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

Sunday 4th December, 2022 - 22:29

Georgi Tyufekchiev
University of Luxembourg
Email: 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
Email: 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 paper 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 someones 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 [5]. Such schemes also provide a solution to the ”Byzantines Generals Problems”[9]. 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* [6] 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 [7], 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 [10] (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 [8] can be found in zero knowledge proofs. They are used to store a large amount of data into elliptic curve points. The goal is to prove to a verifier that some computations have taken place, without him redoing all the computations. 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 paring 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, threading and networking.

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 posses 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 let 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 $a_1, .., a_{k-1} \in F_q$
- Evaluate the polynomial $f(x) = a_0 + a_1 x + ... + a_{k-1}x^{k-1}$ at points $i \in \{1, ..., n\}$ 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

Today the topic of blockchain is very popular because of concepts like Web3 and crypto-coins. Many heavy computations happen and a lot of data is stored. Usually the node has to prove that it did all of those computations. However, recomputing everything and sending it to a verifies is expensive and inefficient. In that case the node can just "commit" the data into a single elliptic curve point, which will be used as part of the verifying process. But why is all of this needed, when one can just hash the data? In that case, the values would still need to be transmitted, which is not what

we want. For this reason, we need polynomial commitments. Even when the data is compressed to a single elliptic curve point, the original data remains "separated" and we can do mathematical operations on it, without violating the integrity of the data. 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 $e : GxG \rightarrow G_T$ 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 α and the public key PK is $(G, G_T, e, g, g^{\alpha}, g^{\alpha^2}, ..., g^{\alpha^t}, h, h^{\alpha}, h^{\alpha^2}, ..., h^{\alpha^t})$. The SK is not needed of the rest of the construction.
- **Commit**$(PK, f())$: Computes the commitment $Com = g^{f(a)}h^{r(i)}$ using the PK, where $r() \in Z_p$ 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 $g^{f(a)}h^{r(i)}$ 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, intersection both axis and is 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 trough 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 $P + P + P + ... + P$, 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.

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 $p, qorr$, but that is not the case. Thanks to the computational Diffie Hellman problem and the discrete logarithm problem, this leakage remain 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 Fq^2 where the equation becomes $y^2 = x^3 + 4(1 + i)$.

4.4. DPSS

Now we will look at the DPSS defined in [7]. 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, and since the each sharing is a polynomial by itself, the party can execute the **Commit**$(PK, f())$ with $f()$ being the first share. The random polynomial 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 failed to validate a share, they can accuse the party that sent it. For each accusation, the party will publish the shares and witnesses, which they sent. All parties will check the validity of the shares and if the check fails, the party is regarded as corrupted. Otherwise, the accusing party uses the shares and witness, which were sent to them.
- **Verification.** The parties need to check that the commitments from the Setup protocol were computed correctly. Since opening the polynomials leaks the generated shares, they generate two additional ones, which will be used to mask the others. 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 this phase we have a new set of parties, to whom the secret will be transferred, meaning we have $2n$ parties 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, from which the first pair is the same share and second pair is also the same but different from the first. 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

In this section we provide benchmarking of our implementation of the DPSS system. We evaluate only the case, when the parties are acting honest. Since we do not have a network, it is important to know that most of the code is sequentially executed, meaning in some parts of the system, one party has to wait for the previous one 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 quadraticaly. 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.

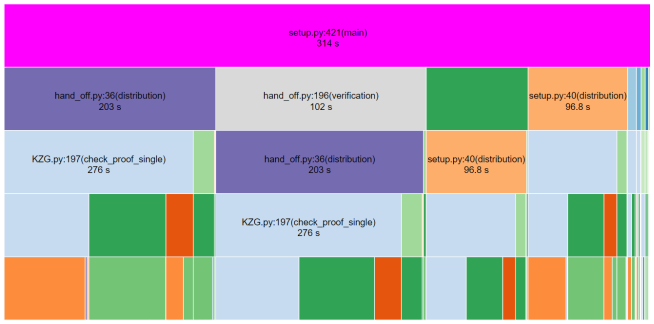


Figure 1: Fully Sequential Execution - 10 parties

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 he 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 $3x$ was achieved.

