

# Proactive Dynamic Secret Sharing: Implementation, Benchmarking, Applications to Distributed Password Authenticated Symmetric Key Encryption

Monday 12<sup>th</sup> December, 2022 - 12:43

Georgi Tyufekchiev  
University of Luxembourg  
Email: [georgi.tyufekchiev.001@student.uni.lu](mailto:georgi.tyufekchiev.001@student.uni.lu)

This report has been produced under the supervision of:

Qingju Wang, Marius Lombard-Platet  
University of Luxembourg  
Email: [qingju.wang@uni.lu](mailto:qingju.wang@uni.lu)  
Email: [marius.lombard-platet@uni.lu](mailto:marius.lombard-platet@uni.lu)

**Abstract**—Data breaches are a serious threat to both the businesses and their customers. Compromised passwords can lead to irreparable damage in the life of anyone. For this reason, strong password authentication protocols are needed to protect the personal information of the user. The aim of this report is to look at how dynamic secret sharing schemes can help the development of such protocols. We will provide a proof-of-concept implementation of such a scheme, which was developed for retrieving secrets from the blockchain.

## 1. Introduction

Over the past few decades, technology has evolved significantly - from the invention of the World Wide Web to the emergence of cloud based services. However, the more complex the systems become, the more they are prone to security vulnerabilities. Hackers are getting creative in the ways they attack and exfiltrate data. With information being one of the most valuable assets, it is crucial that we protect our personal data.

The majority of the population is using social media applications, peer-to-peer communication or playing massive multiplayer online games. We heavily rely on companies to deploy strong security mechanisms to protect their customers. Still, every year we can witness data breaches not only in small businesses but major ones as well. The users themselves can become a victim of phishing or social engineering attacks, which results in stolen credentials and identity theft.

One of the reasons for such problems is the way we create and use passwords. It is cumbersome to make and remember a new long and complex password for every application. For this reason, many create a simple one (and derivations of it) and use it in more than one place.

However, this can make the job of our adversaries much simpler. So how can we protect ourselves?

One solution is for the users to use and store strong encryption keys. Even so, not everyone has the technical knowledge for this, and if someone has it, key management is a difficult task. The solution to this problem is using distributed password authentication protocols. Such protocols store the encryption key on multiple servers, releasing the end-user from the responsibility of managing keys as well as protecting them from different attacks. The aim of this Bachelor Semester Project is to look at proactive dynamic secret sharing, which can aid in the development of such protocols.

## 2. What is DPSS and its application?

Dynamic proactive secret sharing is a method to update distributed keys after certain periods of time, such that the attacker does not have the time to compromise the key(s). It is a useful technique for when the attacker can dynamically corrupt parties (servers), providing us with optimal-resilient systems [6]. Such schemes also provide a solution to the "Byzantine Generals Problems"[10]. In the problem, Byzantine generals communicate only with messengers, which can be corrupt and provide false information to other parties with the goal of confusing and sabotaging them.

Another usage for proactive secret sharing is in combination with the *witness encryption* [7] concept. The authors, who came up with the idea, want to find a solution to the following problem: "Can we encrypt a message so that it can only be opened by a recipient who knows a witness to an NP relation?". In this case, we only care if the recipient knows the solution to some NP-complete problem such as the battleship puzzle or the travelling salesman problem. Our main focus is the paper on how to store and retrieve secrets

from the blockchain [8], which combines the concepts of DPSS and extractable witness encryption.

## 2.1. Domains

**2.1.1. Scientific.** The scientific domain for the project is cryptographic protocols. We will look at several cryptographic methods, which are necessary to achieve dynamic secret sharing.

**2.1.2. Technical.** A proof-of-concept implementation of DPSS will be shown, which covers the techniques presented in the scientific part. A few parts of the source code will be explained.

## 2.2. Preliminaries

In this section, we will provide a high-level overview of the scientific concepts and technical tools used in this paper.

**2.2.1. Shamir Secret Sharing.** The Shamir Secret Sharing scheme [11] (SSS) is a  $(k,n)$  threshold scheme, used to split a secret into  $n$  shares, where any  $k$ -subset of  $n$  can recover the secret. It provides us with homomorphic encryption, meaning we can do addition and scalar multiplication on the shares.

**2.2.2. Polynomial Commitments.** Polynomial commitments [9] can be found in zero knowledge proofs. They are used to store large amounts of data into elliptic curve points. The goal is to prove to a verifier that some computations have taken place, without him redoing all of them. It finds application in blockchain technologies, where storing large amounts of data is expensive.

**2.2.3. Elliptic Curve Cryptography.** Elliptic curves are an algebraic structure over finite fields, which are used in public key cryptography. We will focus on pairing friendly curves, which are used in the polynomial commitments.

### 2.2.4. Technical Tools.

**2.2.5. Python.** The Python programming language was created by Guido van Rossum in 1991. It is a high-level, dynamically typed language, which supports object oriented and functional programming.

**2.2.6. PyCharm.** PyCharm is a powerful Python IDE, which supports smart code completion, code inspections, on-the-fly error highlighting and quick-fixes etc.

## 3. Pre-requisites

For this project, no previous knowledge in cryptography is required. For the implementation of the DPSS, some experience in the Python programming language is required. As our goal is to provide some type of benchmarking, it is important to know about data structures and multiprocessing.

## 4. Scientific Research

### 4.1. Secret Sharing

To understand Shamir secret sharing, first let us see why secret sharing is needed. Imagine our actor Alice has a password "PASSWORD" for her banking account and she wants to store the password somewhere and can access it anytime. Doing so can put Alice at risk because that storage place is a single point of failure. If a hacker manages to compromise the storage place, Alice's password could be stolen. Another threat is the storage place being destroyed or just unavailable. To solve this issue, Alice can "share" her password with a few of her friends. She can split the word into pieces like "PA", "SSW", "ORD" and give a piece to different people. When a certain number of them gather, Alice can recover her password. In that case, even if someone is her enemy, they will possess only one piece of the password and will not be able to compromise it. It is important to notice that if Alice has enemies above a certain threshold then her password will certainly be stolen or even lost. Now we look at a proper secret sharing scheme based on Adi Shamir's work.

**4.1.1. Shamir Secret Sharing.** Shamir Secret Sharing is a  $(k,n)$ -threshold scheme, where a secret  $S$  is divided among  $n$  parties such that any  $k$ -subset of them can recover it. Knowing  $k - 1$  shares or less, makes  $S$  unrecoverable. The scheme works as follows:

- Generate a finite field  $F_q$  where  $q$  is larger than the shares being generated.
- Represent the secret  $S$  as the coefficient  $a_0 \in F_q$
- Generate  $k - 1$  coefficients  $a_1, \dots, a_{k-1} \in F_q$
- Evaluate the polynomial  $f(x) = a_0 + a_1x + \dots + a_{k-1}x^{k-1}$  at points  $i \in \{1, \dots, n\}$  to produce the pair  $(i, f(i))$ .
- Give a pair denoted as  $[s]_i$  to each party  $P_i$ . The whole sharing is denoted as  $[s]_d$  where  $d$  is the degree of the polynomial.

To reconstruct the secret  $S$ , Lagrange's polynomial interpolation can be used

$$a_0 = f(0) = \sum_{j=0}^{k-1} y_j \prod_{m=0}^{k-1} \frac{x_m}{x_m - x_j}$$

### 4.2. Polynomial Commitments

Blockchain technology is gaining popularity due to the emergence of Web3 and cryptocurrency. The technology relies on heavy computations and data storage. In order to prove that these computations have been carried out, nodes must recompute everything and send the results to a verifier, which is costly and inefficient. One solution to this problem is to use elliptic curve points to "commit" the data, which can be used in the verification process. This approach is superior to simply hashing the data because it allows for mathematical operations on the original data without

compromising its integrity. Polynomial commitments are necessary to ensure that the data remains "separated" even when it is compressed into a single elliptic curve point. The most famous scheme, known as KZG, was made by A.Kate, G.Zaverucha and I.Goldberg. Next we show the formal definition of the scheme as given by the authors.

