Implementation of Privacy-Preserving Computation Techniques

Priyanshu Gupta
19750534
Georg-August-University in Göttingen
Göttingen, Germany
priyanshu.gupta@stud.uni-goettingen.de

Claas Kochanke
22646052
Georg-August-University in Göttingen
Göttingen, Germany
c.kochanke@stud.uni-goettingen.de

***Abstract*—This report investigates the implementation of Privacy-Preserving Computation (PPC) techniques, which aim to enable secure data processing without exposing sensitive information to third parties. Three different techniques, Partial Homomorphic Encryption (PHE), Secure Multi-Party Computation (SMPC), and Zero-Knowledge Proofs (ZKP), are analyzed and implemented in a multi-server architecture developed in Python using Flask. The report examines the performance, limitations, and benefits of these techniques.**

Homomorphic encryption allows operations on encrypted data, but introduces significant performance and memory overhead, limiting its practical use for large datasets. In SMPC, various libraries were explored, but challenges arose due to outdated documentation. Shamir's Secret Sharing was successfully implemented for secret distribution and reconstruction across bank servers, enabling decentralized decision-making. Finally, the ZKP implementation employed the Schnorr protocol for secure authentication, integrated with JWT, but was hindered by computational overhead compared to standard authentication methods. The report concludes that while these techniques provide strong privacy guarantees, further research is required to improve their performance and scalability for real-world applications.

***Index Terms*—Privacy-Preserving Computation, Homomorphic Encryption, Secure multi-party Computation, Shamir's Secret Sharing, Zero-knowledge Proofs, Schnorr protocol**

I. Introduction [CK]

The use of cloud computing services has increased significantly worldwide in recent years. Companies as well as individuals are increasingly using these services to outsource their computing workloads and reduce costs. These Services offer numerous advantages, but there are also growing concerns about data protection. Sensitive data is placed in the hands of cloud providers, creating potential security risks. This creates a need for techniques that allow secure data processing without granting access to these third party cloud providers. Traditional approaches only secure the transport of the data, but the data must be decrypted before processing.

This report investigates the implementation of Privacy-Preserving Computation (PPC) techniques, which can be a promising solution to this problem. As part of a proof-of-concept project, three PPC techniques were analysed: partial Homomorphic Encryption (PHE), Zero-Knowledge Proof (ZKP) and Secure Multi-Party Computation (SMPC). For this purpose, a server architecture has been developed in Python using Flask to secure communication between different parties. This architecture makes it possible to test the above techniques in a real application and evaluate their performance. Thus the report provides an insight into PPC techniques and their application.

A. Report Structure [CK]

In Section II, the report introduces the fundamental concepts of Privacy-Preserving Computation (PPC), explaining the term and outlining the techniques covered in this report. It also provides the theoretical foundations and basics of these methods. Section III presents a high-level overview of the architecture implemented in the example project, offering insights into its design and structure.

The following three sections dive into the specific technical implementations of each PPC technique. Beginning with Section IV, the report details the implementation of homomorphic encryption, highlights the challenges faced during development, and justifies the choice of libraries used. Section V focuses on the implementation of Secure Multi-Party Computation (SMPC), discussing the encountered problems and how they were overcome. In Section VI, the implementation of zero-knowledge proofs is examined, including the chosen algorithm and a breakdown of how the implementation functions.

Section VII provides a comprehensive evaluation of the implemented techniques, summarizing key insights, assessing their practical usability, and reviewing overall findings. Finally, Section VIII provides a summary of the report's findings and reflections on the project as a whole. It also discusses potential directions for future research in this area.

II. Privacy-Preserving Computation [CK]

Privacy-Preserving Computation (PPC) refers to all techniques that allow for computations on data without exposing the underlying information. This enables secure data processing even if third parties are involved. PPC is particularly useful in areas where data is highly sensitive. Examples include healthcare, finance and cloud computing.
There are many different PPC techniques. This report focuses on three key techniques.

- **Homomorphic Encrytion**: It allows for computation on encrypted data.

- **Secure Multi-Party Computation (SMPC)**: Data is split among multiple parties for collaborative processing.
- **Zero-Knowledge Proofs**: It Validates a statement's truth without revealing the data itself

These techniques are implemented together in a project to provide an easy to understand example of the use of PPC techniques. For better comprehension, these techniques will be described in more detail, albeit briefly, in the following subchapters.

A. *Homomorphic Encryption [CK]*

Homomorphic encryption allows for computation on encrypted data without the need to decrypt it first. This property is crucial for privacy-preserving computations in scenarios where sensitive data must remain confidential. An example of an application would be to send data to a cloud provider who then performs calculations on it, but does not need to decrypt the data, so confidentiality can be maintained.

