GEORG-AUGUST-UNIVERSITÄT GÖTTINGEN IN PUBLICA COMMODA SEIT 1737

Implementation of Privacy-Preserving Computation (PPC) Techniques

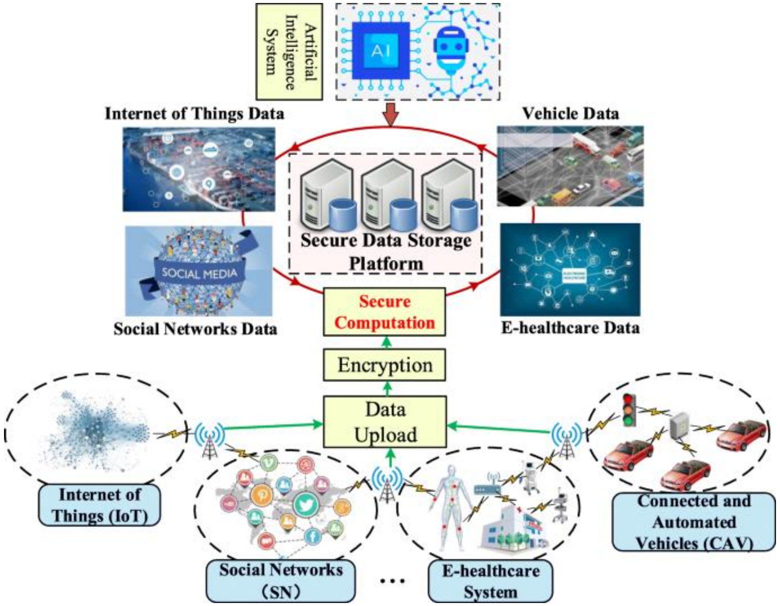
Presented by Priyanshu Gupta (19750534) and Claas Kochanke (22646052)

Structure

1. Introduction
2. System Architecture
3. Homomorphic Encryption
4. Secure Multi-Party Computation
5. Zero-Knowledge Proof
6. Conclusion

Introduction

Privacy-Preserving Computation (PPC) allows **secure computations** on data without revealing the underlying information. This technique is valuable in sensitive fields like healthcare, finance, and cloud computing, enabling secure data processing even with third-party involvement.