#### 4.2.1. KZG Polynomial Commitments.

- **Setup**( $1^k, t$ ): computes two groups  $G$ , and  $G_T$  of prime order  $p$  (providing  $k$ -bit security) such that there exists a symmetric bilinear pairing  $e : G \times G \rightarrow G_T$  and for which the t-SDH assumption holds. Then it randomly samples generators  $g, h \in G$  and  $\alpha \in [p-1]$ . The secret key SK is  $\alpha$  and the public key PK is  $(G, G_T, e, g, g^\alpha, g^{\alpha^2}, \dots, g^{\alpha^t}, h, h^\alpha, h^{\alpha^2}, \dots, h^{\alpha^t})$ . The SK is not needed of the rest of the construction.
- **Commit**( $PK, f()$ ): Computes the commitment  $Com = g^{f(\alpha)} h^{r(i)}$  using the PK, where  $r() \in \mathbb{Z}_p$  is a random polynomial of degree  $t$ .
- **VerifyPoly**( $PK, Com, f(), r()$ ): If the degree of either  $f()$  or  $r()$  is larger than  $t$  then it outputs 0. Otherwise it computes  $g^{f(\alpha)} h^{r(i)}$  using PK and compares with  $Com$ . If the result matches  $Com$ , it outputs 1, otherwise it outputs 0.
- **CreateWitness**( $PK, f(), i, r()$ ): It first computes  $f'(x) = \frac{f(x)-f(i)}{x-i}$  and  $r'(x) = \frac{r(x)-r(i)}{x-i}$ . Then it computes  $w_i = g^{f'(\alpha)} h^{r'(\alpha)}$  using PK. Finally, output  $(i, f(i), r(i), w_i)$ .
- **VerifyEval**( $PK, Com, i, f(i), r(i), w_i$ ). It checks whether  $e(Com, g) = e(w_i, \frac{g^\alpha}{g^i}) e(g^{f(i)} h^{r(i)}, g)$ . If true it outputs 1, Otherwise, output 0.

### 4.3. Elliptic Curves

In this section we will refer to the book "Elliptic Curves in Cryptography" [1]. Elliptic curves are mainly used in cryptography as public key encryption. The reason for that is because elliptic curve cryptography has low CPU and memory usage, resulting in fast encryption and decryption. The shape of elliptic curves is not an actual ellipse but rather a looping line, intersecting both axis while being symmetric along the x-axis. The equation for elliptic curves usually has the form  $y^2 = x^3 + ax + b$ . It is important to know that elliptic curves are groups, which means they only have one binary operation usually called "addition".

Let  $P$  and  $Q$  be two distinct points. A straight line through  $P$  and  $Q$  must intersect the curve at a third point  $R$ . Reflecting the point  $R$  along the x-axis will produce  $P+Q$ . To double a point, meaning  $P + P$  the tangent line has to be taken from  $P$ . This line then must intersect with the curve at only one point  $R$ . Again,  $R$  is reflected along the x-axis to obtain  $P + P$ . In order not to write  $P + P + P + \dots + P$ , it is accepted to write  $nP$ . The equivalent to zero is the "point at infinity"  $O$ , allowing the operation  $P + O = P$ . Elliptic curves also have an "order", which means that there is a

number  $n$  such that  $P * n = O$ . Finally, there exists the so called "generator", which is supposed to be the number 1 in most cases, although in theory the generator can be any point on the curve. Next, we will look at pairing curves.

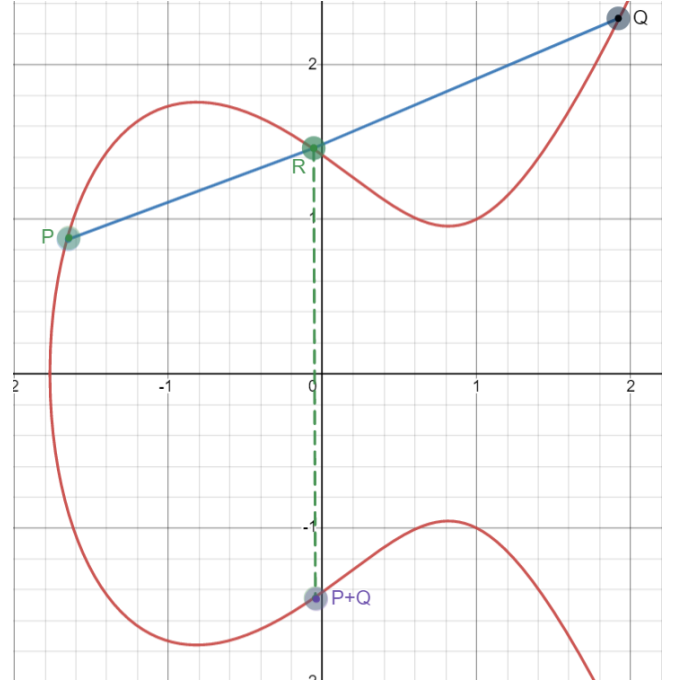


Figure 1: Adding two points on elliptic curve

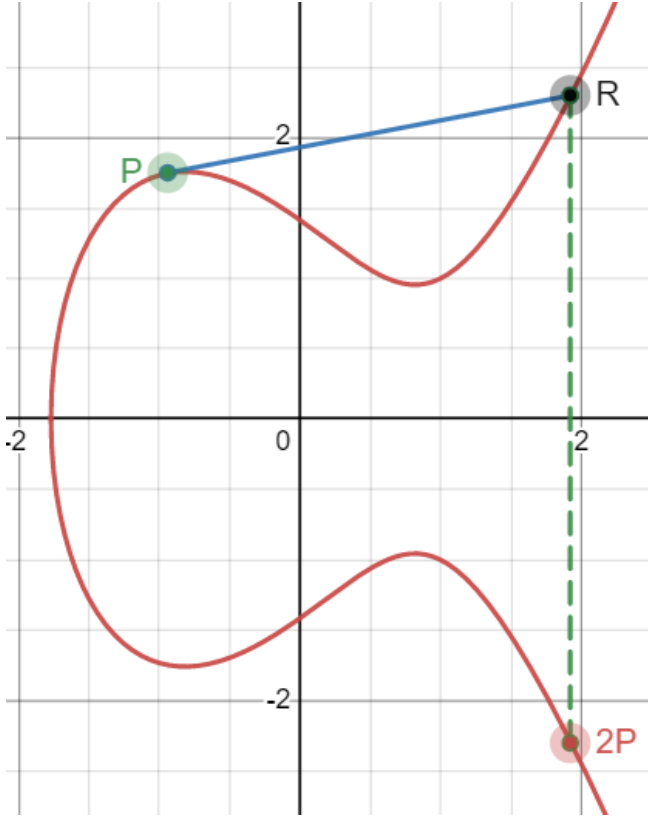


Figure 2: Point doubling on elliptic curve

#### 4.3.1. Pairing Friendly Elliptic Curves.

For pair-friendly elliptic curves we will use an article by Vitalik Buterin [4], which nicely explains how everything works. Pairing is an advanced method, which allows the computation of more complicated equations. If we have  $P = pG$ ,  $Q = qG$  and  $R = rG$  it is possible to check if  $p * q = r$ . It would seem that this can leak information about the secrets  $p, q$  or  $r$ , but that is not the case. Thanks to the computational Diffie Hellman problem and the discrete logarithm problem, this leakage remains computationally infeasible for the time being. Further benefits of using pairing is that it allows to check quadratic constraints like  $e(P, Q) * e(G, 5G) = 1$ . The  $e$  here stands for "bilinear mapping", which basically allows operations like  $e(P + S, Q) = e(P, Q) * e(S, Q)$ . This is great when using simple numbers, but they are not suitable for cryptography, so pairing has to be combined with elliptic curves. However, it is not possible to pair any elliptic curve point. One has to come up with a function  $e(P, Q)$  such that the output is an element in  $F_{p^{12}}$  (this is not true for every pairing, but for our use case we will need this element). The group  $F_{p^{12}}$  is used for a procedure called "final exponentiation", where the result of the above function can be raised to the power  $z = (p^{12} - 1)/n$  and  $p^{12} - 1$  is the order of the group. The mathematics behind all of this is a whole paper by itself, so we will not go any further than that. Since pairing cannot be applied to any elliptic curve, we will briefly look at one pair-friendly elliptic curve, which is used in the implementation

of the system.

#### 4.3.2. BLS12-381.

The BLS12-381 curve was designed and implemented by Sean Bowe [3] and is part of the BLS family described in [2]. The 12 here refers to the embedding degree and 381 is the amount of bits required to represent a coordinate. With this construction, not one but two curves are used. The first one has the equation  $y^2 = x^3 + 4$  and is over the finite field  $F_q$ . The second one is a field extension from  $F_q$  to  $F_{q^2}$  where the equation becomes  $y^2 = x^3 + 4(1 + i)$ .

#### 4.4. DPSS

Now we will look at the DPSS defined in [8]. As stated in the paper they "allow users to encode a secret along with a release condition". Their system is designed for active adversary with threshold of  $t/n < 1/2$  and achieve communication complexity of  $O(n)$  (amortized) and  $O(n^2)$  (non-amortized). There are 3 main phases, each composed of a few protocols. We give a brief description of the phases, more detailed information and security proofs can be found in the paper.