In general, homomorphic encryption can be divided into partial homomorphic encryption (PHE) and full homomorphic encryption (FHE). PHE allows only certain types of operations on encrypted data, whereas FHE enables an unlimited number of operations. In practice, there are implementations that are nearly FHE but still come with certain limitations. The field of homomorphic encryption is actively researched, and it is expected to take some time before a fully practical implementation of FHE becomes available.

A common example of homomorphic encryption uses RSA, which is known to be multiplicative homomorphic, meaning it is a partially homomorphic encryption scheme. Consider two numbers a and b, with their encrypted forms represented as $E(a) = a^e mod n$ and $E(b) = b^e mod n$, where e is the public exponent and n is the modulus. When these two encrypted values are multiplied, the following equation holds:

$$E(a) \cdot E(b) = (a^e \mod n) \cdot (b^e \mod n) = (a \cdot b)^e \mod n \tag{1}$$

The product of $E(a) \cdot E(b)$ is equivalent to $E(a \cdot b)$. Decrypting this result yields $a \cdot b$ demonstrating that RSA is multiplicative homomorph. [1] This property is a feature in certain applications of homomorphic encryption, although RSA is limited to supporting only multiplicative operations.

B. *Secure Multi-Party Computation [PG]*

Secure Multi-Party Computation (SMPC) is a cryptographic method that enables several parties to collaboratively compute a function based on their respective inputs while maintaining the confidentiality of those inputs. The primary objective of SMPC is to facilitate computations across distributed data without disclosing individual inputs to any participant. This approach finds extensive application in fields where privacy and security are paramount, including finance, healthcare, and secure voting systems.

SMPC allows multiple entities to collaborate on a computational problem while ensuring that no single party gains access to the private data of the others. This ensures confidentiality and privacy, making SMPC a valuable tool for distributed computation across untrusted entities. The basic principle behind SMPC involves breaking down data into encrypted or obfuscated shares, which are distributed among parties. These parties jointly perform the computation on their respective shares without needing to reveal the actual data. [2]

C. *Zero-Knowledge Proof [PG]*

Zero-Knowledge Proofs (ZKP) are crucial for modern authentication security in IT and application development. They enable a prover to demonstrate knowledge of a secret, such as a password, without revealing the secret itself. This capability ensures that authentication, authorization, and accounting (AAA) operations are secure while keeping private information confidential.

The core principle behind ZKP is the Elliptic Curve Discrete Logarithm Problem (ECDLP). In elliptic curve cryptography, ECDLP involves determining the scalar multiple of a point on an elliptic curve given the result of that multiplication. This mathematical problem underpins the security of various ZKP protocols. [3]

III. System Architecture [CK]

The system approach implemented in this project is based on a multi-server architecture that was developed using Python and Flask to implement various privacy-preserving computation techniques. Communication between the various parties takes place via HTTP requests. Figure 1 shows a high-level view of the architecture.

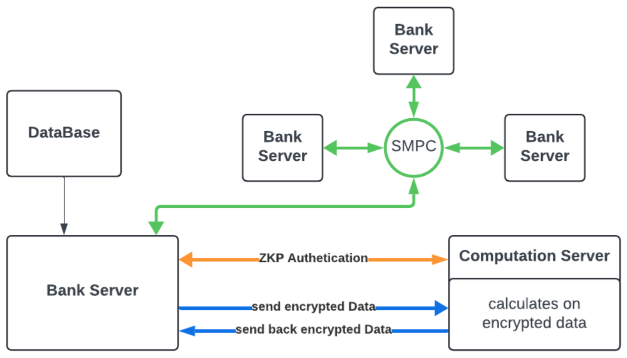
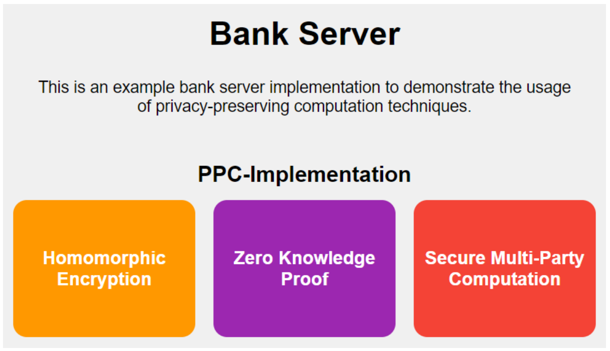


Fig. 1: Architecture of the multi server implementation

A central component of the system is the bank server, which sends encrypted data to a computation server, for example a cloud server such as Amazon AWS. The computation server performs calculations on this encrypted data and sends the result back to the bank server. The bank server decrypts the data and receives the correct result without the computation server ever having access to the data in plain text. As shown in Figure 1, only the bank server has access to the database;

the computation server does not have direct access to it at any time. In addition, a proof-of-concept for zero-knowledge proof authentication was implemented in which the bank server authorizes itself with the computation server. This approach was deliberately implemented in isolation from other procedures to ensure simplicity and traceability. Another component of the system is Secure Multi-Party Computation (SMPC), which is based on Shamir's Secret Sharing. Here, a secret is divided into several parts, for example six, and distributed to different parties. A threshold value determines how many parts are required to restore the original secret, for example a key. This method ensures that a transaction can only be carried out if enough parties work together to reconstruct the key.

