this section, we will use the parameter λ to refer to the number of bits of security we expect from each protocol, and we follow [13] in assuming t is super-polynomial in λ. This allows for a meaningful comparison between the VDF candidates, where minimising t is the priority.

Setup The first step of Setup in our P-VDF is to generate an RSA modulus N which must be a Blum integer (Section 3.1).

The RSW-based VDFs share a similar setup. They all require the generation of an RSA modulus. In Pietrzak's construction, the primes are required to be safe primes. That is, a prime p such that $(p-1)/2$ is also prime [25]. Therefore, the RSW-based VDFs share the same asymptotic complexity of Setup, $O(\lambda^3)$. This is in contrast to the isogeny-based VDF which has a longer setup of $O(t\lambda^3)$ [13]. That is, a notable gap in efficiency is present between RSW-based VDFs and the isogeny-based VDF.

The main constraint of our P-VDF is that Setup must be run for each new challenge. This is in contrast to the other VDF candidates. However, Figure 1 and Figure 2 in Section 6 both show that the mean and median run time (respectively) of the Setup algorithm on a desktop PC is efficient. The figures show that Setup generates a new 2048 bit modulus and creates a new challenge in 1 to 2 seconds.

Eval The definition of a VDF indicates that the time complexity of computing Eval should be t regardless of the amount of computational power used [5]. However, if a proof is included in Eval the run time is increased.

Our P-VDF and the isogeny-based VDF have empty proofs, and hence require t sequential steps for honest evaluation. Contrastingly, the other RSW-based VDFs require a proof in their construction. It takes $O(\sqrt{t})$ group operations to construct Pietrzak's proof, and $O(t)$ operations to construct Wesolowski's proof [6].

In an implementation study by Attias et al., the time spent generating the proof was similar to the evaluation time for both constructions [1]. Both VDFs address this limitation by indicating that the computation of the proofs can be parallelised to reduce the run time. However, the run time overhead of proof computation in Eval is a constraint when compared to a VDF with an empty proof.

Verify Our P-VDF runs Verify in $O(1) \ll \mathsf{polylog}(t)$, as shown in Theorem 13. This is more efficient than the $O(\lambda^4)$ used by the isogeny VDF and Wesolowski's VDF, and the $\log(t)$ used in Pietrzak's VDF [13]. Additionally, our P-VDF shares the benefit of being non-interactive with the isogeny-based VDF. This will be analysed further in Section 5.2.

Storage requirements The RSW-based VDFs require the storage of an RSA modulus, and the challenge parameters. Whilst our P-VDF has an empty proof, the RSW-based VDFs have an additional proof to store. Wesolowski's proof has the size of one group element, whereas Pietrzak's proof has size $\log_2(t)$ elements [6].

The isogeny-based VDF has an empty proof, but still requires the evaluator to use large amounts of storage. In the case where the evaluator runs the VDF in optimal time, $O(t)$ storage is required. However, this can be improved upon using a time-memory trade off, meaning $O(t/n)$ storage is required instead, at the cost of increasing evaluation time to $O(t \log n)$. The authors used the parameter $n = 1244$ when benchmarking this scheme [13]. In Table 1 we prioritise efficient evaluation over storage, therefore n is set to 1.

VDF	Setup	Sequential Eval	Parallel Eval	Verify	Memory cost	Fiat Shamir
Wesolowski	$O(\lambda^3)$	$O((1 + \frac{2}{log(t)})t)$	$O((1 + \frac{2}{s\ log(t)})t)$	$O(\lambda^4)$	$O(\lambda^3)$	Yes
Pietrzak	$O(\lambda^3)$	$O((1 + \frac{2}{\sqrt{t}})t)$	$O((1 + \frac{2}{s\sqrt{t}})t)$	$O(\log(t))$	$O(\log(t))$	Yes
Isogeny	$O(t \log(\lambda))$	$O(t)$	$O(t)$	$O(\lambda^4)$	$O(t)$	No
P-VDF	$O(\lambda^3)$	$O(t)$	$O(t)$	$O(1)$	$O(\lambda^3)$	No

Table 1: VDF candidates' comparison: We see that our P-VDF has the best asymptotic complexity across every column, in particular the O(1) verification time, and additionally does not require Fiat-Shamir.

Efficiency considerations Table 1 illustrates that the main advantages of our P-VDF is its lack of proof and the efficiency of Verify. The isogeny-based VDF shares the benefit of having an empty proof. However, the isogeny-based VDF has a Setup algorithm which grows linearly with t.

Our P-VDF and the isogeny-based VDF have the most efficient evaluation due to their lack of proofs. However, both of these candidates have less efficient setups when compared to the RSW-based VDFs: the isogeny-based VDF has a setup of order $O(t)$, making it impractical for long-running VDFs. Our P-VDF has a similar length setup as the RSW-based VDFs, but with the limitation that it is single-use (Section 6). The other RSW-based VDFs have the additional strength that challenges with different time parameters can be generated for each setup, which is not possible our P-VDF and the isogeny-based VDF. Therefore, in any setting which requires many challenges with varying values of t to be generated, Wesolowski and Pietrzak's VDFs are the correct candidates.

However, our P-VDF and the isogeny-based VDF excel where fast verification is desired. Both have very efficient verification procedures which are non-interactive. In Attias et al., the proving time of both RSW-based VDFs was similar to the evaluation time, which is undesirable [1]. Given these considerations, we now compare our P-VDF with the isogeny-based VDF.