**4.4.1. Setup Phase.** To generate the public key for the KZG scheme either a trusted third party can be used or if this is not possible an MPC protocol can be executed. This phase is composed of the following protocols

- **Distribution.** Each party  $P_i$  will generate two sharings using Shamir's scheme. Since each sharing is a polynomial by itself, the party can execute **Commit**( $PK, f()$ ) with  $f()$  being the first share. The random polynomial  $r()$  here is the second sharing. With each commitment, they also compute the witnesses for their shares. Each party sends the  $j - th$  shares and witness to the  $j - th$  party. Finally, they invoke the **VerifyEval** to verify the validity of the received shares. The commitments are also published.
- **Accusation-Response.** If a party fails to validate a share, they may accuse the party that sent it. When an accusation is made, the accused party must publish the shares and witnesses they sent. All parties will then verify the validity of these shares. If the verification fails, the accused party is considered corrupted. If the verification is successful, the accusing party may use the shares and witnesses that were provided.
- **Verification.** The parties need to check that the commitments from the Distribution protocol were computed correctly. Since opening the polynomials leaks the generated shares, they compute two random additional ones, which will be used to mask the initial. If this verification fails, the parties take *fail* as output. If this happens they go to the **Single-Verification**( $P_i$ ) protocol, which is used to identify the corrupted party.
- **Output.** This protocol will generate random sharings, which will be sent to the client and used to

mask the secrets. One sharing will be used per secret. Instead of generating each share individually, which takes time, the parties can batch it. This is achieved using a predetermined Vandermonde matrix. The commitments and witnesses of the sharings are computed by each party.

- **Fresh.** The client receives the shares and validates them. After  $t + 1$  validations pass, the client reconstructs the sharing and uses it mask the secret. The mask is published and each party can individually compute their share of the secret.

**4.4.2. Hand-off Phase.** In the next phase we have a new set of parties, to whom the secret will be transferred, bringing the total number of participants to  $2n$ , making this phase the most difficult. The protocols are essentially the same, with some differences. Instead of generating only two shares, they new committee has to create 4 shares, with two pairs that are identical but different from each other. The new committee will distribute everything generated to the old committee as well. Only the final protocol is different.

- **Refresh.** The old committee will mask their share of the secret with the shares generated in the Output protocol. Then they will choose a *king* party from the same committee and send their shares of the secret along with the witness. All of the commitments have been published. The *king* will validate the shares, reconstructs the masked secret. All parties will check if the reconstruction is valid. If not they regard the party as corrupt. Otherwise, the parties in the new committee proceed with computing their new share of the secret. Again the compute the commitments and witness of the share.

**4.4.3. Reconstruction Phase.** This is the simplest phase, composed of one protocol. All parties in the current committee send their share of the secret along with the witness. The client will verify the shares and after the first  $t + 1$  pass, the secret can be reconstructed.

## 4.5. Benchmarking

This section presents the results of our benchmarking of the DPSS system. We only evaluate the case where all parties are acting honestly. Since we do not have a network, it is important to note that most of the code is executed sequentially, meaning that some parts of the system require one party to wait for the previous party to finish.. For benchmarking, a tool provided by Python is used called "cProfiler" and for visualisation of the data, we use SnakeViz.

In the first version of the system, everything is sequential and some part of the system scale quadratically. The figure below represents the result from the profiling tool when we execute the system with 10 parties, which increase to 20 in the hand-off phase.



Figure 3: Fully Sequential Execution - 10 parties

For now we only care about the top rectangle, which shows the total amount of time it took for the system to finish - 300 seconds was already too much for only 10 parties. In that case, we decided to parallelize some parts of the code, which acted as a bottleneck. The speedup was achieved using multiprocessing. Since we incorporated parallelism, which heavily affects the results, the specifications of the computer are very important

- Processor Intel(R) Core(TM) i5-9300H CPU @ 2.40GHz, 2400 Mhz, 4 Core(s), 8 Logical Processor(s)
- Cache Sizes from L1 to L3 - 256KB, 1MB, 8MB
- 16 GB RAM
- at the time of execution the CPU speed was between 3500Mhz-4000Mhz

After the parallelization, we get the following result for the same test case. As we can see, a speedup of

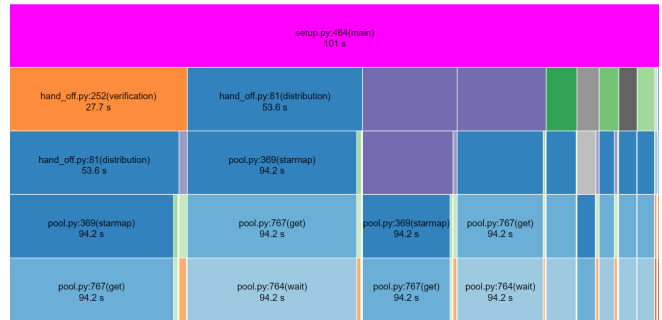


Figure 4: Partially Sequential Execution - 10 parties

3x was achieved. Now we can provide a more detailed benchmarking.

Figure 5 shows the total time it took for the system to finish with 10, 20, 30, 40 and 60 parties. We can see it scales linearly, but spikes at 60 parties. However, these numbers do not represent the time it takes for one party to finish their computations. Figure 6 shows the time per party.

Next, we look at how each phase performed at figure 7. Again this is the total time and we can see that the Reconstruction phase is constant. However, we should

properly divide the total time by the number of parties to get more accurate data. On graph 8 we can clearly see the scaling is again linear. Slight anomalies can be observed, but this is normal considering the system has many processes, which affect the performance.

Next we look at how the different protocols performed of each phase. We skip the Reconstruction phase as it is only one protocol and we already saw it is nearly constant time. Only the total time will be shown, but as we saw when we divide by the number of parties the graph is clearly linear.

On figure 9 we can see the Setup phase. As the graph shows, the distribution and verification protocol are the most time consuming. This is due to the computations of the witnesses and the validation of shares, which become heavier the more parties there are. The Fresh protocol is almost constant complexity.

Figure 10 shows the Hand-off phase. The distribution and verification protocols are again the most time consuming. The output and refresh protocols are almost identical to those of the setup phase.

## 4.6. Assessment

We covered all of the research topics, required to proceed with the technical deliverable. The benchmarking provides useful information for the scaling of our implementation and whether this method is efficient for securely sharing a secret.

## 5. DPSS Implementation

### 5.1. Requirements

In this section we describe the functional and non-functional requirements the system will have. For the non-functional requirements we will use the international standard **ISO/IEC 25051:2014**

#### 5.1.1. Functional Requirements.

**FR#1.** In the **Setup phase** and **Hand-off Phase** the parties are able to distribute shares and witnesses among each other. Each party is able to publish their commitments.

**FR#2.** If validation of shares fails, the party is able to accuse the party, which sent them. The accused party is able to publish the shares and witnesses for further validation.

**FR#3.** In the **Verification protocol**, the party is able to publish certain shares and witnesses, used for the verification.

**FR#4.** In the **Fresh protocol**, the parties are able to send their shares from the **Output protocol** to the client.

**FR#5.** The client is able to validate the shares. They are also able to generate a random polynomial and use it to mask their secret. Finally, they are able to publish the mask. Each party is able to compute their share of the secret.

**FR#6.** In the **Refresh protocol** the old committee is able to choose a *king party* and send their shares and witnesses to that party. The *king* is able to reconstruct shares. The other parties are able to accuse the king of being corrupted.

**FR#7.** The client is able to request the shares of the secret from the parties. The parties can send their shares. The client then is able to verify the validity and reconstruct the secret.

#### 5.1.2. Non-Functional Requirements.

**NFR#1.** *Functional completeness*, meaning all the protocols must be implemented for system to function as intended.

**NFR#2.** *Functional correctness*, meaning all functions must perform accurate calculations and send the correct results to the appropriate party.

**NFR#3.** *Integrity*, meaning dishonest parties cannot violate the integrity of the secrets.

**NFR#4.** *Accountability*, meaning all dishonest parties can be identified in the case they decide to cheat.

**NFR#5.** *Modularity*, meaning the system is composed of discrete components, facilitating changes and code maintenance.

**NFR#6.** *Reusability*, meaning a function can be used to build different components of the system.

**NFR#7.** *Modifiability*, meaning components can be changed without introducing defects or degrading the system.

Additional non-functional requirements, which are specific to the system.

**NFR#8.** *Multiprocessing*, which allows to parallelize parts of the system.

**NFR#9.** A professional cryptographic library must be used, which is recognised by the public.

**NFR#10.** Either BLS or BN curves can be used in the system.

**NFR#11.** Only SHA-3 hashes can be used in the system.

### 5.2. Development Process

In this section we describe the software development process of the system. We will follow the incremental process,

which allows us to be flexible and spot mistakes early. The development process will have two main phases.

### 5.2.1. Honest Setting.

During this phase, we assume our nodes will act honestly during every protocol. The **Single-Verification** protocol will not be implemented as its usage is for when the nodes are acting malicious. The rest of the protocols must be implemented at least partially such that the client can share their secret and then reconstruct it when requested. Verifiability of the shares can be included, so that it is ready for the next phase. For validation, we will use the main paper [8], by making sure the system is working as described in the paper. For verification, we will make sure the system is built right i.e. non-functional requirements such as modularity, reusability and PEP8 styling is used.

### 5.2.2. Malicious Setting.

During this phase we assume our nodes can be malicious. It is mandatory that all protocols are implemented. Commitments and witnesses must be computed, where required, and shares must be validated when necessary. In case the validation fails, the system must react in some kind of way as described in the different protocols. As for system validation, we again make sure the software is developed as in the paper and perform the same verification from the previous phase.

