PulpFHE: Complex Instruction Set Extensions for FHE Processors

Omar Ahmed and Nektarios Georgios Tsoutsos

University of Delaware
{oaaa, tsoutsos}@udel.edu

Abstract. The proliferation of attacks to cloud computing, coupled with the vast amounts of data outsourced to online services, continues to raise major concerns about the privacy for end users. Traditional cryptography can help secure data transmission and storage on cloud servers, but falls short when the already encrypted data needs to be processed by the cloud provider. An emerging solution to this challenge is fully homomorphic encryption (FHE), which enables computations directly on encrypted data, and recent works have focused on developing new processor designs tailored for native processing of FHE data. In this work, we introduce PulpFHE, an optimized instruction set extension tailored for the next generation of FHE processors. Our proposed FHE instructions offer native support for non-linear operations on encrypted data, and enable significantly faster homomorphic computations for a broad range of realistic applications.

Keywords: Homomorphic Encryption, Privacy-Preserving Computation, Encrypted Processor, Cloud Computing.

1 Introduction

The recent advancements in cloud technologies and the convenience of on-demand computing resources for data storage and processing have motivated organizations and individuals to opt for outsourced computations in place of in-house resources. This shift helps avoid the complexities of managing a network of servers, including the maintenance, and administration costs of owning the hardware. At the same time, cloud servers have become attractive targets for cyber criminals; a recent survey reports 26 distinct attacks, each threatening the sensitive data stored in the cloud, as well as the availability of these services [21]. Likewise, a 2021 study on cloud security [1] reports that one of the most challenging aspects of cloud computing is gaining the trust of cloud users, as consumers remain reluctant to accept the privacy guarantees offered by cloud providers. Indeed, even with robust hardening techniques for protecting the network and server infrastructure against cyberattacks, user privacy can still be compromised by insider threats that attempt to access users' sensitive data.

One potential solution is to leverage traditional cryptography, which offers privacy protections for sensitive data during storage or transit. Nevertheless, *processing* encrypted data remains incompatible with traditional cryptosystems; as a result, a cloud server would need to retain a copy of the user's secret key to decrypt and process the data, which is yet another target for cyberattacks. Effectively, to support data processing, users' sensitive data must be exposed to the cloud provider, and a semi-honest cloud may leverage this data for profit (e.g., to monitor user behavior or offer targeted advertisement).

An emerging solution to this challenge is to use Fully Homomorphic Encryption (FHE) [10]. This special form of cryptography represents a significant advancement over traditional cryptosystems, as it allows arbitrary computations to be performed *directly on encrypted data* without the need for any decryption. In effect, this allows sensitive data to remain encrypted at all times, both during transmission and storage, as well as during processing by cloud servers.

While FHE is powerful and versatile, harnessing its benefits comes with several challenges. First, an inherent property of each FHE ciphertext is its *noise level* [10], which ensures the ciphertext remains secure. During processing, however, this noise increases and requires careful management to remain under a critical threshold (beyond which the ciphertext is destroyed). As a result, programmers need to carefully tune FHE encryption parameters and manually apply noise reduction methods to implement meaningful algorithms on encrypted data. Thus, while several FHE libraries have been developed, such as Microsoft Seal [20] and OpenFHE [2], configuring the right parameter set, and managing FHE noise growth during computation still requires significant effort from programmers.

A second challenge is the computational cost of FHE itself, which is mostly attributed to the native encoding of FHE ciphertexts as very large polynomials of high degree. As a result, a typical ciphertext can be several kilobytes in size, and homomorphic operations over FHE data can be significantly slower compared to their plaintext counterparts [12]. Notably, since modern computer architectures, such as x86 and ARM, do not offer native support for FHE operations, programmers have to abstract the algebraic operations to an underlying software library.

Another challenge is the fact that writing programs over encrypted data is significantly harder compared to writing the equivalent programs for regular (plaintext) data. FHE supports a functionally complete set of operations, namely addition and multiplication, so expressing arbitrary algorithms takes the form of arithmetic or Boolean circuits [13], where input and intermediate ciphertexts are "wires", and FHE operations are "gates" of the conceptual circuit. Therefore, basic programming primitives, such as assignments and control flow decisions, should be expressed using a dataflow model, rather than an imperative paradigm that regular software programmers are familiar with.

To address these usability and performance challenges of FHE, the research community has proposed native FHE processor architectures. One of the most recent works, the open-source Juliet processor [11], allows non-crypto savvy programmers to express arbitrary programs over encrypted data as a sequence of high-level instructions. In this case, each instruction is implemented as a circuit of FHE Boolean gates using the state-of-the-art TFHE [6] and cuFHE [7] libraries.