The key difference is that our P-VDF has a significantly more efficient Setup algorithm. However, the public parameter output by Setup can only be used once, due to N being factored during Eval. Therefore, if many challenges are required using the same value of t, and fast verification is important, the isogeny-based VDF is the most appropriate candidate.

For VDFs where t is large, or where there are few challenges issued relative to the time parameter t, our P-VDF is the most suitable, due to the impractical length of the isogeny setup.

5.2 Trust and Security Assumptions

Trust assumptions Our P-VDF relies upon a trusted setup to generate the Blum integer, N. As verification requires checking the factors of N, it must be instantiated in such a way that the prover never learns the factors p and q.

The three other VDF candidates have similar assumptions. Both RSW-based VDFs require the generation of a group of unknown order, such as an RSA modulus. Using an RSA group requires a trusted setup. If the factorisation of N is known, then the RSW time-lock assumption is bypassed (Corollary 3,Appendix A.1).

To solve this problem, it is proposed in [31] to use class groups of an imaginary quadratic field instead of an RSA modulus. However, this is a non-standard security assumption, lacking the rigorous cryptanalysis of the RSA assumption. Furthermore, the isogeny-based VDF also requires a trusted setup to generate initial parameters. Therefore, whilst all four candidates require a trusted setup in their simplest form, the use of class groups mean that the RSW-based VDFs have a promising method to avoid this.

Security assumptions Both of the other RSW-based VDFs are interactive by nature, meaning that the prover and verifier must complete a protocol to show that the calculation was correct. This is undesirable, as it requires to prover to be available whenever a party wishes to verify the computation is correct. In both cases this can be made non-interactive using the Fiat-Shamir heuristic, which involves modelling a hash function as a random oracle. Whilst this is a standard cryptographic procedure, it weakens the security model by relying on an additional assumption [3, 23]. By contrast, our P-VDF and the isogeny-based VDF do not use this heuristic.

An additional requirement of Wesolowski's VDF is a hash function H_{prime}, which maps an input to the set containing the first $2^{2\lambda}$ primes, and it is assumed that this hash function is a uniformly distributed random oracle. Such a hash function does not exist in public literature, and it is unclear how to efficiently achieve a function which selects primes in a cryptographically secure pseudo-random manner[3]. Reliance on the H_{prime} hash function increases the assumptions for the security of this construction. This means that any implementation at present would rely on a heuristic not subjected to full cryptanalysis, testing, and approval.

In summary, our P-VDF is built upon well studied mathematical problems and standard assumptions. From a practical perspective, this is beneficial as it does not rely on strong assumptions. Furthermore, our P-VDF being RSW-based is easier to understand and cryptanalyse than the isogeny-based VDF.

[3] NIST standards FIPS 180-4 [29] and FIPS 202 [30] specify the various hash functions given approval by NIST; neither of which includes H_{prime}.

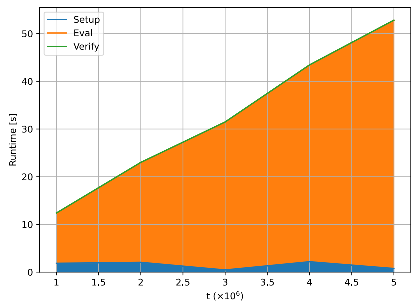

Fig. 1: The impact of adjusting parameter t on the run time of Setup, Eval, and Verify algorithms when run on a desktop PC with a 2048 bit modulus. The primary effect is on Eval, which displays a linear increase.

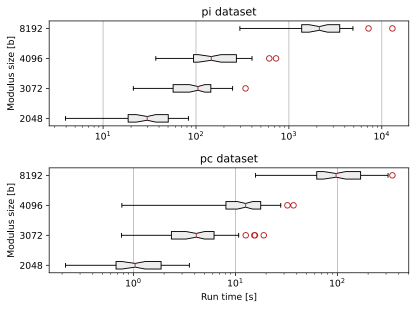

Fig. 2: The spread of setup time across modulus sizes and machines. Setup time increases in response to an increase in modulus size. The dispersion of run times is similar across different devices.

6 Implementation Analysis

In this section we describe the implementation and performance analysis of our P-VDF. Recall from Section 5 that we unify the algorithms Gen and Setup, so that when we write Setup, we are referring to Setup followed by Gen. The software implementation is written in Python 3 and the code is publicly available at https://github.com/wsAJMYbR/pvdf. Furthermore, we have ensured that the authors of this code are anonymized.

Our testing platform consisted of two different hardware environments: a Raspberry Pi cluster, and a desktop PC. The Pi cluster consisted of four Raspberry Pi 3 Model B computers networked together. Each Pi node utilises a quad-core 1.2 GHz CPU, with 1 GB of available memory. This enabled us to run experiments on four different modulus sizes in parallel. The use of Raspberry Pi devices provides an affordable and ubiquitously available device with a consistent configuration. This facilitates the replication of our experiments and comparison with other VDFs. Furthermore, as Raspberry Pi devices are lower power, they represent a lower bound for hardware that may reasonably be expected to be used in practice outside of embedded applications.

We also executed performance tests on a consumer grade desktop PC. The machine used a quad-core 3.2 GHz Intel i5 processor, with 16 GB of available memory. We wished to confirm that the statistical properties remained constant over different hardware types. Additionally, this dataset provides a more pragmatic view of performance on commercial hardware.