A graphical user interface (GUI) was developed for both the bank server and the compute server to provide real-time monitoring and visualization of operations. This interface allows users to view data and gain a clearer understanding of the processes occurring behind the scenes. Figure 2 illustrates the GUI for the bank server, which displays the various privacy-preserving computation (PPC) methods implemented. Users can select each method to reveal a custom interface specific to that PPC technique. Figure 3 shows the compute server's GUI, where incoming requests and their data are displayed. An interactive GUI was developed for each PPC technique, allowing users to trace the processes and better understand how each method functions.


Bank Server

This is an example bank server implementation to demonstrate the usage of privacy-preserving computation techniques.

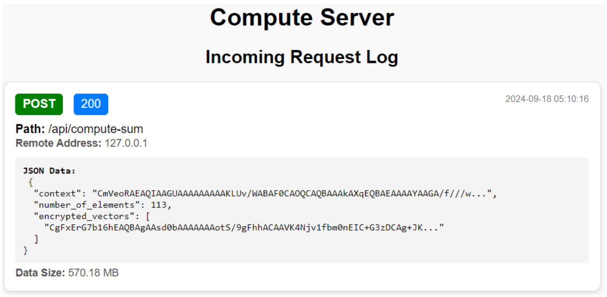
PPC-Implementation

Fig. 2: GUI of the bank server

IV. Homomorphic Encryption [CK]

To implement homomorphic encryption, the first step is to choose an appropriate library. There are many available options, but two prominent ones stood out. These are outlined below:

1) **pyfhel PY**thon **F**or **H**omomorphic **E**ncryption **L**ibraries implements various homomorphic encryption operations in python. It is build on top of Afhel, an Abstraction for Homomorphic Encryption Libraries in C++, designed to serves as a common API across different backends.


Compute Server

Incoming Request Log

POST 200 2024-09-18 05:10:16

Path: /api/compute-sum
Remote Address: 127.0.0.1

```
JSON Data:
 {
  "context": "CmVeoRAEAQIAAGUAAAAAAAAAKLUv/WABAF0CAOQCAQBAAAkAXqEQBAEAAAAYAAGA/f///w...",
  "number_of_elements": 113,
  "encrypted_vectors": [
    "CgFxErG7b16hEAQBAgAAsd0bAAAAAAAotS/9gFhhACAAVK4Njv1fbm0nEIC+G3zDCAg+JK..."
  ]
}
```

Data Size: 570.18 MB

Fig. 3: GUI of the compute server

These backends include the underlying encryption libraries SEAL and PALISADE. [4]

2) **TenSEAL** TenSEAL is a library designed for homomorphic encryption operations on tensors and it uses Microsoft SEAL as a backend. It offers a user-friendly Python API and performs most of its operations using C++ for efficiency. [5]

Both libraries are based on the open-source, regularly updated, and well-documented homomorphic encryption library Microsoft SEAL. In the analysis of basic code examples, no significant functional differences were observed between the two. However, TenSEAL proved slightly easier to use, making it the preferred choice for the current project. In future projects that focus only on homomorphic encryption, it may be worth exploring the differences between these two libraries in more detail. Since both rely on Microsoft SEAL, it is unlikely that the differences will be significant.

A. Selecting TenSEAL Parameters

In homomorphic encryption, it is important to distinguish between integer and floating-point numbers. The BFV scheme is used for integers, while CKKS is for floating-point numbers. BFV generally provides higher accuracy but is slower, whereas CKKS is faster but introduces more noise, reducing precision. In this example, a banking system with two decimal places was used, so values were multiplied by 100 to convert them to integers and achieve higher accuracy by using the BFV scheme. Additionally two key paramters must be carefully selected when working with TenSEAL:

- poly modulus degree
- plain modulus

The poly modulus degree defines the maximum size of a ciphertext, impacting both security and performance. It must be a power of two, starting at 4096. Larger values provide stronger security but decrease performance and require more memory. The maximum number of elements that can fit in a ciphertext is $\frac{poly\_modulus\_degree}{2}$. A poly modulus degree of 8192 is often recommended as a balanced trade-off between security and performance.

The plain modulus, specific to the BFV scheme, defines the range of values that can be encrypted. It should be set high enough to accommodate the largest values but kept as low as possible. Larger values lead to increased noise and reduced accuracy. For this example, a plain modulus over 500 million was selected, allowing encoding of values up to 5 million, which is sufficient for the application. [5]

More detailed benchmarks on the influence and selection of these parameters are available on the TenSEAL GitHub repository [6].