The Instruction Set Architecture (ISA) of recent FHE processors offers native support for basic bitwise and arithmetic operations to manipulate encrypted data, yet it lacks more complex instructions that are frequently used in realistic applications on FHE data. For example, the Juliet processor does not natively support ReLU or SQRT instructions that are inherent to machine learning or data analytics applications. Instead, programmers have to write lengthy functions using the provided domain-specific language that comes with the FHE processor to simulate such operations using the more primitive operations of the ISA.

Motivated by the limitations of current state-of-the-art FHE processors, in this work, we introduce PulpFHE: a general-purpose ISA extension that brings new complex instructions to encrypted processor architectures. The PulpFHE instructions are implemented directly as FHE circuits that are judiciously optimized for performance. Notably, PulpFHE is designed for high parallelism that is tailored for multi-core hardware targets. Overall, our contributions are summarized as follows:

- We design the PulpFHE ISA extensions, enabling a more concise and usable programming model for end-users.
- We pair PulpFHE with a new set of complex functional units for homomorphic evaluation, bringing high-performance non-linear operations to modern FHE processors.
- We introduce low-level performance optimizations and runtime parallelism, while abstracting the complexities of evaluating homomorphic circuits from the user.

2 Preliminaries

2.1 Homomorphic Encryption Schemes

There are several homomorphic encryption schemes, and each one offers different capabilities and encoding for the underlying computation. At a high level, HE schemes are classified into three classes: partially homomorphic schemes, leveled homomorphic schemes, and fully homomorphic schemes.

Partial HE (PHE) supports only one operation, either multiplication or addition, to be applied an unlimited number of times between ciphertexts. Examples of PHE include the Paillier, RSA, and ElGamal systems. Conversely, a leveled homomorphic encryption (LHE) scheme allows both addition and multiplication to be performed between ciphertexts. Nevertheless, a downside of LHE is that it does not support evaluating very deep arithmetic circuits because of ciphertext noise growth. The most prominent LHE schemes are CKKS [4], BFV [9], and BGV [3], which are supported by SEAL [20] and OpenFHE [2] homomorphic encryption libraries.

The most powerful form of HE is Fully Homomorphic Encryption (FHE) that supports evaluating arithmetic circuits of any size with both addition and multiplication operations. In particular, FHE employs a special noise reduction technique called *bootstrapping*, which allows an unbounded number of operations on ciphertexts by preventing the noise from corrupting the underlying values. At the same time, since bootstrapping is the most intensive operation when evaluating an FHE circuit, the cost of FHE is higher than LHE. The current state of the art in FHE is the CGGI [5] cryptosystem that is implemented by both the TFHE [6] and OpenFHE [2] libraries.

2.2 Related Work

The authors of [19] have proposed a hardware accelerator for FHE operations. This design implements a scheduler and hardware units for expensive FHE primitives such as number-theoretic transforms (NTTs) and modular arithmetic operations. Their approach focused on minimizing the data movements within FHE programs.

Likewise, the authors of [14] have proposed runtime optimizations for FHE operations by introducing parallelism on both the ciphertext level and the operation level. Their approach focused on accelerating bootstrapping, as well as the key and modulus switching operations of FHE. Similarly, the authors of [18] focused on developing architectures for accelerating the underlying NTTs, given the high cost of these operations.

A parallel research direction aims to make FHE more usable by developing custom compilers for expressing FHE programs in a high-level language before translating them to the target FHE scheme. For instance, the authors of [8] present a bespoke compiler for neural network inference on the FHE domain. Likewise, HECATE [15] is a compiler framework that aims to reduce ciphertext sizes to improve runtime performance.

3 Our PulpFHE Extensions

Existing FHE processors support only basic instructions, which renders them less usable for real-life applications, such as machine learning. While the end user could simulate the missing functionality using more primitive instructions, this goes against one of the claimed goals of such frameworks, which is enhancing usability. For instance, the encrypted division operation is not natively supported by modern FHE processors; even if this basic operation could be implemented using existing instructions, the outcome will be less efficient compared to an atomic division instruction.

To address this challenge, we propose the PulpFHE processor-agnostic ISA extensions that complement FHE processor designs and enable new functionalities, such as non-linear operations. Moreover, a key objective in our approach is to further optimize the performance of native FHE instructions, which are the building blocks for more complex constructions. Towards that end, we investigate how the basic instructions, such as addition, can be decomposed to independent primitives that can be parallelized for significantly faster runtime evaluation.