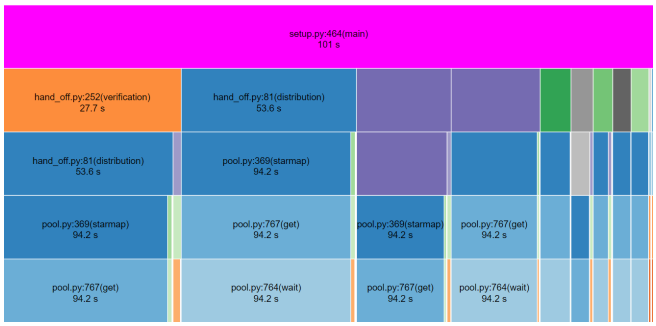
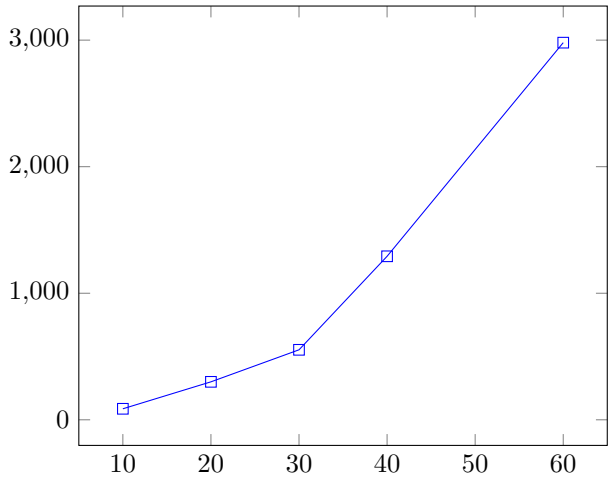


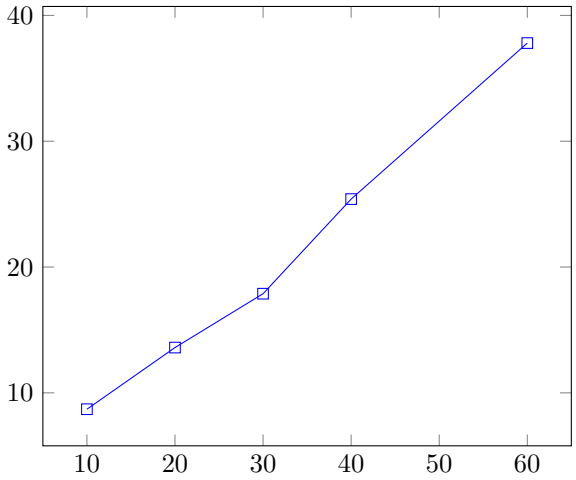
Figure 2: Partially Sequential Execution - 10 parties

Now we can provide a more detailed benchmarking.

The figure below 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, which is logical when we take into account that the code is still executed sequentially. However, these numbers do not represent the time it takes for one party to finish their computations. The figure below shows the time per party.



Next, we look at how each phase performed.

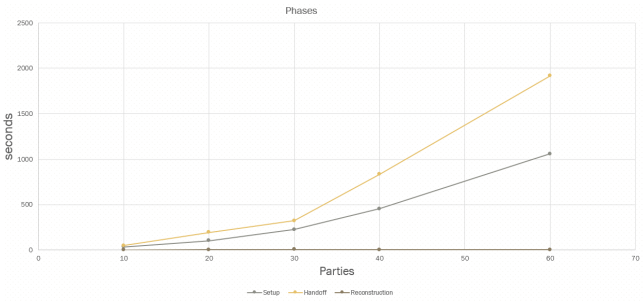


Figure 3: Time per Phase

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 this graph we can clearly see the

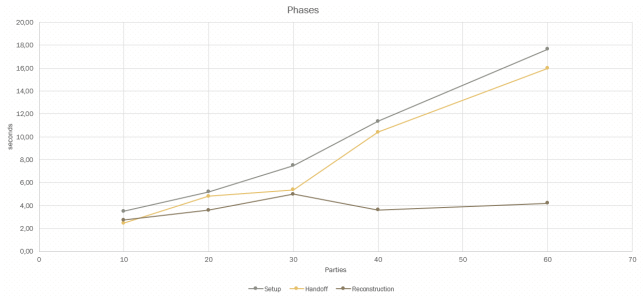


Figure 4: Time per Phase per Party

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.