### 5.2.3. Testing.

All testing will be done manually. Functional tests will be performed to verify the functional correctness of each function's output. For validations we use the acceptance testing, which verifies if the system satisfies the business requirements. Finally, we do performance testing by using different number of parties to determine the complexity of the code, identify bottlenecks and possibly improve those parts of the code.

## 5.3. Design

### 5.3.1. Object Oriented Programming.

Python allows us to use the object-oriented paradigm. For this reason, we split our system into modules, which contain classes or only functions. . To ensure proper encapsulation, some attributes and methods may be marked as private and not accessible to users. For simplicity, however, most of these will be set as public. We will aim to adhere to the PEP8 styling guide in our code, although some standards may be inadvertently broken.

### 5.3.2. Class Relationships.

In this section we describe how the classes of the system are related to each other. First, we provide a simplified UML class diagram 11 for the relationship between the different classes. There is not separate class for the **Reconstruction phase** since it is composed of one protocol. For this reason, the reconstruction is put in the **Hand-off phase**. There is a unidirectional association between the *Handoff* and *Setup*

classes, because we need to send some of the *Setup* objects to the *Handoff* class. The *SSS class* stands for Shamir Secret Sharing, and its objects exist only in the instances of the *Setup* and *Handoff* classes. The same goes for the *KZG class*. The *bls* class here is actually a whole package, called **py\_ecc**, which is open source and is composed of multiple modules, but for simplicity we represent it as a single class. It is associated with the *KZG class* as we need to store the generators for creating the public key. The class also depends on the *fft* and *mulcombs* classes for certain mathematical operations. The 3 classes were written by Dankrad Feist and are publicly available in the Ethereum research repository in GitHub. Only small parts of the KZG class have been modified to suit our needs. The *utils* class is holding useful methods, which will be used in certain protocols.

### 5.3.3. Data Structures.

On figure 12 and figure 13 we have put more detailed UML class diagrams of the different classes. For simplicity, we do not provide detailed specifications for the data types of the parameters. Since we are not implementing the system on a network, we will represent the parties as a dictionary collection, where the keys are party IDs and the values are various data structures representing commitments, witnesses, and other data. When we say that parties have published something, we mean that there is a shared data structure that can be accessed by all parties.

### 5.3.4. Cryptography.

When the parties have to compute the witness, they also need a random point at which to evaluate the polynomials. Usually, this point they get from another party. However, we do not want to have additional communication cost for this, so we swap this interaction with the Fiat-Shamir heuristic [5]. As a hashing algorithm we will use the SHAKE-256, which is part of the SHA-3 family, a standard released by NIST in 2015.

### 5.3.5. Multiprocessing.

The Python program is a process that is executed on the main thread. However, in our system we want to speed up certain parts by running the code in parallel i.e. run tasks concurrently. Python already provides us with a module, which is capable of spawning new processes. More specifically, we will use the *Process Pool*. The pool is responsible for creating the processes when needed and making them wait without consuming computational resources. As per the documentation we have "A process pool object which controls a pool of worker processes to which jobs can be submitted. It supports asynchronous results with timeouts and callbacks and has a parallel map implementation."

## 5.4. Production

In this section we go into details of how the system is implemented. First, we provide the libraries, which are used. Parts of the source code, will be shown with the idea to capture as best as we can, the way the system works.



#### 5.4.1. NumPy.

NumPy is a package, mainly used for scientific computing in Python. They provide us with a large selection of mathematical operations such as creating multi-dimensional arrays (matrices), generating and evaluating polynomials efficiently and much more.

#### 5.4.2. PyCryptodome.

The package as described in their documentation "Is a self-contained Python package of low-level cryptographic primitives". Their code is open source and is widely used and supported by the public. We are able to use their implementation of the SHAKE256 hash algorithm, as well as their secure random number generator, suitable for cryptography.

#### 5.4.3. py\_ecc.

This package provides elliptic curve cryptography and is open sourced. Even though the code is not audited and may contain experimental code, it has been developed by people working for the Ethereum Foundation, including Vitalik Buterin. From this package, we will use an optimized version of the BLS12-381 curve.

#### 5.4.4. System Implementation.

In this section we present only the certain parts of the code, which are the most important. First, we have the generation of the public key.

The code in **Listing 1** will generate a list, containing our elliptic curve generators, which are coordinates of the form  $(x, y, z)$ . The size will be the threshold, which depends on the number of parties. This list will be used for computing commitments and witnesses. The  $G_1$  and  $H$  points are defined over  $F_q$ , while  $G_2$  is twisted over  $F_{q^2}$ . The  $(x, y)$  points ideally should be 381 bits long, except for  $z$ , which is the number 1. Finally, we show how the validation is done in **Listing 2**

With this function we want to check the following equation

$$e([C - y]^{-1}, G_2) * e(w, \alpha G_2 + i G_2^{-1}) = 1$$

Where  $C$  is the commitment,  $y = f(i)G_1 + r(i)H$  and  $w$  is the witness. The  $i$  here is the random point, which we generate using the Fiat-Shamir heuristic. When, we say the equation is equal to 1, we mean the coordinate  $(1, 1, 0)$  defined over  $F_{q^2}$ . In this module, there exist functions for computing multi proofs and doing multi validations, but the code and idea behind it becomes way more complex, thus we will not cover it. Next we look at we can use the KZG module in our system.

Each party (**Listing 3**) in the committee takes their polynomials for the  $u$  and  $v$  sharing and computes the tuple  $(w, f(x), h(x), x)$ . Initially, the complexity for computing everything was  $O(n^2)$  and it was the bottleneck of the system. For this reason, we parallelised this section. This requires to put the code in separate function so that they can be executed concurrently. This is done in the following

way (**Listing 4**). First we compute our random point with the *fsh* (Fiat-Shamir) function from our *Utils* module. Then we can start our multiprocessing. Since our function require multiple parameters, we have to prepare in a tuple (or just any iterable structure). The *starmap* allows us to provide the target function with all parameters. The results are stored in a dictionary. Doing all of this reduces the complexity to  $O(n)$ . But we can do even better by computing multiple proofs at once, which further reduces the complexity. We need to modify the *computeWitness* method in the following manner (**Listing 5**).

Here we also need to compute the root of unity. However, all of this gets tricky to do. First, the order of the root of unity has to satisfy the equation  $(curve.order - 1) \bmod root\_of\_unity.order = 0$ . Next, when evaluating the polynomials, we can use at most *root\_of\_unity.order* points (an example of correct order is 16). Ideally we want the order to match the threshold, but this cannot always happen (unless we do some complicated changes to the curve order). It is possible to use order, which is less than the threshold, but for the time being, we have not tested this enough to make sure it is consistent. Though, if we manage to do it, it has high impact on the performance since we have to compute just 3 "aggregates witnesses" instead of  $n$ . The validation remains  $O(n)$ .

In **Listing 2**, we added additional checks for the sign of the values i.e.  $y, y_r, x$ , where  $y, y_r$  are the evaluation of a polynomial at point  $x$ , because it is possible that they are negative. In that case we have to be careful, because multiplying the generators by a negative scalar will stall the program and most likely throw an error. To solve this we can take the inverse of the generator and multiply the scalar by  $-1$ . The negative values are possible to come from the code in **Listing 6**. In that block of code we want to compute

$$w_j([s]_t; [z]_t) = w_j(s + r; z + \psi) / w_j([r]_t; [\psi]_t)$$

$$Com_{KZG}([s]_t; [z]_t) = Com_{KZG}(s + r; z + \psi) / Com_{KZG}([r]_t; [\psi]_t)$$

Since the witnesses and commitments are elliptic curve points, this division is translated into subtraction. However, this subtraction has to be also done on the values of the different polynomial evaluations. From here we can get those negative values, thus the need for additional sign check.

### 5.5. Assessment

First, we evaluate the functional requirements. In the last version of the implementation, all functional requirements are present and automated so that it can act as a real system. From the non-functional requirements only **NFR1**, **NFR2**, **NFR3** have been partially completed. This is due to the fact that the system can identify dishonest parties, but does not do anything about it at the moment. If a protocol takes *fail* as output, the system will just crash and has to be restarted. For this reason the malicious setting phase is also partially



completed. We have also done all of the aforementioned testing to ensure the system works correctly under the honest setting. The last version of the system is not the most efficient but at least we managed to lower the complexity to  $O(n)$ . Of course for a cryptographic protocol we need to perform much faster than the results from the benchmarking.

## Acknowledgment

I Georgi Tyufekchiev, would like to thank my tutors Dr. Qingju Wang and Dr. Marius Lombard-Platet for supporting me during this project.

## 6. Conclusion

In conclusion, the research conducted in this report has provided valuable insights into the topic of proactive secret sharing. The implementation and benchmarking of such a protocol indicate that this is indeed an effective method for securely sharing sensitive information among multiple parties. Doing more research in this topic can help organizations that need to share sensitive information in an efficient manner. It is also useful for the development of future blockchain technologies, which could be used by everyone one day.