We summarize our complex ISA extensions in Table 1. Notably, implementing these new FHE instructions in a parallel fashion is a challenging process that requires judicious scheduling of the gates comprising the target homomorphic circuit. For example, consider a Blake3 instruction that involves additions and other bitwise operations of an encrypted value. A naive implementation would yield a large sequential circuit; however, PulpFHE identifies independent parts of the circuit that can be evaluated in parallel, such as XORing and rotating, thereby reducing the overall circuit depth.

In the next paragraphs, we discuss six families of new instructions in PulpFHE. We also detail the algorithms for the division and square root instructions, as their implementation in the encrypted domain is a major challenge due to noise management. For the sake of succinctness, we have abstracted subtle implementation details (such as data movements between registers), as well as algorithms for instructions where the underlying function is well-known.

Table 1. Twelve new complex instruction implemented in PulpFHE

Instruction	**Description**	**Instruction**	**Description**
div(A,B)	Returns A / B	mod(A,B)	Returns A % B
max(**A**)	The maximum value in the list **A**	min(**A**)	The minimum in the list **A**
mean(**A**)	The average of the values in the list **A**	sqrt(A)	The square root of A
std(**A**)	The standard deviation of the values in **A**	var(**A**)	The variance of the values in the list **A**
relu(A)	Returns the rectified linear unit of A	blake3(M_0,M_1)	A quarter-round of the Blake3 hash
rotr(A)	Right-rotate the bits of A	rotl(A)	Left-rotate the bits of A

3.1 Encrypted Division and Modulus

To the best of our knowledge, no existing FHE library implements an encrypted division operation. While prior works in [16, 17] support division, these approaches assume that the divisor is public. Conversely, PulpFHE implements a fast integer division instruction where both the dividend and the divisor are encrypted values. As a result, using this operation, PulpFHE can further support an encrypted modulus operation over FHE. We remark that implementing division as an atomic instruction is a challenge, as it requires iterative algorithms and comparisons which introduce large amount of FHE noise.

The division instruction in PulpFHE inputs two encrypted values that are encoded in a two's complement binary representation. This allows us to evaluate the division algorithm using additions and bit-wise operations, yielding the parallelizable instruction of Algorithm 1.

Algorithm 1: Encrypted Division in PulpFHE

```
   Data: $N_1$, $N_2$, $nbits$
1  $Q \leftarrow N_1$
2  $A \leftarrow 0$
3  $M \leftarrow N_2$
4  for $i \leftarrow 0$ to $nbits$ by $1$ do
5      $Q, msb \leftarrow shift\_left(Q, 0)$
6      $A_{tmp}, \_ \leftarrow shift\_left(A, msb)$
7      $A \leftarrow A_{tmp} - M$
8      $A_{msb} \leftarrow A[nbits - 1]$
9      $not\_A_{msb} \leftarrow NOT(A_{msb})$
10     $Q[0] \leftarrow not\_A_{msb}$
11     $A \leftarrow A_{msb}\ ?\ A + M : A$
12 end
13 return $Q$, $A$
```

In more details, the division instruction operates on arrays of encrypted bits of size $nbits$. Index 0 of each array holds the least significant bit (LSB), while the last index, $nbits - 1$, holds the most significant bit (MSB). Line 11 is a ternary statement that selects between either the value $A + M$ or A based on the bit value of A_{msb}. Notably, changing the runtime control flow of an encrypted program is not possible since the value that is used a condition is encrypted, and thus the outcome of the selection not known at runtime. In our approach, we employ an encrypted multiplexer gate, which works as follows:

$$A \leftarrow A_{msb} \times (A + M) + (1 - A_{msb}) * A \tag{1}$$

Therefore, if A_{msb} is 0 (false), then A does not change ($A \leftarrow A$); otherwise, if A_{msb} is 1 (true), then A becomes equal to $A + M$.

3.2 Max, Min, and ReLU

PulpFHE offers three complex comparison instructions: maximum, minimum, and a Rectified Linear Unit (ReLU). Here, the `max` and `min` instructions operate on a vector of encrypted values, while `relu` operates on a single encrypted value. Computing the maximum and the minimum value of list values is a common operation in many applications that involve sorting, searching, or data analysis. Likewise, ReLU is a widely used activation function in machine learning applications, which adds non-linearity in neural networks. If the encrypted input is positive, ReLU returns it directly; otherwise, it outputs an encrypted zero.