B. Implementation

In this project, a multi-server architecture was implemented, as discussed in Section III. The central component of this system is a bank server, which manages a database containing user information and transaction records. This bank server utilizes external computational resources, such as a cloud provider, to perform its operations. In the proof of concept design, a Flask server was implemented to handle the computations.

The project is organized into several directories. The ”Bank_Server” directory contains the code for the bank server, with the ”homomorphic_enc” folder housing the implementation of homomorphic encryption. The ”Computation_Server” folder includes the code for the ”external” server responsible for performing computations using homomorphic encryption. The proof of concept system is designed to be easily extendable, allowing for the addition of more operations in the future.

A route has been implemented for summing transactions, which can be extended to support other types of computations. The computation process is as follows:

1) The bank server retrieves the necessary data from its database.
2) It initializes a TenSEAL context, which holds the required keys for encryption.
3) The bank server encrypts customer transaction data by dividing it into batches, followed by encrypting each batch individually.
4) The context, encrypted data, and the number of encrypted elements are serialized.
5) These serialized components are sent via JSON to the computation server.
6) The computation server performs the sum operation on the encrypted data.
7) The encrypted result is returned to the bank server.
8) Finally, the bank server decrypts the result.

While this example focuses on summing transactions, the implementation can be easily adapted for other operations, such as computing averages or adding and subtracting scalars. Sending the number of encrypted elements is optional, but it can be useful for specific operations like calculating averages. Since the underlying concept remains largely the same across different operations, no additional operations were implemented beyond the initial demonstration.

C. Performance analysis

During the development of the server architecture and testing with the TenSEAL library, it became clear that only a limited number of operations are supported. Since the data is encrypted, filtering or selectively applying operations to specific elements is not possible. TenSEAL encrypts data in batches, though it is possible to encrypt each piece of data individually. However, encrypting data individually results in extremely poor performance. Additionally, each TenSEAL-encrypted vector maintains the same size, regardless of whether it contains only a few elements or the maximum number.

Following the implementation, significant delays were observed, with computations sometimes taking over 12 seconds, even for a few hundred numbers. This led to a close look into the performance of the homomorphic encryption library. A detailed Jupyter notebook was created for analysis, which included various performance graphs. Operations were tested on datasets containing tens of thousands of elements, as using smaller datasets would defeat the purpose of outsourcing data to an external cloud provider to save on compute costs. The key findings from this analysis are summarized below. All measurements were conducted on a system equipped with a Ryzen 5800X 8-Core processor and 64GB of RAM.”

1) TenSEAL Sum Operation: The sum operation involves first summing all elements of a single encrypted vector, followed by summing the results from each encrypted vector, as the data is split into batches when dealing with large amounts of numbers. Figure 4 illustrates the performance of each step in the sum operation compared to NumPy. While NumPy can sum an array of numbers in less than a millisecond, TenSEAL requires 212 ms, which is several thousand times slower. The figure also shows that while encryption time increases, the decryption time remains constant, as the encrypted vectors are reduced to a single result.

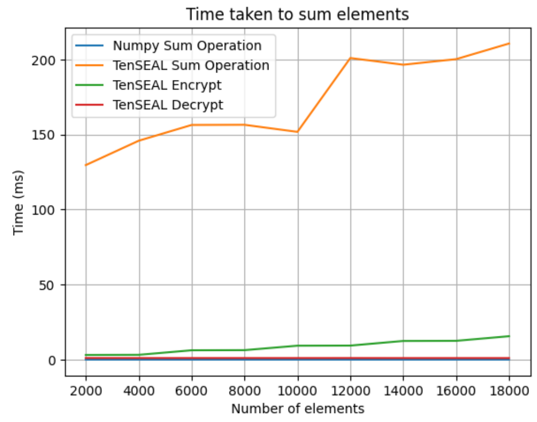


Fig. 4: Performance of TenSEAL sum operation

2) TenSEAL Additional Operations: When examining additional operations, Figure 5 shows that multiplication in TenSEAL is faster than both addition and subtraction. However, compared to NumPy, which operates at near-zero time, TenSEAL is significantly slower.