## 7. Plagiarism statement

I declare that I am aware of the following facts:

- As a student at the University of Luxembourg I must respect the rules of intellectual honesty, in particular not to resort to plagiarism, fraud or any other method that is illegal or contrary to scientific integrity.
- My report will be checked for plagiarism and if the plagiarism check is positive, an internal procedure will be started by my tutor. I am advised to request a pre-check by my tutor to avoid any issue.
- As declared in the assessment procedure of the University of Luxembourg, plagiarism is committed whenever the source of information used in an assignment, research report, paper or otherwise published/circulated piece of work is not properly acknowledged. In other words, plagiarism is the passing off as one's own the words, ideas or work of another person, without attribution to the author. The omission of such proper acknowledgement amounts to claiming authorship for the work of another person. Plagiarism is committed regardless of the language of the original work used. Plagiarism can be deliberate or accidental. Instances of plagiarism include, but are not limited to:
  - 1) Not putting quotation marks around a quote from another person's work
  - 2) Pretending to paraphrase while in fact quoting
  - 3) Citing incorrectly or incompletely

- 4) Failing to cite the source of a quoted or paraphrased work
- 5) Copying/reproducing sections of another person's work without acknowledging the source
- 6) Paraphrasing another person's work without acknowledging the source
- 7) Having another person write/author a work for oneself and submitting/publishing it (with permission, with or without compensation) in one's own name ('ghost-writing')
- 8) Using another person's unpublished work without attribution and permission ('stealing')
- 9) Presenting a piece of work as one's own that contains a high proportion of quoted/copied or paraphrased text (images, graphs, etc.), even if adequately referenced

Auto- or self-plagiarism, that is the reproduction of (portions of a) text previously written by the author without citing that text, i.e. passing previously authored text as new, may be regarded as fraud if deemed sufficiently severe.

## References

- [1] Ian Blake et al. *Elliptic curves in cryptography*. Vol. 265. Cambridge university press, 1999.
- [2] Dan Boneh, Ben Lynn, and Hovav Shacham. "Short Signatures from the Weil Pairing". In: *Advances in Cryptology - ASIACRYPT 2001, 7th International Conference on the Theory and Application of Cryptology and Information Security, Gold Coast, Australia, December 9-13, 2001, Proceedings*. Ed. by Colin Boyd. Vol. 2248. Lecture Notes in Computer Science. Springer, 2001, pp. 514–532. DOI: 10.1007/3-540-45682-1\\_30. URL: [https://doi.org/10.1007/3-540-45682-1%5C\\_30](https://doi.org/10.1007/3-540-45682-1%5C_30).
- [3] Sean Bowe. *BLS12-381: New ZK-snark elliptic curve construction*. Mar. 2017. URL: <https://electriccoin.co/blog/new-snark-curve/>.
- [4] Vitalik Buterin. *Exploring elliptic curve pairings*. July 2022. URL: <https://medium.com/@VitalikButerin/exploring-elliptic-curve-pairings-c73c1864e627>.
- [5] Amos Fiat and Adi Shamir. "How to Prove Yourself: Practical Solutions to Identification and Signature Problems". In: *Advances in Cryptology - CRYPTO '86, Santa Barbara, California, USA, 1986, Proceedings*. Ed. by Andrew M. Odlyzko. Vol. 263. Lecture Notes in Computer Science. Springer, 1986, pp. 186–194. DOI: 10.1007/3-540-47721-7\\_12. URL: [https://doi.org/10.1007/3-540-47721-7%5C\\_12](https://doi.org/10.1007/3-540-47721-7%5C_12).
- [6] Y. Frankel et al. "Optimal-resilience proactive public-key cryptosystems". In: *Proceedings 38th Annual Symposium on Foundations of Computer Science*. 1997, pp. 384–393. DOI: 10.1109/SFCS.1997.646127.

- [7] Sanjam Garg et al. “Witness encryption and its applications”. In: *Symposium on Theory of Computing Conference, STOC’13, Palo Alto, CA, USA, June 1-4, 2013*. Ed. by Dan Boneh, Tim Roughgarden, and Joan Feigenbaum. ACM, 2013, pp. 467–476. DOI: 10.1145/2488608.2488667. URL: <https://doi.org/10.1145/2488608.2488667>.
- [8] Vipul Goyal et al. “Storing and Retrieving Secrets on a Blockchain”. In: *Public-Key Cryptography - PKC 2022 - 25th IACR International Conference on Practice and Theory of Public-Key Cryptography, Virtual Event, March 8-11, 2022, Proceedings, Part I*. Ed. by Goichiro Hanaoka, Junji Shikata, and Yohei Watanabe. Vol. 13177. Lecture Notes in Computer Science. Springer, 2022, pp. 252–282. DOI: 10.1007/978-3-030-97121-2\\_10. URL: [https://doi.org/10.1007/978-3-030-97121-2%5C\\_10](https://doi.org/10.1007/978-3-030-97121-2%5C_10).
- [9] Aniket Kate, Gregory M. Zaverucha, and Ian Goldberg. “Constant-Size Commitments to Polynomials and Their Applications”. In: *Advances in Cryptology - ASIACRYPT 2010 - 16th International Conference on the Theory and Application of Cryptology and Information Security, Singapore, December 5-9, 2010. Proceedings*. Ed. by Masayuki Abe. Vol. 6477. Lecture Notes in Computer Science. Springer, 2010, pp. 177–194. DOI: 10.1007/978-3-642-17373-8\\_11. URL: [https://doi.org/10.1007/978-3-642-17373-8%5C\\_11](https://doi.org/10.1007/978-3-642-17373-8%5C_11).
- [10] Leslie Lamport, Robert Shostak, and Marshall Pease. “The Byzantine Generals Problem”. In: *ACM Transactions on Programming Languages and Systems* (July 1982), pp. 382–401. URL: <https://www.microsoft.com/en-us/research/publication/byzantine-generals-problem/>.
- [11] Adi Shamir. “How to Share a Secret”. In: *Commun. ACM* 22.11 (1979), pp. 612–613. DOI: 10.1145/359168.359176. URL: <http://doi.acm.org/10.1145/359168.359176>.

## 8. Source Code

```

1 from py_ecc import optimized_bls12_381 as b
2 from fft import fft
3 from multicombs import lincomb
4 def generate_setup(s, size):
5     """
6     # Generate trusted setup, in coefficient form.
7     # For data availability we always need to
8     # compute the polynomials
9     # anyway, so it makes little sense to do things
10    # in Lagrange space
11    """
12
13    return (
14        [b.multiply(b.G1, pow(s, i, MODULUS)) for
15         i in range(size + 1)],
16        [b.multiply(b.G2, pow(s, i, MODULUS)) for
17         i in range(size + 1)],
18        [b.multiply(b.H, pow(s, i, MODULUS)) for i
19         in range(size + 1)]
20    )

```

Listing 1: PK Generation.

```

1 def check_proof_single(commitment, proof, x, y,
2                        y_r, setup):
3     """
4     Check a proof for a Kate commitment for an
5     evaluation f(x) = y
6     """
7     # Verify the pairing equation
8     # # e([commitment - y], [1]) = e([proof], [s -
9     # x])
10    # equivalent to
11    # e([commitment - y]^(-1), [1]) * e([proof],
12    # [s - x]) = 1_T
13    #
14
15    s_minus_x = b.add(setup[1][1], b.multiply(b.
16    neg(b.G2), x))
17
18    if y < 0 and y_r < 0:
19        g_f = b.multiply(b.G1, y * -1)
20        h_r = b.multiply(b.H, y_r * -1)
21    elif y < 0:
22        g_f = b.multiply(b.G1, y * -1)
23        h_r = b.multiply(b.neg(b.H), y_r)
24    elif y_r < 0:
25        g_f = b.multiply(b.neg(b.G1), y)
26        h_r = b.multiply(b.H, y_r * -1)
27    else:
28        g_f = b.multiply(b.neg(b.G1), y)
29        h_r = b.multiply(b.neg(b.H), y_r)
30
31    commitment_minus_y = b.add(commitment, b.add(
32    g_f, h_r))
33    pairing_check = b.pairing(b.G2, b.neg(
34    commitment_minus_y), False)
35    pairing_check *= b.pairing(s_minus_x, proof,
36    False)
37    pairing = b.final_exponentiate(pairing_check)
38    return pairing == b.FQ12.one()

```

Listing 2: Validation.

```

1 def computeWitness(self, committee, i, x):
2     uPoly = committee[i][0][1]