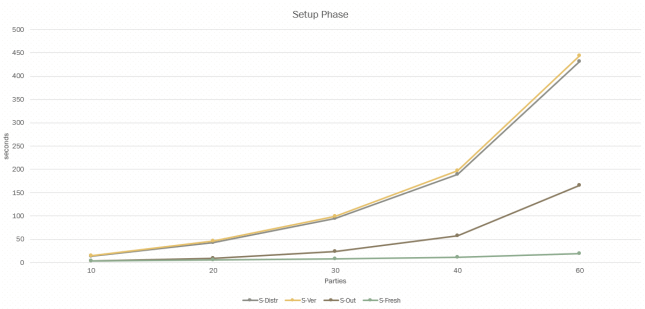


Figure 5: Time for Setup Phase

On the graph 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 and witnesses and the validation of shares, which become heavier the more parties there are. The Fresh protocol is almost constant complexity.

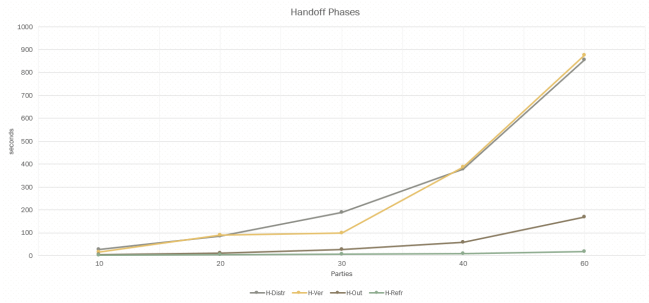


Figure 6: Time for Hand-off Phase

Again the distribution and verification protocol are the most time consuming. The output and refresh protocols are almost identical to those of the setup phase.

4.6. Assessment (± 15% of section's words)

Provide any objective elements to assess that your deliverables do or do not satisfy the requirements described above.

5. A Technical Deliverable 1

For each technical deliverable targeted in section 2.2 provide a full section with all the subsections described below. The cumulative volume of all deliverable sections represents 75% of the paper's volume in words. Volumes below are indicated relative the the section.

5.1. Requirements (± 15% of section's words)

cf. section 5 applied to the technical deliverable

5.2. Design (± 30% of section's words)

cf. section 5 applied to the technical deliverable

5.3. Production (± 40% of section's words)

cf. section 5 applied to the technical deliverable

5.4. Assessment (± 15% of section's words)

cf. section 5 applied to the technical deliverable

Acknowledgment

The authors would like to thank the BiCS management and education team for the amazing work done.

6. Conclusion

The conclusion goes here.

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.

[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] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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/.

[10] 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. Appendix

All images and additional material go there.

8.1. Source Code

The following environment shows the correct and mandatory way to insert your code.

Listing 1: Caption example.

```
import numpy as np

def incmatrix(genl1,genl2):
    m = len(genl1)
    n = len(genl2)
    M = None #to become the incidence matrix
    VT = np.zeros((n*m,1), int) #dummy variable

    #compute the bitwise xor matrix
    M1 = bitxormatrix(genl1)
    M2 = np.triu(bitxormatrix(genl2),1)

    for i in range(m-1):
        for j in range(i+1, m):
            [r,c] = np.where(M2 == M1[i,j])
            for k in range(len(r)):
                VT[(i)*n + r[k]] = 1;
                VT[(i)*n + c[k]] = 1;
                VT[(j)*n + r[k]] = 1;
                VT[(j)*n + c[k]] = 1;

                if M is None:
                    M = np.copy(VT)
                else:
                    M = np.concatenate((M, VT), 1)

                VT = np.zeros((n*m,1), int)

    return M
```