A STUDY ON EARLY DECRYPTION MECHANISM AT VERIFIABLE DELAY FUNCTIONS

A Thesis Submitted to
the Graduate School of
İzmir Institute of Technology
in Partial Fulfillment of the Requirements for the Degree of

MASTER OF SCIENCE

in Computer Engineering

by
Oğulcan ÖZDEMİR

February 2022
İZMİR

ABSTRACT

A STUDY ON EARLY DECRYPTION MECHANISM AT VERIFIABLE DELAY FUNCTIONS

In computer science, can we measure the passage of time in accordance with Earth time and use this measurement mechanism as a time lock to decrypt encrypted data? The search for answers to these questions has not yet been given a definite, straightforward answer. Because there is no fixed definition of time in computer science.

Research on the use and measurement of "time-locked cryptography" in computer science is based on the research of Time-Lock Puzzles and Timed-Release Crypto by Rivest et al. In 2017, two studies were published that accelerated development in this area: Simple Verifiable Delay Functions and Efficient Verifiable Delay Functions. In both studies, timing requirements are defined as Verifiable Delay Functions (VDF).

However, current VDF solutions do not have a controlled early decryption feature for time locking mechanism. The contributions we intend to make to the VDF protocol in this study focus on the design, verification, and implementation of a new VDF protocol that both guarantees the time lock mechanism requirements defined by VDF and provides the ability to open the time lock in a controlled manner by authorised individuals before the target time. VDF solution to be developed, unlike similar VDF protocols, should also include the blockchain Ethereum component and work flexibly with any of the defined VDF time lock algorithms, depending on which one is chosen.

ÖZET

DOĞRULANABİLİR GECİKME FONKSİYONLARINDA ERKEN ŞİFRE ÇÖZME MEKANİZMASI ÜZERİNE BİR ÇALIŞMA

Bilgisayar biliminde, zamanın geçişini Dünya saati ile uyumlu bir şekilde ölçebilir ve bu ölçme mekanizmasını şifrelenmiş verilerin çözülmesi için bir zaman kilidi olarak kullanabilir miyiz? Bu sorulara cevap arayışı teknolojinin de hızlı gelişimi nedeniyle, henüz kesin, net bir cevap verilmemiş, araştırmaları sonlandıracak tek bir çözüm üretilememiştir. Çünkü, bilgisayar biliminde zamanın sabit bir tanımı yoktur.

Bilgisayar biliminde "zaman kilitli kriptografi" kullanımı ve ölçümü üzerine yapılan araştırmalar, Rivest ve arkadaşlarının Time-Lock Puzzles and Timed-Release Crypto araştırmasına dayanmaktadır. 2017 yılında bu alanda gelişimi hızlandıran iki araştırma yayınlanmıştır: Simple Verifiable Delay Functions ve Efficient Verifiable Delay Functions. Her iki araştırmada zamanlama gereksinimlerini Doğrulanabilir Gecikme Fonksiyonları (Verifiable Delay Functions-VDF) olarak tanımlanmaktadır.

Ancak mevcut VDF çözümleri, protokol şemalarında zaman kilitleme mekanizmaları için kontrollü bir erken açma işlevine sahip değildir. Oysa, bazı mahkeme kararları veya vasiyetnamelerin şifreli olarak saklanan belgelerin, hedeflenen gizlilik kalkış tarihi gelmeden de açılmasını gerektirebilir ve var olan VDF mekanizmaları bu yeteneğe sahip değillerdir.

Bu çalışmada VDF protokolüne hedeflediğimiz katkılar; VDF tarafından tanımlanan zaman kilidi mekanizması gereksinimlerinin garanti altına alan, ihtiyaç durumunda zaman kilidinin yetkili kişiler tarafından kontrollü olarak hedeflenen zamanından önce açılmasının olası olduğu yeni bir VDF protokolü tasarlanması, bu protokolün doğrulanması, atak analizlerinin yapılması ve uygulanmasına odaklanmaktadır. Kurulacak olan VDF çözümünün ayrıca, benzer VDF protokollerinden farklı olarak Blockchain-Ethereum bileşenini içermesi, ve tanımlı olan tüm VDF zaman kilit algoritmalarının hangisi seçilirse, hepsi ile de çalışabilme esnekliğinde olması hedeflenmektedir.

TABLE OF CONTENTS

LIST OF FIGURES

Figure **Page**

LIST OF TABLES

Table **Page**

CHAPTER 1

INTRODUCTION

How can the passage of time be measured appropriately and used in a time lock mechanism for encrypted data in computers? The concept of time itself is a complex and multi-layered entity. If we want to explain time using traditional physics, we must use the periodic motion of the earth around the sun. If we want to explain it using modern relativity physics, it is a concept that changes depending on the observer's speed and the gravity to which he is subjected. If we look at it from a psychological perspective, it is a subjective concept that changes depending on the current mood of the observer.

As we can deduce, time definitions differ from the perspective of the identifying domains. In computer science, the time has no direct constant definition. It is tied to the measurement assumption of universally recognised devices or systems. For example, we can use an electrical circuit to measure the round-trip time of an electrical pulse or write a program to compute the solution to a nondeterministic polynomial problem and measure time. Alternatively, we can use atomic clocks to observe the reactions of atoms of the element strontium to electromagnetic radiation. Certain frequency changes are accepted as time units, and these measurements are transmitted to the computer as a clock reference. Regardless of which of the defined solutions we choose, it is necessary to rely on a reference system or a third trusted party that is not in the research context of computer science to measure time.

The idea of sending messages to the future using the Time-Lock Puzzles protocol was first put forward by Timothy C. May [64]. The responsibility for the confidentiality of the messages in the protocol and the opening after the specified time limit is given to the "independent trustees" as third party participants. The encrypted messages and keys are fragmented and distributed among the "independent trustees". This is to prevent "independent trustees" from opening messages independently. Messages and keys that are opened are forwarded by "independent trustees" to their recipients.

In the following years, Rivest et al. [1] proposed an RSA-based puzzle scheme in the "Time-lock Puzzles and Timed-release Crypto" research for the operation of Time-Lock Puzzles. The purpose of this scheme is to establish a puzzle (delay) mechanism that cannot be opened before the specified time, even against attackers with high computational power without the need for a third party trusted agent. The scheme provides the sequentiality required for this mechanism with the Successive Squaring algorithm.

The Verifiable Delay Function (VDF) scheme declares a protocol that provides a

publicly verifiable system based on time-lock puzzles. VDF defines concrete boundaries to sequential computation schemes on which both VDF and "Time-Locks Puzzles" are based. So far, VDF implementations are "trusted public randomness" on distributed systems, such as fair contract signings, bids in auctions, and coin tosses.

Existing solutions do not have a controlled opening function for time-lock mechanisms in their protocol schemes. Our contribution focuses on ensuring time-lock and VDF mechanism requirements and designing, verifying, and implementing a new VDF and Time-Lock hybrid protocol so that authorised individuals' controlled opening of the time-lock(delay) function is possible before the target time, if needed.

In other words: We aim to combine the ability to send encrypted messages to the future from the Time-Lock puzzle and the VDF's "publicly verifiable system" and add a mechanism for early decryption. With the early decryption mechanism, we want to ensure that future encrypted messages can be decrypted early under certain conditions. We have named this new behaviour in the VDF scheme Early Decryptable Verifiable Delay Functions (ED-VDF). Use cases of the ED-VDF scheme include real-world applications such as declassification of secret information, legal testaments and auction systems.

We focus on legal testaments use case in this work. Testaments are binding papers signed by both the testator and the trustee. Testaments can be disclosed before the decedent's death when certain circumstances are met. For example, testaments of decedents who lose intellectual capacity may be unsealed before the decedent's death. The testament procedure is public and legally verifiable by anybody using our ED-VDF protocol. As a result, we eliminate the need for a third party to keep the testament secure until it is revealed. ED-VDF protocol ensures that the criteria of the testamentary procedure are met.

The work is organised as follows: First, in chapter 2, we give information about the VDF and Time-Lock protocol schemes and their delay algorithms. Then, in the Related Works chapter, we compare and contrast similar researches. In the Foundational Technologies for Proposed Solution chapter, we explain our addition to the VDF scheme in detail. The Design and Experimental Works chapter describes our protocol design and implementation on the Ethereum blockchain environment. Subsequently, in the Attacks Analysis chapter, we study the techniques of intrusion and the countermeasures against our protocol. Eventually, we mention extensibility and improvement areas in the Future Work chapter. Finally, we briefly summarise our research and contribution to the VDF scheme.

CHAPTER 2

VDF and TIME-LOCK MECHANISMS

In this chapter, the mechanism of the VDF and Time-Lock protocols is explained in detail. The VDF and Time-Lock protocols are closely related. However, there are approach differences between the two protocols.

Time-Lock protocol applications provide the cryptographic realisation of the "sending encrypted messages to future" concept. In order to provide this concept, the Time-Lock mechanism relies on algorithms that provide sequential work. Such as the Successive Square algorithm. We explain the Successive Square algorithm in section 2.5.1.

VDF protocol structuring Time-Lock protocol phases(Setup-Eval-Verify) to a more streamlined scheme. In addition, VDF protocol focuses on public verifiability of the sequential work output of delay(time-lock puzzle) algorithms. This verification time is exponentially shorter than evaluation time and easily verifies the evaluation output's correctness.

We are focusing on merging these two protocols in such a way that guarantees the Time-Lock and VDF mechanism requirements to designing, verifying, and implementing a new VDF and Time-Lock hybrid protocol that provides a mechanism to authorised individuals(secret-sharer) controlled opening of the time-lock(delay) function is possible before the target time.

Firstly we define mutual requirements in section 2.1 for VDF and Time-Lock protocols. Then we give a brief definition of VDF and Time-Lock protocols shared steps in section 2.2. In addition, we mention common properties of VDF in section 2.3. Finally, we state application domains and real-world use cases of VDF protocol in section 2.4.

2.1. Requirements

A sender (encryptor) prepares an encrypted message and sends it to the recipient (decryptor) in the general scenario. In order to fulfil the protocol premise, the encrypted message can be automatically and autonomously decrypted at the time specified without the sender's presence so that the recipient can read it in time. In addition, senders and recipients should not use any computational resource to perform a delay (puzzle) mechanism. Therefore, existing and proposed solutions must support following requirements:

- **Non-Interactive:** The sender of the message should not be present during the decryption phase. The recipient can open the encrypted message by himself without any interaction.

- **No-Trusted Setup:** Third parties or illegitimate parties should not be involved in keeping the decryption key to perform functions. Thus provide a true idea of sending encrypted messages into the future without interference. In our proposed ED-VDF structure, a new mechanism is proposed to give a chance to legitimate parties to resolve the delay mechanism earlier than the aimed decryption date under legal judgments. We explain the detail of this mechanism in section 4.

- **No-Resource Restrictions:** When the target decryption time has elapsed, the receiver who wants to decrypt the ciphertext should not perform computationally intensive operations to decrypt it. However, this property does not apply to delay(puzzle) mechanisms. To enable the notion of trustworthy timing, there should be a mathematically reliable structure that is not tied to any particular message or person in the system nor any processed function. The related algorithms and their mathematical structures are defined in section 2.5.

2.2. Protocol Phases

The structure of VDFs is built by one-to-one mapped functions. This means that each input x of these functions should have only one valid output y. VDFs must be valid for the following three steps according to two researches [3,4];

- **Setup:** Accepts the time limit (T) input from the sender. Then prepare the delay functions' input parameters as public parameters (pp and x). These are used for the eval and verify steps

- **Eval:** Takes inputs as a delay function and public parameters. Then outputs the solution to the function and the proof of this evaluation.

- **Verify:** Takes input of public parameters, delay function, and solution. Outputs the decision of whether a correct evaluation was given for the delay function. Then the recipient receives a decrypted message.

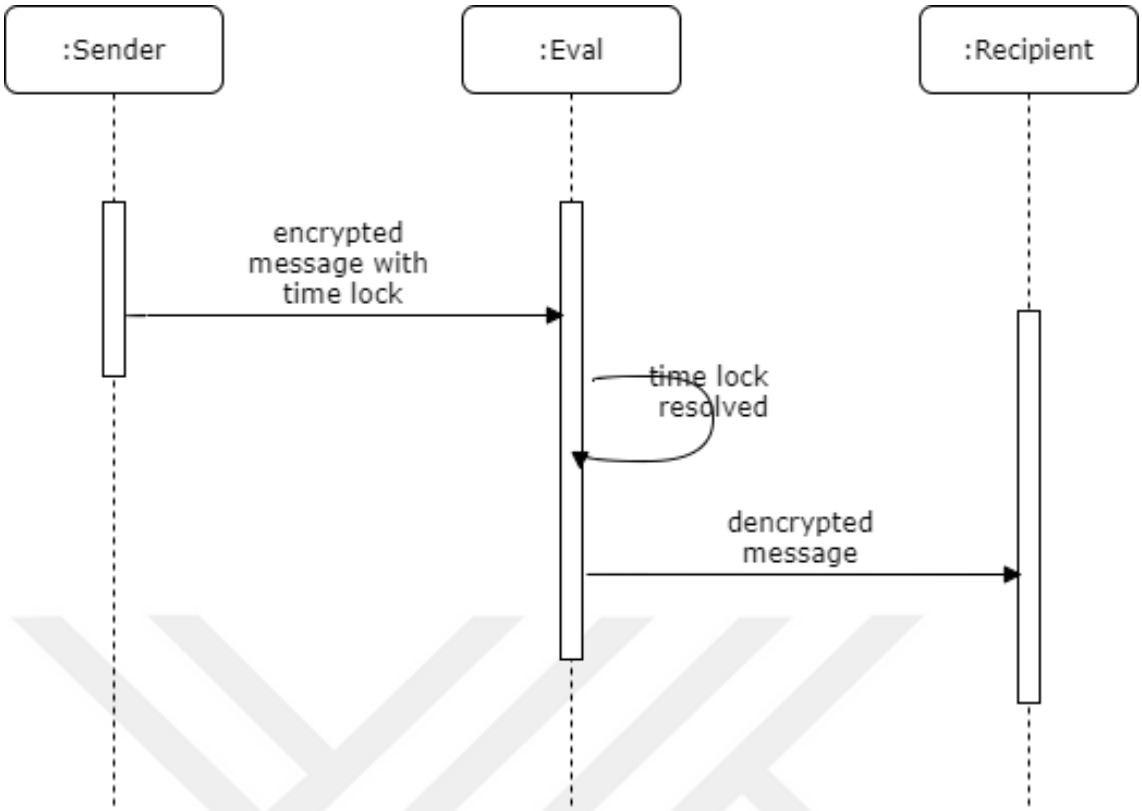


Figure 2.1. VDF Protocol Overview

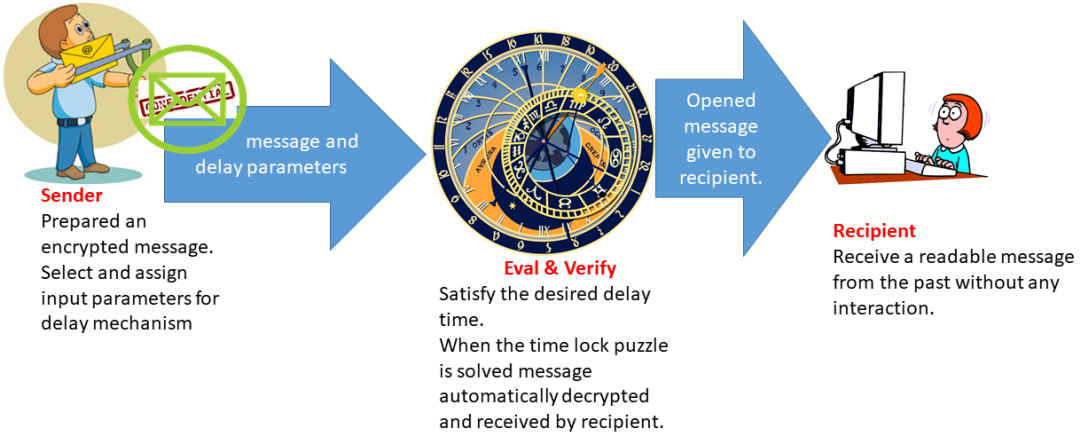


Figure 2.2. VDF Protocol Story Demonstration

Figure 2.1 represents protocol sequence diagram, and Figure 2.2 visualise protocol flow.

2.3. Properties

The most important part of the VDF structure is building a unique delay function evaluation context; otherwise, the requirements mentioned above cannot be satisfied. Due to that reason, Boneh et al. [5] state that the VDF protocol should realise the following properties with described main three steps in section 2.2.

- **Sequentiality:** Eval delay algorithm should run in timeless than T
- **Uniqueness:** For the input x ε X, verify step considers one to one map exactly one y ε Y.

Notations:

- T: Time constraint computed in Setup Phase
- x: public parameters generated in Setup Phase.
- X: public parameters space in Setup Phase.
- y: evaluated output equal to the private parameter in Setup Phase.
- Y: evaluated output equal to the private space in Setup Phase.

2.4. Application Domains and Real World Use Cases

Verifiable delay functions have a wide range of applications, such as auctions and multilateral legal agreements. For instances:

- A person wishes to encrypt his/her testaments(will)in such a way that it cannot be decoded until a certain time and with an event has elapsed.
- Declassification of secret information withheld in public authority records.
- A bidder in an auction wants to seal his bid, so it cannot be opened until the bidding period is closed.
- A community that wants to make sure voting results open after a predetermined time.

2.5. Suitable Algorithms for Protocols

In this section we examine algorithms that provide a delay mechanism for Time-Lock and VDF protocols. Common features of the algorithms require sequential computational steps to calculate the result in aimed delay time, while the time to check the result is relatively concise.

Successive Squaring, Weak Keys and Successive Hashing (Hash Chains) algorithms are used for Time-Lock. In VDF protocol Nakamoto Consensus with Continuous VDF, Random Oracle Model, Witness Encryption, and Random Encodings could be used.

2.5.1. Successive Squaring

Rivest et al. suggest a scheme based on a successive square algorithm in research for Time-lock Puzzles and Timed-release Crypto [1]. In this scheme, puzzle generation steps follow;

1. The sender generates n which is the product of two large random primes denoted p and q.

$$n = pq \tag{2.1}$$

2. Then computes

$$\Phi(n) = (p-1)(q-1) \tag{2.2}$$

3. The sender calculates t with T multiplied by S. T is an expected reveal time in seconds. S is the number of squarings per second. Finally, t is the total squarings modulo that required the complete Time-Lock solving phase, which is equivalent to the VDF eval phase. Note that, S value changes are based on predictions for average computational power solver (evaluators) nodes in protocol network.

$$t = TS \tag{2.3}$$

4. The sender calculates a random key(K) for a symmetric encryption scheme such as RC5 [2]. The encryption key should be strong enough to resist brute force prevent early decryption before the predetermined time of the puzzle(delay).

5. The sender using the RC5 algorithm encrypts message(M) with key(K) to get the

ciphertext.

$$C_M = RC5(K, M) \tag{2.4}$$

6. The sender choose a random a modulo n (with 1 < a < n), and encrypts K as

$$C_K = K + a^{2^t} mod n \tag{2.5}$$

   (a) For efficiency, the sender first calculates

$$e = 2^t (mod \phi(n)) \tag{2.6}$$

   (b) Then computes

$$b = a^e (mod n) \tag{2.7}$$

7. Finally the sender has a output public parameters as $(n, a, t, C_K, C_M))$ and gives the puzzle solver (evaluator as VDF protocols states).