```

```

3     vPoly = committee[i][1][1]
4     xi = x[i]
5     w = []
6
7     for j in range(self.PARTIES):
8         # evaluate the polynomials of u and v
9         # at point x
10        polyEvals = self.__eval(uPoly[1],
11                                vPoly[1], xi[j])
12        # compute the witness which is the
13        # tuple (proof, f(x), h(x), x)
14        # the witness has to be prepared by
15        # each party for each party
16        w.append([compute_proof_single(uPoly
17                                      [1], vPoly[1], xi[j], self.__setup), polyEvals
18                                      [0], polyEvals[1],
19                                      xi[j]])
20
21    return w, i
22
23    def validateShares(self, committee, commit, i)
24    :
25        for j in range(self.PARTIES):
26            witness = committee[i][-1][j]
27            w, fx, hx, xi = witness
28            oCommit = commit[j]
29            if not check_proof_single(oCommit, w,
30                                     xi, fx, hx, self.__setup):
31                print("Validation BAD %d %d" % (i,
32                                                 j), flush=True)
33                # prepare the accusation for the
34                # Accusation protocol
35                self.validations.append((j,
36                                         witness, oCommit))

```

Listing 3: Commitment and witness computation.

```

1     for i in range(self.PARTIES):
2         hashing = fsh(self.__setup, commit[i])
3         x.append(int.from_bytes(hashing.read
4                                 (4), byteorder='big'))
5
6     with Pool() as p:
7         items = [(committee, i, x) for i in
8                 range(self.PARTIES)]
9         for result in p.starmap(self.
10                                computeWitness, items):
11             w, i = result
12             proof[i] = w

```

Listing 4: Parallelizing computations.

```

1    def computeWitness(self, committee, i, x,
2                        rootOfUnity):
3        uPoly = committee[i][0][1]
4        vPoly = committee[i][1][1]
5        xi = x[i]
6        w = []
7
8        coset = [xi * pow(rootOfUnity, n, MODULUS)
9                  for n in range(self.THRESHOLD)]
10       ys = [eval_poly_at(uPoly, z) for z in
11             coset]
12       ys_r = [eval_poly_at(vPoly, z) for z in
13              coset]
14       proof = compute_proof_multi(uPoly, vPoly,
15                                   xi, self.THRESHOLD, self.__setup)
16       w.append([proof, ys, ys_r, xi])
17
18       coset = [xi * pow(rootOfUnity, n, MODULUS)
19                 for n in range(self.THRESHOLD, 2 * self.
20                                THRESHOLD)]

```

```

14       ys = [eval_poly_at(uPoly, z) for z in
15             coset]
16       ys_r = [eval_poly_at(vPoly, z) for z in
17              coset]
18       proof = compute_proof_multi(uPoly, vPoly,
19                                   xi, self.THRESHOLD, self.__setup)
20       w.append([proof, ys, ys_r, xi])
21
22       coset = [xi * pow(rootOfUnity, n, MODULUS)
23                 for n in range(2 * self.THRESHOLD, 3 * self.
24                                THRESHOLD)]
25       ys = [eval_poly_at(uPoly, z) for z in
26             coset]
27       ys_r = [eval_poly_at(vPoly, z) for z in
28              coset]
29       proof = compute_proof_multi(uPoly, vPoly,
30                                   xi, self.THRESHOLD, self.__setup)
31       w.append([proof, ys, ys_r, xi])
32
33       return w, i

```

Listing 5: Batching witness computations.

```

1  for i in range(self.PARTIES):
2      r_phyWitness, rEval, phyEval, x =
proof[i][0]
3      srEval, zphyEval = self.__eval(sr,
zphy, x)
4      publicSharingWitness =
compute_proof_single(sr, zphy, x, self.__setup
)
5      proofs[i] = [curve.add(
publicSharingWitness, curve.neg(r_phyWitness))
, x, srEval - rEval,
6          zphyEval - phyEval]
7      r_phyCommitment = commitments[i][0]
8      partyCommitments[i] = curve.add(
commitment, curve.neg(r_phyCommitment))

```

Listing 6: Last step of the Fresh protocol.