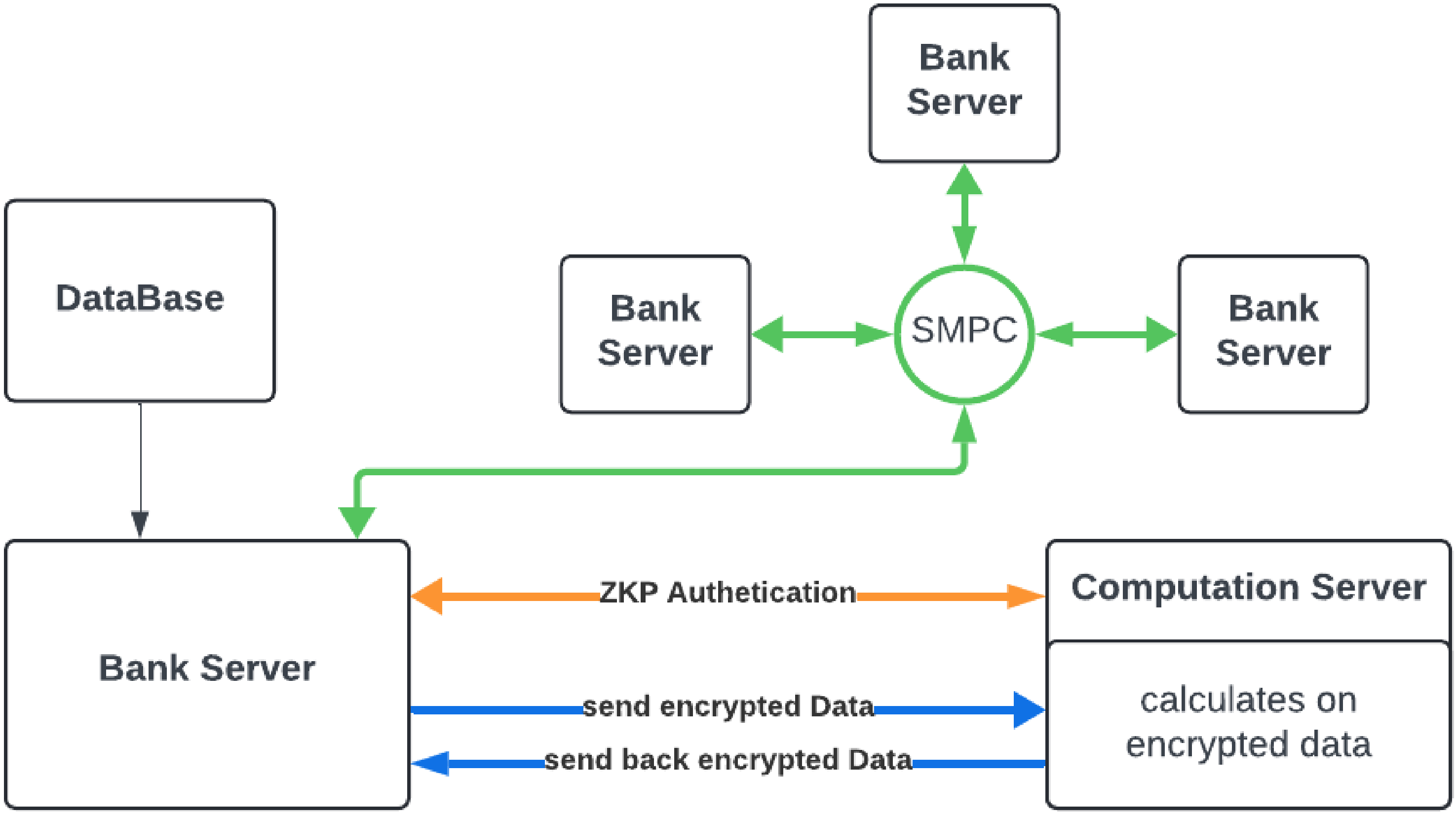


Source url: https://cybersecurity.springeropen.com/articles/10.1186/s42400-020-00057-3

Privacy Preserving Computation Techniques

- Many different techniques exist
- Used approaches:
    - Homomorphic encryption
    - Secure Multi-Party Computation
    - Zero-Knowledge Proof

System Architecture



Implemented using Python & Flask

Bank Server

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

PPC-Implementation

Homomorphic Encryption

Zero Knowledge Proof

Secure Multi-Party Computation

Compute Server

Incoming Request Log

POST **200** 2024-09-24 17:05:44

Path: /api/login
Remote Address: 127.0.0.1

JSON Data:


```
{
  "signature": "{\"params\": {\"alg\": \"sha3_256\", \"curve\": \"secp256k1\", \"salt\": \"svp6y11R..."
}
```


Data Size: 185 bytes

POST **200** 2024-09-24 17:05:26

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

JSON Data:


```
{
  "context": "CmVeoRAEAQIAAGUAAAAAAAAAAKLUv/WABAF0CAOQCAQBAAAkAXqEQBAEAAAAYAAGA/f///w...",
  "number_of_elements": 113,
  "encrypted_vectors": [
    "CgFxEv66b16hEAQBAgAAft0bAAAAAAAotS/9gFhhACAAtKwNjv1fYW0nEIC+G1YQOHETL4..."
  ]
}
```


Data Size: 570.18 MB

Homomorphic Encryption

- enables **computation** on **encrypted** data
- Sensitive data remains sensitive

- Full Homomorphic Encryption (**FHE**)
  - Unlimited number of operations
- Partial Homomorphic Encryption (**PHE**)
  - Only certain types of operations are supported

RSA – Partial Homomorph

- RSA is **multiplicative** homomorph
- $E(a) = a^e \bmod n$
- e: public exponent; n: modulus

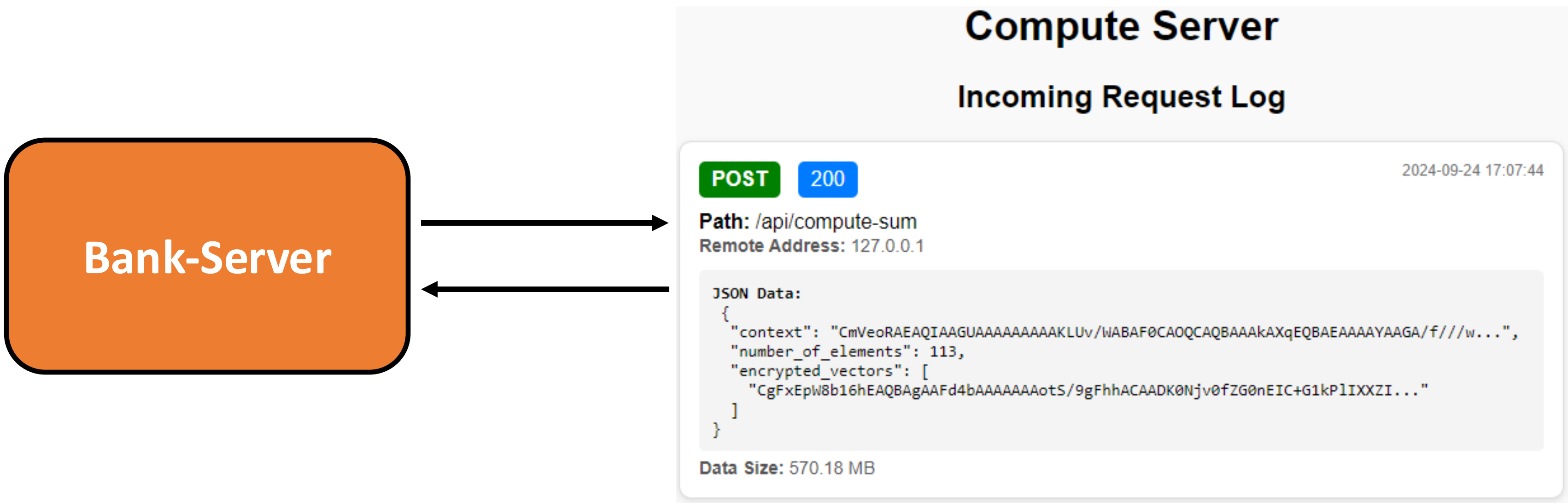
- Example [1]:

$$
\begin{aligned}
&E(a) \cdot E(b) \\
&= (a^e \bmod n) \cdot (b^e \bmod n) \\
&= (a \cdot b)^e \bmod n \\
&= E(a \cdot b)
\end{aligned}
$$

