Homomorphic Encryption for Data Privacy

Omar Loudghiri
oxl51@case.edu

Nick Harms
nmh84@case.edu

Kaia Kanj
kmk233@case.edu

Tom Hua
yxh1165@case.edu

Kamsi Eneh
kre40@case.edu

Stephen Yen
sxy747@case.edu

ABSTRACT

This paper explores the application and impact of homomorphic encryption in safeguarding data privacy in various domains, including healthcare, banking, and defense. Homomorphic encryption allows for computations on encrypted data, enabling the use of third-party computational services without compromising data privacy. This feature is increasingly relevant in today's world, where data-driven models are widely used and very important in decision-making processes. The paper discusses the problem statement, conducts a literature search to understand the current state of the art, outlines the methodologies for a proof of concept matrix multiplication implementation (the basis of many other matrix operations and models), and provides the data collected from our implementation.

KEYWORDS

homomorphic encryption, data privacy, secure computation, healthcare, banking, defense

ACM Reference Format:
Omar Loudghiri, Nick Harms, Kaia Kanj, Tom Hua, Kamsi Eneh, and Stephen Yen. 2024. Homomorphic Encryption for Data Privacy. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Homomorphic encryption represents a critical advancement in the field of data privacy and secure computation. With the increasing reliance on external computational resources and cloud-based services for data processing and analysis, ensuring the privacy and security of sensitive data is paramount. This technology enables the execution of computations on encrypted data, producing an encrypted result that, when decrypted, matches the outcome of operations performed on the plaintext. This capability is particularly relevant in sectors like healthcare, banking, and defense, where the privacy of data is not just a requirement but a necessity.

2 APPLICATIONS

There are multiple different applications in which homomorphic encryption can be utilized including taxes, banking, elections, healthcare, and the military.

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

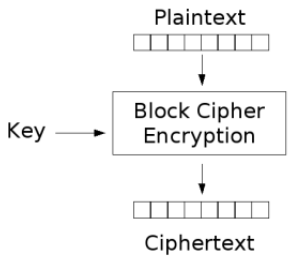

Figure 1: Homomorphic Encryption Visualized

Homomorphic encryption offers a solution for tax authorities seeking to conduct computations on sensitive financial information while respecting taxpayer privacy. By encrypting financial data prior to submission, taxpayers can prevent unauthorized access while facilitating seamless data analysis by tax authorities.

In terms of banking, FHE allows financial institutions to observe spending patterns and detect fraudulent activities across a vast customer base without compromising individual privacy rights, thereby increasing the integrity of banking operations.

By encrypting ballots before submission, voters safeguard the secrecy of their choices, while election authorities utilize homomorphic encryption to tally votes and verify results without compromising voter anonymity, thereby improving electoral integrity.

In the realm of healthcare, there are multiple different uses for homomorphic encryption. Its application in telemedicine platforms ensures the secure processing of patient data during remote consultations, safeguarding sensitive information from unauthorized access. Additionally, homomorphic encryption revolutionizes data analysis in healthcare by enabling computations on encrypted medical records, fostering research, and facilitating disease diagnosis while upholding patient confidentiality.

Security and confidentiality are necessities in military operations, making homomorphic encryption very important in protecting sensitive communications and data from adversaries. Whether in secure communications or logistical planning, homomorphic encryption empowers military entities to safeguard operational details while ensuring effective coordination and strategic decision-making.

3 LIBRARIES

OpenFHE, formerly known as PALISADE, is a library for fully homomorphic encryption (FHE) implementations. It has broad functionality and optimization capabilities and supports a variety of encryption schemes. OpenFHE is designed for performance and scalability which is optimal for individuals performing research on

large datasets. However, OpenFHE's intricacy and the significant effort required to become proficient in its use is a considerable task, particularly for beginners. Additionally, since it is written in C++, there would be extra steps for integration with different programming tools and languages, which could slow down our development process and limit the findings of our project.

Therefore, our project utilizes pyFHE, an extension of the OpenFHE library, which is designed for modularity and simplicity of extending cryptographic schemes. This allows us to seamlessly integrate new cryptographic approaches with minimal adjustments to the existing framework.