## 9. Appendix

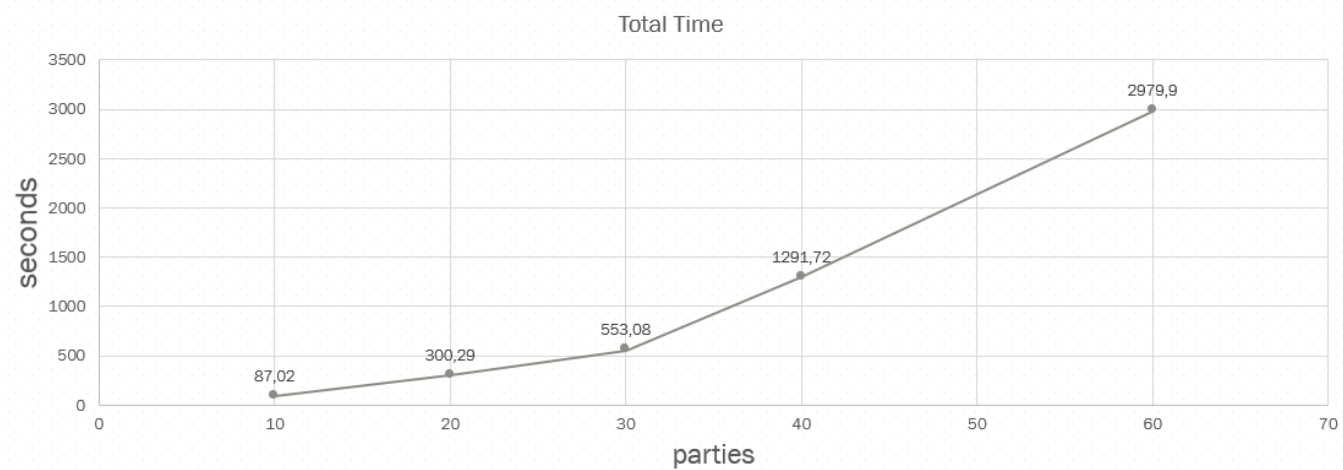


Figure 5: Total Time

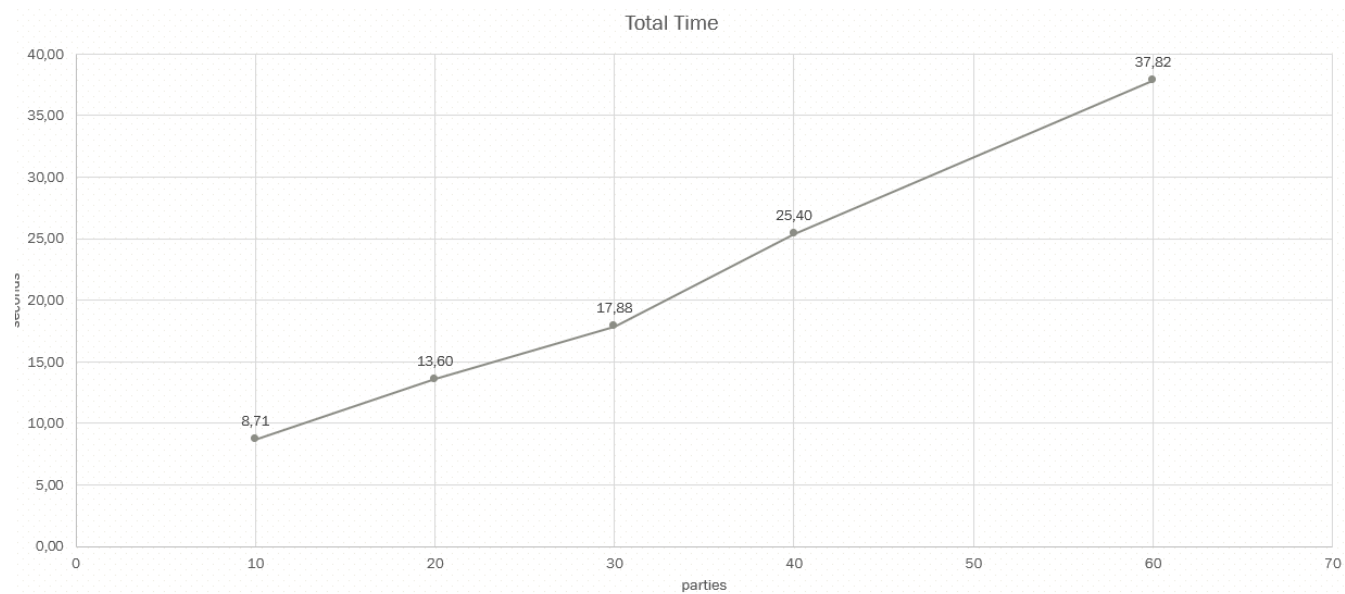


Figure 6: Total Time per Party

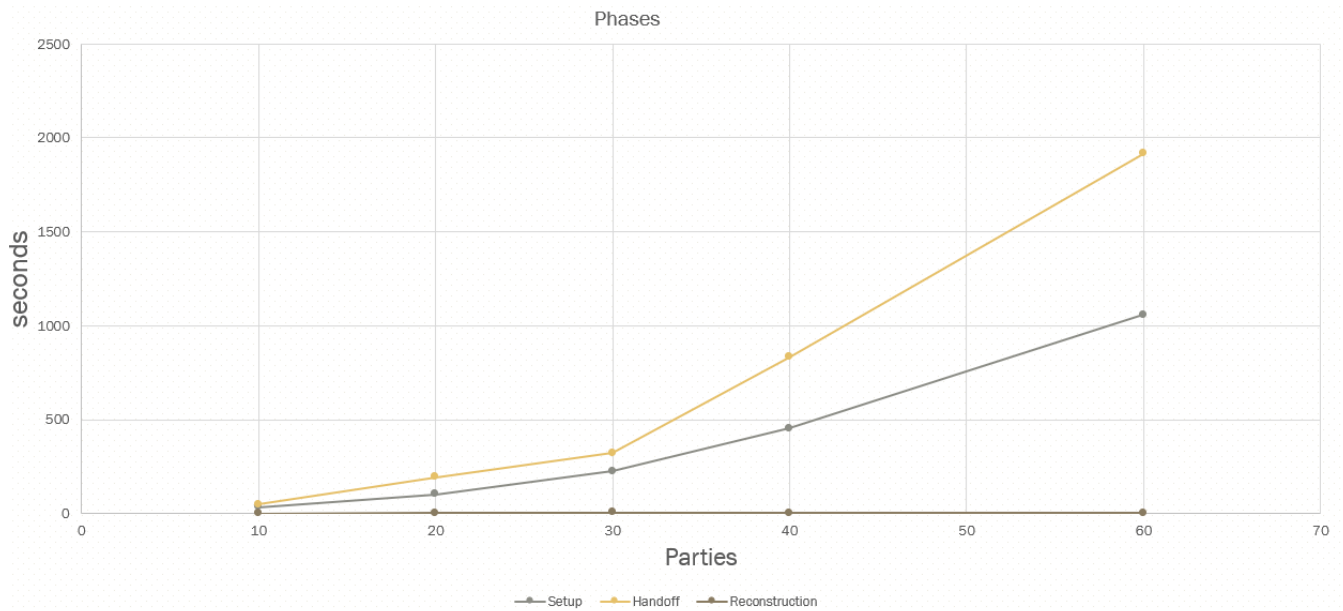


Figure 7: Time per Phase

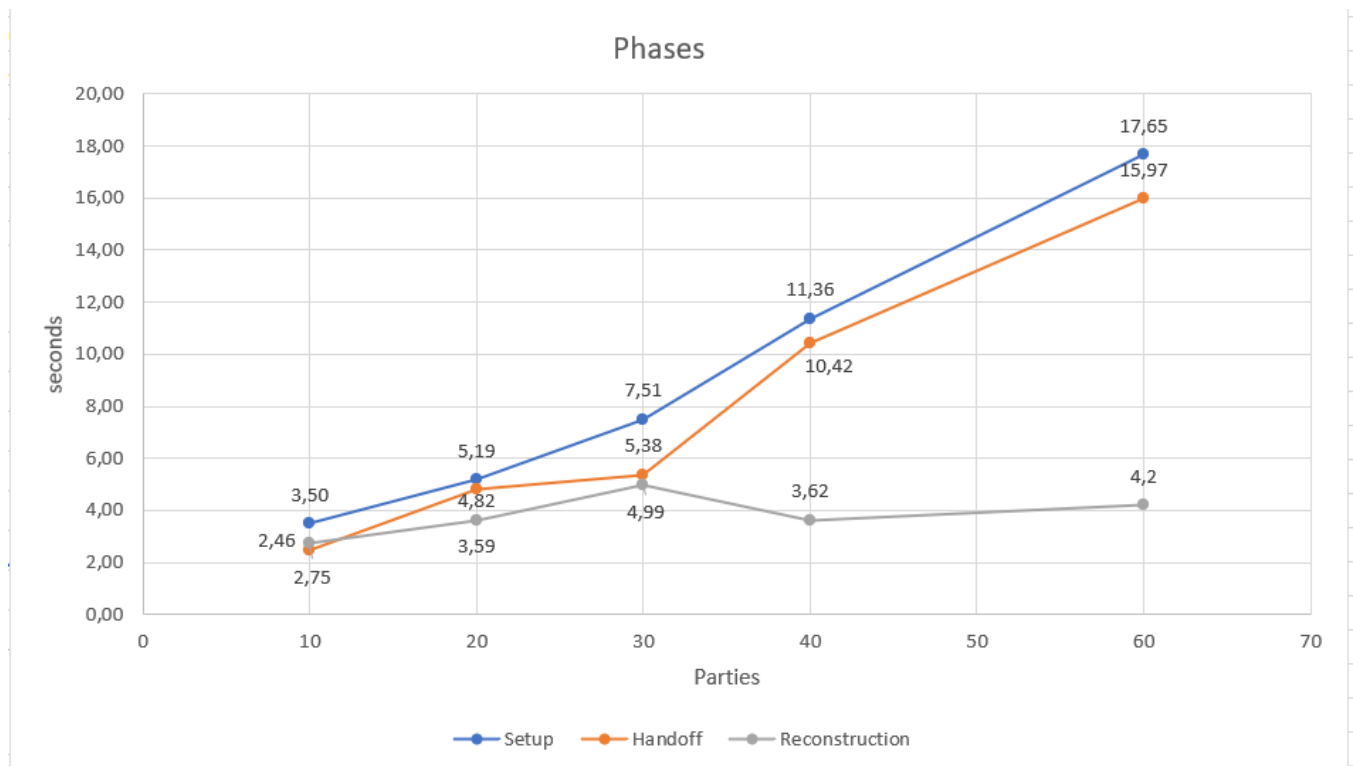


Figure 8: Time per Phase per Party

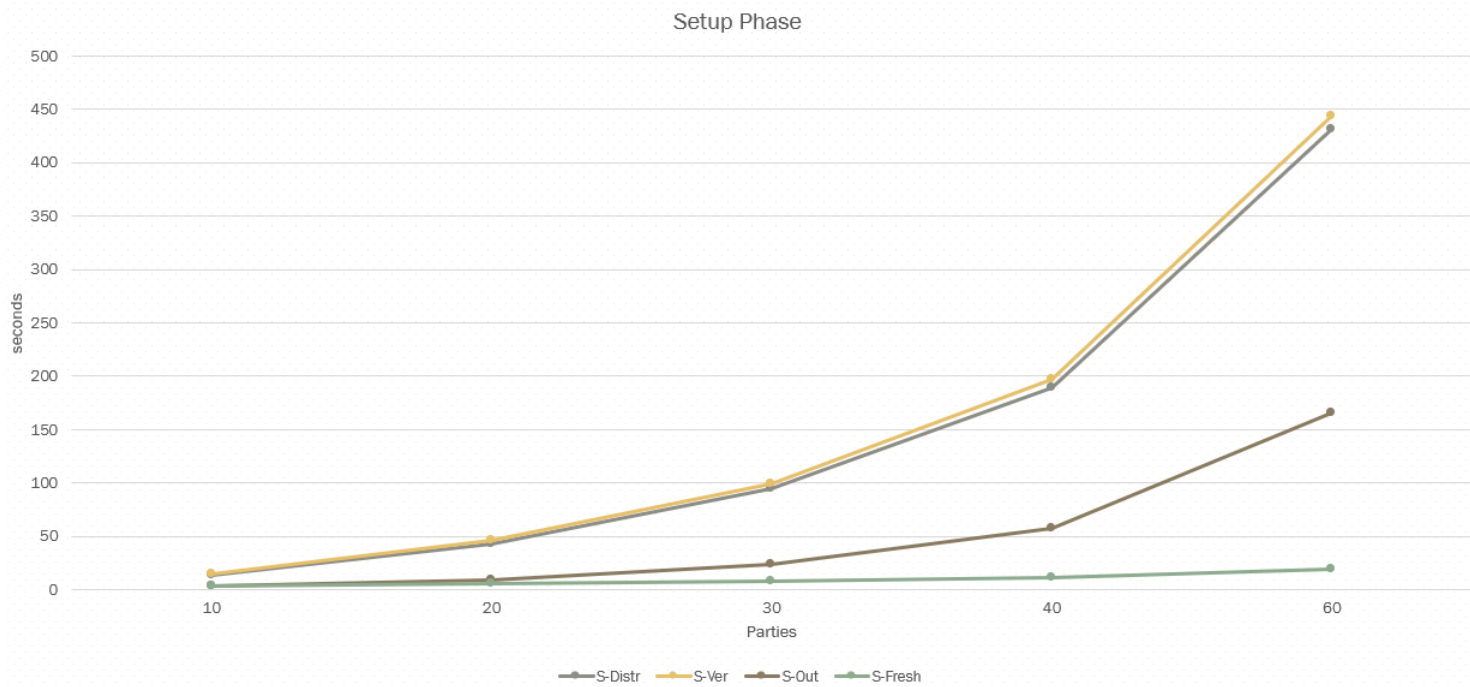


Figure 9: Time for Setup Phase

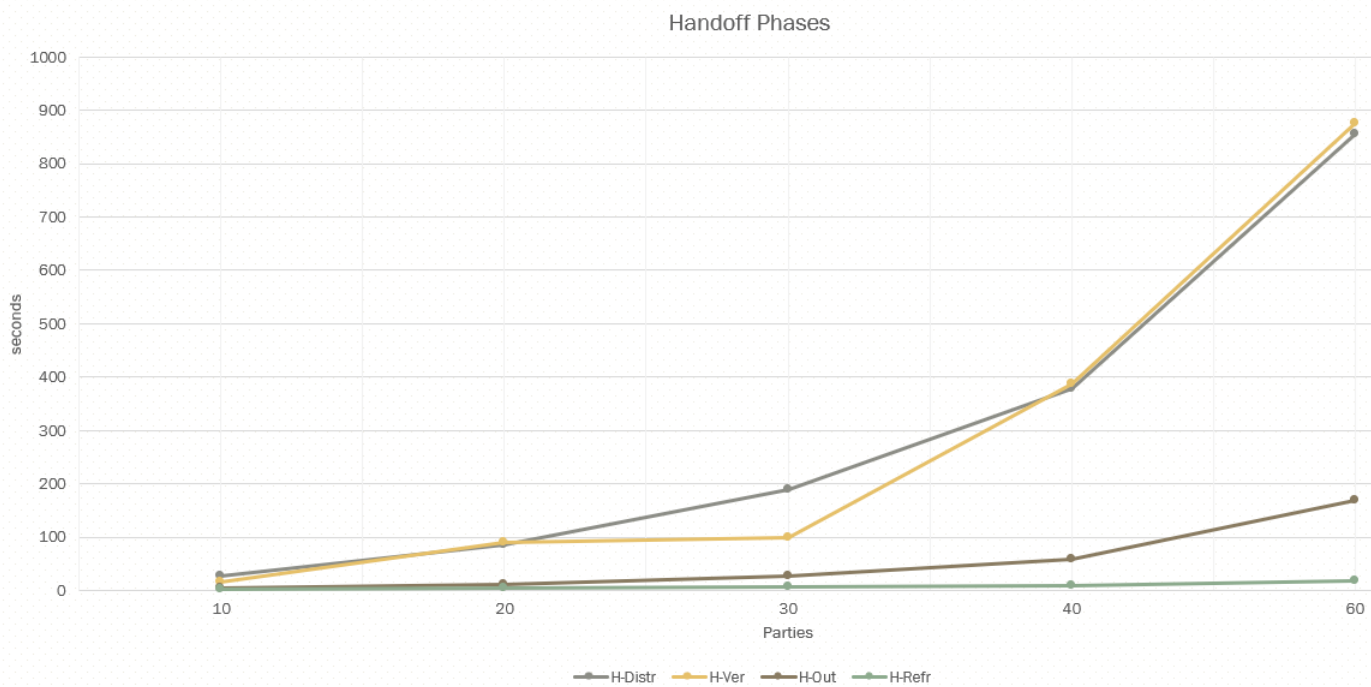


Figure 10: Time for Hand-off Phase



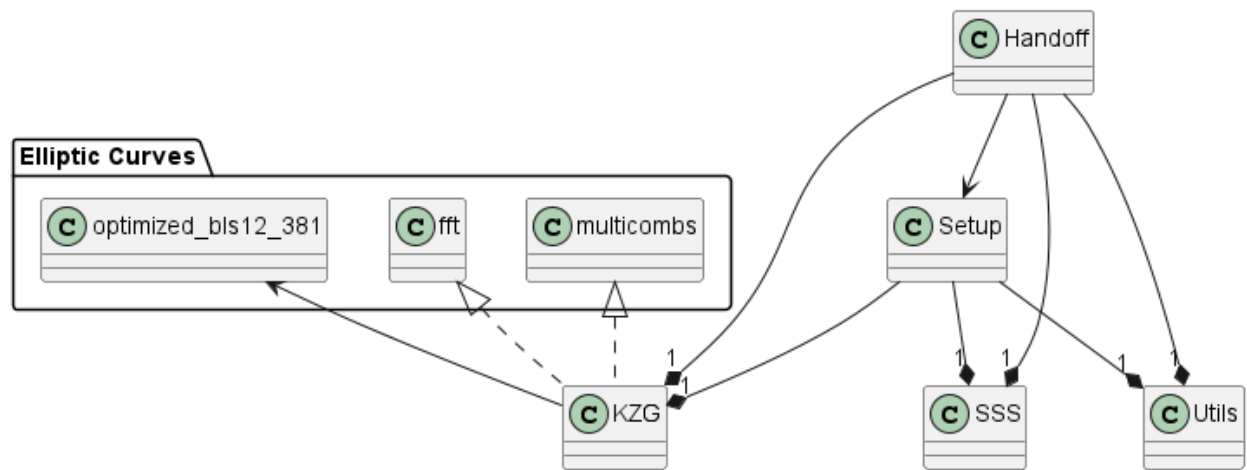


Figure 11: Relationship between the classes


C Utils	
●	vanderMatrix(N:Integer,t:Integer):Integer[]
●	computeVanderElem(vander:Integer[],shares:Array<Tuple>):Integer[]
●	challenge(parties:Integer):Integer[]
■	fiatShamir(PK:Array<Array>,commitment:Tuple):String


C Setup	
○	validations:Array<Tuple>
○	parties:Integer
○	threshold:Integer
○	commit:Dictionary
●	distribution():Dictionary
●	accusation():Boolean
●	verification(committee:Dictionary):Boolean
●	output(committee:Dictionary):Tuple<Dictionary>
●	fresh(r:Dictionary, phy:Dictionary, commitments:Dictionary, proof:Dictionary):Tuple<Dictionary>
●	computeWitness(committee:Dictionary, i:Integer, x:Integer[], rootOfUnity:Integer):Tuple<Dictionary,Integer>
●	validateShares(committee:Dictionary, commit:Dictionary, i:Integer):Boolean
●	outputCommitments(committee:Dictionary,i:Integer):Tuple<Dictionary,Integer>
●	validate(proof:Dictionary, commitments:Dictionary, j:Integer):Boolean
●	main()

C Handoff	
○	nValidations:Array<Tuple>
○	oValidations:Array<Tuple>
○	commit:Dictionary
○	parties:Integer
○	threshold:Integer
●	distribution():Tuple<Dictionary>
●	accusation():Boolean
●	verification(committeeNew:Dictionary,committeeOld:Dictionary):Boolean
●	output(committeeNew:Dictionary,committeeOld:Dictionary):Tuple<Dictionary>
●	refresh(r, phy, s, z, rTilde, phyTilde, szWitnesses, szCommitments, rphyWitnesses, rphyCommitments):Tuple<Dictionary>