Figure 1 demonstrates our first experiment. This experiment shows how the run time of Setup, Eval, and Verify is impacted by the time parameter t for a 2048 bit modulus when run on a desktop PC. The figure shows that as t increases the run time for Eval also increases in a predictable linear manner. The linear variance in run time of Algorithm 3.5 Eval as t varies supports the RSW time-lock assumption in Definition 1. Furthermore, we see that Algorithm 3.3 Setup and Algorithm 3.6 Verify remain consistently low, regardless of the size of t. This is expected as both algorithms reduce the parameter t by the group order using the Fermat-Euler Theorem 14. We also observe that Setup has minor variations in the run time when compared to Eval and Verify. This is a result of the randomised nature of Setup in comparison to the deterministic behaviour of Eval and Verify. Finally, the figure also illustrates the efficiency of Verify, as outlined in Theorem 13.

For our next experiments we select $t = 5 \times 10^6$ to provide a total run time appropriate for repeat testing. We performed experimentation over four modulus sizes $m \in \{2048, 3072, 4096, 8192\}$ bit, selected to cover common modulus sizes in use. For each modulus size, we run 70 experiments, which allows us to estimate values with a 90% confidence interval with a 10% margin for error. The 8192 bit modulus is included as an edge case to demonstrate performance at the upper bound present in real world applications.

In Figure 2, we plot the spread of run times for the Setup algorithm. The primary metric of interest is the spread of the stochastic algorithm. For both the Pi and PC datasets, an increase in the modulus size increases the median run time. However, there is some overlap between modulus sizes, particularly between $m = 3072$ bit and $m = 4096$ bit. This discrepancy can be attributed to more efficient computation afforded when $\log_2 m \in \mathbb{N}$. In particular, the Miller-Rabin primality implementation can use a fast Fourier transform which is most efficient when dealing with powers of two [24]. The dispersion of the data points follows a similar pattern across both data sets, with an offset in median speed afforded by the relative difference in processor speed.

In Figure 3 we plot the run times of the Eval and Verify algorithms for both datasets against the modulus size. We use the run time means as a metric to eliminate variations caused by other processes on the machine, which we assume to be Gaussian. This leaves us with a more accurate indication of the run time of the deterministic algorithms. As with the Setup algorithm, we see similar increases in run time as a function of modulus size for both Eval and Verify. However, we note the large difference between the run times of Verify and Eval. Above each bar we plot the ratio of the run time of Verify to the run time of Eval. We see that, while there is a small increase in the ratio in relation to the modulus size, the difference between the two remains marked. Even at the edge case, when Eval runs in excess of four hours on the Pi when $m = 8192$ bit, Verify doesn't exceed 30 s. For most practical cases, Verify is often results in sub-second evaluations, demonstrating practicality even in constrained environments. As Eval factors N, our P-VDF is single-use. However, as we have seen in our experiments, this property is not an obstacle for practical use. Even in more computationally constrained environments such as the Pi, the Setup and Verify algorithms do not require an impractical time cost. Although we would recommend for a standard use case to use $m \leq 4096$ to keep the setup run time within an appropriate bound. This leaves the value for t as the primary parameter dictating the length of the delay. As we saw in Figure 1, the value for

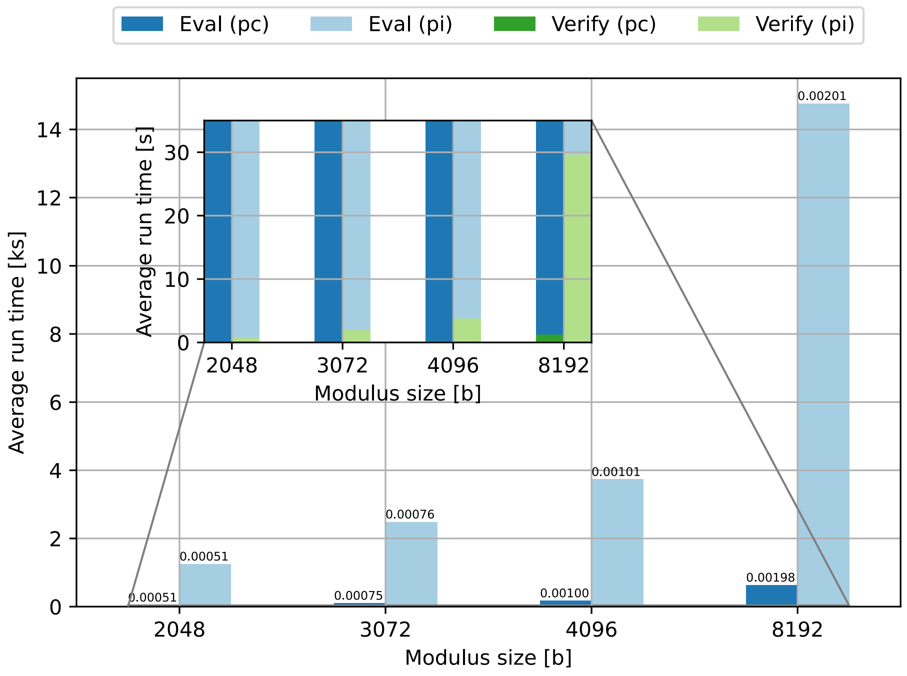

Fig. 3: Verify takes significantly less time to run than Eval across modulus sizes. The inset shows a zoomed in view of the bar chart which is necessary for the effect of Verify to be observed. Above each bar is the ratio Eval to Verify run time.

t can be set with reasonable accuracy to introduce a desired delay for the target hardware.