3) TenSEAL Complex Operations: When performing more complex operations, such as squaring each element of a vector

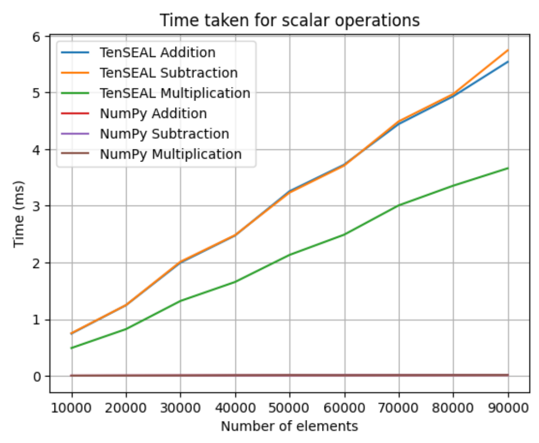


Fig. 5: Performance of simple Operations in TenSEAL

or summing all elements in a vector, performance significantly decreases. Figure 6 TenSEAL requires almost 800 ms to sum 90,000 elements, while NumPY completes the operation in less than a millisecond.

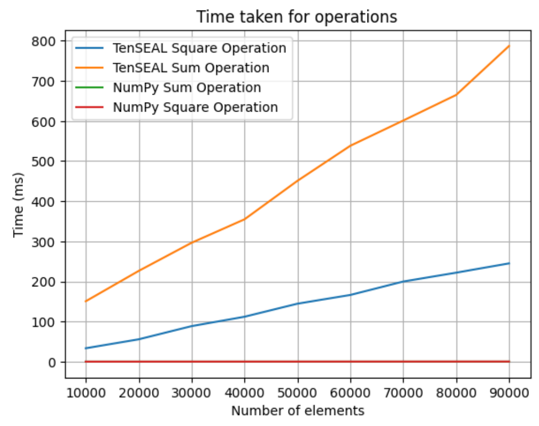


Fig. 6: Performance of Complex Operations in TenSEAL

4) TenSEAL Memory Overhead: In addition to the performance decrease, homomorphic encryption also significantly increases the memory required to store vectors compared to plain NumPy arrays. As mentioned earlier, the memory usage depends on the polynomial modulus degree. The higher the degree, the more memory is needed. Figure 7 illustrates how different modulus degrees affect memory consumption, with up to 140 MB required to store 500,000 elements.

The TenSEAL context contains all the cryptographic parameters that define the characteristics of homomorphic encryption operations. It includes all the essential components such as the encryption scheme, polynomial modulus degree, and encryption keys, with the option to include only the public keys if necessary. This context must be transmitted to the computation server at least once and whenever any changes occur, ensuring the server can perform operations on the encrypted data effectively. With increasing size of the poly modulus degree the context grows significantly in size up to 567 MB as shown in Table I.

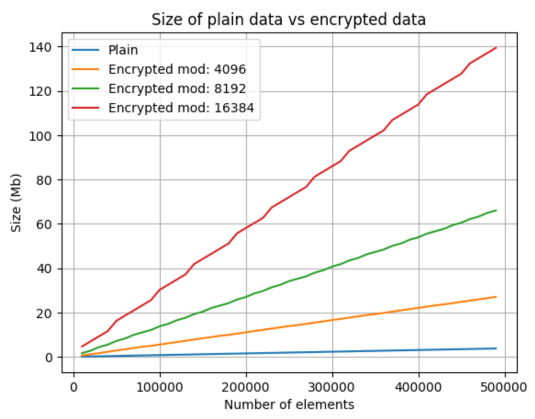


Fig. 7: Size of Encrypted data in TenSEAL

TABLE I: Size of Context for Different Poly Modulus Degree Values

Poly Modulus Degree	Size of Context (MB)
4096	8.2419
8192	69.6447
16384	567.8529

D. *Practicality*

Homomorphic encryption, in its current state, only supports a limited range of operations and is significantly slower, as shown in the previous chapter. Performance was found to be more than a thousand times slower than traditional methods. Additionally, it requires much more memory. Based on the research, homomorphic encryption does not yet appear feasible for use in real-world applications. However, it remains an area of active research. While significant progress has been made in recent years, further improvements are necessary before it can become practically usable.

V. Secure multi-party Computation [PG]

In exploring SMPC, attempts were made to implement the technique using popular Python libraries such as Crypten (developed by Facebook), PySyft, and MPYC. These libraries, designed for privacy-preserving computation, presented several limitations:

1) **Crypten:** Crypten [7] is intended for secure and private machine learning. However, the documentation was sparse and lacked resources for practical use cases. Although computations on encrypted data were possible, the computation server could decrypt the data, thereby violating the core principle of SMPC. This represents a significant limitation in ensuring privacy during the computation.
2) **PySyft:** PySyft [8] is another Python library designed for privacy-preserving, federated learning, and SMPC. However, similar to Crypten, the library faced issues

with insufficient documentation for the specific use case. Older example implementations were no longer functional due to changes in dependencies, and compatibility issues arose with newer versions of the library. Despite its potential, PySyft's limitations in practical applications for the SMPC use case necessitated the exploration of alternative solutions.