The architectural design of pyFHE facilitates the incorporation of new cryptographic schemes without extensive modifications. This structure features a clear separation of components such as encryption schemes, cryptographic methods, and the underlying operations in the polynomial ring, thus enhancing the library's adaptability and extensibility.

Functionality-wise, pyFHE is robust, covering a comprehensive range of operations necessary for full homomorphic encryption (FHE), including key generation, encryption, decryption, relinearization, and bootstrapping.

At the core of pyFHE's operations is the mathematical foundation built on the Ring Learning with Errors (RLWE) problem. It utilizes operations within polynomial rings, where coefficients are subjected to modular reduction. This establishes the basis for the security and functionality of FHE:

$$\text{Polynomial operations in } \mathbb{Z}[x]/(x^N + 1)$$

These operations, fundamental to pyFHE, include addition, multiplication, and modulus operations in the ring:

$$\mathbb{Z}[x]/(x^N + 1)$$

For the CKKS scheme, pyFHE implements the bootstrapping process, which is crucial for reducing noise accumulation and enabling an unlimited sequence of arithmetic operations on ciphertexts. This process involves complex transformations and inversions of data between coefficient and slot representations, utilizing Fourier Transform-like operations for encoding and decoding.

4 PROBLEM STATEMENT AND LITERATURE SEARCH

Homomorphic encryption (HE) is a sophisticated form of encryption that allows computations to be performed on encrypted data without the need to decrypt it first or access the secret key. As outlined in the OpenFHE and pyFHE papers, this technology has evolved significantly, drawing on design ideas from previous projects like PALISADE, HElib, and HEAAN, and introducing new concepts such as bootstrapping support, hardware acceleration, and user-friendly operational modes.

OpenFHE is an open-source library that, from its inception, assumes all implemented FHE schemes will support bootstrapping and scheme switching. It also incorporates a Hardware Abstraction Layer (HAL) to support multiple hardware acceleration backends, enhancing its flexibility and performance. The library is designed to be both user-friendly and compiler-friendly, automating complex operations like modulus and key switching, or delegating them to external compilers.

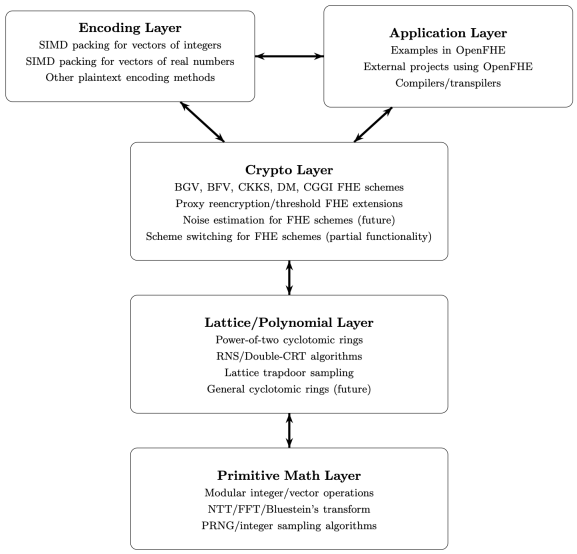

Figure 2: Layers of OpenFHE

On the other hand, pyFHE fills a unique niche by providing a Python-based library for FHE, catering to the language's simplicity and readability. It supports key FHE schemes such as the Brakerski-Fan-Vercauteren (BFV) and the Cheon-Kim-Kim-Song (CKKS) schemes, with a particular focus on bootstrapping for CKKS—a crucial operation for maintaining the usability and security of homomorphic encryption systems over extended computations.

The practical applications of HE, particularly in sensitive fields like healthcare, are profound. By enabling computations on encrypted data, HE allows healthcare providers to adopt advanced analytics and machine learning without compromising patient privacy. This capability is especially crucial for third-party analytics services, where traditional methods pose significant privacy risks. Moreover, even if a data breach occurs, the encrypted nature of the data ensures its security and integrity, mitigating potential damage.