7 A Practical Delay Encryption Scheme

Burdges and De Feo introduce the primitive of Delay Encryption (DE) [7], where parties are allowed to encrypt messages in such a way that they can only be decrypted once a certain amount of time has passed. A DE scheme provides a solution to electronic voting and auctions, which circumvents the need for a commitment scheme[4]. The authors provide an instantiation based upon the isogeny-based VDF. However, they highlight that this construction comes with implementation issues, such as large storage requirements. For example, they indicate that an evaluator requires approximately 12 TiB for a one-hour delay.

In this section we recall the formal definitions of a DE scheme from [5] and present an instantiation derived from our P-VDF. Our resulting scheme is secure and inherits the efficiency benefits described in Section 5. Specifically, this

[4] We refer the reader to [7] for a detailed discussion of DE and related primitives such as TRE and time lock puzzles.

means that the Practical Delay Encryption (P-DE) scheme has a more efficient setup and requires significantly less storage that the isogeny-based instantiation. However, this comes at the cost of having a single-use setup.

7.1 DE Definitions

We follow the definitions given in [7], altered to fit the public key encryption paradigm, rather than the KEM-DEM paradigm [20]. A Delay Encryption scheme consists of four algorithms: $\mathsf{DE.Setup}$, $\mathsf{DE.Extract}$, $\mathsf{DE.Encrypt}$, and $\mathsf{DE.Decrypt}$. To be consistent with the parties in our P-VDF scheme we note that $\mathcal{V}$ is the encrypting party (verifier), $\mathcal{P}$ is the decrypting party (prover), $\mathcal{A}$ is the adversary, and $\mathcal{C}$ is the challenger.

- $(\mathsf{ek},\mathsf{pk}) \leftarrow_{\text{R}} \mathsf{DE.Setup}(1^\kappa, t)$. $\mathsf{DE.Setup}$ is an algorithm run by $\mathcal{V}$ that takes a security parameter κ and a time delay t, and produces extraction key ek, and encryption key pk. DE.Setup must run in time poly(κ, t).
- $\mathsf{idk} \leftarrow \mathsf{DE.Extract}(\mathsf{ek},\mathsf{id})$. $\mathsf{DE.Extract}$ is an algorithm run by $\mathcal{P}$ that takes the extraction key ek and a session identifier id, and outputs a session key idk.
- $CT \leftarrow_{\text{R}} \mathsf{DE.Encrypt}(\mathsf{pk}, \mathsf{id}, M)$. $\mathsf{DE.Encrypt}$ is an algorithm run by $\mathcal{V}$ that takes the encryption key pk, session id id and a message M, and outputs a ciphertext CT. DE.Encrypt must run in time poly(κ).
- $\{M, \perp\} \leftarrow \mathsf{DE.Decrypt}(\mathsf{pk}, \mathsf{id}, \mathsf{idk}, CT)$. $\mathsf{DE.Decrypt}$ is an algorithm run by $\mathcal{P}$ that takes the encryption key pk, a session identifier id, the session key idk and a ciphertext CT, and outputs either a message M, or a failure symbol $\perp$. DE.Decrypt must run in time poly(λ).

A delay encryption scheme must be correct and satisfy Δ-Delay indistinguishably under chosen plaintext attacks.

```
1 $\mathcal{V}$ generates encryption and extraction keys: $(\mathsf{ek},\mathsf{pk}) \leftarrow_{\text{R}} \mathsf{DE.Setup}(1^\kappa, t)$.;
2 $\mathcal{V}$ delay encrypts a message: $CT \leftarrow \mathsf{DE.Encrypt}(\mathsf{pk},\mathsf{id},\mathsf{M})$.;
3 $\mathcal{P}$ computes the session key: $\mathsf{idk} \leftarrow \mathsf{DE.Extract}(\mathsf{ek},\mathsf{id})$.;
4 $\mathcal{P}$ decrypts the ciphertext: $\{M, \perp\} \leftarrow \mathsf{DE.Decrypt}(\mathsf{pk}, \mathsf{id}, \mathsf{idk}, CT)$.;
  A Delay Encryption scheme is correct if $\mathcal{P}$ outputs $M$ with probability
    $1 - \mathsf{negl}(\kappa)$.
```