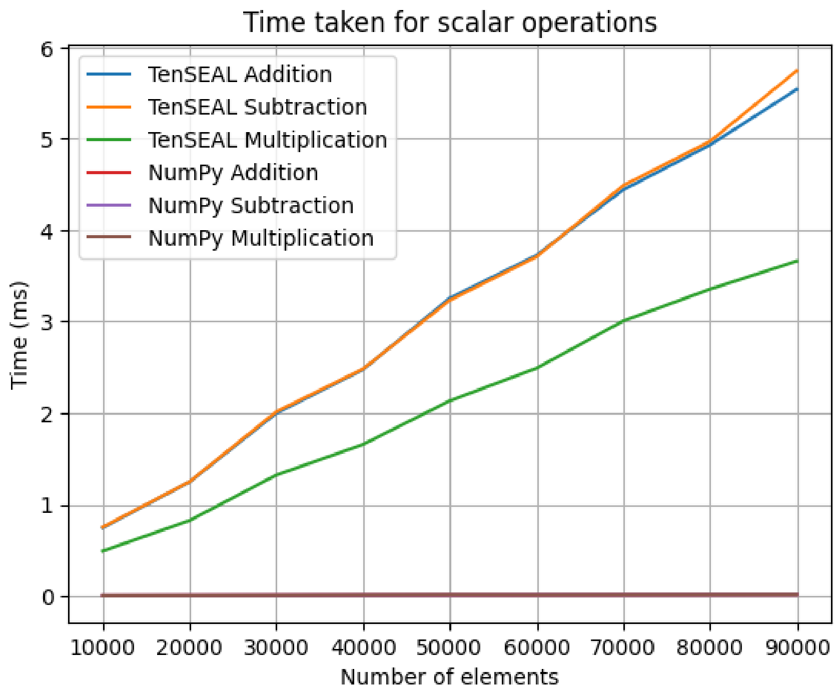
Homomorphic Encryption Libraries

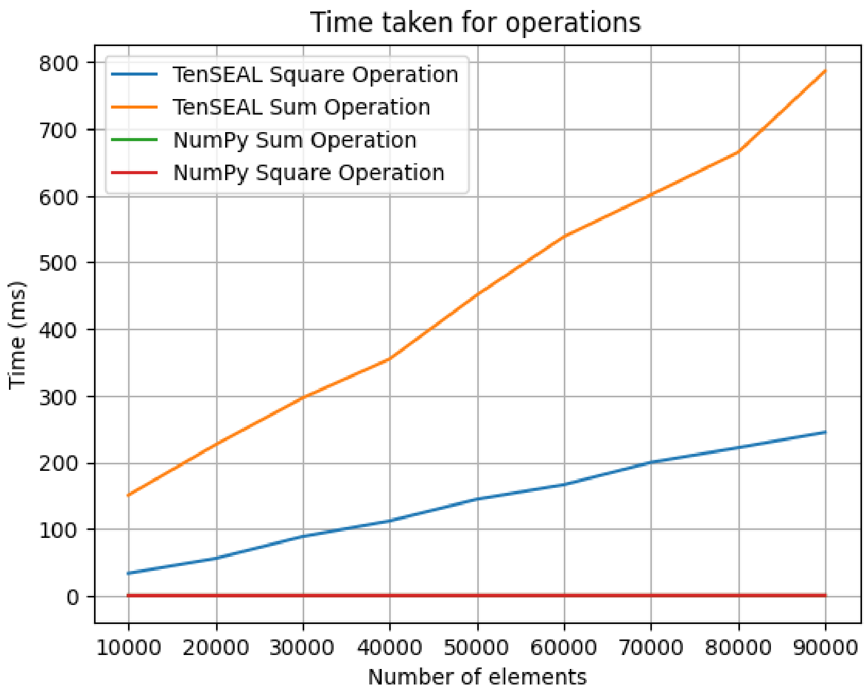
- Many different libraries are available
- Microsoft SEAL
  - Homomorphic encryption Library
  - Open-source in C++
  - Regularly updated
  - Well documented
- Wrappers for Python:
  - Pyfhel [4]
  - TenSEAL [5]