●	reconstruction(shares:Integer[],commitments:Dictionary,proofs:Dictionary):Integer
●	computeProofs(committee:Dictionary, i:Integer, x1:Integer[],x2:Integer[], rootOfUnity:Integer):Tuple<Dictionary,Integer>
●	validateNewCommittee(committee:Dictionary, commit:Dictionary, i:Integer):Boolean
●	validateOldCommittee(committee:Dictionary, commit:Dictionary, i:Integer):Boolean
●	outputCommitmentsOld(committee:Dictionary,i:Integer):Tuple<Dictionary,Integer>
●	outputCommitmentsNew(committee:Dictionary,i:Integer):Tuple<Dictionary,Integer>
●	validateKing(proof:Dictionary, commitments:Dictionary, i:Integer):Boolean
●	validateClient(proofs:Dictionary,commitments:Dictionary,i:Integer):Boolean


Figure 12: Local View-01 of the System

## Elliptic Curves

 optimized\_bls12\_381

 multicombs

 fft

 SSS

□ n : Integer  
□ k : Integer  
□ p : Integer

● init(n:Integer,k:Integer)  
■ polynomial(s:Integer): Integer[]  
● splitSecret(s:Integer,polyCoeff:Integer[]): Array<Tuple>  
● reconstruct(shares:Array<Tuple>):Integer  
● coupledSharing():Array<Tuple>  
● sampleSharing():Array<Tuple>  
■ interpolate(x:Integer[],y:Integer[]):Integer  
● getPoly(shares:Array<Tuple>):Integer[]

 KZG

□ primitiveRoot:Integer  
□ modulus:Integer

● generateSetup(s:Integer,size:Integer)Array<Array>  
■ getRootOfUnity(order:Integer):Integer  
● commitToPoly(polynomial:Integer[],randPoly:Integer[],setup:Array<Array>): Tuple  
● computeProofSingle(polynomial:Integer[],randPoly:Integer[],x:integer,setup:Array<Array>): Tuple  
● checkProofSingle(commitment: Tuple,proof: Tuple,ys:Integer[],ys\_r:Integer[],x:integer,setup:Array<Array>): Boolean  
● computeProofMulti(polynomial:Integer[],randPoly:Integer[],x:integer,setup:Array<Array>): Tuple  
● checkProofMulti(commitment: Tuple,proof: Tuple,ys:Integer[],ys\_r:Integer[],x:Integer,setup:Array<Array>): Boolean

Figure 13: Local View-02 of the System