Homomorphic encryption not only extends public-key cryptography by allowing for computational homomorphisms between plaintext and ciphertext spaces but also supports diverse computational needs through various schemes categorized into partially, somewhat, leveled fully, and fully homomorphic encryption. Each type offers different capabilities and trade-offs in terms of circuit evaluation depth and types of operations supported, addressing the key challenge of multiplicative depth in encrypted computations.

In addition to OpenFHE and pyFHE, there have been numerous significant contributions to the field of homomorphic encryption. Projects like Microsoft's SEAL, IBM's HElib, and Google's Private Join and Compute have advanced the practicality and efficiency of HE implementations. These libraries not only demonstrate the versatility of HE in different programming environments but also underline the increasing demand for privacy-preserving technologies across various industries. Each of these projects contributes to the ecosystem by providing different optimizations, such as improved performance of arithmetic operations on encrypted data and expanded support for complex algorithms.

5 PROPOSED SOLUTION: MATRIX MULTIPLICATION USING FULLY HOMOMORPHIC ENCRYPTION

Implementing complex operations using homomorphic encryption presents significant challenges due to the intricate nature of securely performing computations on encrypted data. One such complex operation is matrix multiplication, a fundamental component in many computational tasks, which has not yet been fully implemented in practical applications of homomorphic encryption, in open source projects, despite theoretical proofs of its feasibility.

Our implementation begins by generating two square matrices filled with random complex numbers, simulating real-world data sets that might be sensitive, hence the need for encryption. The encryption process for these matrices involves the CKKS scheme, particularly suited for handling complex numbers. Here, the matrix multiplication is executed as follows:

(1) **Matrix Generation**: We generate two matrices, `matrix1` and `matrix2`, each containing complex numbers, using the function `create_complex_matrix(size)`.
(2) **Encryption of Vector**: The columns of `matrix2` are processed using the `encrypt_vector()` function, which breaks down the matrix into manageable blocks, encodes them, and encrypts them.
(3) **Encrypted Matrix Multiplication**:
   - We iterate over each encrypted column vector of `matrix2`.
   - For each column, we perform encrypted multiplication with corresponding blocks of `matrix1` using nested loops. The multiplication and addition operations are carried out in the encrypted domain, utilizing `CKKSEvaluator`.
(4) **Decryption and Decoding**: After the encrypted computations, the results are decrypted and decoded back into complex numbers to obtain the final matrix product.

These steps are detailed in pseudo code in **Algorithm 1** This implementation aims to validate the practicality of performing matrix multiplication on encrypted data. By leveraging the CKKS encryption scheme, we handle the complexities of encrypted calculations while ensuring the confidentiality and integrity of the data.

The python code for the implementation is attached to this report.

Algorithm 1 Pseudocode for Encrypted Matrix Multiplication

```
procedure EncryptVector(encoder, encryptor, vector, scale, blockSize)
    Initialize an empty list encryptedBlocks
    for each block in vector do
        encoded ← encoder.encode(block, scale)
        encrypted ← encryptor.encrypt(encoded)
        Append encrypted to encryptedBlocks
    end for
    return encryptedBlocks
end procedure
procedure MatrixMultiply(matrix1, matrix2, keys, blockSize, scale)
    for each column in matrix2 do
        encryptedVector ← EncryptVector(...)
        for each block in matrix1 do
            product ← evaluator.multiplyMatrix(encryptedVector,
block, . . . )
            aggregatedProduct ← aggregate product
        end for
        decryptedColumn ← decrypt and decode
aggregatedProduct
    end for
    return the result matrix
end procedure
```

5.1 Expected Output

The primary expectation for matrix multiplication using homomorphic encryption is that the output should closely resemble that of traditional matrix multiplication in both structure and numerical value. Specifically, the encrypted output should mathematically mirror the result of a plaintext multiplication of matrices A and B, producing matrix C with high numerical fidelity to an error tolerance as tight as 0.001. This level of precision ensures that despite the computations being carried out on encrypted data, the decrypted results should be practically indistinguishable from those obtained through non-encrypted methods.

