WEB3CLUBS FOUNDATION LIMITED

Course Instructor: DR. Cyprian Omukhwaya Sakwa
PHONE: +254723584205 Email: cypriansakwa@gmail.com

Foundational Mathematics for Web3 Builders

Implemented in RUST

Lecture 40

August 5, 2024

Fully Homomorphic Encryption (FHE)

- Fully Homomorphic Encryption (FHE) is a type of encryption that allows calculations to be done on ciphertexts, yielding an encrypted result that, when decrypted, corresponds to the result of operations conducted on the plaintext.
- This means you can process data without first decrypting it, preserving data privacy and security even when it is in use.
- Fully Homomorphic Encryption is an important advancement in cryptography, with the potential to provide extremely secure data processing in a variety of sensitive and privacy-critical applications.

15.1 Key Properties of FHE

1. Homomorphic Addition and Multiplication: Fully Homomorphic Encryption (FHE) methods allow for addition and multiplication operations on ciphertexts, which can be utilized to create more complicated calculations.
2. Encryption and decryption: The process of encrypting and decrypting data is identical to that of other encryption systems, however FHE ensures that the encrypted data can be computed.
3. Security: FHE retains the same level of security as traditional encryption systems, guaranteeing that data is protected from unauthorized access.

Applications of FHE

1. Secure Cloud Computing: Users can store and process encrypted data on cloud servers without revealing it to the server.
2. Privacy-Preserving Data Analysis: Enables the analysis and processing of sensitive data, such as medical or financial records, while maintaining privacy.
3. Secure Voting Systems: Allows for the secure computation of votes without revealing individual preferences.
4. Encrypted Databases: Enables queries to be executed on encrypted databases without decrypting the data, ensuring confidentiality.

Challenges and Limitations

1. Performance overhead: FHE operations are computationally demanding and significantly slower than plaintext operations.
2. Complexity: FHE schemes demand a thorough comprehension of cryptographic principles and mathematical foundations.
3. Noise Management: Noise is introduced into ciphertexts during computation, which must be handled in order to avoid decryption errors.

Zama Bounty Program

- Zama is a cryptography company that develops Fully Homomorphic Encryption (FHE) and other privacy-protecting solutions.
- They developed the experimental bounty scheme to encourage community developers to join them in improving the FHE space.
- The Zama Bounty Program provides monetary rewards for completing particular challenges.
- This effort intends to motivate and incentivize the developer community to create FHE applications and solve problems that will propel FHE technology forward by a decade! As a result, their bounty program prioritizes innovation and contribution over bug fixes.

18.1 Zama's Season 6 bounties

- Every season, they introduce bounties for a single Zama library.
- All submissions are evaluated on the basis of code quality and, more significantly, speed performance.
- At the end of each season, they will reward up to three submissions per bounty.

1. Enhance the performance of the TFHE-rs library €10,000 $\approx$ Ksh $14,000,000$

- Improving the efficiency of the TFHE-rs (a Rust library for Torus Fully Homomorphic Encryption) entails optimizing various library components.
- The purpose of this bounty is to improve the efficiency of the TFHE-rs library by implementing any optimizations, including cryptographic or algorithmic methods.

The focus of the optimization should be on:

- Cryptographic primitives: Programmable bootstrapping or keyswitch.
- Integer operations: Any operations with a focus on addition, multiplication, or division of numbers larger than or equal to 64 bits.
- These could be used in cases where both inputs are encrypted or where one is encrypted while the other is in clear text (also known as scalar).

Definition 13 (Programmable Bootstrapping)

Programmable Bootstrapping is a technique for refreshing ciphertexts and reducing noise while potentially adding data transformations throughout the process.

✓In FHE, each homomorphic action on encrypted data raises the noise level in the ciphertext. If the noise becomes too great, the ciphertext will no longer be decrypted properly.

- ✓ Bootstrapping is a technique that resets the noise level in ciphertext, allowing for more calculations without exceeding noise limitations.
- ✓ Traditional bootstrapping just decreases noise.
- ✓ Programmable bootstrapping minimizes noise while also allowing for the application of a specific function or change to the ciphertext during the bootstrapping process.
- ✓ This can be beneficial for complex computations in which some processes must be repeated frequently without causing significant noise buildup.

Definition 14 (Keyswitch)

An approach for changing a ciphertext's encryption key from one to another, allowing for secure and flexible calculation across multiple keys without decryption.

Expectations: In your optimization, you should consider the following constraints:

- Cryptographic parameters can be adjusted, but security must be at least 128 bits and error probability should be at most $2^{(-40)}$ for the entire process, including linear operations, a keyswitch, and a programmable bootstrapping . Any new cryptographic techniques must provide proof that these constraints are met.
- Compatibility with other operations: The new techniques should not degrade existing functionality. For example, if a new encoding for accelerating multiplication is introduced, a casting mechanism must be available to convert between the new and classical encodings. Any overhead associated with the new approaches should be clearly described and adequately benchmarked. Generally, all tests must be run to demonstrate that correctness has been confirmed.

18.2 Techniques to help increase their performance:

- Algorithmic Optimizations

  a) Choose the right parameters (e.g., key size, noise level) to strike a balance between security and performance.

  b) Implement approaches for managing and reducing noise growth during homomorphic procedures, such as modulus switching or key switching.

  c) Use Fast Fourier Transforms (FFT) or its variants to speed up polynomial multiplication, which is a common operation in FHE schemes.

- Parallelization and Concurrency
    a) Use Rust's multi-threading features to parallelize independent homomorphic processes.
    b) Use SIMD (Single Instruction, multiple Data) instructions to perform operations on several data points at the same time.
    c) Use technologies such as CUDA or OpenCL to offload computationally heavy activities to GPUs.
- Memory Management
    a) Use efficient data structures to store and handle ciphertexts and keys while minimizing memory overhead.
    b) Implement memory pooling to reuse memory blocks and reduce the overhead associated with frequent allocation and deallocation.

- Code Optimizations
    a) Inlining and loop unrolling if applied can help to reduce function call overhead and enhance loop efficiency.
    b) Use Rust's zero-cost abstractions to develop high-level programs with minimal runtime overhead.
    c) Profile the library to detect and optimize hot pathways for increased performance.
- Leveraging Rust Features
    a) Use Rust's ownership and borrowing system to assure secure and efficient memory usage that eliminates the need for garbage collection.
    b) Use iterator adaptors to create short and efficient loops.
    c) Use Rust's concurrency primitives, such as channels and async/await, to efficiently manage concurrent processes.

- Profiling and Benchmarking

    a) Use profiling tools such as perf, Valgrind, or Flamegraph to find performance bottlenecks.

    b) Continuously benchmark the library using crates-like criteria to track performance improvements and regressions.