At puzzle solving(eval) phase evaluator should found K in order decrypt message(M) which is encoded as

$$C_K - a^{2^t} (mod n) = K \tag{2.8}$$

However evaluator only knows $(n, a, t, C_K, C_M))$ public parameters as stated in preparation step 7. Thus the evaluator must calculate t squarings sequentially each time squaring the previous result. This process is called successive(repeated) squaring which is an intrinsically sequential process. Calculation continues until finding a satisfying value of K that equal C_K in the finite field $GF(n)$. Here, the delay function is built on the factorization problem of n, due to that large prime factors of n have to be chosen.

Remarks:

1. In our experiments, we use successive squaring algorithm. However, we change the encryption algorithm RC5. We explain these mechanisms in the experimental work in Section 5.2.

2. In preparation step 3, we state that the number of successive squaring computations calculated with find average computational power solver(evaluators) nodes in the network. This property is feasible only in Time-Lock or VDF protocols that run in

blockchain based public networks. We also use a blockchain-based network in our experimental works, and we will explain its details in section 5.2.

3. It is crucial to calculate the time cost of a single operation of the Time-Lock and VDF to ensure the desired delay time before opening the message. For that purpose, we calculate time cost at the setup phase of our protocol. We explain the setup phase in section 4.1.1. We also explain the calculation of the average computational power of the blockchain network in Appendix A.

2.5.2. Weak Keys

The weak-key algorithm is based on short-length symmetric keys chained together to increase resistance against brute-force guessing. A weak key can be found in an intolerable fraction of the time. However, if we use short keys consecutively to encrypt message repeatedly. Decryption time of message can be adjusted one of short key; brute force time probability of the short key.

This algorithm does not address the serialisation/computational power argument. Some actors can break any short key much faster with computational resources, and it makes little difference if it is one of many weak keys. They are still able to open it before others. Therefore, the expected delay time cannot be met.

2.5.3. Successive Hashing (Hash Chains)

Successive hashing uses the output of each previous hash as the input for the next hash. This creates a sequential hash chain from the primary hash to the last successor hash. For taking advantage of this property of iterative hashing, the last successive hash can be used as the symmetric key to encrypt the message. The decryptor repeats the hash steps exactly as many times as the encryption phase. Hash steps a predetermined number that adapts the current computing power to the recomputation time for a hash.

In 2006, Chalkias et al. [6] proposed a solution based on successive hashes and bilinear pairings. They use successive hashes as timestamps and public key infrastructure as integrity and confidentiality mechanisms in their proposed system.

2.5.4. Nakamoto Consensus with Continuous VDF

Wei et al. [12] propose an algorithm based on Nakamoto consensus and Continuous VDF. Nakamoto consensus is a multi-party agreement model used in the Bitcoin blockchain system. Nakamoto consensus is based on the proof-of-work idea. Participants validate their commitment to the blockchain network by computing hard polynomial problems at predetermined intervals.

Continuous VDF (cVDF) [13] research to improve the successive squaring algorithm of Rivest et al. [1]. cVDF reduces the time complexity of the verification step in the successive squaring algorithm.

2.5.5. Random Oracle Model

In cryptography, a random oracle (a theoretical black box) responds to each query with a (truly) random answer chosen uniformly from its output space. When a query is repeated with the same input, it responds in the same way for each query. We can thus assume that such an algorithm can be used as a delay function in VDFs.

Mahmoody et al. [8] explain the random oracle algorithm in more detail. Random oracle algorithm defined is a delay function generator. The output of (where A is the random input and H is the random oracle) is a pair (M, V): the puzzle M and a solution checker V. Given M, the solution checker must output a solution x such that V(x) = 1. When a delay function has a single solution, V compares its input with this constant value to hide an encrypted message.

2.5.6. Witness Encryption

Witness Encryption use NP problems as a delay function. Since it is an NP condition, the computational complexity of finding a satisfying input (called a witness for the language) grows rapidly.

Garcia et al. [11] give the construction of such an encryption scheme based on a multilinear map (a generalisation of a bilinear map, but with arbitrarily many inputs). Then we only need to know the instruction NP to encrypt text, but we need a witness of the appropriate length to decrypt the ciphertext again. The construction of the example witness encryption follows;

- Assume $(x, w) \epsilon R$ where x is statement and w is witness

- Encrypt message m as $c \leftarrow WE.Enc(x, m)$
- With witness w satisfies $(x, w) \epsilon R$ used decrypt ciphertext c as $m = WE.Dec(c, w)$

2.5.7. Random Encodings

Random encodings is a way to compute every $f(x)$ there should be $f(x; r)$ satisfies;

- *Privacy*: $f(x; r)$ should not leak x besides $f(x)$.
- *Correctness*: $D(f(x; r)) = f(x)$ such that D is algorithm for given input x and random r mapping for f(x)

Thus, by the concept of random encoding, the coding of two computations with the same output is indistinguishable, even if the computation behaves differently before the output. For example, one computation may solve a complex problem while the other stops and then produces a hard-coded solution to a complex problem. Jain et al. [14] state the architecture follows this intuition:

1. Encyrptor chooses intricately hard problem f(x) for the delay function.
2. Encryptor generates random encodings to map every x to be relatively simple $f(x; r)$.
3. When decryptor solve f(x) and find y encryptor easily verifies a y is correct solution with that $f(x; r))$ mapping that compute in step 2.

2.5.8. Comparison of Algorithms for Requirements

In the tables below, the compliance status of the algorithms mentioned in section 2.5 to the requirements defined in section 2.1 is compared. Tables are divided into two according to protocol and application use cases. About the column naming of the following tables;

- For the setup phase, the “Non-Trusted Setup” requirement addresses that a third party agent is not needed.
- In the eval phase, the "Non-Interactive" requirement checks whether the algorithm does not need a sender or a receiver to guarantee the aimed delay time.

- For all phases, “No-Resource Restrictions” checks whether the algorithm has no computational requirement constraints for its realisation.
- Implementable column checks whether an algorithm has been implemented with the current software technology capabilities.

Table 2.1. Time-Lock delay algorithms comparison against requirements in section 2.1

	Non-Interactive	Non-Trusted Setup	No-Resource Restrictions*	Implementable
Successive Squaring	+	+	-	+
Weak Keys	+	+	-	+
Successive Hashing (Hash chain)	+	+	-	+

Table 2.2. VDF delay algorithms comparison against requirements in section 2.1

	Non-Interactive	Non-Trusted Setup	No-Resource Restrictions*	Implementable
Nakamoto Consensus with Continuous VDF	+	+	+	+
Random Oracle Model	+	+	-	-**
Witness Encryption	+	+	-	+***
Random Encodings	+	+	-	+

* “No-Resource Restriction” requirement only satisfies if the blockchain network has enough distributed computing power on the network.

** True Random Oracle function is not feasible due to the limitation described in The Church-Turing Thesis [47] as “no function computable by a finite algorithm can implement a true random oracle”.

*** Witness Encryption based on NP-complete problems, and every NP problem does not have a known solution. However, Liu et al. [9] researched in “How to build a time-lock encryption” paper to show a real-world implementation of Witness Encryption.

CHAPTER 3

RELATED WORKS

The most recent research related to our contribution are Simple Verifiable Delay Functions [3], Efficient verifiable delay functions [4] and Homomorphic Time-Lock Puzzles and Applications [10].

3.1. Simple Verifiable Delay Functions

K. Pietrzak offered a technique to create a Verifiable Delay Function (VDF) [3] by demonstrating how the Rivest-Shamir-WagnerTime-Lock puzzle may become publically verifiable.

The answer to this problem is to conceal the group order in some way, such that it can only be used to efficiently test if a particular solution is right, rather than to speed up its calculation. There is no known implementation of this technique at the moment.

K. Pietrzak devised a technique in which a prover P can persuade a verifier V that it computed the correct result $y = x^{2^T} mod(N)$ (Successive Square algorithm described section 2.5.1) SVDF's interactive protocol is public-coin, however it may be turned non-interactive and hence yield a VDF through the Fiat-Shamir transformation [68].

In this case, the prover's messages are substituted by a random function applied to the transcript. When performed constant-round public-coin interactive proof system, the Fiat-Shamir transformation yields a solid non-interactive proof system.

Although SVDF's proof is not constant-round, it can nevertheless demonstrate transformation works, it provides a sound non-interactive proof method in comparison to a random function. In reality, the random function is realized with a real hash function, such as SHA256; soundness holds only computationally; such systems are referred to as arguments, not proofs.

3.2. Efficient Verifiable Delay Functions

Efficient Verifiable Delay Functions(EVDFs) [4] is a protocol that although evaluating takes a certain quantity of consecutive stages, the outcome may be rapidly confirmed.

B. Wesolowski constructed EVDFs, with a trapdoor VDF. A trapdoor VDF is essentially a VDF that can be evaluated efficiently by parties who know a secret (the trapdoor). EVDFs construction is also built on groups of unknown order class imaginary quadratic field such as RSA.

The construction output is succinct, and the verification of accuracy is quite efficient. Similarly Rivest et al. research [1]: given as input an RSA group $(Z/N)^x$, N is a product of 2 large, primes, a arbitrary component x ε $(Z/N)^x$, and a timing constraint t to compute x^{2^t}.

This task necessitates t consecutive squarings in the group without the factorisation of N. This design is just a time-lock problem, not a VDF, since there is no feasible technique to verify that supplied an output $y = x^{2^t}$.

The EVDF design involves of answering an instance of the Time-Lock puzzle [1] and calculating a proof of correctness that allows anybody to easily check the outcome. Set the time parameter Δ, the security degree k, and the group G. The essential characteristics are associated with EVDF properties:

1. It is $\Delta - sequential$ To calculate, it takes (Delta) consecutive steps, given Time-Lock [1] constraint of cite1 in the group G.
2. It is fair to say that it is impossible to create a valid argument for a wrong output given particular group-theoretic criteria on G, which are thought class groups of quadratic imaginary number fields.
3. Every one of output and the evidence of correctness is an unique member of the group G.
4. The evidence can be generated through group operations in $O(\Delta/log(\Delta))$.

The suggested structure is a trapdoor VDF, from which a genuine VDF may be derived, with the following use scenario: Alice has a secret key sk (the trapdoor) and an attached public key pk. Provided a data x, trapdoor verifiable delay functions allow a hidden trapdoor to calculate an output y from x or Δ(time) a sequential work. Trapdoor verifiable delay functions have the following steps:

- $keygen \rightarrow (pk, sk)$ is a key production technique that generates Alice's public and secret keys, pk and sk.
- $trapdoor_{sk}(x, \Delta) \rightarrow (y, \phi)$: Receives the data x ε X as input and utilizes the secret key sk to create the output y from x, as well as a proof ϕ. The variable Δ shows

the amount of sequential effort necessary to calculate the identical result y in the absence of the secret key.

- $eval_{pk}(x, \Delta) \rightarrow (y, \phi)$ technique for evaluating the delay function on x with public key pk for a certain number of sequential work Δ. It generates the output y given x and an evidence ϕ. This process is designed to be impracticable in less than Δ.
- $verify_{pk}(x, y, \phi, \Delta) \rightarrow trueorfalse$: Examine if y is the right result for x, as related public key pk and evaluation period Δ.

Without knowing of the secret key sk, the right result cannot be created in less than Δ time. Our contribution also relies on a trapdoor verifiable delay functions scheme that bridges the Time-Lock puzzle and VDF protocol.

3.3. Homomorphic Time-Lock Puzzles and Applications

Homomorphic Time-Lock Puzzles (HTLP) [10] is a Time-Lock puzzle which enables anybody to homomorphically assess a circuit C across sets of puzzles (Z1,..., Zn). Without being knowledge of the secret messages (s1,..., sn) encoded in puzzles. The resultant output (puzzle) includes the circuit output C(s1,..., sn), and timing difficulty of this problem is independent of the length of evaluated circuit C. (compactness).

Construction of such a scheme using successive squaring [1] described for parameters

$$(N, T, x, x^{2^T}.k, Enc(k, s)) \tag{3.1}$$

where N=p.q are RSA prime integers, s secret key, and T time constraint tuples. Satisfies (x, k) samples in Z_N^* and Enc(k, s) symmetric encryption with s. Using paillier cryptosystem [15], group $Z_N^{2^s}$ can be redefined as a product of (1+N) and a group of N-th residues $X^N : x \in Z_N^s$) order p(N) parameters written as

$$(N, T, x, x^{N^{2^T}}.(1+N)^s)) \tag{3.2}$$

for random $x \in Z_N^s$ with fixed N equal to

$$(N, T, x.y, x^{N2^T}.y^{N2^T}.(1+N)^s.(1+N)^s) = (N, T, (x, y), (x.y)^{N.2^T}.(1+N)^{s+s}) \tag{3.3}$$

Prove linearity of homomorphic property of successive squaring on paillier cryptosystem.

The main focus of Malavolta et al. [10] research on HTLP is to reduce the consumption of computational resources when solving blocking (delay) functions for VDFs. To achieve this, the generated delay function must satisfy homomorphic scheme properties. This allows the aggregation of lock(delay) functions into an overall solution of the delay function. However, homomorphic schemes determine the use of delay function algorithm types. Therefore, different approaches for delay functions with these requirements should be considered.

We also provide a configurable, verifiable runtime for the delay function for the first time, which is an improvement for Homomorphic Time-Lock Puzzles and Applications [10]. This makes it easy to use HTLP as cost-saving prevention. For example, independent delay functions can be solved with a single block as long as they satisfy the homomorphic scheme.

CHAPTER 4

FOUNDATIONAL TECHNOLOGIES for PROPOSED SOLUTION

Time-Lock puzzles define a solution to sending encrypted messages into the future. In addition to that, Verifiable Delay Functions(VDFs) are concise and well-formed protocol that specify steps to verify delay(puzzle) algorithms solution proofs publicly. However, neither of these protocols provided a solution to the secure early decryption mechanism. This section provides a detailed explanation for our contribution to existing protocols. Afterwards, we describe foundational technologies that we use for our contribution.

4.1. Main Structure of Proposed ED-VDF Protocol

We propose a solution integrating a secret-share algorithm to construct a new Time-Lock and VDF protocol hybrid scheme that provides an early decryption mechanism. Our contribution is based on Efficient Verifiable Delay Functions(EVDFs) [4]. We named this scheme Early Decryptable Verifiable Delay Functions(ED-VDF). ED-VDF scheme is constructed as following main steps;

1. **SETUP:** The generation of the delay(puzzle) and the public and private parameters is prepared in the setup phase. Then the message is encrypted with a private parameter.
2. **EARLY-DECRYPTION SETUP:** Private parameter used as a shared secret key constructed with a Shamir Secret Share algorithm in the early decryption setup phase.
3. **EVAL:** Delay(puzzle) algorithm evaluated to provide sequential work in this phase using public parameters and a delay(puzzle).
4. **EARLY-DECRYPTION:** Suppose the Secret Share Participants want to open the encrypted message before the specified time. Secret share participants reconstruct the private parameter key and decrypt the encrypted message with a reconstructed private parameter.

5. **VERIFY:** If step 4 is not executed before the Eval phase is complete. The validation phase begins by evaluating the output of the delay(puzzle) algorithm. Then, the delay(puzzle) output is used to decrypt the encrypted message.

In Figure 4.1, the overview of the proposed ED-VDF solution is presented, and details of the design will be explained in the following subsections 4.1.1.

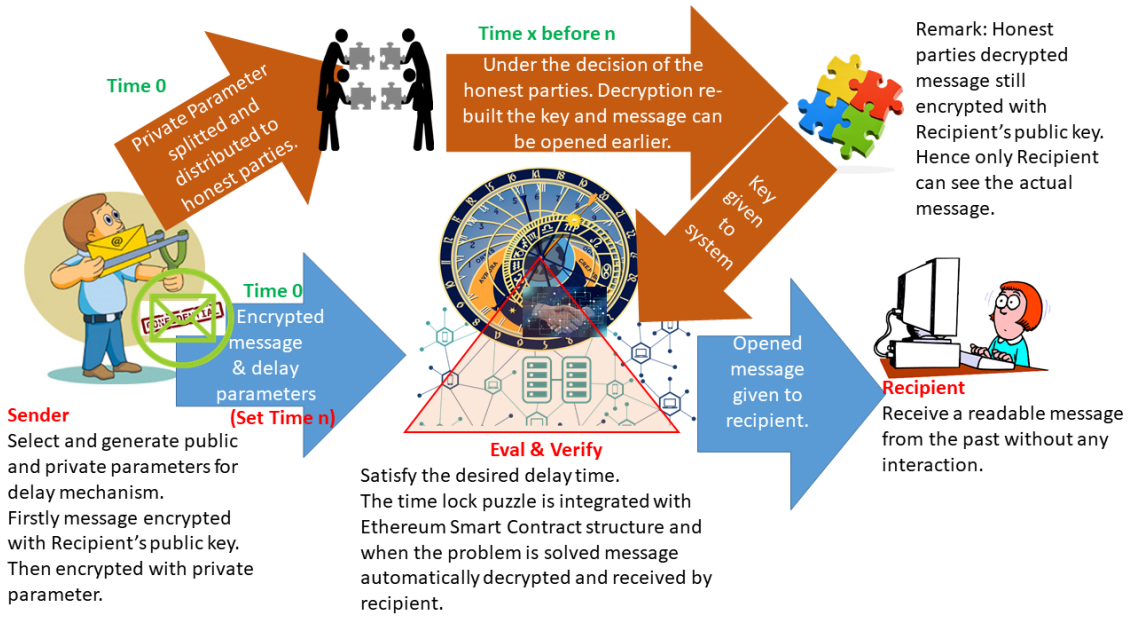


Figure 4.1. ED-VDF Protocol Demonstration

4.1.1. Setup Phase

Setup phase, the same as the EVDFs scheme, we construct x ε X is input where the VDF will be assessed which X is finite group G. The group G is $(Z/NZ)^x$ where $N = pq$ given a pair of different prime numbers p and q, using $(p-1)(q-1)$ as the secret key. For a given time variable t, the aim evaluation time is provided by $\Delta = t\delta$, where δ is the time-rate (number of sequential work) to calculating an unique squaring in group G.

After computation of public and private parameters, the sender encrypts her/his message with the receivers' public key then encrypts the message with private parameter, as shown in Figure 4.1. Then public parameters and encrypted messages are published to the Ethereum network as a transaction. Transaction distributed through Ethereum network as sealed and immutable state.

4.1.2. Early-Decryption Setup Phase

Shamir's Secret Sharing algorithm is based on the Lagrange interpolation theorem. Algorithm aim is to divide secret S into n pieces of data $S1$, ..., Sn (known as shares) Then set reconstruction threshold k; that:

- Knowledge of any k or more pieces makes S constructible.
- Knowledge of any k-1 or fewer pieces leaves S non constructible.

In ED-VDF protocol, private parameter split into shares and set threshold value as Shamir Secret Share reconstruction. Afterwards, private parameter shares(parts) send the Secret Share Participants' Ethereum Account through to the Ethereum network as a transaction.

We note that Secret Share Participants' shares are encrypted with their account public key. In addition, the threshold value for the Early Decryption phase send it as an Ethereum Smart Contract transaction. Ethereum Smart Contract and Account systems are explained in detail in sections 4.2.2 and 4.2.3.

4.1.3. Eval Phase

In ED-VDF Eval phase is processed at one of the Ethereum Miner nodes. The selection of the miner node is random in the Ethereum network. Evaluate phase similar to EVDF mechanism for input x;

1. The main notion is that sender can easily calculate g^{2^t} using two exponentiations for every t ε Z > 0 by first calculating $e = 2^t mod|G|$, then g^e.
2. The runtime in t is logarithmic. Anyone else who not know |G| could calculate g^{2^t} by doing t sequential squarings with a runtime of t.
3. As a result, everyone can calculate $y = g^{2^t}$, however only sender can accomplish it quickly, and any other participant must spend time linear in t.