To validate the precision and reliability of homomorphic encryption in matrix multiplication, the implementation must pass several rigorous tests. These include precision tests to verify that the decrypted results align within a minimal margin of error from standard computations, performance benchmarks to assess computational efficiency against traditional methods, scalability tests to ensure that larger matrix sizes are handled effectively, and comprehensive security audits to guarantee data integrity and encryption robustness throughout the computational process.

Algorithm 2 Homomorphic Encryption-based Matrix Multiplication

```
procedure HomomorphicMatrixMultiply(A, B, keys)
    n ← number of rows in A
    m ← number of columns in B
    Initialize matrix C of dimension n × m with zeros
    for i ← 1 to n do
        for j ← 1 to m do
            sum ← 0
            for k ← 1 to n do
                encrypted_value ← Encrypt(A[i][k] ×
B[k][j], keys)
                sum ← HomomorphicAdd(sum, encrypted_value)
            end for
            C[i][j] ← Decrypt(sum, keys)
        end for
    end for
    return C
end procedure
```

However, implementing matrix multiplication under homomorphic encryption faces several significant challenges. Sometimes it even affects the accuracy of the output. The computational overhead associated with encryption and decryption processes can not only affect performance by increasing computation time but also impact

the output quality, as longer processing times can increase the likelihood of computational errors and reduce precision. Additionally, as matrix sizes and the complexity of operations increase, maintaining computational stability and precision within acceptable error margins becomes more challenging. This can lead to outputs that, while still encrypted and secure, may vary slightly from expected results in non-encrypted scenarios. Integrating such encrypted operations into existing systems also presents complexities, requiring extensive modifications that can introduce new sources of error, further influencing the final output's accuracy and reliability. These influencing factors determine that we need to conduct rigorous calculations and conduct necessary verifications after calculations.

5.2 Error Correction

In the implementation of matrix multiplication using homomorphic encryption, meticulous attention to error handling and mitigation strategies is critical for ensuring the accuracy and reliability of the expected output. Small errors in computations, if not properly addressed, can propagate through multiple stages of encrypted operations, leading to significant deviations from expected results. Therefore, developing error detection and correction mechanisms is essential. These mechanisms must be capable of identifying and rectifying errors that arise due to the encryption process itself or from numerical instabilities during matrix operations. Additionally, implementing redundancy checks and validation at each step of the computation can help maintain output integrity. This proactive approach to error management not only enhances the accuracy of the results but also adds to the overall confidence in the use of homomorphic encryption for complex mathematical tasks. By prioritizing these strategies, we can minimize the impact of potential computational errors and ensure that the output properly represents the intended calculations, preserving both the functional and numerical integrity of the data.

6 EVALUATION METRICS AND RESULTS

The advantages of homomorphic encryption are clear; however, its efficacy depends on various factors that can greatly enhance its utility or contribute to its shortcomings. In this segment, we will explain the critical factors that impact the performance and viability of homomorphic encryption.

Metric, Time to Run Program: The time to run a program is a critical metric in evaluating the efficiency of homomorphic encryption operations, especially when considering parallelization. In a research article on homomorphic encryption, it demonstrate that parallel algorithms can yield up to a 15x speedup is pivotal, showing the substantial benefits of parallelization in reducing computation time. This speedup not only signifies an improvement in raw computational time but also implies broader applicability of homomorphic encryption in time-sensitive or resource-constrained environments. This metric serves as a benchmark for evaluating the practicality of homomorphic encryption solutions, emphasizing the importance of optimization in algorithm design to enhance performance and scalability.

Comparing the Accuracy: Accuracy in homomorphic encryption is also an extremely important factor, Science ensures that operations on encrypted data yield valid and reliable results upon decryption. The process of validating the decrypted results of encrypted multiplication to an absolute tolerance is crucial in demonstrating the integrity of homomorphic encryption operations. This validation process ensures that despite the complex nature of encrypted computations, the end results remain consistent and accurate, a fundamental requirement for the adoption of homomorphic encryption in fields where precision is critical, such as in finance or healthcare.