Like division, implementing these comparison instructions entails iterating over the encrypted bits of the input values, comparing the corresponding bits, and handling the carry bit from the lower bits comparisons. As discussed earlier, these instructions aim to maximize independent code sequence to allow for parallel execution.

3.3 Rotations

There are numerous privacy-preserving applications that require manipulating bits of data, such as data compression algorithms, hashes, and error detection algorithms. PulpFHE introduces the new `rotr` and `rotl` instructions that enable right and left rotations by manipulating the encoding of the encrypted integer values. These rotations are critical for implementing more complex instructions, such as the Blake3 quarter-round.

3.4 Square Root

The square root (sqrt) is another key operation that is not natively supported by existing FHE schemes. The non-linear nature of the square root makes it very challenging for end users to implement accurately due to excessive noise growth. PulpFHE implements an efficient and accurate integer square root instruction, as presented in Algorithm 2. Here, *n_iters* is a constant corresponding to the number of iterations required to achieve the required accuracy. According to our experiments, 5 iterations are sufficient for 8-bit numbers, 10 iterations work well for 16-bit numbers, and 20 iterations are required for 32-bit numbers.

Algorithm 2: Square Root

```
  Data: $N$
1 $X \leftarrow N$
2 $Y \leftarrow 0$
3 for $i \leftarrow 0$ to $n\_iters$ by $1$ do
4     $tmp \leftarrow X + Y$
5     $X \leftarrow tmp/2$
6     $Y \leftarrow N/X$
7 end
8 return $X$
```

3.5 Mean, Variance, and Standard Deviation

In real-life applications where users outsource a computation, they typically have large inputs to be analyzed. In this case, a common requirement is to compute statistical metrics over the input data. The most common operations in this case are the mean, the variance, and the standard deviation. Notably, these statistical measures are non-linear functions since they use the division and square root operations, which make their FHE implementation a challenge. Like `min` and `max` discussed earlier, PulpFHE supports `std`, `var` and `mean` over a list of encrypted values an a native instruction, which offers better usability and performance.

3.6 Blake3 Quarter Round

Hash functions are cryptographic primitives that consume an input of arbitrary length and produce a fixed-length digest that looks random. One such function is Blake3[1] which is released in 2020. The motivation behind supporting Blake3 in PulpFHE is to enable homomorphically hashing encrypted data, which offers integrity protections. Using our new atomic instruction, a user implements the Blake3 hash and authenticate her sensitive data without ever decrypting. In PulpFHE, the `blake3` instruction takes four initial vectors and two 32-bit messages to execute a quarter round.

One challenge in implementing the Blake3 algorithm is the large inputs it consumes. Our design takes this into consideration and minimizes the data movement between the registers of the FHE processor. Furthermore, our implementation allows each quarter round to be executed in parallel with other quarter rounds, thus each instruction can operate on independent parts of the original data.

3.7 PulpFHE in Action

The PulpFHE ISA extensions are agnostic to the underlying encrypted processor design. Without loss of generality, we deploy PulpFHE over the Juliet processor [11]. Notably, the original design of Juliet is limited by only 15 basic instructions, while our PulpFHE extensions brings the total to 27 instructions. As Juliet employs the eJava programming language to express programmer's intent, we defined new built-in functions in eJava to realize our new instruction set. One of the limitations of Juliet is that its basic instruction set is inherently sequential; conversely, PulpFHE parallelizes the implementation of new and existing instructions for the target hardware to improve performance.

In more detail, we have scrutinized all encrypted instructions for independent code segments that can be executed in parallel. For instance, PulpFHE comes with its optimized adder circuit instead of using the original Juliet circuit (that uses a basic adder comprising two XOR gates, two AND gates and an OR gate). Although a basic circuit has a small number of gates, it is not parallelizable. Instead, PulpFHE employs the Kogge-Stone adder circuit in FHE, which has more logic gates, but offers faster performance when executed in parallel.

4 Experimental Results

To evaluate the efficiency of PulpFHE extensions, we incorporate our extended ISA to the open-source Juliet processor. Then, to assess the performance optimizations attributed to PulpFHE, we run the same set of benchmarks both on the original Juliet, as well as on Juliet with our optimized PulpFHE extensions. To maintain a comprehensive and fair comparison, we use the same exact configurations and the same exact code of the benchmarks as used in [11], except for varying the problem size to offer a more comprehensive comparison between the two architectures.