Notations:

- t: Time bound parameter
- g: Base prime

- e: RSA coprime
- G: Group order

4.1.4. Early-Decryption Phase

If Secret Share Participants decide to early decryption. They can decrypt encrypted message using the reconstruction of private parameter with their shares from the Early-Decryption Setup Phase. In order to start the early decryption process. They send their secret share parts through Ethereum Smart Contract transactions.

If Secret Share Participants reach a threshold value computed at the Early-Decryption Setup phase. Reconstructed private parameter use to decrypt encrypted message first layer. Then the message is sent to the recipient, and the recipient decrypts the encrypted message with his or her private key. We explain the public key cryptography system in the Ethereum Account system in section 4.2.3.

4.1.5. Verify Phase

Protocol verification phase similar to EVDFs scheme with value y as output of Eval phase not have a shortcut to the apparent strategy consisting in recomputing g^{2^t} and checking if it matches. To solve this issue, B. Wesolowski proposes the following public-coin succinct argument to prove that $y = g^{2^t}$. Detail explaination at EVDFs [4] paper Section 4.1. When verification of Eval phase output, use decrypt encrypted messages. Then the message is sent to the recipient and the recipient decrypts the encrypted message with his or her private key.

4.2. Foundational Technologies

This section explains the technologies that form the basis of our ED-VDF scheme. As we mentioned in Section 2.2, the protocol consists of three phases. With the ED-VDF scheme, we add two intermediate phases that provide an early decryption mechanism, as mentioned in section 4.1.

To satisfy the requirements of both the existing Time-Lock and VDF protocols with the addition of ED-VDF, we chose to use Ethereum blockchain and smart contract technologies. Ethereum provides a “no-resource-restriction” runtime environment. Ethereum

Smart Contract provides a public medium that meets “non-interactive” and “non-trusted setup” as mentioned requirements in sections 2.1 and 4.1 to implement the ED-VDF protocol.

4.2.1. Ethereum Blockchain

Ethereum Blockchain technology allows a network of computers to periodically agree on the true state of a public ledger [50]. Activities on a blockchain are stored in chronological order, resulting in an irreversible record. Depending on how the system is set up, transactions on a blockchain can be more confidential or anonymous. The network’s ledger is scattered across many people and does not exist in a single location. Ethereum Blockchain data is stored in each actively participating node in the network. The types of data stored in the Ethereum Blockchain are; currencies, digitised claims, proprietary information, identifications, and inventory quantities.

Ethereum is a cryptographically secure transactional shared-state singleton machine [16]. Cryptographically secure means that advanced mathematical methods protect Ethereum block production. This makes it extremely difficult to commit fraudulent transactions in Ethereum.

The term “transactional singleton machine” refers to a machine with only a single version that is accountable for all transactions performed on the network; in other words, there is now only a single concrete shared state that everyone on the Ethereum Node Network [55] accepts. A “shared state” refers to the fact that the state stored on this system is public and accessible to everyone.

The Ethereum Virtual Machine (EVM) [32] is a distributed state machine built on transactions in computing. A state machine evaluates a series of inputs and afterwards move into a new state based on those inputs. Ethereum’s state machine starts with a genesis state, which resembles a clean canvas until any network transactions have taken place. From this genesis state, it transitions to a final state after transactions have been processed, with this final state reflecting Ethereum’s current state at any given time. Every Ethereum Node has an identical copy of EVM.

Each Ethereum transaction is added to a block in the Ethereum blockchain. A transaction must be legitimate to move from one state to another. A transaction must go through a verification process called "mining" to be declared genuine. The "mining" process is performed by one of the nodes on the Ethereum network, using its computing resources to create a block of legitimate transactions. The mining process is a crucial function for our ED-VDF protocol. We use this function to distribute the ED-VDF phases among the mining nodes.

Ethereum Blockchain mainstay of ED-VDF protocol. Ethereum Blockchain allows us to meet the requirements of VDF, we state in section 2.1. Also, satisfy ED-VDF early decryption requirements at the same time in section 4.1. We will explain the use of Ethereum in detail in our ED-VDF protocol in section 5.2.

4.2.2. Smart Contract

The smart contract is a program that runs on the Ethereum virtual machine [32]. Like real contracts, smart Contracts [51] can set rules and implement them using code itself. It is a set of commands, functions, data, and state stored at a single Ethereum blockchain address. The contract data needs to be allocated to a memory or storage space. Contracts can be used by another contract.

Smart contracts are not managed by a user but are installed on the network and work with triggered transactions. Moreover, real user accounts can connect to a smart contract by sending transactions that perform a function defined by the smart contract.

Smart Contracts on Ethereum are public and can be considered open APIs. Still, Smart Contracts cannot communicate directly outside of the Ethereum Virtual Machine because the contracts cannot establish HTTP calls. However, Ethereum network nodes can interact with the smart contract and execute transaction requests from users. In addition, users can call other users' smart contracts to expand their smart contract capabilities significantly.

We use smart contracts with decentralised data storage and autonomous function call properties to satisfy the ED-VDF requirements. We solve the problem of non-interactive requirements with autonomous function calls of smart contracts. Once the ED-VDF smart contract is deployed in the Ethereum blockchain; Its receiver can decrypt a message without the sender of the contract being present.

In addition, we provide secure, independent, secret message storage for VDF contracts with the decentralised data storage feature of smart contracts, as each smart contract has its own unique and confidential storage on the Ethereum blockchain.

4.2.3. Ethereum Account

An Ethereum account is entity in the Ethereum that can send transactions to Ethereum network. All participants of protocol should have an account.Accounts can be managed by the user or distributed as smart contracts. The Elliptic Curve Digital Signature Algorithm is used to produce the account's public key from the private key [69].

The private key is used to sign communications and transactions that produce a signature. Other account holders can then use the signature to generate a public key for the account, validating the message's authorship.

4.2.4. The Work in Proof-of-Work Concept

The proof-of-work algorithm forces miners to compete in a guessing game to find the nonce for a block. Only blocks with a valid nonce were allowed to be joined to the chain. A miner racing to produce a block will continually send a dataset through a cryptographic algorithm that can only be achieved by getting and executing the whole chain. The dataset is used to generate a mixHash that is less than a nonce generated by the block challenge.

The hash's aim is determined by the challenge. The lower the threshold, the fewer valid hashes produced when there are. When a hash is produced, all other miners and clients may easily validate it. Despite the fact that just one transaction changed, the hash would be different, suggesting forging.

4.2.5. Chain Finality

Because miners employ a decentralized mechanism, numerous genuine blocks may be mined at the same time, resulting in a temporary chain fork. The Ethereum network selects which of these chains will eventually be recognized as the main chain.

On rare cases, transactions refused on the transitory fork may be accepted on the permitted chain. This suggests that transactions might be reversed. Ethereum's proposed length is six blocks, or little more than one minute. Following the first six blocks, it is reasonable to assume that the transaction was successful. As a result, finality is determined by the number of blocks passed before declaring a transaction permanent.

4.2.6. Block Mining Times

The Ethereum network is set up to generate a block of 12 second periods. Block timings may fluctuate depending on the time needed to create a hash that matches the current mining difficulty. Twelve seconds was selected as a period that is as quick as feasible while yet being significantly longer for network latency.

The 12-second configuration's primary goal is to enable the network to disseminate

blocks as quickly as possible while preventing the detection of a significant percentage of stationary blocks by miners. Decker and Wattenhofer published a study in 2013 in Zurich that assessed Bitcoin network delay [66] and discovered that it takes 12.6 seconds for a new block to spread to 95% of nodes. Appendix A has a full description of the block mining difficulty computation.

CHAPTER 5

DESIGN of ED-VDF PROTOCOL and EXPERIMENTAL WORKS

This section provides design and implementation details to our contribution for empirical proof on real-world application. Then we give use case scenarios for our contribution in section 5.3.

5.1. Design

Our ED-VDF scheme design is visualised as 2 parts. Those charts are namely high-level protocol flowchart and protocol implementation sequence diagram.

5.1.1. High-Level Protocol

Below Figures 5.1-2 describes the first part of our design flowchart the start to the evaluation phase. Pseudocode represent experimental works section 5.2 contains the start to the evaluation phase.

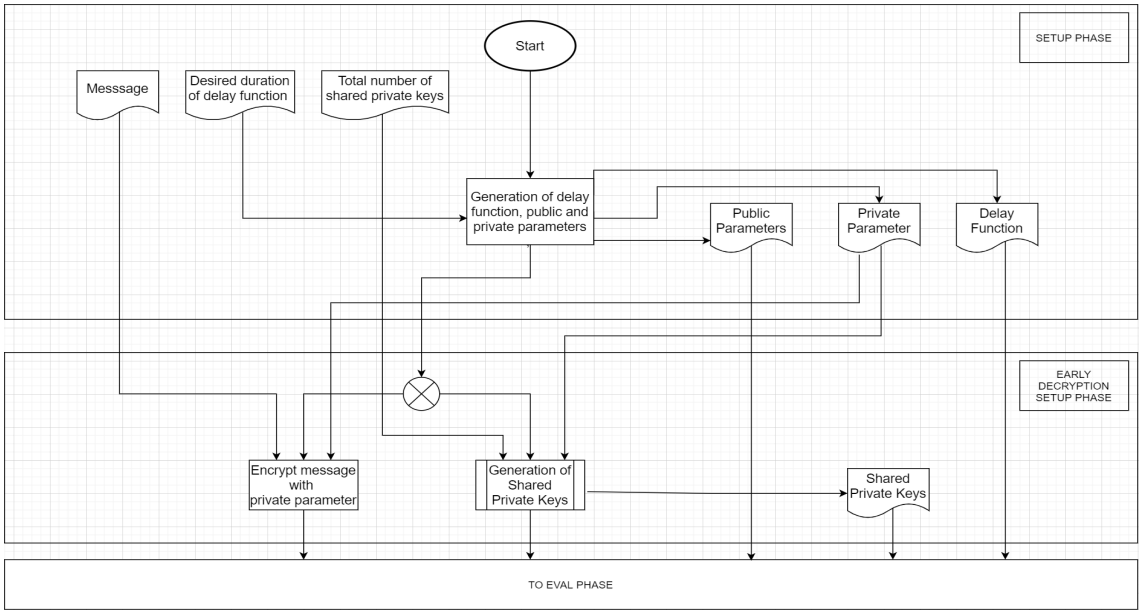


Figure 5.1. High-Level Protocol Flow Chart representation for Setup Phases

```
function Setup(DelayDuration) do
    PublicParameters, PrivateParameter, DelayFunction = GenerateDelayFunction(DelayDuration)
    return PublicParameters, PrivateParameter, DelayFunction
end

function EarlyDecryptionSetup(PrivateParameter, TotalNumberOfPrivateKeyShares, NumberOfReconstructionThreshold) do
    //Shamir Secret Share Construction
    SharedPrivateKeys = GenerationOfSharedPrivateKeys(TotalNumberOfPrivateKeyShares, NumberOfReconstructionThreshold, PrivateParameter)
    return SharedPrivateKeys
end
```

Figure 5.2. High-Level Protocol Pseudocode representation for Setup Phases

As shown in Figure 5.3, the protocol starts with a message, delay duration, recipient public key(Ethereum account public address) and a total number of private key shares as input from the sender. Then the Setup phase prepares the delay function, public and private parameters for next protocol phases.

```
// INPUT : PlainMessage, TotalNumberOfPrivateKeyShares, NumberOfReconstructionThreshold, RecepientPublicKey
PublicParameters, PrivateParameter, DelayFunction = Setup(DelayDuration)

EncrpyedMessageForRecepient = EncryptMessage(PlainMessage, RecepientPublicKey) // Asymmetric Encryption
EncrpyedMessage = EncryptMessage(EncrpyedMessageForRecepient, PrivateParameter) // Symmetric Encryption

SharedPrivateKeys = EarlyDecryptionSetup(PrivateParameter, TotalNumberOfPrivateKeyShares, NumberOfReconstructionThreshold)
```

Figure 5.3. High-Level Protocol Pseudocode representation for Setup Flow

Then, the message is asymmetrically encrypted with the recipient public key. Afterwards, the message is symmetrically encrypted with the private parameter. Simultaneously, the private parameter is split into the desired number of secret shares using Shamir's Secret Sharing Scheme in the Early Decryption Setup phase, as shown in the Generation Of Shared Private Keys subflow in Section 5.1.2. Subsequently, the delay function eval phase starts.

Figures 5.4 and 5.5 describes the second part of our design, which contains the Eval, Verify and the Early Decryption Setup phases. The Eval phase can be terminated by two cases: (i) when the desired delay time has expired, or (ii) when the legitimate secret share participants can decide to terminate it earlier than the desired delay time. Secret share participants can decrypt the message early until the eval phase is completed.

Figure 5.6 describes if secret share participants decide to early decryption. Private parameter reconstructed with participants' secret shares using the Shamir Secret Share scheme, as shown in Section 5.1.2 Reconstruction the Private Parameter flow. Alternatively, In the Eval phase, the evaluator of the delay function finds private parameter after the delay duration.

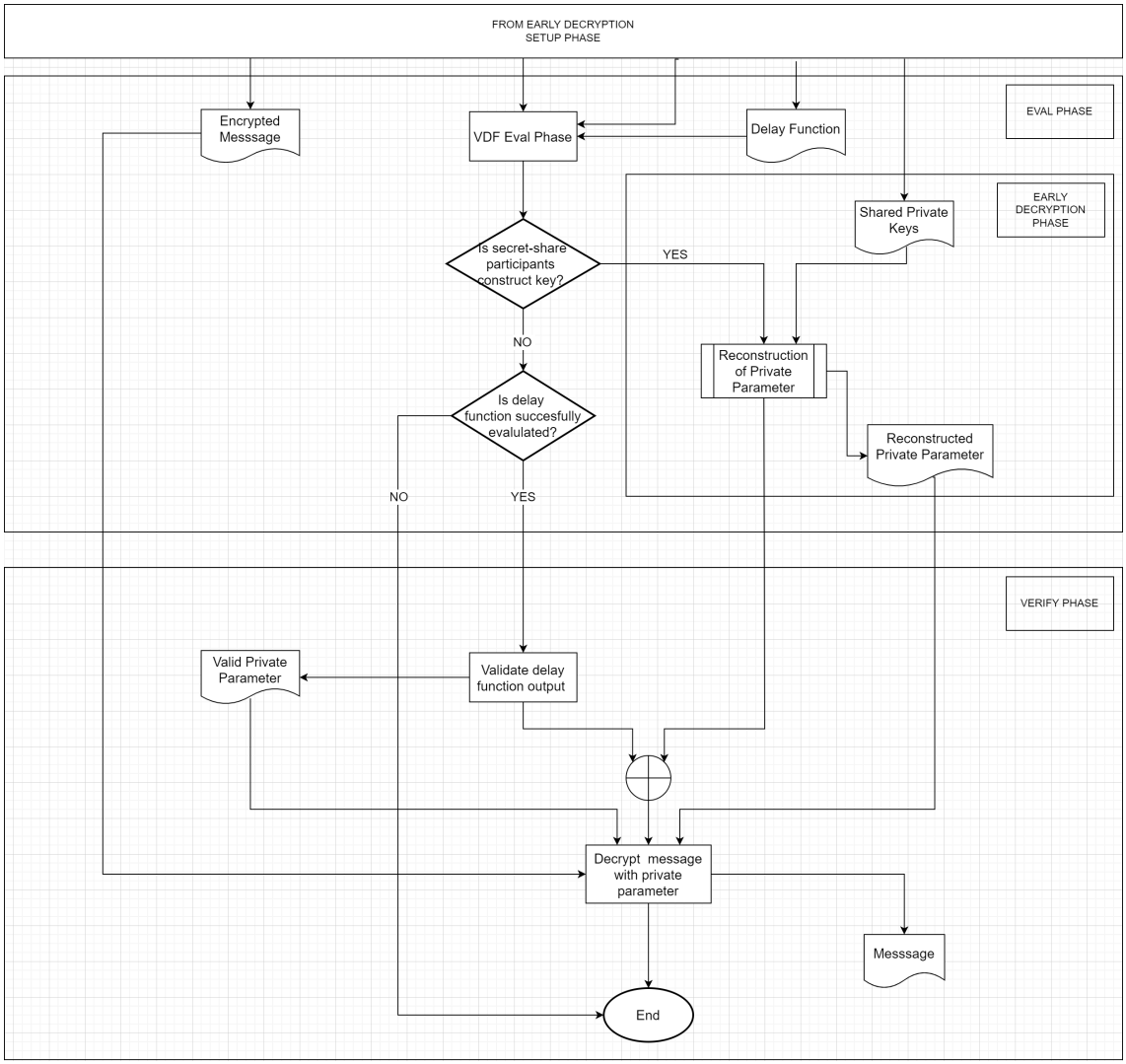


Figure 5.4. High-Level Protocol Flow Chart Second Part for Eval and Verify Phases

```
function Eval(PublicParameters, DelayFunction) do
    PrivateParameter = EvaluateDelayFunction(DelayFunction, PublicParameters)
    return PrivateParameter
end

function EarlyDecryption(SharedPrivateKeys, NumberOfReconstructionThreshold) do
    // Shamir Secret Share Reconstruction
    ReconstrcutedPrivateParameter = ReconstrcutionOfPrivateParameter(SharedPrivateKeys, NumberOfReconstructionThreshold)
    return ReconstrcutedPrivateParameter
end

function Verify(PrivateParameterToBeValidated, PublicParameters) do
    return ValidateDelayFunctionOutput(PrivateParameterToBeValidated, PublicParameters)
end
```

Figure 5.5. High-Level Protocol Pseudo Code Second Part for Eval and Verify Phases

```
if Is_Secret_Share_Participants_Decide_Early_Decryption == TRUE do
    Output_PrivateParameter = EarlyDecryption(SharedPrivateKeys, NumberOfReconstructionThreshold)
    EncrpyedMessageForRecepient = DecryptMessage(Output_PrivateParameter, EncrpyedMessage)
end
else do
    Output_PrivateParameter = Eval(PublicParameters, DelayFunction)
end

IsOutputPrivateParameterVerified = Verify(Output_PrivateParameter, PublicParameters)
if IsOutputPrivateParameterVerified === TRUE do
    EncrpyedMessageForRecepient = DecryptMessage(Output_PrivateParameter, EncrpyedMessage)
    PlainMessage = DecryptMessage(RecepientPrivateKey, EncrpyedMessageForRecepient)
end

// OUTPUT: PlainMessage
```

Figure 5.6. High-Level Protocol Pseudo Code Second Part for Eval and Verify Phases

Then the output private parameter will be verified for VDF evaluation requirements. Consecutively validated output using as a private parameter to decrypt encrypted message to encrypted message with the recipient public key form. Then the message is decrypted by the Recipient with his private key to the plain message form.

As we describe in the above flowcharts, our contribution to the VDF protocol as an ED-VDF scheme is that the current VDF protocols do not support an early decryption mechanism. We decrypt the message with an additional phase for private key distribution using Shamir Secret Share and Early Decryption capability using the thresholds of the distributed shares private key.

Present researches on VDF protocols are based on specific fix delay algorithms tailored to the problem domain. Contrary to other VDF researches, our ED-VDF scheme delay function(algorithm) is loosely coupled, and any delay function(algorithm) as presented under Section 2.5 can be prefered.

5.1.2. Early Decryption Phase

We use the Shamir Secret Share algorithm [59] as described in the figures below. We implement a succinct and cryptographically secure early decryption system to our ED-VDF protocol with this algorithm. We visualise the Generation of Shared Private Keys and Reconstruction of Private Parameter flows in Figures 5.7 and 5.8 previously referenced in the design flowchart in section 5.1.1.