Game 7.1: DE Correctness Game

- A delay encryption scheme is *correct* if any encrypted message can be decrypted using the session key with overwhelming probability. This is made precise in the DE correctness game, Game 7.1 [7].
- Suppose an adversary $\mathcal{A}$ chooses a message M_0, and sends it to the challenger $\mathcal{C}$. $\mathcal{C}$ encrypts this message, and also chooses a random string of the same length, before sending one of these at random to the adversary. A DE scheme is *secure* if no efficient adversary can distinguish between these possibilities. This is made precise in the Δ-Delay IND-CPA game, Game 7.2 [7].

```
1 $\mathcal{A}$ receives the encryption and extraction keys $\mathsf{ek}$,$\mathsf{pk}$ as input, and outputs an
    algorithm $\mathcal{D} \leftarrow \mathcal{A}(\mathsf{ek},\mathsf{pk})$.;
2 $\mathcal{A}$ selects a message $M_0$, and sends it to $\mathcal{C}$.;
3 $\mathcal{C}$ computes $CT_0 \leftarrow \mathsf{DE.Encrypt}(\mathsf{pk},\mathsf{id},M_0)$, and chooses $CT_1 \in \{0,1\}^\kappa$
    uniformly at random.;
4 $\mathcal{C}$ then picks a random bit $b \in \{0,1\}$ and then outputs $(\mathsf{id}, CT_b)$.;
5 Guess. The algorithm $\mathcal{D}$ is run on session identifier $\mathsf{id}$, and a ciphertext $CT_b$.;
  $\mathcal{A}$ wins if $\mathcal{D}$ terminates in time less than $\Delta$, and the output is such that
    $b \leftarrow \mathcal{D}(\mathsf{id}, CT_b)$.;
  A Delay Encryption scheme is $\Delta$-Delay indistinguishable under chosen
    plaintext attacks if no efficient adversary $\mathcal{A}$ can win the game with a
    non-negligible advantage.
```

Game 7.2: Δ-Delay IND-CPA Game

7.2 Instantiation using P-VDF

We now show how to instantiate a correct and secure DE scheme using our P-VDF and RSA-OAEP [20]. The overview of our P-DE scheme is as follows:

A summary of the inputs and outputs for the DE.Setup, DE.Extract, DE.Encrypt, and DE.Decrypt algorithms of our P-VDF construction are as follows:

$$\begin{aligned}
\mathsf{pk} := (e, k_0, k_1, G, H),&\\
\mathsf{ek} := (C, t), \mathsf{id} := N &\xleftarrow{R} \mathsf{DE.Setup}(1^\kappa, t)\\
\mathsf{idk} := (p', q') &\leftarrow \mathsf{DE.Extract}(\mathsf{ek},\mathsf{id})\\
CT &\xleftarrow{R} \mathsf{DE.Encrypt}(\mathsf{pk},\mathsf{id}, M)\\
M &\leftarrow \mathsf{DE.Decrypt}(\mathsf{pk},\mathsf{id}, \mathsf{idk}, CT)
\end{aligned}$$

Algorithm 7.3: DE.Setup generates the extraction key ek, the encryption key pk, and the session ID id for our P-DE.

```
  input  : $1^\kappa, t$
1 $\mathsf{id} := N \xleftarrow{R} \mathsf{Setup}(1^\kappa), \mathsf{ek} := (C := (x, x_0, x_{-t}), t) \xleftarrow{R} \mathsf{Gen}(\mathsf{pp}, t)$
2 $e := 65537$
3 $k_0, k_1, G, H \leftarrow \mathtt{params}(1^\kappa)$
4 $\mathsf{pk} := (e, k_0, k_1, G, H)$
  output: ek,pk,id
```

In our P-DE we generate the extraction key ek, the encryption key pk, and the session identifier `id`, with the algorithm DE.Setup where N is a Blum integer, e is an RSA exponent such that $\mathtt{gcd}(e, \phi(N)) = 1$, C, t are the challenge and time parameter output from Algorithms 3.3 Setup and 3.4 Gen in our P-VDF, and k_0, k_1, G, H are RSA-OAEP parameters. We note that k_0, k_1 are integers fixed by RSA-OEAP, and G, H are random oracles - which are cryptographically secure hashing functions. The Algorithm DE.Setup can be seen in Algorithm 7.3.

Algorithm 7.4: DE.Extract takes the extraction key ek and the session identifier id and outputs the session key idk with Δ-Delay.

```
  input  : ek,id
   // ek := $(C, t)$, id := $N$
1 $y := (p', q') \leftarrow \mathsf{Eval}(N, C, t)$
2 idk := $y$
  output: idk
```

We see on line 1 that our P-VDF Algorithm 3.3 Setup is called to output the Blum integer and Algorithm 3.4 Gen is called to output the challenge, and time parameter. Next, we see the public RSA exponent e defined. Typical RSA implementations set $e = 2^{16} + 1 = 65537$, as 65537 is a prime number with a low Hamming weight [20]. Finally, the function `params` outputs the public RSA-OEAP parameters, which we set as the encryption key pk. DE.Setup outputs ek,pk,id.

Next, the DE.Extract algorithm calls Algorithm 3.5 Eval from our P-VDF to evaluate N, C, t to output idk $:= (p', q')$. The Algorithm DE.Extract can be seen in Algorithm 7.4.

Algorithm 7.5: DE.Encrypt takes the encryption key pk, and the session identifier id, and encrypts message M using the RSA-OEAP encryption scheme.

```
  input  : pk,id,$M$
   // pk := $(e, k_0, k_1, G, H)$, id = $N$
1 $M' := M \mid\mid 0^{k_1}$                         // Zero pad to $n - k_0$ bits
2 $r := \mathrm{rand}(k_0)$              // Generate a random $k_0$ bit number
3 $X := M' \oplus G_{n-k_0}(r)$                   // Hash $r$ to length $n - k_0$
4 $Y := r \oplus H_{k_0}(X)$                         // Hash $X$ to length $k_0$
5 $M'' := X \mid\mid Y$                            // Create message object
6 $CT := M''^e \bmod N$                                  // RSA encrypt
  output: $CT$
```

Next the DE.Encrypt algorithm uses the encryption key and session identifier and encrypts a message M using RSA-OEAP encryption and outputs the ciphertext CT. Using RSA-OAEP, parties can encrypt messages to this modulus, so that such messages can only be decrypted after algorithm DE.Extract has terminated, implying Δ time has passed. The Algorithm DE.Encrypt can be seen in Algorithm 7.5. Note that these algorithms are not sequential: The DE.Encrypt algorithm can be run by any party using the encryption key and the session identifier, whilst Extract is running.