3) **MPYC:** MPYC [9] is a library focused on secure multi-party computation based on the MP-SPDZ framework. However, the heavy reliance on multi-threading in MPYC's architectural design hindered its use in the implementation. The documentation was convoluted and not directly applicable to the client-server architecture. The lack of dedicated solutions for the use case and the complexity of its multi-threaded approach rendered it infeasible for implementation.

A. *Attempt to Solve Millionaire's Problem*

To address the limitations of existing libraries, an implementation of the Millionaire's Problem was explored. This classic cryptographic challenge involves two parties determining which is wealthier without disclosing their actual wealth. Initial efforts referenced a GitHub solution and Yao's 1982 research paper [10], which presented secure multi-party computation protocols utilizing techniques such as garbled circuits for secure comparisons.

The GitHub implementation explored for solving the Millionaire's Problem uses RSA encryption. Each party generates an RSA key pair: the public key encrypts a random number, while the private key is used for decryption. One party encrypts a random number, adjusts it by subtracting their wealth, and sends the adjusted value to the other party. The receiving party decrypts this value and generates a set of modified values by adding a range of integers. These values are transformed using a prime number to facilitate comparison. The final step involves determining if one party's wealth exceeds the other's based on these transformed values.

Despite following this theoretical approach, the outdated GitHub code faced compatibility issues with modern libraries and dependencies. Bugs related to ciphertext manipulation and issues with handling encrypted values as integers led to unresolved problems. Even after updating the comparison logic and dependencies, the implementation did not successfully resolve the problem due to these persistent issues.

B. *Shamir's secret sharing Theory*

Shamir's Secret Sharing (SSS) is a cryptographic algorithm developed by Adi Shamir in 1979. It is designed to divide a secret into multiple shares such that only a specified number of shares are required to reconstruct the original secret. This technique is based on polynomial interpolation and offers robust security through distributed secret management. It finds application in various domains such as secure key management, decentralized data storage or distribution, and multi-party computation. The use case for Shamir's Secret Sharing involves distributing shares to multiple banks, where a decision or transaction is validated only if a threshold number of banks agree. This approach ensures that critical decisions or sensitive transactions require consensus among multiple participants, rather than relying on a single entity. It is particularly useful for secure multi-party computations and collaborative decision-making. [11]

C. *Implementation*

The implementation features a client-server architecture with the following components:

- **Main Bank Server:** This server is responsible for splitting the secret into n shares, where n is specified by the client. It also determines the threshold k, the minimum number of shares required to reconstruct the secret.
- **Bank Servers:** The main bank distributes the generated shares to multiple bank servers running on different ports. Each bank server stores and manages one or more shares.
- **Client Side:** The client specifies the number of shares n and the threshold k. The client interacts with the main bank to request the splitting of the secret and later retrieves the shares for reconstruction.

1) Secret Splitting: The main bank server performs the following steps to split a secret into shares:

a. The secret S is embedded in a polynomial of degree $k-1$:

$$f(x) = S + a_1 x + a_2 x^2 + \cdots + a_{k-1} x^{k-1} \tag{2}$$

where $a_1, a_2, \ldots, a_{k-1}$ are randomly chosen coefficients, and x represents the share index.

b. Each share (x_i, y_i) is computed by evaluating the polynomial at x_i:

$$y_i = f(x_i) \mod p \tag{3}$$

where p is a large prime number($2^{127} - 1$) used as a modulus to ensure operations are performed within a finite field.

c. The computed shares are distributed among smaller bank servers, each running on different ports.

2) Share Collection: To reconstruct the secret, the following steps are taken:

a. A client requests shares from the bank servers via GUI. Each server responds with its share.

b. When the number of collected shares meets or exceeds the threshold k, the secret can be reconstructed using Lagrange interpolation:

$$S = \sum_{i=1}^{k} y_i \cdot \prod_{\substack{1 \le j \le k \\ j \ne i}} \frac{x_j}{x_j - x_i} \mod p \tag{4}$$

where x_i and y_i are the coordinates of the shares used for interpolation.

In summary, Shamir's Secret Sharing provides a secure and efficient method for distributing and reconstructing secrets through polynomial interpolation. The client-server implementation effectively demonstrates the practical application of this cryptographic technique, ensuring secure management and reconstruction of secrets when necessary.

VI. Zero-Knowledge Proof [PG]

In this use case, a ZKP-based authentication system is implemented to ensure secure communication between a main bank server and a computation server.

Several ZKP protocols offer distinct advantages and trade-offs. Three notable protocols evaluated include zk-SNARKs, zk-STARKs, and the Schnorr protocol.