The sender uses this solution after the encryption of the message. The secret key is divided into many pieces and distributed to legitimate parties, as in Figure 5.7, who have the grant to open this encrypted message under specific conditions, i.e. a decision from a court. As shown in Figure 5.8, the legitimate owners of shared keys combine these partial values according to a threshold number of partial keys, and the secret key can be rebuilt.

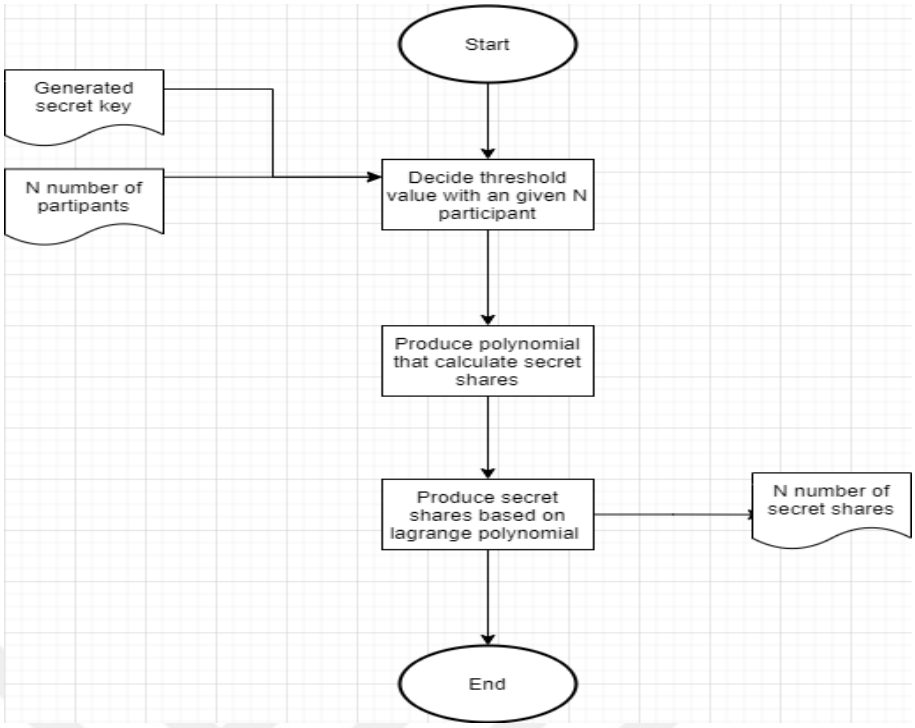


Figure 5.7. Generation of Shared Private Keys

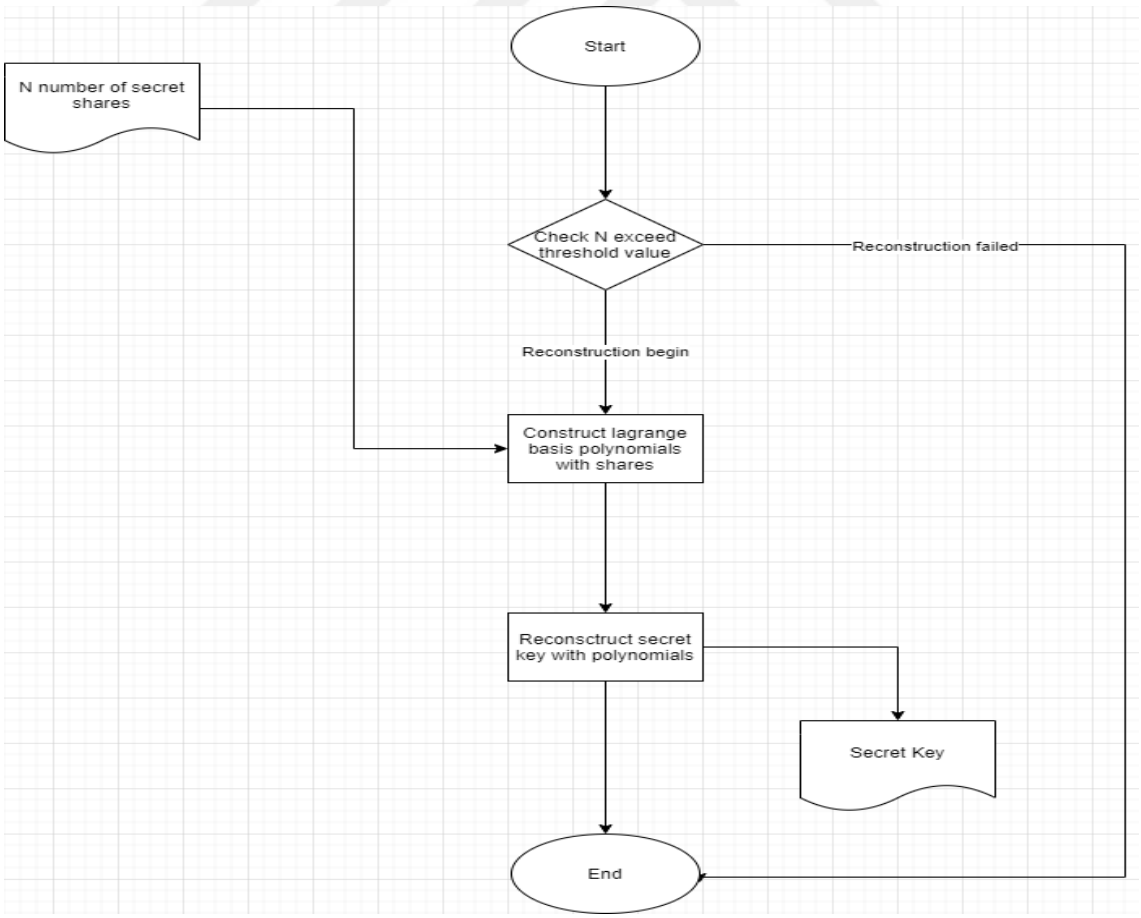


Figure 5.8. Reconstruction of Private Parameter

5.2. Experimental Works

We use the Ethereum Decentralized Applications (DApps) [29] structure to implement our ED-VDF protocol design. DApps is a blueprint for applications built on Ethereum smart contracts. We chose the DApp Hardhat [54] infrastructure to scaffold ED-VDF experimental works. Hardhat provide a convenient development environment. We use Solidity Smart Contract Language [70] to write ED-VDF Smart Contract. Experimental works published publicly on our Github repository [71]. In addition we add full ED-VDF smart contract visualisation in Appendix B. Our application structure is divided into 4 modules: Artifacts, Contracts, Scripts and Tests.

- Artifacts: Contains compiled contracts and their build information.
- Contracts: Holds contracts written with Solidity programming language.
- Scripts: Embody code that provides a bridge between ethereum virtual machine (EVM) [32] and application code that manages interaction, transaction and listening events on VDF smart contract at EVM.
- Test: Enclose unit, functional and attack scenario tests for implementation.

As mentioned in section 5.1.1, implementation follows ED-VDF protocol phases. These protocol phases are implemented in the Ethereum Blockchain system.

First, we implement the ED-VDF phases in Smart Contract format, and Appendix B presents an example of the Smart Contract format created by our implementation. We use the Solidity programming language [31] to implement the Smart Contract. The Smart Contract represents the ED-VDF protocol in the Ethereum Virtual Machine (EVM) [32]. To use the ED-VDF smart contract, we compile the Application Binary Interface (ABI) [53] and deploy it to Ethereum Network as a transaction using the Ethereum client [55]. ABI is a gateway for data exchange between the application code and the EVM.

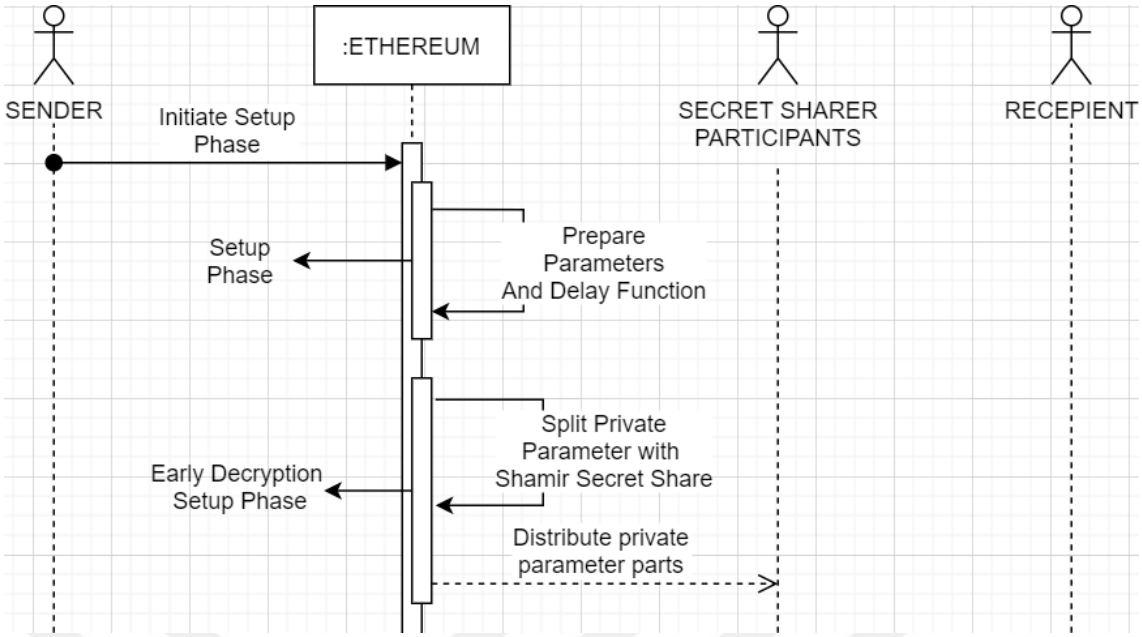


Figure 5.9. Protocol Implementation Sequence Diagram First Part

The above sequence diagram describes when the sender wants to initiate a new ED-VDF instance, it performs a new Setup and Early Decryption Setup as Ethereum transaction [56] using below code snippet in Figure 5.10.

```
function Setup(uint256 _N, uint256 _T, string memory _RECIPIENT) public
{
    require(SENDER == msg.sender);
    require(keccak256(abi.encodePacked(STATUS)) == keccak256('INIT'), 'ED_VDF should be at INIT status before Setup phase');

    INIT_TIMESTAMP = block.timestamp;

    STATUS = 'SETUP';
    Time = _T;
    N = _N;
    RECIPIENT = _RECIPIENT;

    x = uint256(blockhash(block.number - 1)) % N;

    emit SETUP(x);
}

function EarlyDecryptionSetup(string[] memory _SECRET_SHARER_PARTICIPANTS) public
{
    require(SENDER == msg.sender);
    require(keccak256(abi.encodePacked(STATUS)) == keccak256('SETUP'), 'ED_VDF should be at SETUP status before EarlyDecryptionSetup phase');
    require(bytes(encryptedMessage).length != 0, 'encryptedMessage should be set before EarlyDecryptionSetup phase');

    STATUS = 'EARLY_DECRYPTION_SETUP';

    SECRET_SHARER_PARTICIPANTS = _SECRET_SHARER_PARTICIPANTS;
    emit EARLY_DECRYPTION_SETUP(SECRET_SHARER_PARTICIPANTS);
    emit EVAL(N, Time, x);
}
```

Figure 5.10. ED-VDF Smart Contract First Part

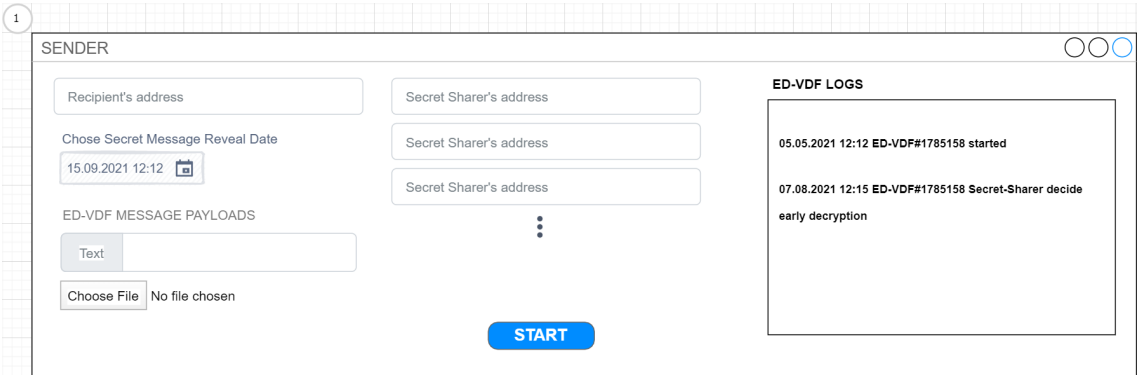


Figure 5.11. UI Mockup design for Sender

Eventually, transactions are processed as a block by one of the Ethereum miner nodes [55] in the Ethereum network. After the setup transactions are successfully processed, the new ED-VDF instance stores the sender, receiver and secret-sharer information as Ethereum account address [57] and encrypted message in EVM.

Then, the Eval transaction is automatically triggered after the setup transactions are completed. Subsequently, one of the miner nodes starts evaluating the delay function.

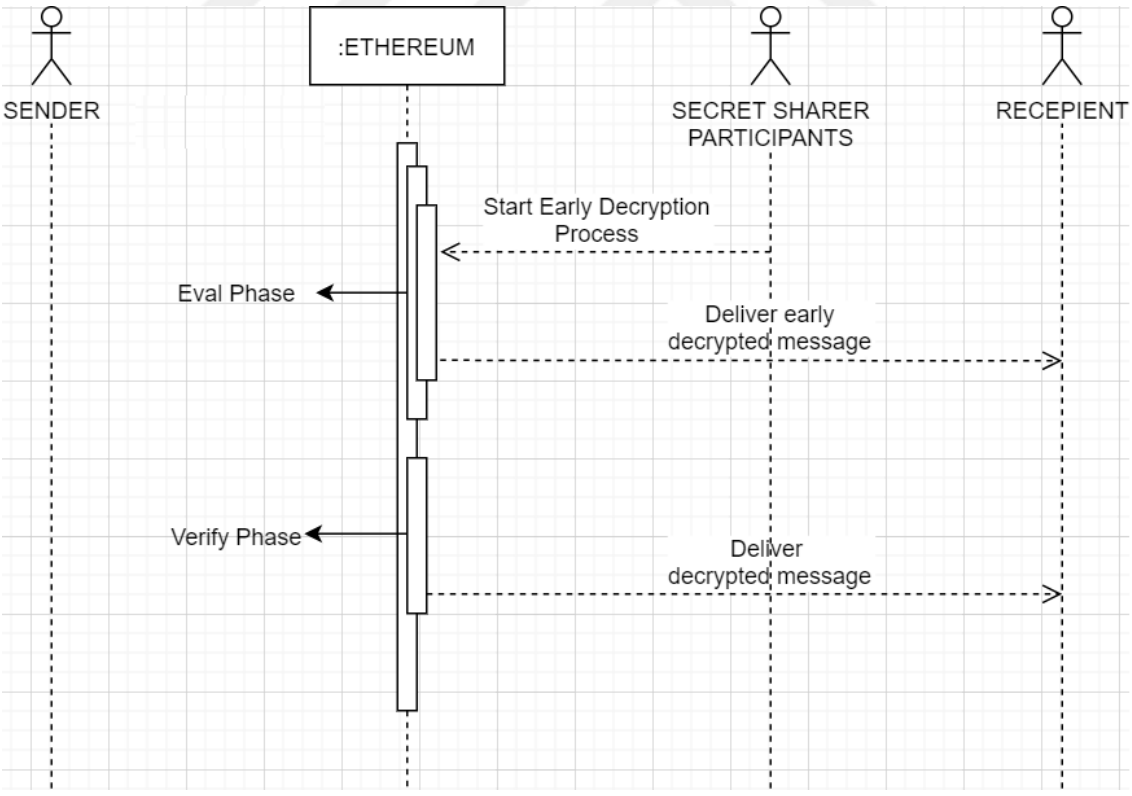


Figure 5.12. Protocol Implementation Sequence Diagram Second Part

In the above sequence diagram, while the evaluation continues if the secret sharer participants decide to decrypt the encrypted message before the specified time. They initiate an early decryption transaction with their private parameter shares using Figure 5.13. When the number of their secret shares reaches the threshold, the encrypted message is revealed, and the miner node is notified to abort the eval phase process.

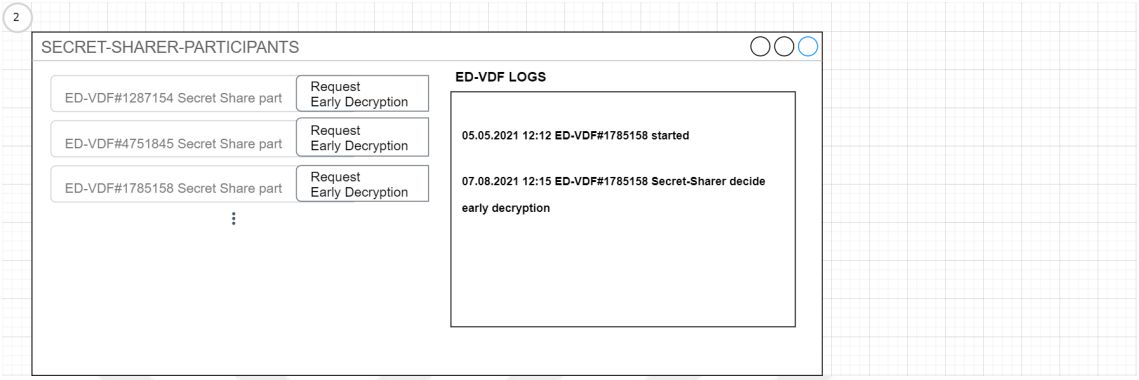


Figure 5.13. UI Mockup design for Secret-Sharer Participants

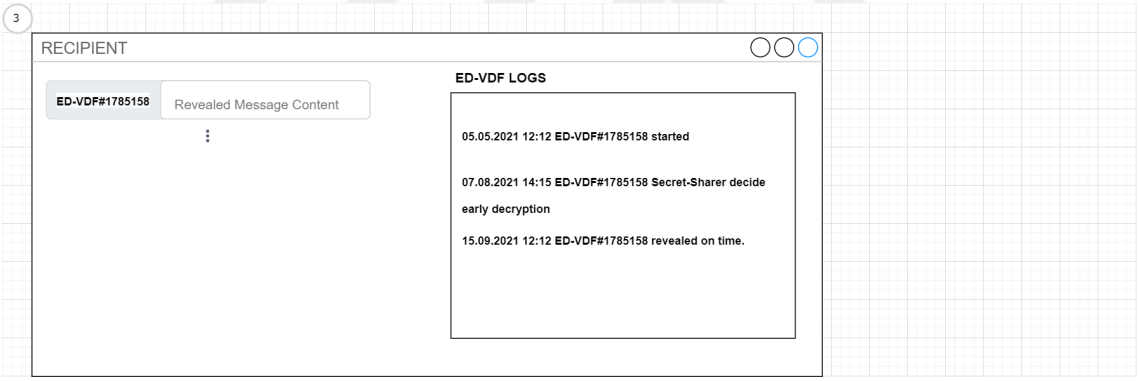


Figure 5.14. UI Mockup design for Recipient

If the delay function resolves at a predetermined time, Verification transactions trigger automatically, and the evaluation phase's output is used to decrypt the encrypted messages. The message appears in the recipient's UI, as shown in Figure 5.14.

We tested the scenarios created for 5.3 Use Cases and 6.1 Attack Analysis sections of the ED-VDF protocol using BDD for validation. We chose BDD style test cases because of the improved test scenario visualisation. Also, provide a better binding between ED-VDF protocol test scenarios and written experimental work codes in a human-readable form.