Homomorpic Encryption

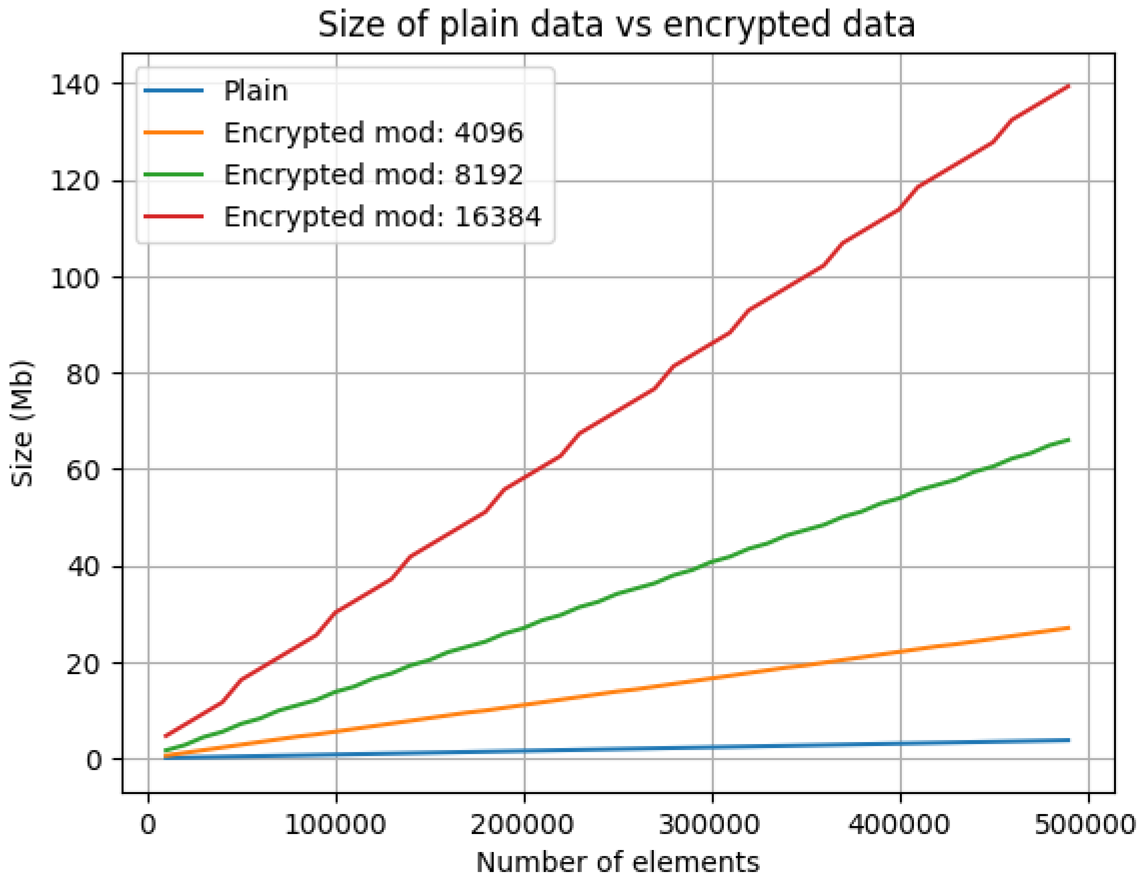


TenSEAL - Performance





TenSEAL – Memory Usage



Secure Multi Party Computation

- SMPC enables multiple parties to compute a function **without revealing their private inputs**.
- Ensures **privacy and confidentiality** in collaborative computations.
- Applications: **Finance, healthcare, and secure voting systems**. [2]

Implementations tried:

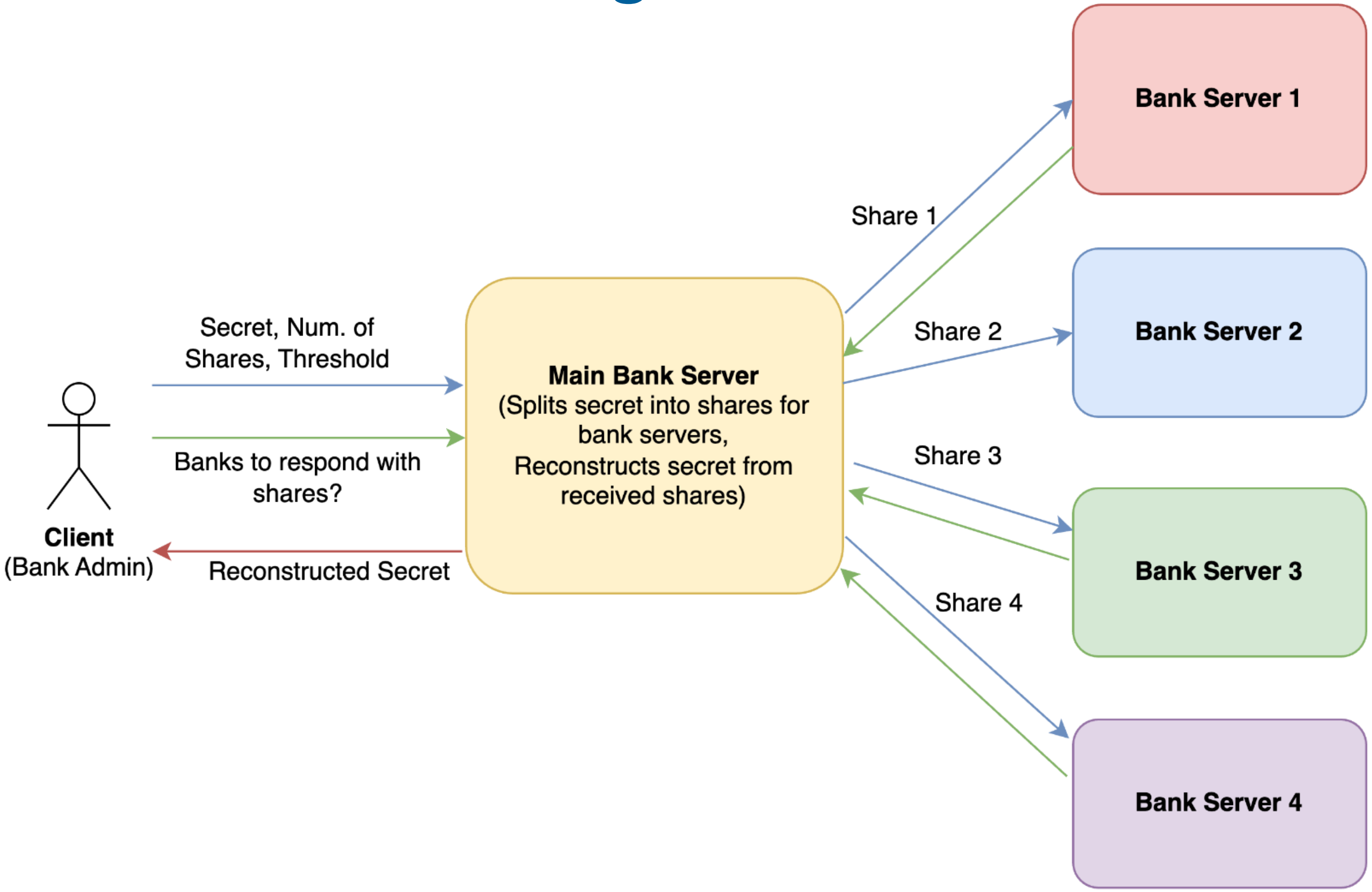
1. Millionaire's Problem: Compare wealth **without revealing** actual amounts. [9]

2. Shamir's Secret Sharing: Split sensitive data into parts, requiring a minimum number to **reconstruct** the original secret. [10]

Comparison of Python Libraries for SMPC

Library	Use Case	Limitations
Crypten	Secure & Private Machine Learning [6]	Decryption possible during computation.
PySyft	Privacy-preserving, federated learning, and SMPC [7]	Insufficient documentation, old examples not functional due to compatibility issues with modern library versions.
MPyC	Secure multi-party computation based on the MP-SPDZ framework [8]	Complex multi-threading architecture, poor documentation.

Shamir's Secret Sharing Architecture



Shamir's Secret Sharing Implementation [10]

1. Polynomial Formation:

Embed the secret $S = 123$ in a polynomial of degree $k - 1$(k = threshold = 3):

$f(x) = 123 + a_1x + a_2x^2$

Random coefficients:

- $a_1 = 97$
- $a_2 = 3$

Polynomial: $f(x)= 123 + 97+ 3x^2$

2. Share Computation:

Each share (x_i, y_i) is computed for $x_i = 1,2,3,4,5$ *(Total 5 Shares)*:

- $y_1 = f(1) = 223$, $y_2 = f(2) = 329$, $y_3 = f(3) = 441$, $y_4 = f(4) = 559$, $y_5 = f(5) = 683$
- **Shares Generated:** *(1, 223), (2, 329), (3, 441), (4, 559), (5, 683)*

3. Shares are distributed to smaller bank servers on different ports.

Shamir's Secret Sharing Implementation [10]

Secret Reconstruction:

When collected shares meet or exceed threshold k (e.g., *k=3*), reconstruct the secret using Lagrange interpolation:

$$S = \sum_{i=1}^{k} y_i \cdot \prod_{\substack{1 \leq j \leq k \\ j \neq i}} \frac{x_j}{x_j - x_i} \mod p$$

Using shares *(1, 223), (2, 329), (3, 441)*:

$S = 1239 \cdot L_1 + 1248 \cdot L_2 + 1269 \cdot L_3$

Calculate Lagrange Coefficients:

L1 = ((2/(2-1))(3/(3-1))) = 3, L2 = -3, L3 = 1*

Final Calculation:

$S = 223 \cdot 3 + 329 \cdot (-3) + 441 \cdot 1 = 123$ *(Original Secret)*

Shamir's Secret Sharing GUI

1. Input Secret

Enter the Secret:

1234

Enter no. of Shares:

5

Enter Threshold Shares(Required for secret reconstruction):