6.1 Collected Data

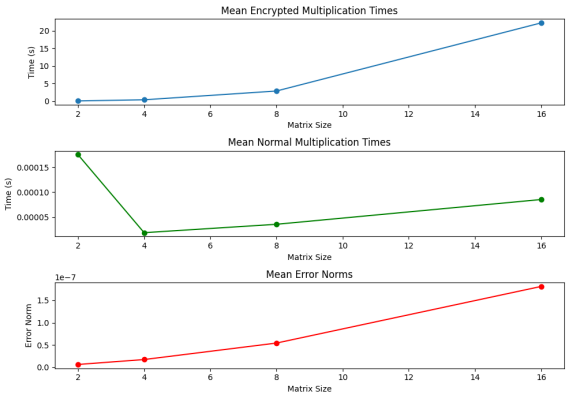

Figure 3: Collected Results

The figure displays three graphs, each representing different aspects of matrix multiplication performance and accuracy across various CPUs including the Apple M1, the Intel i7-8750H, and the HPC(pioneer).

The first plot, titled 'Mean Encrypted Multiplication Times,' shows the time it takes to multiply encrypted matrices as a function of their size. As the size of the matrix increases, so does the computation time, which is expected due to the additional complexity of encrypted operations. It is an average of the data collected accross all three CPUs. It is important to note that Apple M1 was the best performer out of the three likely due to optimizations of python3.12 on ARM M1 chips

The second plot, 'Mean Normal Multiplication Times,' benchmarks the time taken by NumPy to perform unencrypted matrix multiplications. The graph shows that as the size of the matrix increases, the time taken for these multiplications does not increase substantially.

The third plot, 'Mean Error Norms,' measures the average error norm, which provides insight into the accuracy of the encrypted matrix multiplication. This error is likely a result of the noise inherent in the encryption process. As the matrix size increases, so does the error norm, suggesting a degradation in precision with larger data sets. This might be of particular importance when considering the balance between computation time and accuracy in encrypted

data processing. However at a certain threshold (HuggingFace Accuracy threshold) bootstrapping can be used to remedy to the error generated.

7 FUTURE WORK AND CONCLUSION

In conclusion, this paper offers a comprehensive analysis of homomorphic encryption as a method to ensure data privacy in sectors like healthcare, banking, and defense. It discusses how homomorphic encryption allows for computations on encrypted data, thereby supporting secure and private data processing. This study not only highlights the utility of libraries such as OpenFHE and pyFHE in enabling these processes but also serves as a proof of concept for these applications. Future improvements could focus on enhancing parallelization and optimizing block sizes to improve the performance and usability of homomorphic encryption. As development continues, it will be vital to balance computational efficiency with data protection, ensuring that homomorphic encryption remains a pivotal technology in privacy preservation.

7.1 Parallelization

An important part of any encryption algorithm is for it to be done in a timely matter. In our project we used the Cheon-Kim-Kim-Seong (CKKS) scheme, a pre-integrated homomorphic encryption scheme within pyFHE. Despite producing some efficient looking results for our encryption and decryption scheme, it can still be improved. In future works. In future endeavors we can aim to incorporate a Blocked Matrix-Vector Multiplication to manage our matrices and data. An example algorithm is provided below. This encryption breaks down the matrix and vectors into smaller parts, making encryption and computation faster. There are multiple ways to approach parallelizing the encryption process: blocked, row parallel, and parallel. The blocked/serial implementation encrypts the data block by block with no parallelization. The row parallel encryption parallelizes the outer loop, enabling con- current encryption of each block. This parallel method parallelizes both the inner and outer for loops of the encryption algorithm. Parallizing both for loops ensures the simultaneous encryption of individual data within the matrices. Utilizing these encryption schemes on the blocked matrix-vector multiplication and CKKS scheme, data can securely and quickly be encrypted. Additionally, an alternative or expanded approach involves batching and multi-threading techniques utilizing threading and multiprocessing packages. Multithreading divides the algorithm into separate "threads" or processes executed concurrently within the same program, using the same resource and memory. Batching breaks down large computational tasks into smaller, independent pieces for concurrent execution. Future endeavors will involve conducting experiments to evaluate the practicality of implementing these parallelization techniques, assessing efficiency improvements, and determining the optimal scenarios for each approach.