Remarks:

- The current EVM implementation is not in an ideal condition to process evaluation of the delay function due to limitations of heap and stack aspects. Nevertheless, we implement experimental works to idealise test scenario conditions.
- For the purpose of experimental work, we choose the Successive Squaring algorithm as the algorithm for the delay function. This algorithm satisfies the VDF requirements. However, the current computational performance slightly compromises the delay timing.
- We plan to do experimental works with cVDF [12] to improve the usability of ED-VDF on the current Ethereum network.

5.3. Use Cases

5.3.1. Use Case #1: Testaments

A testament is a formal document describing a testator's desires regarding how their assets should be dispersed following their pass away. Type of testaments are nuncupative (non-culpatory), holographic, self-proved, notarial, will in solemn form. To ensure the legality of testaments, individuals must agree with the statutory trustee. Legal trustees are usually attorneys or law firms

Testaments are bound documents after the testator and the trustee sign the will. However, wills can be disclosed before the decedent's death if certain conditions occur. For example, wills of decedents who lose intellectual capacity may be opened before the decedent's deceased. The following Figure presents the sequence diagram of the Testaments. Testaments should have the following requirements to be considered valid:

1. There must be evidence that the testator actually created the will, which can be proved through the use of witnesses, handwriting experts, or other methods.
2. The beneficiaries should receive their portion of assets exactly as stated in the testament after the testator's pass away.
3. If the testator loses his/her intellectual capacity after the testament is prepared. The beneficiaries can request the testament process before the testator's pass away.

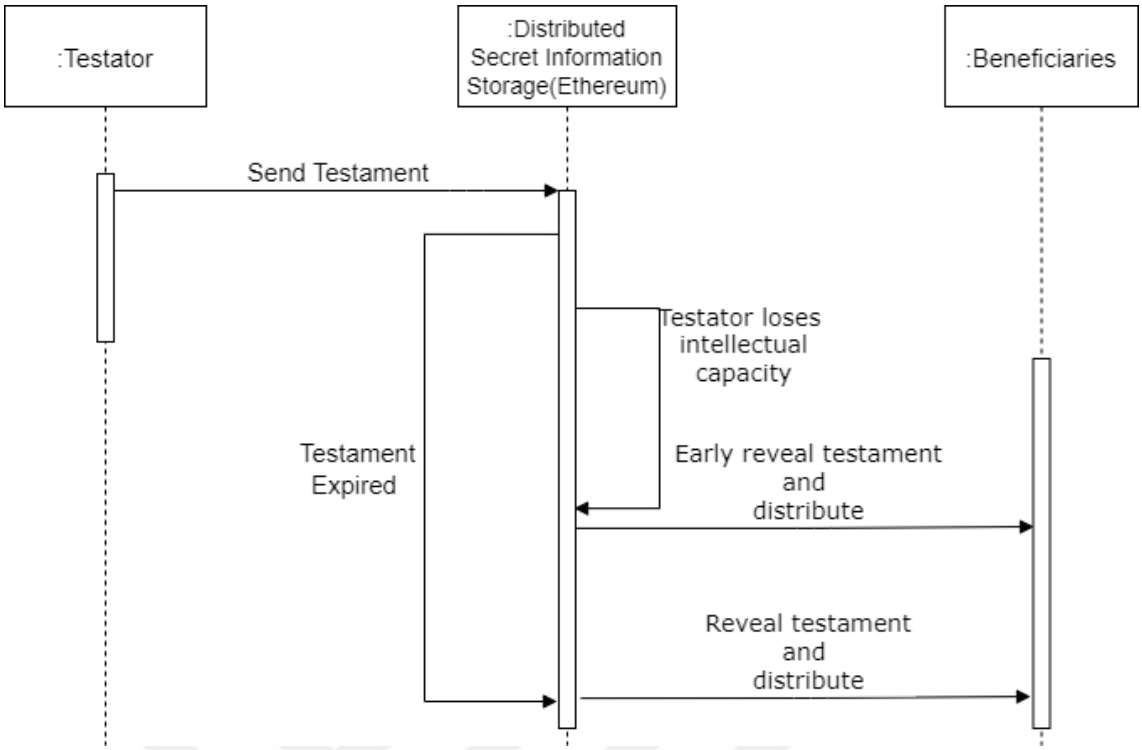


Figure 5.15. Use Case #1: Testaments Sequence Diagram

As seen with the requirements above, testament documents need a third legal authority for the accuracy and formality of the document. At the same time, the identities of the testator and beneficiaries must be proven. With our ED-VDF protocol, the testament process is public and legally verifiable by anyone. Hence we eliminate the third party requirement for keeping testament secure until reveal time. Testament process requirements are ensured with ED-VDF protocol as follows;.

- **First requirement:** Since that person must encrypt the testament with a key in the setup step of the ED-VDF protocol, authentication will be made.
- **Second requirement:** The testator specifies testament's beneficiaries during the setup phase of the ED-VDF protocol that cannot be changed later thanks to Ethereum Blockchain.
- **Third requirement:** ED-VDF protocol early decryption mechanism based on Shamir Secret Share for situations where the testament needs to be opened early during setup (i.e. testator loses his/her intellectual capacity) by the secret decryption key.

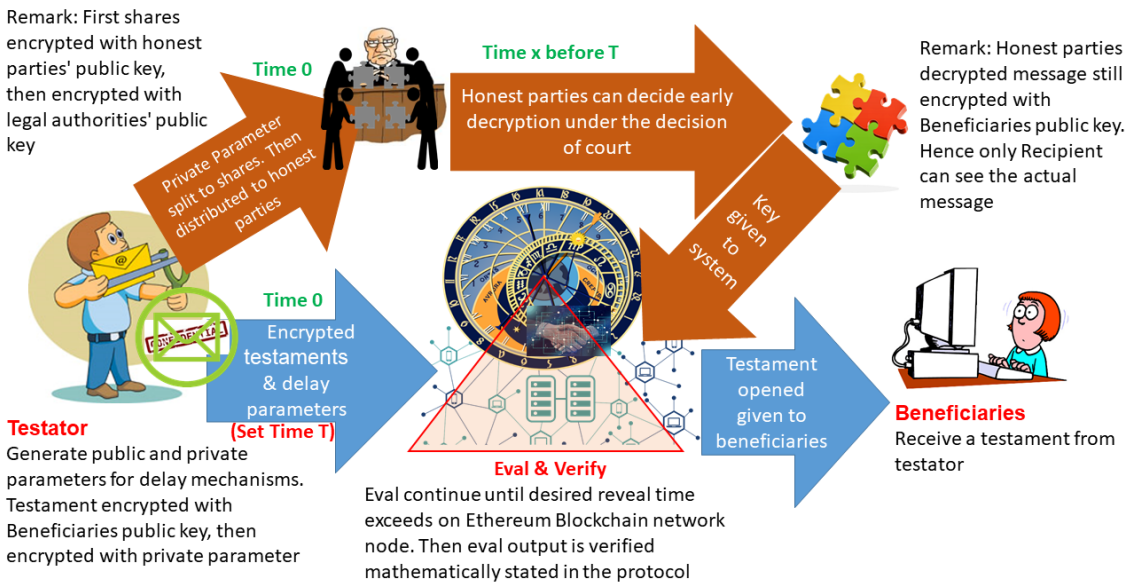


Figure 5.16. ED-VDF Hybrid Protocol for Testaments

5.3.1.1. Testament - ED-VDF Protocol Integration

The above figure demonstrates Testament use case integration to ED-VDF protocol phases. Testament steps in ED-VDF protocol as follows;

1. Testator starts protocol with generating public-private parameters and delay duration for Testament. Then Testator encrypted his/her Testament with the Beneficiarys' public key. Afterwards, Testator once again encrypt Testament using private parameter as a symmetric key.

2. Consecutively private parameter split into shares using Shamir Secret Share algorithm with threshold value for reconstruction. Thereupon shares encrypted with honest parties public key, then encrypted with the court (legal authority) public key.

3. Suppose honest parties decide early decryption of the Testament. Honest parties submit Early Decryption request to the court. If the court allows early decryption, honest parties' private parameter shares are decrypted by the court and sent to the honest parties.

4. If the honest parties not decide to early decryption of Testament. Eval phase continue until desired delay time.

5. Verification starts with reconstructed private parameter from Early Decryption or revealed eval phase output as private parameter. This private parameter is verified mathematically by using parameters stated in section 4.1.5.

6. Verified parameter used as a symmetric decryption key to decrypt Testament second level encryption that previously encrypted first step. Afterwards, the Testament sends to the Beneficiary. Testament still encrypted with Beneficiarys' public key at this step. Then Beneficiary can decrypt the Testament with his/her private key.

The Testament use case involves two-level encryption feature to provide confidentiality of Testament and identification of participants (Testator, Beneficiary, Honest Parties and Legal Authority). Also, ED-VDF protocol has ability to send Testament more than one beneficiary.

- Suppose the beneficiary count is more than one. Part of the testament, which belongs to each Beneficiaries are encrypted with each beneficiary's public key. This ensures that each beneficiary sees only the testament section for him/herself.
- Usage of two level encryption provide;
  - Confidentiality of Testament between Testator and Beneficiaries at first step.
  - In the second step, first-level encryption be done by Honest Parties public key. Thus provide confidentiality and identification of private parameter shares that belongs each Honest Party. Then each share is encrypted with the courts' public key. To give court authorization mechanism for early decryption.
- The integrity of Testament and other parameters of use case provided with Ethereum Blockchain.

Remarks: Ethereum stores every use case step and parameters in blockchain as non-reversible transactions. Ethereum Blockchain is explained detailed in section 4.2.1. Public key cryptography provided with Ethereum Account system. Ethereum Account explained in section 4.2.3.

5.3.2. Use Case #2: Declassification of Secret Information

ED-VDF can be used in various scenarios. Especially when public verifiability of secret information is a necessity.

One of the examples is the declassification of secret information withheld by countries [39]. In the United States, there are laws that require automatic declassification of records after a certain time [40]. For all classified information held by public agencies, the process of declassification begins after 10 years, except under certain conditions. If these conditions are met, some information can remain classified for up to 75 years. Information subject to declassification goes through the legal process.

The Freedom of Information Act [41] gives citizens the right to request disclosure of secret information about public authority records that are kept secret in the United Kingdom.

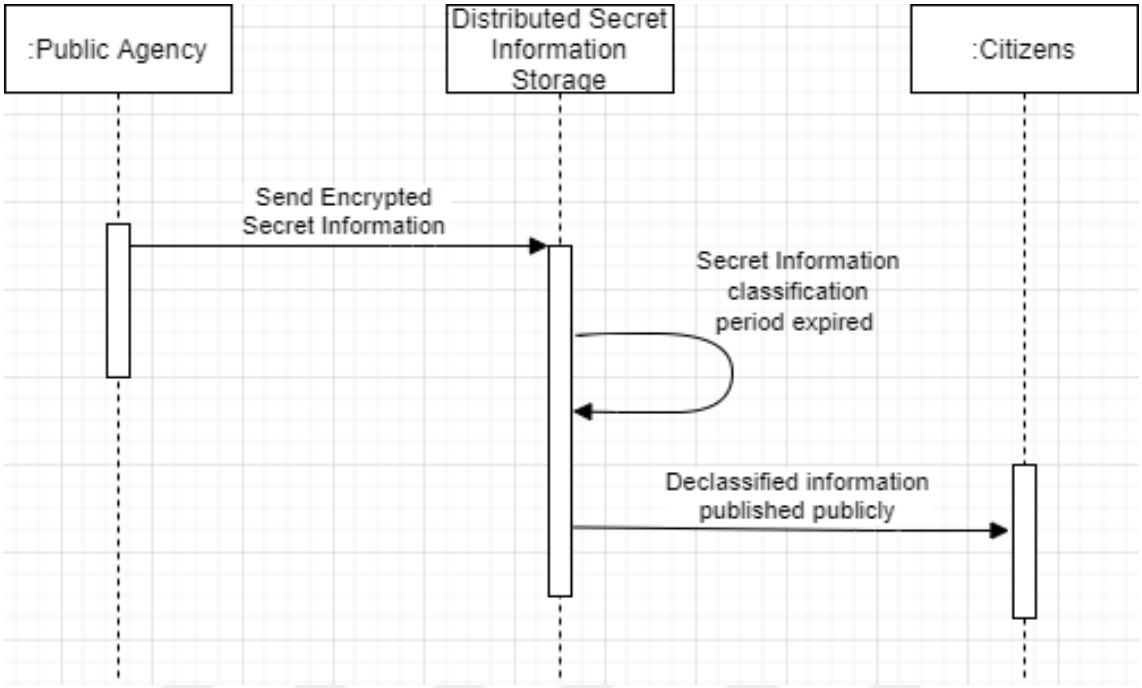


Figure 5.17. Use Case #2 Declassification of secret information Sequence Diagram

Our protocol allows secret information to be disclosed and stored in a publicly auditable system. As shown in Figure 3; Public authorities and citizens ensure that secret information is stored securely and unchanged until the time of publication. Judges can decide to disclose secret information ahead of time in the context of litigation.

CHAPTER 6

ATTACKS ANALYSIS

In this section, we examine attack types for our ED-VDF protocol. Attack types are divided into two main parts: Ethereum Blockchain and Protocol based attacks. Before giving detail for attack analysis, we would like to explain our protocol's attack surfaces briefly.

ED-VDF protocol scheme is implemented in the Ethereum Blockchain network. The Ethereum blockchain network already has countermeasures against its domain attacks. Nevertheless, we will explain these types of attacks in the Ethereum Network against our protocol in Section 6.1.

Protocol-based attack types were simulated and tested using AVISPA [33] protocol validation tool. We will explain in detail these types of attacks in section 6.2.

6.1. Ethereum Blockchain-Based Attacks

Ethereum Smart Contracts' attack surface is primarily circumvented by the distributed Ethereum Virtual Machine [32]. Each protocol phases process by the Ethereum miner nodes, validated by a smart contract function running on the public Ethereum network. These validations minimise the attack surface of the Delay Function processing.

Ethereum Blockchain-based attacks split up 4 scenarios that are ED-VDF protocol phases implemented in the Ethereum Smart Contract system. In order to test the controlled environment, we construct a private Ethereum test network that simulates a real Ethereum network. In the private Ethereum test network we simulate adversaries' attack on a test with interruption and modification trials. In addition we add low level tests to our protocol in Appendix D.

Figure 6.1. ED-VDF Setup Phase

In the above figure, we tested our ED-VDF protocol setup phase. We aim to measure and check whether the expected phase steps are realisation on our Ethereum test network. As we can see, the scenario evaluates successfully.

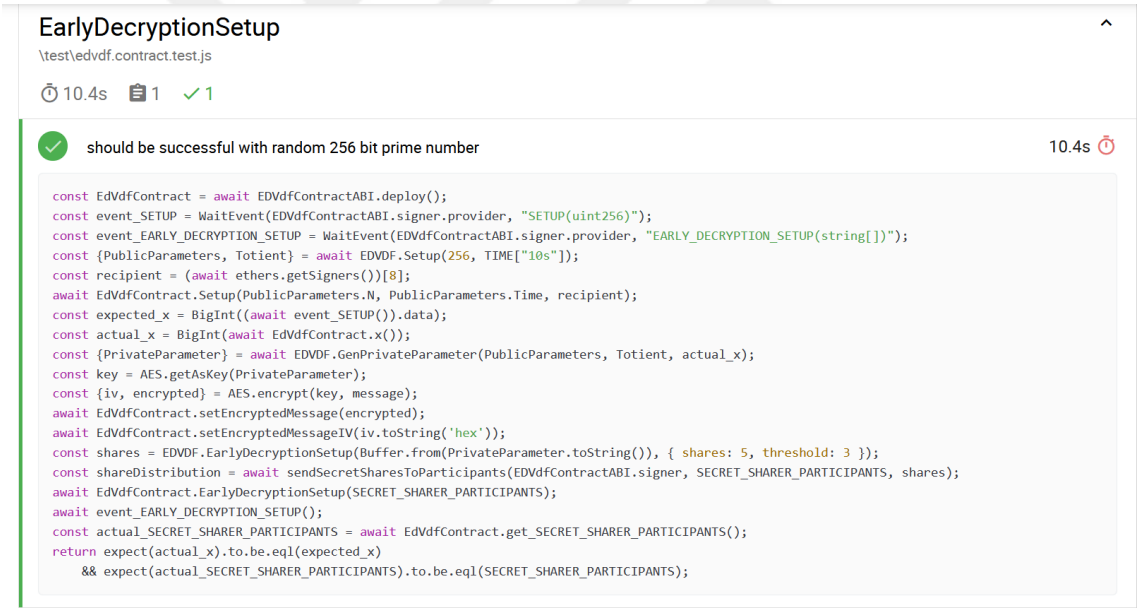


Figure 6.2. ED-VDF Early Decryption Setup Phase

We tested our ED-VDF protocol Early Decryption Setup phase in the above figure. We aim to measure and check whether the expected phase steps are realisation on our Ethereum test network. As we can see, the scenario evaluates successfully.

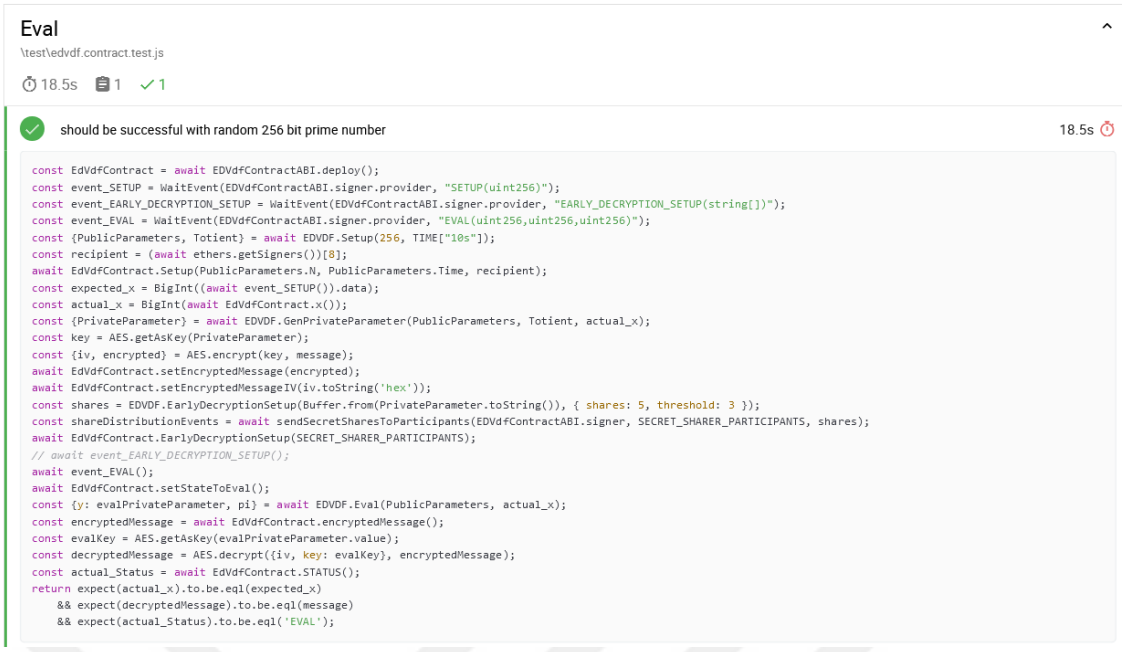


Figure 6.3. ED-VDF Eval Phase

We tested our ED-VDF protocol Eval phase in the above figure. We aim to measure and check whether the expected phase steps are realisation on our Ethereum test network. The scenario evaluates successfully, and the expected delay timing is met.