3

Split Secret

2. Secret Splitting

Secret: 1234

Polynomial Coefficients:


```
[
  1234,
  5.544958592465369e+37,
  5.823013500163691e+37
]
```


Generated Shares:


```
[
  [
    1,
    1.5851540664746653e+38
  ],
  [
    2,
    6.425536267023706e+37
  ],
  [
    3,
    5.750223498925004e+37
  ],
  [
    4,
    1.3825602360450547e+38
  ],
  [
    5,
    1.3637554505553412e+38
  ]
]
```


3. Share Distribution

Central Server is distributing shares to banks:

Central Server

Bank 1: Share 1.5851540664746653e+38

Bank 2: Share 6.425536267023706e+37

Bank 3: Share 5.750223498925004e+37

Bank 4: Share 1.3825602360450547e+38

Bank 5: Share 1.3637554505553412e+38

Shamir's Secret Sharing GUI

4. Share Collection

Select Number of Shares to Send Back:

3

Collect Shares

```
[
  [
    2,
    6.425536267023706e+37
  ],
  [
    1,
    1.5851540664746653e+38
  ],
  [
    3,
    5.750223498925004e+37
  ]
]
```

5. Secret Reconstruction

Reconstruct Secret

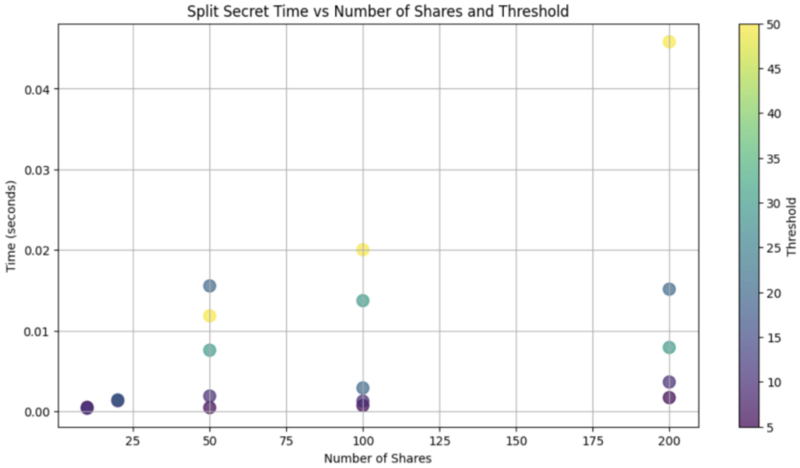
Selected Shares:

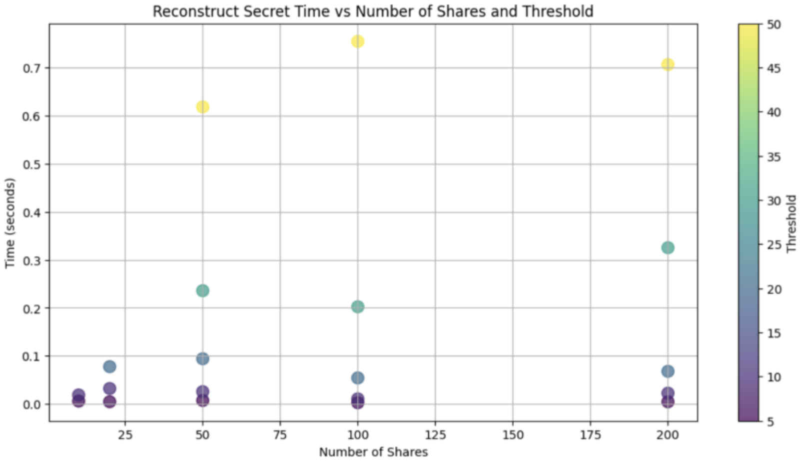
```
[
  [
    2,
    6.425536267023706e+37
  ],
  [
    1,
    1.5851540664746653e+38
  ],
  [
    3,
    5.750223498925004e+37
  ]
]
```

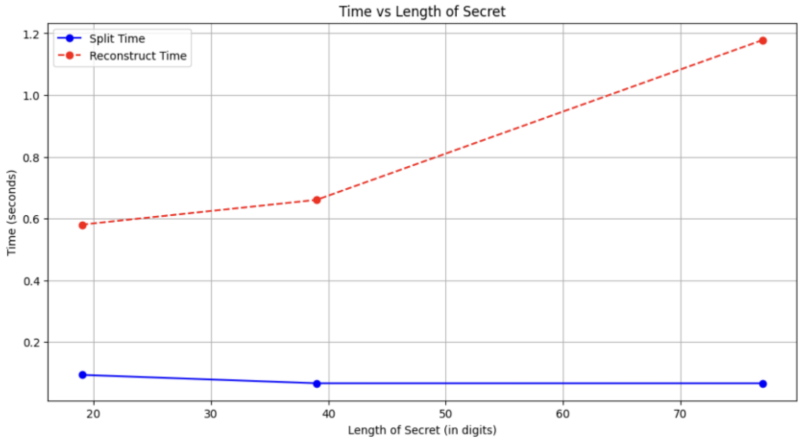
Reconstructed Secret:

1234

Shamir's Secret Sharing Evaluation







Zero-Knowledge Proof

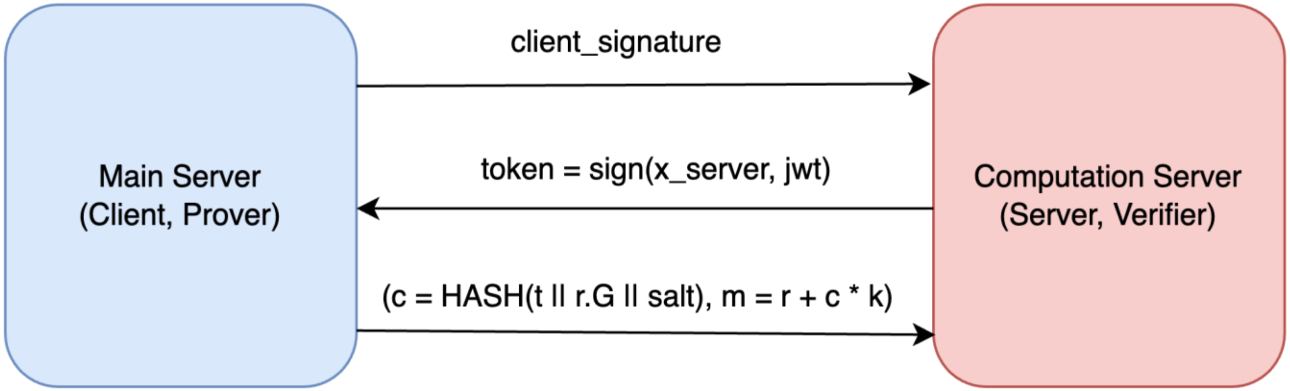
Description:
Cryptographic protocols that allow one party to prove knowledge of a secret to another party without revealing the secret itself. [3]

Use case Implemented:
Secure authentication between the main server and the computation server, ensuring that the main server can verify its identity without revealing sensitive information.

ZKP Implementation(Schnorr Protocol) [3]

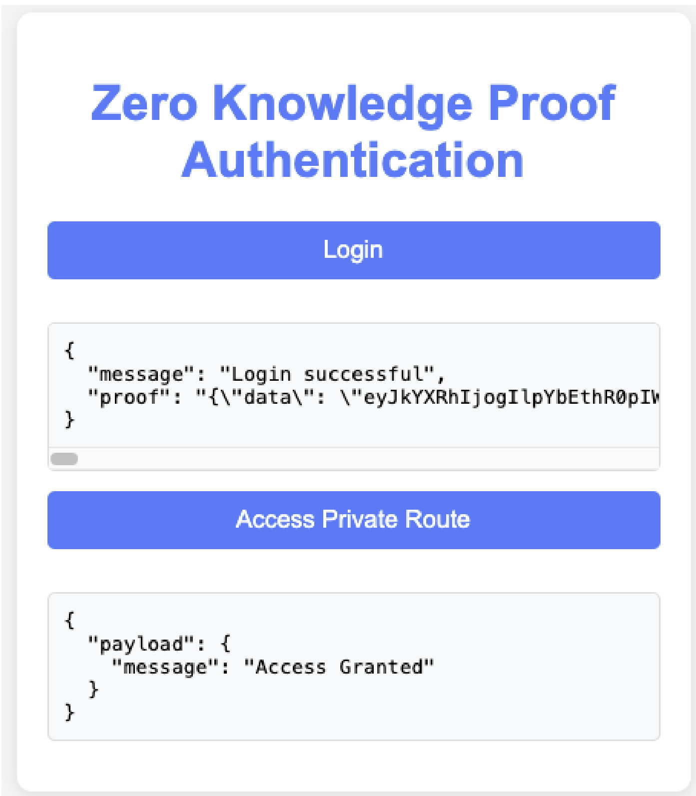
Main server generates a client signature based on the client secret stored on the main server.