7.2 Block Sizes

Block sizes are a fundamental aspect of homomorphic encryption schemes, impacting the efficiency of data encryption and the computational process. When data is encrypted using homomorphic encryption, it is divided into fixed-size blocks. These blocks serve as computation units, with each block undergoing encryption and computation. The size of these blocks directly impacts the granularity of the encryption process and the efficiency of computational operations performed on the encrypted data. In our project, we used a standard block size of 4 for our encryption scheme, regardless of the matrix's dimensions. While this approach provided a consistent framework, it might not have been optimal for all scenarios as different matrix sizes and types of computations may benefit from varying block sizes. For instance, larger blocks might lead to more efficient computations for larger matrices, while smaller blocks could be more suitable for smaller matrices or specific types of operations.

Algorithm 3 Blocked Matrix-Vector Multiplication

```
result ← 0
for i ← 0 to n/b do
    for j ← 0 to n/b do
        Generate rotation keys for multiplication
        prod ← matmul(A[ib : ib+b−1][jb : jb+b−1], v[jb : jb + b − 1])
        result[ib : ib+b−1] ← add(result[ib : ib+b−1], prod)
    end for
end for
return result
```

To explore the impact of block sizes on encryption efficiency and security, we plan to conduct experiments with different block sizes tailored to specific matrix dimensions. By adjusting the blocksize variable in our matrix multiplication scheme for various matrix sizes and recording the performance metrics, aiming to identify the optimal block sizes for different matrix dimensions.

These experiments will provide insights into the relationship between block sizes and encryption efficiency. By optimizing block sizes based on the characteristics of the data and the computational tasks involved, we can enhance the overall performance of our encryption scheme for diverse application scenarios.

7.3 Bootstrapping

Bootstrapping in homomorphic encryption is a necessary process to control the noise growth within ciphertexts, ensuring the system's correctness. The point at which bootstrapping must occur is when the accumulated noise reaches a threshold that could jeopardize decryption accuracy. This threshold is pivotal in the design and operation of homomorphic encryption systems, as it dictates the frequency of bootstrapping and, consequently, impacts the system's overall efficiency. Discussing this aspect sheds light on the balancing act between extending computational capabilities and managing computational overhead, a critical consideration in the practical deployment of homomorphic encryption technologies.

ACKNOWLEDGMENTS

Thank you to Professor Ayday for his help during this project. We would like to acknwledge our use of the CWRU-HPC to run testing on our implementation.

REFERENCES

[1] R.C. Agarwal and C.S. Burrus. 1975. Number theoretic transforms to implement fast digital convolution. *Proc. IEEE* 63, 4 (1975), 550–560. https://doi.org/10.1109/PROC.1975.9791

[2] Alberto Ibarrondo. 2023. PyFHEl Documentation. https://pyfhel.readthedocs.io/en/latest/. Accessed: 3/7/2024.

[3] Ahmad Al Badawi, Jack Bates, Flavio Bergamaschi, David Bruce Cousins, Saroja Erabelli, Nicholas Genise, Shai Halevi, Hamish Hunt, Andrey Kim, Yongwoo Lee, Zeyu Liu, Daniele Micciancio, Ian Quah, Yuriy Polyakov, Saraswathy R.V., Kurt Rohloff, Jonathan Saylor, Dmitriy Suponitsky, Matthew Triplett, Vinod Vaikuntanathan, and Vincent Zucca. 2022. OpenFHE: Open-Source Fully Homomorphic Encryption Library. Cryptology ePrint Archive, Paper 2022/915. https://eprint.iacr.org/2022/915 https://eprint.iacr.org/2022/915.

[4] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. 2014. (Leveled) Fully Homomorphic Encryption without Bootstrapping. *ACM Trans. Comput. Theory* 6, 3, Article 13 (jul 2014), 36 pages. https://doi.org/10.1145/2633600

[5] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. 2017. Homomorphic Encryption for Arithmetic of Approximate Numbers. In *Advances in Cryptology – ASIACRYPT 2017*, Tsuyoshi Takagi and Thomas Peyrin (Eds.). Springer International Publishing, Cham, 409–437.