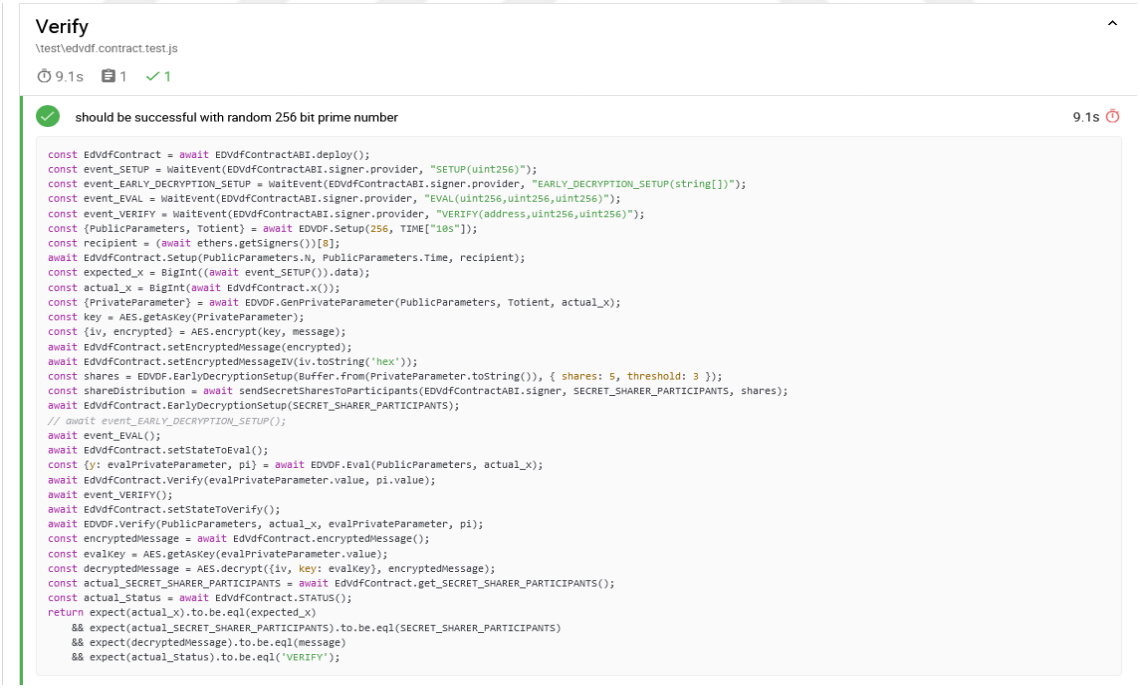


Figure 6.4. ED-VDF Verify Phase

6.2. Protocol Based Attacks

The proposed solution was tested using the security validation tool AVISPA [33] and the visualisation tool SPAN [34]. We simulate On-the-Fly Model Checker (OFMC) [35] and Constraint Logic-based Attack Searcher (CL-AtSe) [37] based attack algorithm against our protocol.

AVISPA stands for Automated Validation of Internet Security Protocols and Applications. AVISPA simulates security protocols and evaluates whether they are secure or insecure. AVISPA validate protocols using automated techniques for implementing tree automaton approximations. The program analyses all backend protocols under the premise of full encryption and exchange of protocol messages over a network controlled by the Dolev-Yao intruder model.

AVISPA uses the high-level protocol specification language (HLPSL) [60] to define and interpolate temporal logic consisting of fundamental roles with transitions between their states. In addition, HLPSL provides a human readable and easy to use programming language.

The runtime of AVISPA allows defining sessions, environment and channel variables for an accurate reenactment of protocol simulation. With those variables, we can simulate various attacks such as passive/active "replay" and "man in the middle" attacks by the adversary.

SPAN was developed to bridge this gap and facilitate formal definition for cryptographic protocols written with AVISPA HLPSL protocol simulations. SPAN uses Message Sequence Charts (MSCs) diagrams to animate AVISPA specifications in SPAN.

AVISPA and SPAN tools are widely used in protocol verification for security algorithms. To evaluate our protocol security, we use the AVISPA tool's two backends: On-the-Fly Model Checker (OFMC) and Constraint Logic-based Attack Searcher (CL-AtSe).

6.2.1. Protocol Reconstruction in AVISPA

We have reconstructed our solution protocol for AVISPA using the HLPSL language. Here are excerpts from that construction. First, we define the roles used in the protocol from our design flowchart described in section 5.1.1 in Figure 5.1.

```
role role_S(S:agent,R:agent,EBC:agent,Kat:symmetric_key,SND,RCV:channel(dy))
played_by S
def=
        local
                State:nat,
                Na,Nb:text,
                Kab:symmetric_key
        init
                State := 0
        transition
                1. State=0 /\ RCV(start) =|>
                   State':=1 /\ Na':=new() /\ Kab':=new() /\ SND({R.Kab'}_Kat) /\ secret(Kab',sec_1,{S,R,EBC})

                2. State=1 /\ RCV({R.Nb'}_Kab) =|> State':=2  /\ SND({Nb'}_Kab)

                   /\ request(S,R,auth_1,Kab)

                   /\ witness(S,R,auth_2,Nb')
end role
```

Figure 6.5. Protocol reconstruction Sender role definition

Role S is the role of the sender who wants to encrypt a message for a certain time. The sender can initiate the protocol described in transition state 0. State 0 begins with the generation of a symmetric key using an encrypted message.

Formerly, an encrypted message is sent over a channel on the Ethereum blockchain network. Then, the sender waits for confirmation of a successfully mined message on Ethereum in state 1. This part of the definition refers start to the eval phase in our protocol design flowchart.

```
role role_EBC(EBC:agent,S:agent,R:agent,Kat,Kbt:symmetric_key,SND,RCV:channel(dy))
played_by EBC
def=
        local
                State:nat,Na:text,Kab:symmetric_key
        init
                State := 0
        transition
                1. State=0 /\ RCV({R.Kab'}_Kat) =|>

                   State':=1 /\ SND({S.Kab'}_Kbt)
end role
```

Figure 6.6. Protocol reconstruction EBC role definition

EBC (Ethereum Blockchain) role is defined as a secure passthrough channel that simulates an ethereum virtual machine. This role receives and sends messages between sender and receiver roles at the transition state description. Part indicates the eval phase in our design flowchart in section 5.1.1 in Figure 5.4.

```
role role_R(R:agent,S:agent,EBC:agent,Kbt:symmetric_key,SND,RCV:channel(dy))
played_by R
def=
        local
                State:nat,Na,Nb:text,Kab:symmetric_key
        init
                State := 0
        transition
                1. State=0 /\ RCV({S.Kab'}_Kbt) =|>

                     witness(R,S,auth_1,Kab')

end role
```

Figure 6.7. Protocol reconstruction Receiver role definition

In role R, the Receiver waits to receive a message from the sender after a certain time. Session role contains a unique statement for AVISPA that describes how to connect roles in the protocol.

```
role session(S:agent,R:agent,EBC:agent,Kat,Kbt:symmetric_key)
def=
        local
                SND3,RCV3,SND2,RCV2,SND1,RCV1:channel(dy)
        composition
                role_S(S,R,EBC,Kat,SND1,RCV1) /\
                role_R(R,S,EBC,Kbt,SND2,RCV2) /\
                role_EBC(EBC,S,R,Kat,Kbt,SND3,RCV3)
end role
```

Figure 6.8. Protocol reconstruction Session definition

Environment role special instructions for AVISPA provide for the interaction (described in section 5.2 at Figure 5.9) between roles and intruders via protocol variables and roles in sessions. This section of protocol simulation refers between eval to end phase in our design flowchart.

Remarks:

- AVISPA does not have the concept of time in protocol definitions. AVISPA reviews the cryptological and network security aspects of the proposed solution.
- Symmetric key variable and trusted agents defined in the environment role in the AVISPA protocol definition are used for simulation purposes only.

```
role environment()
def=
        const
                kat,kbt,kit:symmetric_key,
                alice,bob,trusted:agent, % trusted = ethereum block chain shared enviroment
                sec_1,auth_1,auth_2:protocol_id
        intruder_knowledge = {alice,bob,kit}
        composition

                session(alice,bob,trusted,kat,kbt)

                  /\ session(alice,bob,trusted,kat,kbt)

                  /\ session(i,bob,trusted,kit,kbt)

                  /\ session(alice,i,trusted,kat,kit)
end role
```

Figure 6.9. Protocol reconstruction Environment definition

6.2.2. On-the-Fly Model Checker (OFMC)

Model checking is about developing efficient methods for automatically evaluating the properties of systems represented as transition systems. A model checking programme for security protocols can be created by combining and extending several approaches, as Basin et al. OFMC [35] algorithm shows.

Lazy evaluation decouples the model from the search process and heuristics, allowing the infinite tree to be built dynamically and as needed. It is possible to isolate the model from the search and heuristics by using lazy evaluation. OFMC found 4048 nodes in 16 plies and evaluated their status against adversary intrusion.

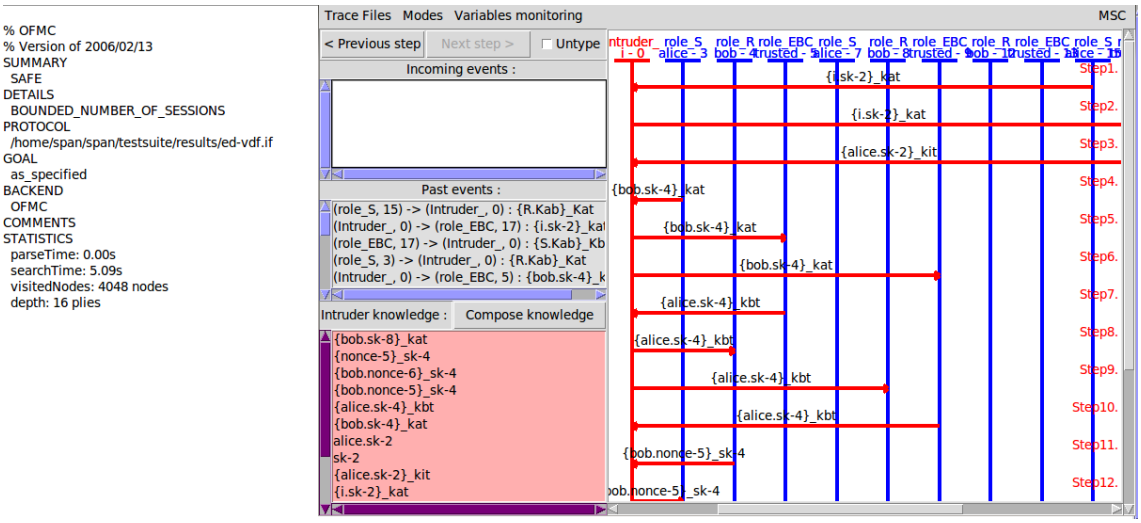


Figure 6.10. OFMC protocol test results

In Figure 6.10, we simulate attack scenarios on SPAN for the OFMC protocol security evaluation algorithm. Simulation results show that our ED-VDF protocol countermeasures all possible attack types that OFMC generates. Therefore ED-VDF protocol was evaluated as “safe” in the OFMC intruder simulation test.

6.2.3. Constraint Logic-based Attack Searcher (CL-AtSe)

Chevalier et al. [36] show security of cryptographic protocols can be evaluated using the CL-Atse. CL-Atse a robust and adaptable automated analysis algorithm. CL-Atse use for rewriting and constraint solving techniques. CL-Atse models checks all possible states of the actors and determines whether or not an attack is possible against the Dolev-Yao intruder [37]. This is done using a protocol provided as a rule set (IF format, created by the AVISPA compiler).

Thanks to a comprehensive modular unification method, any state-based security attribute can be simulated (secrecy, authentication, fairness), as well as the algebraic operators’ xor, or, and exponentiation. Several other relevant constraints can be analysed, such as type constraints, inequalities, and shared knowledge sets (including set operations such as removals and negative tests).

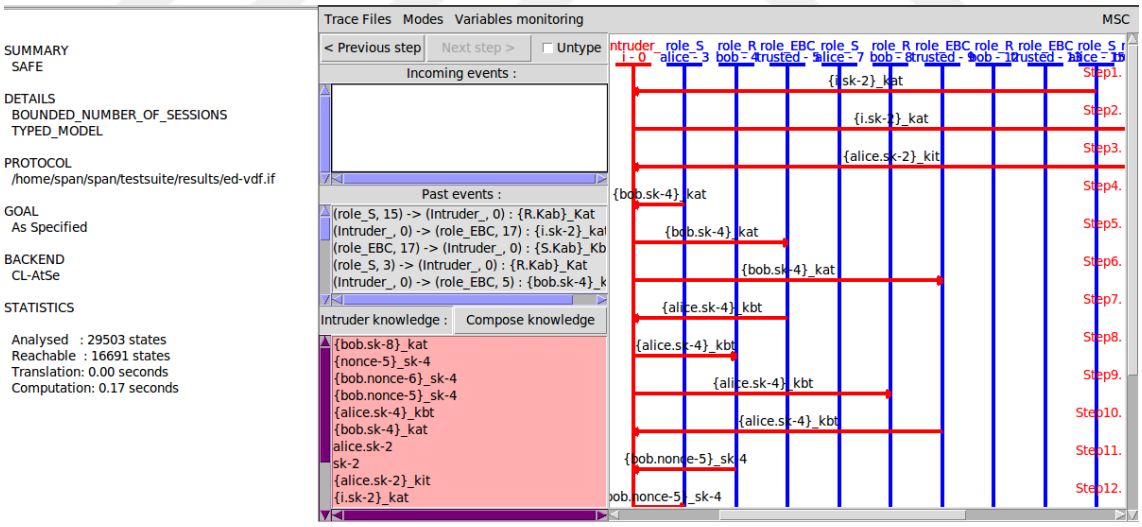


Figure 6.11. CL-AtSe protocol test results

CL-AtSe found 29503 states of potential attack vectors from our protocol. However, the intruder only reaches 16691 states with its public knowledge. In Figure 6.11 we simulate attack scenarios on SPAN for the CL-AtSe algorithm to evaluate protocol security. The simulation results show that our ED-VDF protocol is immune against all possible attack types generated by CL-AtSe. Therefore, CL-AtSe concludes the evaluation of our protocol as secure.

CHAPTER 7

FUTURE WORK

Currently, we plan to improve and explore two areas of our protocol, namely enabling a cross transactional system between different blockchain networks using Cosmos [42] and exploring the impact of quantum computing on our protocol.

Our protocol is based on the Ethereum blockchain network. However, we would like to increase our protocol's usability and transaction interchangeability with other blockchain networks. This is not possible due to the limitations of the Ethereum implementation. In our research, we came across Cosmos. Cosmos provides a way to integrate between different blockchain systems [44] with its Tendermint BFT engine [49]. We will provide mechanisms for using DApps in other blockchain networks [45] with using Cosmos.

Quantum computing is an emerging field of computer science. There are concerns about quantum computing impact on cryptographic security protocols [43]. Since the Ethereum blockchain relies heavily on RSA-based cryptographic functions, these concerns interest our ED-VDF protocol. In our search, we found a candidate that replaces RSA-based public cryptography with resistant post-quantum computing such as NTRU [52]. We would like to improve our protocol based on these implications in the future.

CHAPTER 8

CONCLUSION

In this study, we have investigated Verifiable Delay Functions (VDF) from a computer science perspective. We have compared different approaches to VDF, from successive squaring to witness encryption. Then we declare our contribution to the VDF scheme in four criteria in section 4.1. Then according to aimed contributions; a new VDF and Time-Lock hybrid protocol proposed, design and verified for different attack types; which guarantees the common and main requirements of VDF and Time-Lock puzzle protocols.

Proposed protocol integrates a secret-share algorithm to construct a new VDF protocol scheme that provides an early decryption mechanism. Therefore, it is named as Early Decryptable Verifiable Delay Functions (ED-VDF). Current VDF solutions do not have a controlled early decryption feature for time locking mechanisms in both protocol schemes. This facility is a requirement for social life; for example, court decisions may require encrypted, stored documents to be opened before the intended expiration date of secrecy. Testament process without a trusted third party and an early decryption functionality that provides early testament reveal when Testator loses his mental capacity. ED-VDF could be used for these types of scenarios.

ED-VDF uses Ethereum Blockchain and Smart Contract infrastructure to fulfill VDF and Time-Lock protocol requirements. Moreover, we also use this infrastructure to implement our contribution to VDF protocol named ED-VDF. We also extend our protocol capabilities and real-world application areas with Ethereum Blockchain and Smart Contract infrastructure. In addition, we mention future extensibility opportunities of our ED-VDF protocol in section 7. Hence, we underlined that our proposed protocol design will have new gains from the evolution of Ethereum Blockchain and Smart Contract technologies.

ED-VDF solution has also work with any of the defined VDF or Time-Lock algorithms, depending on which one is chosen. In our experiments, we use a Successive Squaring algorithm. However, with the modularity and usage of the Ethereum Smart Contract structure, the designed solution can use alternative delay(Time-Lock puzzle) algorithms.

We explained the design details of our protocol. Then, we detailed the implementation of the protocol and explained how we solved the early decryption problem using the Ethereum Smart Contract and Shamir's Secret Key Sharing algorithm.

Finally, we analysed and evaluated our protocol attack scenarios in the Ethereum Blockchain network. Then, we simulated our protocol using the security protocol validation tool AVISPA and validated that our protocol is safe against all known network and communication attacks according to the Dolev-Yao intruder model.

REFERENCES

[1] R. L. Rivest, A. Shamir and D. A. Wagner. 1996. Time-lock Puzzles and Timed-release Crypto. Technical Report. Massachusetts Institute of Technology, USA. https://people.csail.mit.edu/rivest/pubs/RSW96.pdf.

[2] R. L. Rivest. 1994. The RC5 Encryption Algorithm. Technical Report. Massachusetts Institute of Technology, USA. https://link.springer.com/content/pdf/10.1007/3-540-60590-8_7.pdf.

[3] K. Pietrzak.Simple Verifiable Delay Functions. Cryptology ePrint Archive, 2018/627, 2018. https://eprint.iacr.org/2018/627.pdf.

[4] B. Wesolowski. Efficient verifiable delay functions. Cryptology ePrint Archive, 2018/623, 2018. https://eprint.iacr.org/2018/623.pdf.

[5] D. Boneh, B. Bunz, and B. Fisch. A Survey of Two Verifiable Delay Functions. Cryptology ePrint Archive, 2018/712, 2018. https://eprint.iacr.org/2018/712.pdf.

[6] K. Chalkias and G. Stephanides (2006) Timed Release Cryptography from Bilinear Pairings Using Hash Chains. In: Leitold H., Markatos E.P. (eds) Communications and Multimedia Security. CMS 2006. Lecture Notes in Computer Science, vol 4237. Springer, Berlin, Heidelberg. https://doi.org/10.1007/11909033_12.

[7] J. Ning, H. Dang, R. Hou and E. C. Chang. Keeping Time-Release Secrets through Smart Contracts. Cryptology ePrint Archive, Report 2018/1166, 2018. https://eprint.iacr.org/2018/1166.pdf.

[8] M. Mahmoody, T. Moran and S. Vadhan. Time-Lock Puzzles in the Random Oracle Model. Cryptology ePrint Archive, 2011. https://www.iacr.org/archive/crypto2011/68410039/68410039.pdf.

[9] Liu, J., Jager, T., Kakvi, S.A. et al. How to build time-lock encryption. Des. Codes Cryptogr. 86, 2549–2586 (2018). https://doi.org/10.1007/s10623-018-0461-x.

[10] G. Malavolta and S. A. K Thyagarajan. Homomorphic Time-Lock Puzzles and Applications. Cryptology ePrint Archive, Report 2019/635, 2019 https://eprint.iacr.org/2019/635.pdf.