Finally, the DE.Decrypt algorithm takes as input the encryption key pk, the session identifier id and the ciphertext and uses the session key idk $:= (p', q')$

to calculate the Euler phi function $\phi(N) = (p' - 1)(q' - 1)$ of N. By having $\phi(N)$, the DE.Decrypt algorithm is able to recover the private RSA key d, where $d := e^{-1} \equiv 1 \bmod \phi(N)$, to subsequently recover the message M. The Algorithm DE.Decrypt can be seen in Algorithm 7.6.

Algorithm 7.6: DE.Decrypt.

```
input  : pk, id, idk, $CT$
// $\mathsf{pk} := (e, k_0, k_1, G, H), \mathsf{id} = N, \mathsf{idk} := (p', q')$
1 $\phi(N) := (p' - 1)(q' - 1)$
2 $d := e^{-1} \bmod \phi(N)$
3 $M'' := CT^d \bmod N$
4 $X := \lfloor M'' \cdot 2^{-k_0} \rfloor$                    // Extract $X$
5 $Y := M'' \bmod 2^{k_0}$                                     // Extract $Y$
6 $r := Y \oplus H_{k_0}(X)$                                   // Recover $r$
7 $M' := X \oplus G_{n-k_0}(r)$                  // Recover padded message
8 $M := M' \cdot 2^{-k_1}$                               // Remove padding
output: $M$
```

7.3 Security

In this section we provide security proofs. Correctness of Our DE scheme relies on the correctness of RSA, which is proved to hold in [21]. RSA encryption with OAEP padding is known to be IND-CCA secure, as proved in [15], and we rely on this to prove that the P-DE is Δ-IND-CPA secure.

Lemma 1. *The P-VDF delay encryption scheme is correct.*

Proof. As shown in Theorem 7, the P-VDF is correct, meaning that the VDF always outputs the factors p and q of N. This allows for the computation of d such that $d \cdot e \equiv 1 \mod \phi(N)$. This ensures that RSA encryption and decryption can be run correctly, and so the result follows from the correctness of RSA.

Lemma 2. *The P-VDF based Delay Encryption scheme (P-DE) is Δ-Delay Indistinguishable Under Chosen Plaintext Attacks*

Proof. Let Δ-IND-CPA^{DE} be the Δ-IND-CPA game for the P-DE, and let IND-CPA^{RSA} be the standard IND-CPA game for RSA.

1) Δ-IND-CPA^{DE} is strictly weaker than the IND-CPA^{RSA}, under the RSW-time lock assumption. To see this, note that the only differences between the two games is that i) an adversary gets no time for meaningful precomputation, as the public parameter and challenge are published at the same time, and ii) the algorithm $\mathcal{D}$ must terminate in time less than Δ.

2) Recall from that Theorem 11 that the P-VDF is sequential, meaning that any algorithm that terminates in time less than Δ has negligible chance of solving

the underlying P-VDF. From (1) and (2), we have that the Δ-IND-CPA^{DE} game is an instance of the IND-CPA^{RSA} game, with the additional constraint that all computation and encryptions performed by an adversary with knowledge of N must take place in time at most Δ.

Recall that RSA with OAEP is IND-CCA secure under the RSA assumption, which implies it is also IND-CPA secure. This implies that if an adversary can break the Δ-IND-CPA^{DE} game, then they can also break the IND-CPA^{RSA} game, therefore we have Δ-IND-CPA security.

7.4 Efficiency

The efficiency of Setup and DE.Extract of this scheme follow directly from the underlying P-VDF. To see this, note that DE.Setup implements Setup and Gen, and additionally specifies the OAEP parameters, and DE.Extract implements Eval. As such, we refer the reader to Section 5 for a discussion of the efficiency of the underlying algorithms.

DE.Encrypt and DE.Decrypt are a textbook case of RSA with OAEP padding, and so we shall not discuss this here.

We conclude this section by noting that we have written code to implement this DE scheme, which can be found at https://github.com/wsAJMYbR/pvdf.

8 Conclusion

In this work we constructed a practical verifiable delay function, which challenges the prover to factor a special class of RSA modulus, known as a Blum integer. We proved security of our construction and compared it to the current published VDF candidates, showing that it has various advantages over the alternatives. Further, we implemented our P-VDF on both a Raspberry Pi and on a desktop PC, showing that it is indeed a practical construction. We then used our scheme to construct the first practical delay encryption scheme, a tool similar to time-lock puzzles, with applications such as e-voting. This is achieved by allowing parties to encrypt messages to the RSA modulus using OAEP, so that they can only be decrypted once the underlying P-VDF has been solved.