[6] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. 2016. Faster Fully Homomorphic Encryption: Bootstrapping in less than 0.1 Seconds. Cryptology ePrint Archive, Paper 2016/870. https://eprint.iacr.org/2016/870 https://eprint.iacr.org/2016/870.

[7] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. 2017. Improving TFHE: faster packed homomorphic operations and efficient circuit bootstrapping. Cryptology ePrint Archive, Paper 2017/430. https://eprint.iacr.org/2017/430 https://eprint.iacr.org/2017/430.

[8] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. 2018. TFHE: Fast Fully Homomorphic Encryption over the Torus. Cryptology ePrint Archive, Paper 2018/421. https://eprint.iacr.org/2018/421 https://eprint.iacr.org/2018/421.

[9] Wei Dai. 2018. cuFHE - *CUDA-accelerated Fully Homomorphic Encryption Library*. https://github.com/vernamlab/cuFHE

[10] Wei Dai and Berk Sunar. 2016. cuHE: A Homomorphic Encryption Accelerator Library. In *Cryptography and Information Security in the Balkans*, Enes Pasalic and Lars R. Knudsen (Eds.). Springer International Publishing, Cham, 169–186.

[11] Saroja Erabelli. 2020. pyFHE - *A Python Library for Fully Homomorphic Encryption*. https://github.com/sarojaerabelli/py-fhe

[12] S. Fan, Z. Wang, W. Xu, R. Hou, D. Meng, and M. Zhang. 2023. TensorFHE: Achieving Practical Computation on Encrypted Data Using GPGPU. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE Computer Society, Los Alamitos, CA, USA, 922–934. https://doi.org/10.1109/HPCA56546.2023.10071017

[13] Austin J. Hartshorn, Humberto A. Leon Liu, Noel F. Qiao, and Scott S. Weber. 2020. *Number Theoretic Transform (NTT) FPGA Accelerator*. Technical Report.

[14] HuggingFace. 2023. How to add a model to HuggingFace Transformers. https://huggingface.co/docs/transformers/add_new_model

[15] Jae Hyung Ju, Jaiyoung Park, Jongmin Kim, Donghwan Kim, and Jung Ho Ahn. 2023. NeuJeans: Private Neural Network Inference with Joint Optimization of Convolution and Bootstrapping. arXiv:2312.04356 [cs.CR]

[16] Hunter Lightman, Vineet Kosaraju, Yura Burda, Harri Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. 2023. Let's Verify Step by Step. arXiv:2305.20050 [cs.LG]

[17] Arthur Meyre, Benoit Chevallier-Mames, Jordan Frery, Andrei Stoian, Roman Bredehoft, Luis Montero, and Celia Kherfallah. 2022. Concrete ML: a Privacy-Preserving Machine Learning Library using Fully Homomorphic Encryption for Data Scientists. https://github.com/zama-ai/concrete-ml.

[18] OpenAI. 2023. GPT-4 Technical Report. arXiv:2303.08774 [cs.CL]

[19] Ardianto Satriawan, Infall Syafalni, Rella Mareta, Isa Anshori, Wervyan Shalannanda, and Aleams Barra. 2023. Conceptual Review on Number Theoretic Transform and Comprehensive Review on Its Implementations. *IEEE Access* 11 (2023), 70288–70316. https://doi.org/10.1109/ACCESS.2023.3294446

[20] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. 2023. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288* (2023).

[21] Zama. 2022. Concrete: TFHE Compiler that converts python programs into FHE equivalent. https://github.com/zama-ai/concrete.

[22] Zama. 2022. TFHE-rs: A Pure Rust Implementation of the TFHE Scheme for Boolean and Integer Arithmetics Over Encrypted Data. https://github.com/zama-ai/tfhe-rs.

[23] Ali Şah Özcan and Erkay Savaş. 2023. Two Algorithms for Fast GPU Implementation of NTT. Cryptology ePrint Archive, Paper 2023/1410. https://eprint.iacr.org/2023/1410 https://eprint.iacr.org/2023/1410.