[11] F. Garcia, M. Ryan and J. Liu. Time-release Protocol from Bitcoin and Witness Encryption for SAT. Cryptology ePrint Archive, 2015. https://dx.doi.org/10.1007/s10623-018-0461-x.

[12] R. Wei and J. Long. Nakamoto Consensus with Verifiable Delay Puzzle. Arxiv ePrint Archive, 2019. https://arxiv.org/pdf/1908.06394.pdf.

[13] C. Freitag, I. Komargodski, N. Ephraim and R. Pass. Continuous Verifiable Delay Functions. Cryptology ePrint Archive, 2019/619, 2019. https://eprint.iacr.org/2019/619.pdf.

[14] A. Jain, B. Waters, N. Bitansky, O. Paneth and S. Goldwasser. Time-Lock Puzzles from Randomized Encodings. Cryptology ePrint Archive, 2015/514, 2015. https://eprint.iacr.org/2015/514.pdf.

[15] P. Paillier. Public-Key Cryptosystems Based on Composite Degree Residuosity Classes. Springer Archive, 1999. https://link.springer.com/content/pdf/10.1007/3-540-48910-X_16.pdf.

[16] "ETHEREUM: A SECURE DECENTRALISED GENERALISED TRANSACTION LEDGER", Jan. 05, 2022. Accessed on: Feb. 26, 2022. [Online]. https://ethereum.github.io/yellowpaper/paper.pdf

[17] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," Tech. Rep., 2008

[18] A. Miller, Y. Xia, K. Croman, E. Shi, and D. Song, "The honey badger of BFT protocols," in Proc. 2016 ACM SIGSAC Conf. Comput. Commun. Secur., 2016, pp. 31–42.

[19] J. R. Douceur, "The Sybil attack," in Proc. Int. Workshop Peer-Peer Syst. Berlin, Germany: Springer, 2002, pp. 251–260.

[20] B. N. Levine, C. Shields, and N. B. Margolin, "A survey of solutions to the Sybil attack," Univ. Massachusetts Amherst, Amherst, MA, USA, Tech. Rep., 2006, vol. 7, p. 224.

[21] G. O. Karame, E. Androulaki, and S. Capkun. (2012). Two Bitcoins at the Price of One Double Spending Attacks on Fast Payments in Bitcoin, Jan. 07, 2022. Accessed on: Feb. 25, 2022. [Online]. Available: http://eprint.iacr.org/2012/248.pdf

[22] A. Kiayias, A. Russell, B. David, and R. Oliynykov, "Ouroboros: A provably secure proof-of-stake blockchain protocol," in Proc. Annu. Int. Cryptol. Conf. Cham, Switzerland: Springer, 2017, pp. 357–388.

[23] J. Kwon, "TenderMint: Consensus without Mining," Draft v.0.6, Tech. Rep., 2014.

[24] I. Eyal and E. G. Sirer, "Majority is not enough: Bitcoin mining is vulnerable," Commun. ACM, vol. 61, no. 7, pp. 95–102, 2018.

[25] V. Buterin. (2014). Slasher: A Punitive Proof-of-Stake Algorithm. [Online]. Available: https://blog.ethereum.org/2014/01/15/slasher-a-punitive-proof- of-stake-algorithm

[26] V. Zamfir. (2015). Introducing Casper 'The Friendly Ghost', Jan. 08, 2022. Accessed on: Feb. 23, 2022. [Online]. Available: https://blog.ethereum.org/2015/08/01/introducing-casper-friendly-ghost

[27] L. Xu et al., "Enabling the sharing economy: Privacy respecting contract based on public blockchain," in Proc. ACM Workshop Blockchain, Cryptocurrencies Contracts, 2017, pp. 15–21.

[28] V. Buterin and V. Griffith. (2017). 'Casper the friendly finality gadget.', Jan. 10, 2022. Accessed on: Feb. 11, 2022. [Online]. Available: https://arxiv.org/abs/1710.09437

[29] "INTRODUCTION TO DAPPS", Jan. 11, 2022. Accessed on: Feb. 15, 2021. [Online]. https://ethereum.org/en/developers/docs/dapps/

[30] "BENEFITS OF DAPP DEVELOPMENT", Jan. 11, 2022. Accessed on: Feb. 15, 2021. [Online]. https://ethereum.org/en/developers/docs/dapps/#benefits-of-dapp-development

[31] "Solidity", Jan. 12, 2022. Accessed on: Feb. 16, 2021. [Online]. https://docs.soliditylang.org/en/latest/

[32] "ETHEREUM VIRTUAL MACHINE (EVM)", October. 26, 2021. Accessed on: Feb. 17, 2021. [Online]. https://ethereum.org/en/developers/docs/evm/

[33] "Avispa", April. 14, 2021. Accessed on: May. 23, 2021. [Online]. https://people.irisa.fr/Thomas.Genet/span/

[34] "Span", September. 24, 2020. Accessed on: May. 25, 2021. [Online]. https://people.irisa.fr/Thomas.Genet/span/

[35] Basin, D., Mödersheim, S. & Viganò, L. OFMC: A symbolic model checker for security protocols. Int J Inf Secur 4, 181–208 (2005). https://doi.org/10.1007/s10207-004-0055-7

[36] Chevalier, Yannick & Vigneron, Laurent. (2002). Automated Unbounded Verification of Security Protocols. LNCS. 2404. 324-337. 10.1007/3-540-45657-0_24.

[37] D. Dolev and A. Yao, "On the security of public-key protocols," in IEEE Transactions on Information Theory, vol. 29, no. 2, pp. 198-208, March 1983, doi: 10.1109/TIT.1983.1056650

[38] Adi Shamir. 1979. How to share a secret. Commun. ACM 22, 11 (Nov. 1979), 612–613. DOI:https://doi.org/10.1145/359168.359176

[39] “Declassification”, Jan. 25, 2021. Accessed on: Jan. 27, 2021. [Online]. https://www.justice.gov/archives/open/declassification/declassification-faq

[40] “Classified information in the United States”, Jan. 21, 2021. Accessed on: Jan. 25, 2021. [Online]. https://www.justice.gov/archives/open/declassification

[41] “What is the Freedom of Information Act”, Jan. 18, 2021. Accessed on: Jan. 25, 2021. [Online]. Available: https://ico.org.uk/for-organisations/guide-to-freedom-of-information/what-is-the-foi-act/

[42] “What is Cosmos?", Sep. 18, 2021. Accessed on: Sep. 20, 2021. [Online]. https://v1.cosmos.network/intro

[43] Moody, Dustin & Chen, Lily & Jordan, Stephen & Liu, Yi-Kai Smith, Daniel & Perlner, Ray & Peralta, Renė. (2016). NIST Report on Post-Quantum Cryptography. 10.6028/NIST.IR.8105.

[44] “INTER-BLOCKCHAIN COMMUNICATION PROTOCOL”, August. 11, 2021. Accessed on: September. 10, 2021. [Online]. https://ibcprotocol.org/

[45] “Explore Cosmos Network”, September. 7, 2021. Accessed on: September. 10, 2021. [Online]. https://v1.cosmos.network/ecosystem/apps

[46] “Will and testament”, Mar. 5, 2021. Accessed on: Mar. 7, 2021. [Online]. https://en.wikisource.org/wiki/Page%3AEB1911_-_Volume_28.djvu/674

[47] “The Church-Turing Thesis”, November. 10, 2017. Accessed on: Jun. 15, 2021. [Online]. https://plato.stanford.edu/entries/church-turing/

[48] David Derler and Daniel Slamanig. 2018. Practical witness encryption for algebraic languages or how to encrypt under Groth—Sahai proofs. Des. Codes Cryptography 86, 11 (November 2018), 2525–2547. DOI:https://doi.org/10.1007/s10623-018-0460-y

[49] “Tendermint”, June. 20, 2021. Accessed on: July. 15, 2021. [Online]. https://github.com/tendermint/tendermint

[50] “What is blockchain technology?”, April. 4, 2021. Accessed on: May. 5, 2021. [Online]. https://www.ibm.com/topics/what-is-blockchain

[51] “INTRODUCTION TO SMART CONTRACTS”, September. 30, 2021. Accessed on: October. 5, 2021. [Online]. https://ethereum.org/en/developers/docs/smart-contracts/

[52] “NTRU (merger of NTRUEncrypt and NTRU-HRSS-KEM)”, September. 30, 2021. Accessed on: October. 5, 2021. [Online]. https://tsapps.nist.gov/publication/get_pdf.cfm?pub_id=927303

[53] “Contract ABI Specification”, May. 17, 2021. Accessed on: June. 15, 2021. [Online]. https://docs.soliditylang.org/en/latest/abi-spec.html

[54] “Ethereum development environment for professionals”, April. 16, 2021. Accessed on: April. 23, 2021. [Online]. https://hardhat.org/

[55] “NODES AND CLIENTS”, April. 10, 2021. Accessed on: April. 15, 2021. [Online]. https://ethereum.org/en/developers/docs/nodes-and-clients/

[56] “TRANSACTIONS”, April. 10, 2021. Accessed on: April. 15, 2021. [Online]. https://ethereum.org/en/developers/docs/transactions/

[57] “ETHEREUM ACCOUNTS”, April. 11, 2021. Accessed on: April. 15, 2021. [Online]. https://ethereum.org/en/developers/docs/accounts/

[58] “Randomized Encodings Practical Issues”, Accessed on: May. 27, 2021. [Online]. http://cryptowiki.net/index.php?title=Succinct_Randomized_Encodings

[59] Adi Shamir. 1979. How to share a secret. Commun. ACM 22, 11 (Nov. 1979), 612–613. https://doi.org/10.1145/359168.359176

[60] von Oheimb, David. (2005). The high-level protocol specification language HLPSL developed in the EU project AVISPA. Proceedings of APPSEM 2005 Workshop.

[61] “Traplottery”, June. 15, 2021. Accessed on: August. 05, 2021. [Online]. https://github.com/mabbamOG/traplottery

[62] “Behavior-driven development”, June. 17, 2021. Accessed on: July. 12, 2021. [Online]. https://dannorth.net/introducing-bdd/

[63] “The history of Ethereum”, July. 03, 2021. Accessed on: September. 12, 2021. [Online]. https://ethereum.org/en/history/

[64] “Timed-Release Crypto”, Feb. 10, 1993. Accessed on: Jan. 27, 2021. [Online]. http://cypherpunks.venona.com/date/1993/02/msg00129.html

[65] Chvojka, P., Jager, T., Slamanig, D., Striecks, C.: Generic constructions of incremental and homomorphic timed-release encryption. IACR Cryptol. ePrint Arch.

[66] Decker, Christian & Wattenhofer, Roger. (2013). Information propagation in the Bitcoin network. 110. 10.1109/P2P.2013.6688704.

[67] “Keccak”, July. 03, 2021. Accessed on: September. 12, 2021. [Online]. https://keccak.team/keccak.html.

[68] Fiat A., Shamir A. (1987) How To Prove Yourself: Practical Solutions to Identification and Signature Problems. In: Odlyzko A.M. (eds) Advances in Cryptology-CRYPTO 86. CRYPTO 1986. Lecture Notes in Computer Science, vol 263. Springer, Berlin, Heidelberg. https://doi.org/10.1007/3540477217_12.

[69] Johnson, D., Menezes, A. & Vanstone, S. The Elliptic Curve Digital Signature Algorithm (ECDSA). IJIS 1, 36–63 (2001). https://doi.org/10.1007/s102070100002.

[70] “Solidity”, December. 20, 2020. Accessed on: January. 02, 2021. [Online]. https://docs.soliditylang.org/en/latest/.

[71] “Early Decryptable Verifiable Delay Functions”, January. 05, 2021. Accessed on: February. 11, 2022. [Online]. https://github.com/Ogulcan-Ozdemir/ed-vdf.

APPENDIX A

BLOCK DIFFICULTY CALCULATION

$$blockTime = currentBlockTimestamp — parentBlockTimestamp \quad \text{(A.1)}$$

$$current\_block\_difficulty = parent\_block\_difficulty \quad \text{(A.2)}$$

$$+ (parent\_block\_difficulty // 2048) * max(1 — (block\_time // 10), -99) \quad \text{(A.3)}$$

$$+ int(2**((current\_block\_number // 100000) — 2)) \quad \text{(A.4)}$$

The complexity level is computed as follows, where // represents integer division and 2** represents over of power two. The int function produces the biggest integer that is smaller than or equal to a specified value.

The first portion calculates how far the block time differs from the predicted block time (10 to 19 seconds). It will calculate an X factor according on how long it takes to mine current block.

This factor will be positive if current block is mined in fewer than 10 seconds, increasing the complexity. The complexity remains constant if block time is around 10 and 19 seconds. When the block duration is higher than or equal to 20 seconds, it will result in a negative value, and difficulty will be reduced.

APPENDIX B

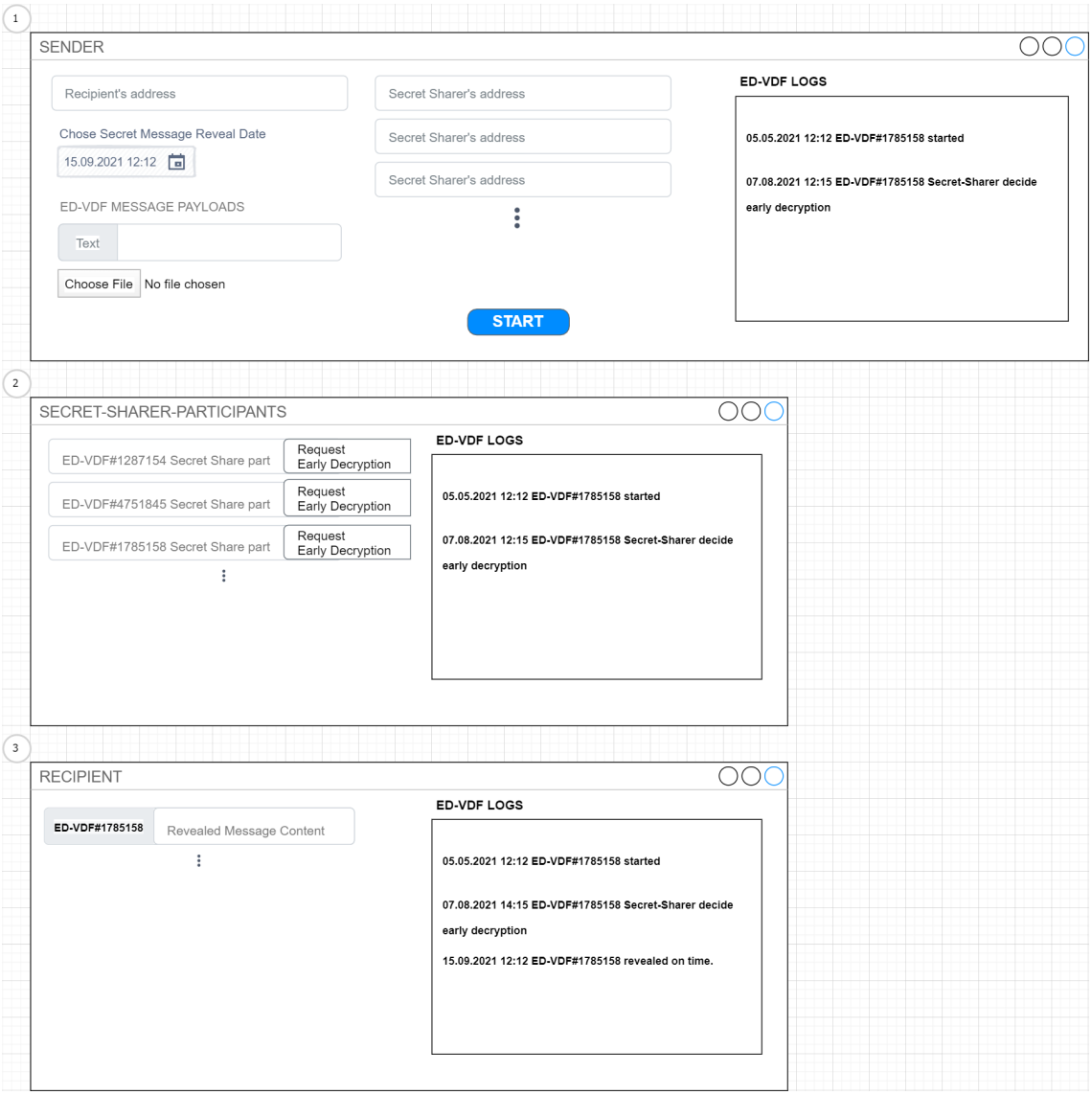
EXPERIMENTAL WORKS CODE SNIPPETS FOR SMART CONTRACTS

```
pragma solidity ^0.6.1;
pragma experimental ABIEncoderV2;
// SPDX-License-Identifier: UNLICENSED

contract ED_VDF {

    string public name = 'ED_VDF';
    string public STATUS = 'NO_INIT';

    address payable public SENDER;
    string[] SECRET_SHARER_PARTICIPANTS;
    string public RECIPIENT;

    uint256 public N;
    uint256 public Time;
    uint256 public x;
    string public encryptedMessage;

    uint256 public INIT_TIMESTAMP = 0 seconds;

    event SETUP(uint256 _x);
    event EARLY_DECRYPTION_SETUP(string[] participants);
    event EVAL(uint256 N, uint256 T, uint256 _x);
    event VERIFY(address from, uint256 _EvalPrivateParameter, uint256 _EvalProof);

    constructor() public {
        SENDER = msg.sender;
        STATUS = 'INIT';
    }

    function Setup(uint256 _N, uint256 _T, string memory _RECIPIENT) public
    {
        require(SENDER == msg.sender);
        require(keccak256(abi.encodePacked(STATUS)) == keccak256('INIT'), 'ED_VDF should be at INIT status before Setup phase');

        INIT_TIMESTAMP = block.timestamp;

        STATUS = 'SETUP';
        Time = _T;
        N = _N;
        RECIPIENT = _RECIPIENT;

        x = uint256(blockhash(block.number - 1)) % N;

        emit SETUP(x);
```

```
function EarlyDecryptionSetup(string[] memory _SECRET_SHARER_PARTICIPANTS, string memory _encryptedMessage) public
{
    require(SENDER == msg.sender);
    require(keccak256(abi.encodePacked(STATUS)) == keccak256('SETUP'), 'ED_VDF should be at SETUP status before EarlyDecryptionS
    encryptedMessage = _encryptedMessage;
    SECRET_SHARER_PARTICIPANTS = _SECRET_SHARER_PARTICIPANTS;
    STATUS = 'EARLY_DECRYPTION_SETUP';
    emit EARLY_DECRYPTION_SETUP(SECRET_SHARER_PARTICIPANTS);
    Eval();
}

function Eval() private
{
    require(keccak256(abi.encodePacked(STATUS)) == keccak256('EARLY_DECRYPTION_SETUP'), 'ED_VDF should be at EARLY_DECRYPTION_SE
    STATUS = 'EVAL';
    emit EVAL(N, Time, x);
}

function Verify(uint256 EvalPrivateParameter, uint256 EvalProof) public
{
    require(keccak256(abi.encodePacked(STATUS)) == keccak256('EVAL'), 'ED_VDF should be at EVAL status before Verify phase');
    require(EvalPrivateParameter <= N &&  EvalProof <= N);
    STATUS = 'VERIFY';
    emit VERIFY(msg.sender, EvalPrivateParameter, EvalProof);
}

function get_SECRET_SHARER_PARTICIPANTS() public view returns (string[] memory) {
    return SECRET_SHARER_PARTICIPANTS;
}

function HPrime(uint256 _N, uint256 _T, uint256 _X, uint256 _Y)  view private returns (uint256)
{
    uint256 p = uint256(keccak256(abi.encodePacked(_N,_T,_X,_Y))) >> 3;
    if (p%2 == 0) p = p+1;
    while (true) {
        if (check_probable_prime(p)) {
            return p;
        }
        p = p+2;
    }
}
```