1) **zk-SNARKs:** Zero-Knowledge Succinct Non-Interactive Arguments of Knowledge are efficient and non-interactive, commonly used in blockchain applications like Zcash. However, they require a trusted setup, which may not be ideal for real-time, lightweight authentication scenarios.
2) **zk-STARKs:** Zero-Knowledge Scalable Transparent Arguments of Knowledge are quantum-resistant and scalable, without requiring a trusted setup. Despite their enhanced security, zk-STARKs are computationally intensive and less suitable for scenarios demanding minimal latency.
3) **Schnorr Protocol:** This interactive ZKP system, based on the discrete logarithm problem, is lightweight and efficient. Its simplicity and effectiveness make it well-suited for real-time applications, such as authentication systems. For the use case, the Schnorr protocol authenticates communication between the main bank server and the computation server, ensuring secure and efficient interactions while keeping client information confidential.

A. Implementation

The Schnorr protocol is used for authenticating the main bank server, ensuring secure communication with the computation server. Implementation details are derived from the research by Ioannis Chatzigiannakis et al. [3]. The `ECPy` [12] Python package was employed for elliptic curve computations in the Zero-Knowledge Proofs (ZKPs) implementation.

This section provides a step-by-step overview of the protocol's implementation.

1) Step 1: The main server generates a client signature based on the client secret(x_{client}). $sha3\_512$ hashing algorithm is used to generate the hash.

$$k = HASH(x_{\text{client}}||salt) \mod n \tag{5}$$

$$S = k * G \tag{6}$$

The signature S, an (x, y) point, is generated by multiplying the hashed password by the elliptic curve's generator point. This publicly shared signature allows future messages to be verified as coming from the same key without revealing the private data used to create it.

2) Step 2: The main server sends the generated client signature to the computation server and requests the challenge. This signature can also be stored on the computation server to restrict access solely to the main server.

3) Step 3: The computation server embeds the client signature into a JWT (JSON Web Token) and signs the token using its server secret(x_{server}) and sends it back to the main server.

$$token = sign(x_{\text{server}}, jwt) \tag{7}$$

Another method is for the computation server to send a random token to the main bank server, which must be stored with the signature in a cache or database with an expiry to prevent reuse. Embedding the signature in a JWT, however, makes the process stateless and removes the need for storage.

4) Step 4: The main server(prover) signs the received token again using the client secret. This additional signature serves as a proof that the main server has verified the token(t).

$$R = r.G \tag{8}$$

where r is a random number of size comparable to the curve.

$$c = HASH(t||R||salt) \tag{9}$$

$$m = r + c * k \tag{10}$$

(c, m) is the proof.

5) Step 5: When accessing a private route, the doubly signed JWT token (proof) is sent to the computation server for verification. The server first checks if the token was signed by it and if it was tampered with: verify(proof.data, server signature).

The JWT is then decoded to check expiration, and the client signature is retrieved. This client signature is used to verify the proof's legitimacy, confirming that the main server has access to the client secret used to create the client signature.

$$M = m.G \tag{11}$$

$$C = c.S \tag{12}$$

$$H(t||M - C||salt) === c \tag{13}$$

If this is verified, the main server is granted access. This demonstrates that the prover has access to the secret without directly sharing it, using zero-knowledge proof to confirm the authenticity.

VII. Evaluation and Results [PG]

In evaluating advanced Privacy-Preserving Computation techniques, the performance and feasibility of Homomorphic Encryption, Shamir's Secret Sharing, and Zero-Knowledge Proofs (ZKP) were assessed. Each technique offers distinct advantages but also faces significant limitations.

Homomorphic encryption enables computations on encrypted data, preserving privacy while outsourcing processing tasks. Despite its theoretical appeal, practical application is hindered by significant performance issues. Specifically, the computational time for homomorphic encryption scales poorly with the number of data elements and increases with encryption complexity. This inefficiency becomes pronounced, making homomorphic encryption impractical for real-time use due to its high computational and cost overhead. In scenarios where data privacy is crucial, organizations are better served by maintaining their own secure servers for computations rather than relying on homomorphic encryption.

Shamir's Secret Sharing effectively divides and reconstructs secrets in a distributed system. However, the time required to split a secret increases significantly with the number of threshold shares. Although the total number of shares also impacts splitting time, the effect is less dramatic compared to changes in threshold shares. For reconstruction, the time doubles with an increase in threshold shares. The size of the secret does not influence these times, while an increase in the prime modulus results in a consistent rise in reconstruction time, though splitting time remains unaffected.

An authentication system was implemented using the Schnorr protocol-based ZKP and compared with traditional JSON Web Token (JWT) authentication. Measurements indicated that ZKP required approximately 50 milliseconds on a Mac M3 Pro (12-core CPU, 18-core GPU), compared to 0.098 milliseconds for JWT authentication. Although ZKP shows a notable performance difference, making it less efficient for conventional applications, it remains highly relevant in high-security scenarios, such as secure financial transactions and voting systems. In these contexts, its ability to verify claims without revealing sensitive information is crucial.