References

1. Attias, V., Vigneri, L., Dimitrov, V.: Implementation Study of Two Verifiable Delay Functions. In: Tokenomics (2020)
2. Bach, E., Shallit, J.: Algorithmic Number Theory, Vol. 1: Efficient Algorithms. MIT Press, Cambridge, MA, USA (1996)
3. Bernhard, D., Pereira, O., Warinschi, B.: How not to prove yourself: Pitfalls of the fiat-shamir heuristic and applications to helios. In: Wang, X., Sako, K. (eds.) Advances in Cryptology – ASIACRYPT 2012. pp. 626–643. Springer Berlin Heidelberg, Berlin, Heidelberg (2012)
4. Blum, L., Blum, M., Shub, M.: A Simple Unpredictable Pseudo-Random Number Generator. In: Journal on Computing. vol. 15, p. 364–383 (1986)

5. Boneh, D., Bonneau, J., Bünz, B., Fisch, B.: Verifiable Delay Functions. In: Advances in Cryptology - CRYPTO 2018 - 38th Annual International Cryptology Conference. pp. 757–788. Santa Barbara, CA, USA (2018)
6. Boneh, D., Bünz, B., Fisch, B.: A survey of two verifiable delay functions. Cryptology ePrint Archive, Report 2018/712 (2018)
7. Burdges, J., Feo, L.D.: Delay encryption. In: Advances in Cryptology - EUROCRYPT 2021 - 40th Annual International Conference on the Theory and Applications of Cryptographic Techniques. Zagreb, Croatia (2021), to appear in
8. Cohen, B., Pietrzak, K.: The chia network blockchain (2019)
9. Cook, J.: Computing Legendre and Jacobi symbols, Algorithm for computing Jacobi symbols. https://www.johndcook.com/blog/2019/02/12/computing-jacobi-symbols (2019)
10. Cormen, T., Leiserson, C., Rivest, R., Stein, C.: Introduction to Algorithms. MIT Press, Cambridge, MA, USA, 3 edn. (2009)
11. Crandall, R., Pomerance, C.: Prime Numbers: A Computational Perspective. Springer-Verlag, New York, NY, USA, 2 edn. (2005)
12. Ephraim, N., Freitag, C., Komardogski, I., Pass, R.: Continuous Verifiable Delay Functions. In: Advances in Cryptology - EUROCRYPT 2020 - 39th Annual International Conference on the Theory and Applications of Cryptographic Techniques. p. 125–154. Zagreb, Croatia (2020)
13. Feo, L.D., Masson, S., Petit, C., Sanso, A.: Verifiable Delay Functions from Supersingular Isogenies and Pairings. In: Advances in Cryptology – ASIACRYPT 2019 – 25th Annual Conference. pp. 248–277. Kobe, Japan (2019)
14. Friedlander, J., Pomerance, C., Shparlinski, I.: Period of the power generator and small values of carmichael's function. In: American Mathematical Society. pp. 1591–1605. Mathematics of Computation 70 (2001) (2000)
15. Fujisaki, E., Okamoto, T., Pointcheval, D., Stern, J.: Rsa-oaep is secure under the rsa assumption. In: Proceedings of the 21st Annual International Cryptology Conference on Advances in Cryptology. p. 260–274. Springer-Verlag, Berlin, Heidelberg (2001)
16. Gauss, C.: Disquisitiones Arithmeticae. Yale University Press, New Haven, CT, USA, 1 edn. (2009)
17. Goodstein, R.: Boolean Algebra. Dover Publications, Mineola, NY, USA, 1 edn. (2007)
18. Griffin, F., Shparlinski, I.: On the linear complexity profile of the power generator. In: IEEE Transactions on Information Theory (Volume: 46, Issue: 6, Sep 2000). pp. 2159 – 2162 (2000)
19. Karatsuba, A., Ofman, Y.: Multiplication of Many-Digital Numbers by Automatic Computers. In: Proceedings of the USSR Academy of Sciences. pp. 293–294 (1962)
20. Katz, J., Lindell, Y.: Introduction to Modern Cryptography, Second Edition. CRC Press, Boca Raton, FL, USA, 2 edn. (2014)
21. Lindenberg, C., Wirt, K., Buchmann, J.: Formal proof for the correctness of rsa-pss. IACR Cryptol. ePrint Arch. p. 11 (2006)
22. Miller, G.: Riemann's Hypothesis and Tests for Primality. In: Journal of Computer and System Sciences, 13(3). pp. 300–317 (1976)
23. Mittelbach, A., Fischlin, M.: The Theory of Hash Functions and Random Oracles. Springer, Cham, 1 edn. (2021)
24. Narayanan, S.: Improving the speed and accuracy of the miller-rabin primality test (2014)
25. Pietrzak, K.: Simple verifiable delay functions. In: 10th Innovations in Theoretical Computer Science Conference, ITCS 201. pp. 601–615. San Diego, California (2019)

26. Rabin, M.: Digitalized signatures and public-key functions as intractable as factorization. In: MIT/LCS/TR-212, MIT Laboratory for Computer Science (1979)
27. Rivest, R., Shamir, A., Wagner, D.: Time-lock puzzles and timed-release crypto. In: MIT/LCS/TR-684, MIT Laboratory for Computer Science (1996)
28. Rosen, K.: Discrete Mathematics and Its Applications. McGraw-Hill Education, New York, NY, USA, 8 edn. (2018)
29. of Standards, N.I., Technology: FIPS PUB 180-4, Secure Hash Standard (2015)
30. of Standards, N.I., Technology: FIPS PUB 202, SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions (2015)
31. Wesolowski, B.: Efficient Verifiable Delay Functions. In: Advances in Cryptology – EUROCRYPT 2019. p. 379–407. Darmstadt, Germany (2019)

A Appendix

A.1 Number Theory

First, we present the Fermat-Euler Theorem.