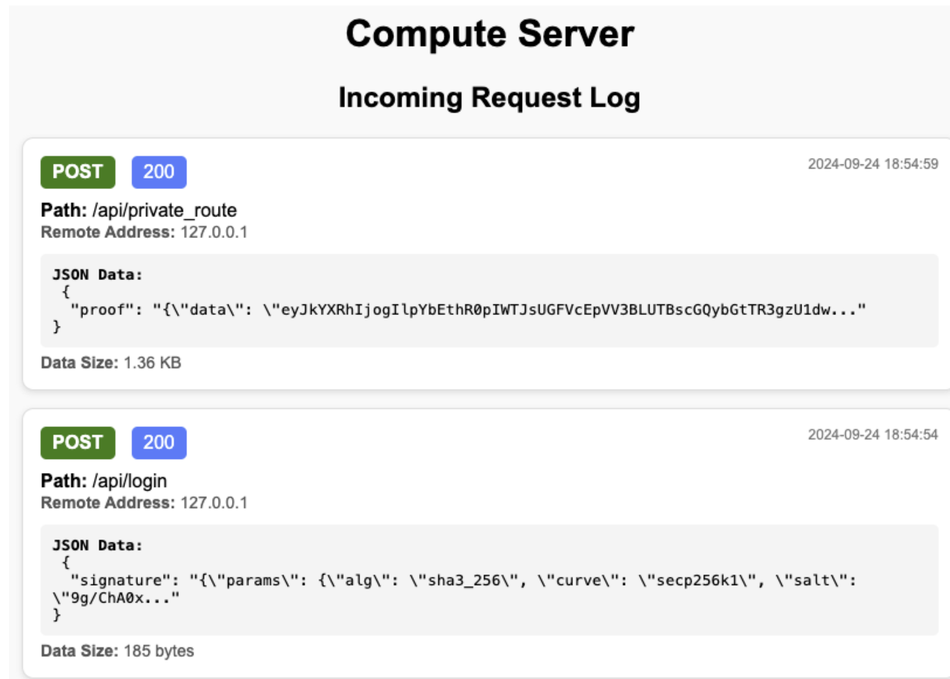
*k = HASH(x_client || salt) mod n, S(client_signature) = k * G*



- *verify(proof.data, server_signature)*
- JWT is decoded to check expiration, and the client_signature(*S*) is retrieved
  M = m.G, C = c.S
  H(t || M − C || salt) === c, if verified, the main server gains access.

ZKP GUI


Zero Knowledge Proof Authentication
Login
{
"message": "Login successful",
"proof": "{\"data\": \"eyJkYXRhIjogIlpYbEthR0pIW
}
Access Private Route
{
"payload": {
"message": "Access Granted"
}
}


Compute Server
Incoming Request Log
POST 200
2024-09-24 18:54:59
Path: /api/private_route
Remote Address: 127.0.0.1
JSON Data:
{
"proof": "{\"data\": \"eyJkYXRhIjogIlpYbEthR0pIWTJsUGFVcEpVV3BLUTBscGQybGtTR3gzU1dw..."
}
Data Size: 1.36 KB
POST 200
2024-09-24 18:54:54
Path: /api/login
Remote Address: 127.0.0.1
JSON Data:
{
"signature": "{\"params\": {\"alg\": \"sha3_256\", \"curve\": \"secp256k1\", \"salt\": \"9g/ChA0x..."
}
Data Size: 185 bytes

ZKP - Evaluation

ZKP Authentication took ***~50 milliseconds*** while a normal **JWT Authentication** took around ***0.098 milliseconds*** on Mac M3 Pro Chip.

Key Insights:
ZKP demonstrates a significant performance difference, making it less efficient for conventional applications.

Relevance in High-Security Scenarios:
Ideal for secure financial transactions and voting systems

Conclusion

Techniques Evaluated:

- **Homomorphic Encryption**: Secure encrypted computations; **Challenge**: Performance degradation with large datasets.
- **SMPC**: Used libraries like **Crypten, PySyft & MPyC**; **Challenge**: Outdated documentation, compatibility issues.
- **Shamir's Secret Sharing**: Reliable decentralized decision-making; **Challenge**: Scalability with large thresholds.
- **Zero-Knowledge Proof (ZKP)**: Secure Schnorr-based authentication; **Challenge**: High computational overhead vs. JWT.

Key Takeaways:

- Techniques implemented in real-world use cases.
- Trade-off between privacy & performance.

Overview

GUI-Implementation

- Bank Server GUI
- Compute Server GUI
- Visualization of the Concepts

Homomorphic Encryption

- Computation on encrypted data
- Slow performance
- Memory overhead

ZKP

- Interactive and non-interactive
- For authentification
- 500 x slower than JWT

SMPC

- Documentation issues
- Compatability issues
- Shamirs-Secret-Sharing

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] M. Research, "Crypten framework," https://github.com/facebookresearch/CrypTen, 2024, accessed: 2024-09-15.
[7] OpenMined, "Pysyft," https://github.com/OpenMined/PySyft, 2024, accessed: 2024-09-15.
[8] B. Schoenmakers, "Mpyc," https://github.com/lschoe/mpyc, 2024, accessed: 2024-09-15.
[9] A. C. Yao, "Protocols for Secure Computations."
[10] 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

ZKP Protocols Evaluated

Protocol	Advantages	Trade-offs
Zk-Snarks	1. Efficient and non-interactive. 2. Widely used in blockchain applications (e.g., Zcash).	1. Requires a trusted setup, which may introduce security risks. 2. Not ideal for real-time, lightweight authentication scenarios.
Zk-Starks	1. Quantum-resistant and scalable. 2. No trusted setup required, enhancing security.	1. Computationally intensive, potentially leading to higher latency.
Schnorr	1. Lightweight and efficient. 2. Simple and effective for real-time applications. 3. Well-suited for secure authentication systems.	1. Interactive, which may introduce some overhead in communication. 2. Less flexible than non-interactive protocols in certain use cases.