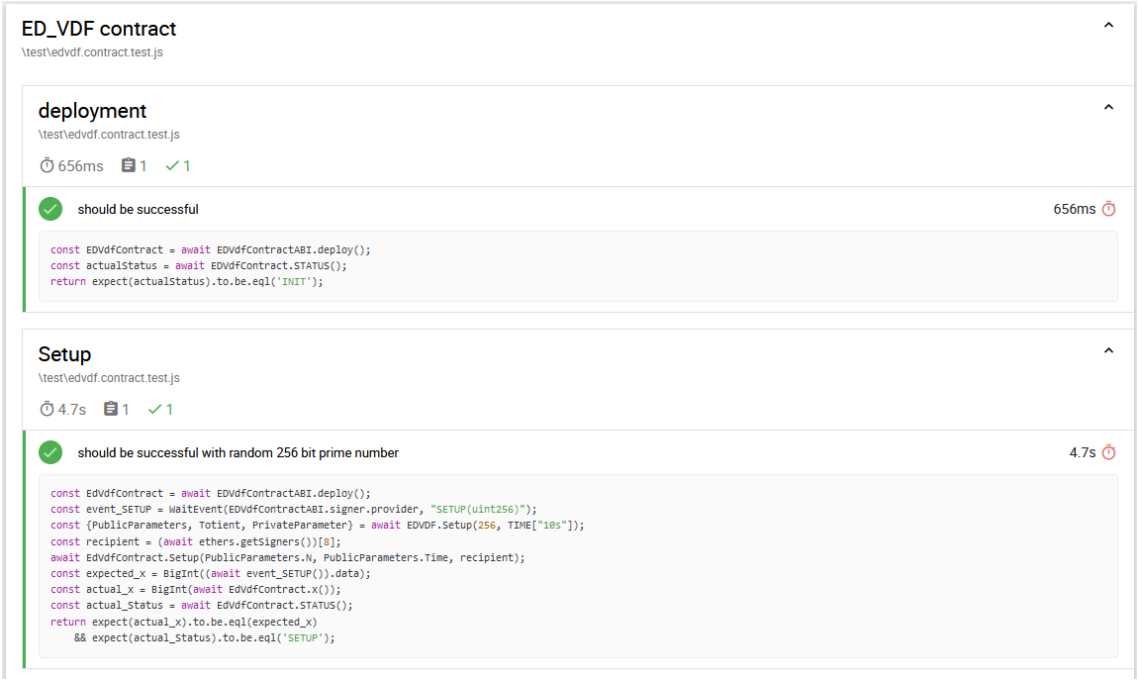
APPENDIX C

UI MOCKUP DESIGN FOR PROTOCOL IMPLEMENTATION



APPENDIX D

VERIFICATION OF EXPERIMENTAL WORKS SCENARIO



EarlyDecryptionSetup

\test\edvdf.contract.test.js

10.3s 1 ✓1

should be successful with random 256 bit prime number 10.3s

```
const EdVdfContract = await EDVdfContractABI.deploy();
const event_SETUP = WaitEvent(EDVdfContractABI.signer.provider, "SETUP(uint256)");
const event_EARLY_DECRYPTION_SETUP = WaitEvent(EDVdfContractABI.signer.provider, "EARLY_DECRYPTION_SETUP(string[])");
const {PublicParameters, Totient} = await EDVDF.Setup(256, TIME["10s"]);
const recipient = (await ethers.getSigners())[8];
await EdVdfContract.Setup(PublicParameters.N, PublicParameters.Time, recipient);
const expected_x = BigInt((await event_SETUP()).data);
const actual_x = BigInt(await EdVdfContract.x());
const {PrivateParameter} = await EDVDF.GenPrivateParameter(PublicParameters, Totient, actual_x);
const shares = EDVDF.EarlyDecryptionSetup(Buffer.from(PrivateParameter.toString()), { shares: 5, threshold: 3 });
const shareDistribution = await sendSecretSharesToParticipants(EDVdfContractABI.signer, SECRET_SHARER_PARTICIPANTS, shares);
const key = AES.getAsKey(PrivateParameter);
const {iv, encrypted} = AES.encrypt(key, message);
await EdVdfContract.EarlyDecryptionSetup(SECRET_SHARER_PARTICIPANTS, encrypted);
await event_EARLY_DECRYPTION_SETUP();
const actual_SECRET_SHARER_PARTICIPANTS = await EdVdfContract.get_SECRET_SHARER_PARTICIPANTS();
return expect(actual_x).to.be.eql(expected_x)
    && expect(actual_SECRET_SHARER_PARTICIPANTS).to.be.eql(SECRET_SHARER_PARTICIPANTS);
```

Eval

\test\edvdf.contract.test.js

21.4s 1 ✓1

should be successful with random 256 bit prime number 21.4s

```
const EdVdfContract = await EDVdfContractABI.deploy();
const event_SETUP = WaitEvent(EDVdfContractABI.signer.provider, "SETUP(uint256)");
const event_EARLY_DECRYPTION_SETUP = WaitEvent(EDVdfContractABI.signer.provider, "EARLY_DECRYPTION_SETUP(string[])");
const event_EVAL = WaitEvent(EDVdfContractABI.signer.provider, "EVAL(uint256,uint256,uint256)");
const {PublicParameters, Totient} = await EDVDF.Setup(256, TIME["10s"]);
const recipient = (await ethers.getSigners())[8];
await EdVdfContract.Setup(PublicParameters.N, PublicParameters.Time, recipient);
const expected_x = BigInt((await event_SETUP()).data);
const actual_x = BigInt(await EdVdfContract.x());
const {PrivateParameter} = await EDVDF.GenPrivateParameter(PublicParameters, Totient, actual_x);
const shares = EDVDF.EarlyDecryptionSetup(Buffer.from(PrivateParameter.toString()), { shares: 5, threshold: 3 });
const shareDistributionEvents = await sendSecretSharesToParticipants(EDVdfContractABI.signer, SECRET_SHARER_PARTICIPANTS, shares);
const key = AES.getAsKey(PrivateParameter.value);
const {iv, encrypted} = AES.encrypt(key, message);
await EdVdfContract.EarlyDecryptionSetup(SECRET_SHARER_PARTICIPANTS, encrypted);
// await event_EARLY_DECRYPTION_SETUP();
await event_EVAL();
const {y: evalPrivateParameter, pi} = await EDVDF.Eval(PublicParameters, actual_x);
const encryptedMessage = await EdVdfContract.encryptedMessage();
const evalKey = AES.getAsKey(evalPrivateParameter.value);
const decryptedMessage = AES.decrypt({iv, key: evalKey}, encryptedMessage);
const actual_Status = await EdVdfContract.STATUS();
return expect(actual_x).to.be.eql(expected_x)
    && expect(decryptedMessage).to.be.eql(message)
    && expect(actual_Status).to.be.eql('EVAL');
```

Verify

\test\edvdf.contract.test.js

20.6s 1 ✓1

should be successful with random 256 bit prime number 20.6s

```
const EdVdfContract = await EDVdfContractABI.deploy();
const event_SETUP = WaitEvent(EDVdfContractABI.signer.provider, "SETUP(uint256)");
const event_EARLY_DECRYPTION_SETUP = WaitEvent(EDVdfContractABI.signer.provider, "EARLY_DECRYPTION_SETUP(string[])");
const event_EVAL = WaitEvent(EDVdfContractABI.signer.provider, "EVAL(uint256,uint256,uint256)");
const event_VERIFY = WaitEvent(EDVdfContractABI.signer.provider, "VERIFY(address,uint256,uint256)");
const {PublicParameters, Totient} = await EDVDF.Setup(256, TIME["10s"]);
const recipient = (await ethers.getSigners())[8];
await EdVdfContract.Setup(PublicParameters.N, PublicParameters.Time, recipient);
const expected_x = BigInt((await event_SETUP()).data);
const actual_x = BigInt(await EdVdfContract.x());
const {PrivateParameter} = await EDVDF.GenPrivateParameter(PublicParameters, Totient, actual_x);
const shares = EDVDF.EarlyDecryptionSetup(Buffer.from(PrivateParameter.toString()), { shares: 5, threshold: 3 });
const shareDistribution = await sendSecretSharesToParticipants(EDVdfContractABI.signer, SECRET_SHARER_PARTICIPANTS, shares);
const key = AES.getAsKey(PrivateParameter.value);
const {iv, encrypted} = AES.encrypt(key, message);
await EdVdfContract.EarlyDecryptionSetup(SECRET_SHARER_PARTICIPANTS, encrypted);
// await event_EARLY_DECRYPTION_SETUP();
await event_EVAL();
const {y: evalPrivateParameter, pi} = await EDVDF.Eval(PublicParameters, actual_x);
await EDVDF.Verify(PublicParameters, actual_x, evalPrivateParameter, pi);
log(new Date().toISOString(), `Verify_Wes18`);
const actual_Verify_TRANSACTION = (await EdVdfContract.Verify(evalPrivateParameter.value, pi.value));
await event_VERIFY();
const encryptedMessage = await EdVdfContract.encryptedMessage();
const evalKey = AES.getAsKey(evalPrivateParameter.value);
const decryptedMessage = AES.decrypt({iv, key: evalKey}, encryptedMessage);
const actual_SECRET_SHARER_PARTICIPANTS = await EdVdfContract.get_SECRET_SHARER_PARTICIPANTS();
const actual_Status = await EdVdfContract.STATUS();
return expect(actual_x).to.be.eql(expected_x)
    && expect(actual_SECRET_SHARER_PARTICIPANTS).to.be.eql(SECRET_SHARER_PARTICIPANTS)
    && expect(decryptedMessage).to.be.eql(message)
    && expect(actual_Status).to.be.eql('VERIFY');
```

ED-VDF

\test\edvdf.test.js

23.6s 9 ✓ 9

Setup should be successfully return {PublicParameters[N, Time_10s], PrivateParameters} 146ms

```
const {PublicParameters, Totient} = await EDVDF.Setup(RSA_KEYS_BIT_LENGTH, constants.TIME["10s"]);
return expect(typeof PublicParameters.N).to.be.eql('bigint')
    && expect(PublicParameters.Time).to.be.eql(constants.TIME["10s"])
    && expect(typeof Totient).to.be.eql('bigint');
```

GenPrivateParameter should be successfully return {y, pi} 1s

```
const {PublicParameters, Totient} = await EDVDF.Setup(RSA_KEYS_BIT_LENGTH, constants.TIME["10s"]);
const {PrivateParameter, pi} = await EDVDF.GenPrivateParameter(PublicParameters, Totient, X);
return expect(BigInteger.isInstance(PrivateParameter)).to.be.true
    && expect(BigInteger.isInstance(pi)).to.be.true;
```

ShortCircuitEval should be successfully return y 142ms

```
const {PublicParameters, Totient} = await EDVDF.Setup(RSA_KEYS_BIT_LENGTH, constants.TIME["10s"]);
const y = await EDVDF.ShortCircuitEval(PublicParameters, Totient, X);
return expect(BigInteger.isInstance(y)).to.be.true;
```

EarlyDecryptionSetup should be successfully return SharedPrivateParameters 143ms

```
const {PublicParameters, Totient} = await EDVDF.Setup(RSA_KEYS_BIT_LENGTH, constants.TIME["10s"]);
const PrivateParameter = await EDVDF.ShortCircuitEval(PublicParameters, Totient, X);
const SharedPrivateParameters = EDVDF.EarlyDecryptionSetup(PrivateParameter, { shares: 5, threshold: 3 });
return expect(Array.isArray(SharedPrivateParameters)).to.be.true;
```

EarlyDecryption ReconstructedPrivateParameter should be equal PrivateParameter 147ms

```
const {PublicParameters, Totient} = await EDVDF.Setup(RSA_KEYS_BIT_LENGTH, constants.TIME["10s"]);
const PrivateParameter = await EDVDF.ShortCircuitEval(PublicParameters, Totient, X);
const SharedPrivateParameters = EDVDF.EarlyDecryptionSetup(PrivateParameter, { shares: 5, threshold: 3 });
const ReconstructedPrivateParameter = EDVDF.EarlyDecryption(SharedPrivateParameters.slice(1, 4));
return expect(ReconstructedPrivateParameter).to.be.eql(PrivateParameter.value);
```

Eval should be successfully return {y, pi} 5.1s

```
const {PublicParameters, Totient} = await EDVDF.Setup(RSA_KEYS_BIT_LENGTH, constants.TIME["10s"]);
const {y, pi} = await EDVDF.Eval(PublicParameters, X);
return expect(BigInteger.isInstance(y)).to.be.true
    && expect(BigInteger.isInstance(pi)).to.be.true;
```

_Eval should be successfully return y 520ms

```
const {PublicParameters, Totient} = await EDVDF.Setup(RSA_KEYS_BIT_LENGTH, constants.TIME["10s"]);
const y = await EDVDF._Eval(PublicParameters, X);
return expect(BigInteger.isInstance(y)).to.be.true;
```

Verify should be successfully return true 12.7s

```
const {PublicParameters, Totient} = await EDVDF.Setup(RSA_KEYS_BIT_LENGTH, constants.TIME["10s"]);
const {y, pi} = await EDVDF.Eval(PublicParameters, X);
const t = await EDVDF.Verify(PublicParameters, X, y, pi);
return expect(t).to.be.true;
```

HPrime should be successfully return Challenge 3.7s

```
const {PublicParameters: {N, Time}, Totient} = await EDVDF.Setup(RSA_KEYS_BIT_LENGTH, constants.TIME["10s"]);
const {y, pi} = await EDVDF.GenPrivateParameter({N, Time}, Totient, X);
const Challenge = await EDVDF.HPrime(N, Time, X, y);
return expect(typeof Challenge === "bigint").to.be.true;
```

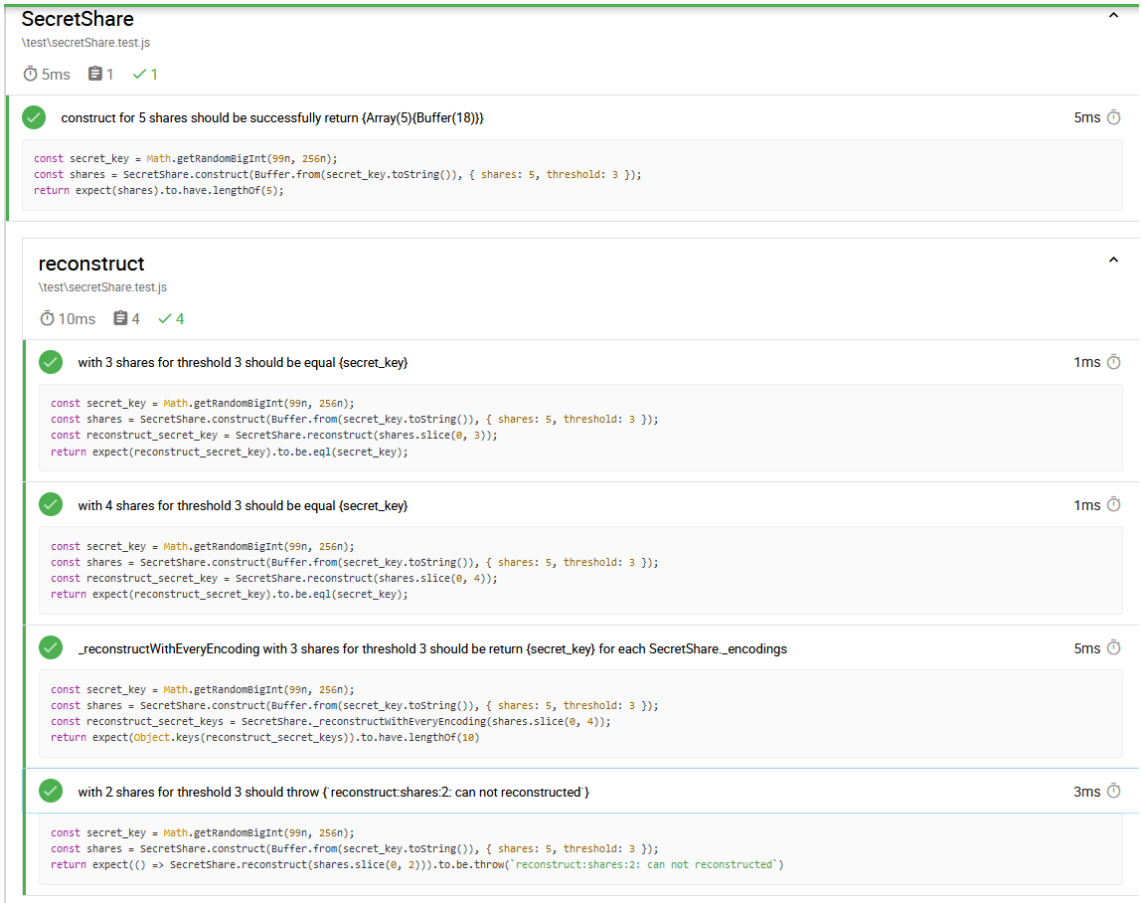
RSA

\test\rsa.test.js

166ms 1 ✓ 1

Setup should be successfully return {SecretKeys, PublicKeys} with bit length 256 166ms

```
const {SecretKeys, PublicKeys} = await RSA.Setup(256);
return expect(typeof SecretKeys.Totient).to.be.eql('bigint')
    && expect(typeof SecretKeys.d).to.be.eql('bigint')
    && expect(typeof PublicKeys.N).to.be.eql('bigint')
    && expect(typeof PublicKeys.e).to.be.eql('bigint')
    && expect(PublicKeys.e).to.be.equal(65537n);
```


SecretShare
\test\secretShare.test.js
5ms 1 1
construct for 5 shares should be successfully return {Array(5){Buffer(18)}} 5ms
const secret_key = Math.getRandomBigInt(99n, 256n);
const shares = SecretShare.construct(Buffer.from(secret_key.toString()), { shares: 5, threshold: 3 });
return expect(shares).to.have.lengthOf(5);
reconstruct
\test\secretShare.test.js
10ms 4 4
with 3 shares for threshold 3 should be equal {secret_key} 1ms
const secret_key = Math.getRandomBigInt(99n, 256n);
const shares = SecretShare.construct(Buffer.from(secret_key.toString()), { shares: 5, threshold: 3 });
const reconstruct_secret_key = SecretShare.reconstruct(shares.slice(0, 3));
return expect(reconstruct_secret_key).to.be.eql(secret_key);
with 4 shares for threshold 3 should be equal {secret_key} 1ms
const secret_key = Math.getRandomBigInt(99n, 256n);
const shares = SecretShare.construct(Buffer.from(secret_key.toString()), { shares: 5, threshold: 3 });
const reconstruct_secret_key = SecretShare.reconstruct(shares.slice(0, 4));
return expect(reconstruct_secret_key).to.be.eql(secret_key);
_reconstructWithEveryEncoding with 3 shares for threshold 3 should be return {secret_key} for each SecretShare._encodings 5ms
const secret_key = Math.getRandomBigInt(99n, 256n);
const shares = SecretShare.construct(Buffer.from(secret_key.toString()), { shares: 5, threshold: 3 });
const reconstruct_secret_keys = SecretShare._reconstructWithEveryEncoding(shares.slice(0, 4));
return expect(Object.keys(reconstruct_secret_keys)).to.have.lengthOf(10)
with 2 shares for threshold 3 should throw { reconstruct:shares:2: can not reconstructed } 3ms
const secret_key = Math.getRandomBigInt(99n, 256n);
const shares = SecretShare.construct(Buffer.from(secret_key.toString()), { shares: 5, threshold: 3 });
return expect(() => SecretShare.reconstruct(shares.slice(0, 2))).to.be.throw(`reconstruct:shares:2: can not reconstructed`)