Theorem 14. *Fermat-Euler Theorem. Let N be an odd prime, or let $N = pq$, where p and q are distinct odd primes. If $\mathsf{gcd}(a, N) = 1$, then $a^{\phi(N)} \equiv 1 \bmod N$, where $\phi(N) = N - 1$ if N is prime or $\phi(N) = (p-1)(q-1)$ if $N = pq$.*

Proof. Proof can be found in [16].

Corollary 3. *Let $x_0 \in \mathbb{Z}_N^*$. If the group order $\phi(N)$ is known, then calculating x_t such that $x_t \equiv x_0^{2^t} \bmod N$ can be done in $\log_2 N$ binary operations.*

Proof. Let $x_t \equiv x_0^{2^t} \bmod N$. If the exponent 2^t is reduced mod $\phi(N)$ we have $2^t = \alpha\phi(N) + \beta$, where β is the remainder of 2^t after the $\phi(N)$ modular reduction. Then, by Theorem 14 we have $x_t \equiv x_0^{2^t} \equiv x_0^{2^t \bmod \phi(N)} \equiv x_0^{\alpha\phi(N)+\beta} \equiv x_0^{\phi(N)^\alpha} x_0^\beta \equiv 1^\alpha x_0^\beta \equiv x_0^\beta \bmod N$. The number of bits in β is $O(\log N)$, and β is input into line 2 of Algorithm 3.1.

Next, we provide proof of Theorem 5.

Proof. (Theorem 5) If N is a Blum integer, then $N \equiv 1 \bmod 4$. By Theorem 4 every $x_0 \in \mathcal{QR}_N$ has four distinct square roots $\pm x$ and $\pm x'$. As $N \equiv 1 \bmod 4$, by the law of quadratic reciprocity $\mathcal{J}_N(x) = \mathcal{J}_N(-x)$ and $\mathcal{J}_N(x') = \mathcal{J}_N(-x')$. It must be the case that $x^2 \equiv x'^2 \bmod N$, which implies $(x-x')(x+x') \equiv 0 \bmod N$, which implies $(x - x') \mid N$ and $(x + x') \mid N$. That is, without loss of generality $(x - x') = k \cdot p$ and $(x + x') = \ell \cdot q$, where $k, \ell \in \mathbb{N}$. Therefore, $\mathcal{J}_p(x) = \mathcal{J}_p(x')$ and $\mathcal{J}_q(x) = \mathcal{J}_q(-x')$. As $p \equiv 3 \bmod 4$, the law of quadratic reciprocity tells us $\mathcal{J}_p(-1) = -1$, we have $\mathcal{J}_q(x) \cdot \mathcal{J}_p(-1) = \mathcal{J}_q(-x') \cdot \mathcal{J}_p(-1)$. This implies that $\mathcal{J}_N(-x) = \mathcal{J}_N(x')$ or written another way $\mathcal{J}_N(x) \neq \mathcal{J}_N(x')$.

Without loss of generality, eliminate the two roots with $\mathcal{J}_N$ equal to -1, say $\mathcal{J}_N(x) = \mathcal{J}_N(-x) = -1$. This leaves $\mathcal{J}_N(x') = \mathcal{J}_N(-x') = +1$. It is the case that only one of $-x'$ or x' has $\mathcal{J}_p = \mathcal{J}_q = 1$ as $p \equiv 3 \bmod 4$. Therefore, without loss of generality, it is only x' that has the property $\mathcal{J}_N(x') = +1$ and $x' \in \mathcal{QR}_N$ [4].

A.2 Jacobi Symbol Algorithm

Finally, we provide the details of Algorithm A.2 Samplex which can efficiently sample the parameter x in the challenge C without knowledge of the factors p and q. Algorithm A.2 can be used in lieu of the while loop on lines 1–6 in Algorithm 3.4 Gen. The proof of correctness of Algorithm A.2 can be found in Bach et al [2]. Therefore, the proof of Corollary 2 is also relevant in the case of Algorithm A.2. The original Python code for the function Jacobi(x, pp) can be found in [9].

Algorithm A.1: Jacobi is run on random input x and public parameter pp to calculate the Jacobi symbol of x .

```
input  : $x, \mathsf{pp}$
1  $j = 1$
2  while $x \neq 0$ do
3      while $x \bmod 2 = 0$ do
4          $x := \frac{x}{2}$
5          if $pp \bmod 8 = 3 \vee pp \bmod 8 = 5$ then
6              $j := -j$
7          end
8      end
9      if $x \bmod 4 = 3 \wedge pp \bmod 4 = 3$ then
10         $j := -j$
11     end
12     $x,\ \mathsf{pp} := \mathsf{pp},\ x$
13     $x := x \bmod \mathsf{pp}$
14 end
15 if $pp \neq 1$ then
16     $j := 0$
17 end
output: $j$
```

Algorithm A.2: Samplex is run on public parameter pp to efficiently sample parameter x without knowledge of p and q.

```
input  : pp
1  do
2      $x := \mathcal{U}(2, pp)$
3      $j := \mathsf{Jacobi}(x, \mathsf{pp})$
4  while $j \neq -1$
output: $x$
```