VIII. Conclusion [PG]

This research evaluated advanced cryptographic techniques—Homomorphic Encryption, Secure Multi-Party Computation (SMPC), Shamir's Secret Sharing, and Zero-Knowledge Proofs (ZKP)—to assess their practical performance and applicability in real-world scenarios. Homomorphic Encryption was explored using libraries like TenSEAL and Pyfhel to perform computations on encrypted bank data, such as verifying user balances by processing transactions. This approach allows banks to securely outsource computations to third parties, but performance degraded significantly with increasing data size and complexity, making it less efficient than standard methods. For SMPC, libraries such as Crypten, PySyft, and MPyC were tested, but outdated documentation hindered implementation of the required use cases. Compatibility issues with modern libraries were encountered during the attempt to solve the Millionaire's Problem. Shamir's Secret Sharing was successfully implemented, enabling secret share distribution across multiple bank servers for decentralized decision-making and transaction verification through consensus. Using polynomial-based splitting and Lagrange interpolation for reconstruction, the system performed reliably. For ZKP, secure authentication was implemented using the Schnorr protocol. The main server generated a signature from its secret, which the computation server then embedded in a JWT. After the main server signed it again as proof, the computation server verified the proof without accessing the secret, ensuring stateless authentication with significant computational overhead compared to standard JWT methods.

Each technique was successfully implemented, with performance and practicality explored in real life scenarios. An interactive GUI was also developed to enhance user interactivity and improve understanding of each mechanism.

A. Further Research

Future research should aim to optimize the efficiency and scalability of privacy-preserving cryptographic techniques. In the case of homomorphic encryption, there is a need to enhance performance, especially when dealing with large datasets and complex operations. Exploring more efficient algorithms or improved configurations within libraries like TenSEAL and Pyfhel could help address the current limitations. For SMPC, better documentation and modernized examples are critical for practical use. Addressing compatibility issues with existing libraries such as Crypten, PySyft, and MPyC, and improving their usability would make these methods more accessible. Additionally, optimizing Shamir's Secret Sharing, particularly in the context of large thresholds and faster reconstruction times, is important for improving its application in distributed systems. Lastly, research on Zero-Knowledge Proofs should focus on reducing the significant overhead associated, making them a more viable option for real-world applications where privacy and performance need to coexist.

In summary, privacy-preserving computation techniques show promise, but require further research to enhance efficiency, scalability, and usability. Researchers should also investigate if new optimizations or unexplored methods exist that could make these techniques more viable for real-world use.

References

[1] B. U. E. S. Lab, "Applying fully homomorphic encryption: Part 1," https://esl.cs.brown.edu/blog/applying-fully-homomorphic-encryption-part-1/, 2024, accessed: 2024-09-15.

[2] C. Zhao, S. Zhao, M. Zhao, Z. Chen, C.-Z. Gao, H. Li, and Y. an Tan, "Secure multi-party computation: Theory, practice and applications," *Information Sciences*, vol. 476, pp. 357–372, 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0020025518308338

[3] I. Chatzigiannakis, A. Pyrgelis, P. G. Spirakis, and Y. C. Stamatiou. Elliptic Curve Based Zero Knowledge Proofs and Their Applicability on Resource Constrained Devices. [Online]. Available: http://arxiv.org/abs/1107.1626

[4] A. Ibarrondo and A. Viand, "Pyfhel: Python for homomorphic encryption libraries," in *Proceedings of the 9th on Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, 2021, pp. 11–16.

[5] A. Benaissa, B. Retiat, B. Cebere, and A. E. Belfedhal, "Tenseal: A library for encrypted tensor operations using homomorphic encryption," 2021.

[6] OpenMined, "Tutorial 3 - benchmarks," https://github.com/OpenMined/TenSEAL/blob/main/tutorials/Tutorial%203%20-%20Benchmarks.ipynb, 2024, accessed: 2024-09-18.

[7] M. Research, "Crypten framework," https://github.com/facebookresearch/CrypTen, 2024, accessed: 2024-09-15.

[8] OpenMined, "Pysyft," https://github.com/OpenMined/PySyft, 2024, accessed: 2024-09-15.

[9] B. Schoenmakers, "Mpyc," https://github.com/lschoe/mpyc, 2024, accessed: 2024-09-15.

[10] A. C. Yao, "Protocols for Secure Computations."

[11] S. A. Abdel Hakeem and H. Kim, "Centralized threshold key generation protocol based on shamir secret sharing and hmac authentication," *Sensors*, vol. 22, no. 1, 2022. [Online]. Available: https://www.mdpi.com/1424-8220/22/1/331

[12] C. Mesnil, "Ecpy," https://pypi.org/project/ECPy/, 2024, accessed: 2024-09-15.