Table 2 presents two sets of experiments. On the left side we compare Juliet against PulpFHE using eight benchmarks from [11]. In addition, on the right side of the Table we compare the individual instructions in PulpFHE against the same operations implemented in Juliet; we remark that since our new instructions were not available in Juliet, we implemented them in the eJava using existing operations (e.g., shifting and additions), as any user would. In all cases, the target benchmarks were compiled into FHE circuits and evaluated on both processors using FHE data. We also note that our experiments table excludes the evaluation of our rotation instructions, as their execution time on both architectures was negligible (less than 1 second), although PulpFHE demonstrated faster performance. Overall, PulpFHE offers a performance improvement of up to 5.4× for the Simon cipher, and up to 28× for the min/max instruction.

All our experiments were conducted on a Linux machine featuring an Intel i9-12900K CPU with 24 cores and 128 GB of RAM. From our experiments, we report that our PulpFHE extensions greatly outperform the original processor design of Juliet, both for benchmark programs and for individual instructions. Likewise, our eJava upgrades, which adapt the new instructions, enable apples-to-apples comparisons and further

[1] https://github.com/BLAKE3-team/BLAKE3/

improve the usability of the whole system. As a result, PulpFHE enables shorter, more readable, and easier-to-maintain codebases.

Table 2. The performance comparison across different program benchmarks and individual instructions between the original Juliet architecture vs. Juliet with PulpFHE extensions. All running times are in seconds. Cells marked with ∅ indicate that the corresponding instruction did not finish within 60 minutes.

Prog.	**Size**	**PulpFHE**	**Juliet**	**Speedup**	**Inst.**	**Size**	**PulpFHE**	**Juliet**	**Speedup**
Simon Cipher	8 bits	4.44	13.39	301%	Blake3	32 bits - 1 round	3.69	9.47	257%
	16 bits	7.36	33.48	455%		32 bits - 4 rounds	13.66	38.17	279%
	32 bits	12.02	65.57	545%		32 bits - 8 rounds	60.12	157.49	262%
Speck Cipher	8 bits	9.69	11.06	114%	Division/ modulus	8 bits	4.55	46.02	1012%
	16 bits	18.98	29.15	154%		16 bits	18.60	138.35	744%
	32 bits	37.10	58.77	158%		32 bits	71.18	464.79	653%
Logistic Regr. Inference	8 bits	32.41	35.65	110%	Max/Min	8 bits - 8 values	2.04	32.00	1572%
	16 bits	112.43	129.39	115%		16 bits - 16 values	7.91	224.04	2831%
	32 bits	413.80	491.50	119%		32 bits - 32 values	29.90	800.62	2678%
Fibonacci	8 bits - 8 values	51.34	58.81	115%	Mean	8 bits - 8 values	5.79	46.42	802%
	16 bits - 16 values	150.98	173.47	115%		16 bits - 16 values	23.84	287.03	1204%
	32 bits - 32 values	487.18	577.64	119%		32 bits - 32 values	90.20	1892.60	2098%
Fibonacci (mux)	8 bits - 8 values	18.01	22.41	124%	ReLU	8 bits	1.59	2.34	147%
	16 bits - 16 values	36.14	44.81	124%		16 bits	5.39	7.58	141%
	32 bits - 32 values	69.88	89.50	128%		32 bits	28.01	38.06	136%
Factorial	8 bits - up to 5	80.88	103.42	128%	SQRT	8 bits	41.26	440.11	1067%
	16 bits - up to 8	301.83	327.73	109%		16 bits	359.45	∅	-
	32 bits - up to 12	948.11	1118.49	118%		32 bits	2767.04	∅	-
Sieve Of Eratosthenes	8 bits - up to 20	28.65	31.26	109%	Standard deviation	8 bits - 8 values	239.05	1061.69	444%
	16 bits - up to 40	212.03	240.66	114%		16 bits - 16 values	484.90	∅	-
	32 bits - up to 60	1234.38	1408.62	114%		32 bits - 32 values	3590.05	∅	-
Private Info. Retrieval	8 bits - 8 values	37.08	44.42	120%	Variance	8 bits - 8 values	24.19	108.72	449%
	16 bits - 16 values	72.49	89.76	124%		16 bits - 16 values	135.30	707.98	523%
	32 bits - 32 values	143.10	177.04	124%		32 bits - 32 values	840.62	∅	-

5 Conclusion

In this work, we present the PulpFHE ISA extensions for modern FHE processors. PulpFHE not only introduces new encrypted instructions that improve runtime performance, but also enables non-linear operations, such as division for the first time. PulpFHE is designed to pair with modern encrypted processors and bring new parallelizable instructions for real-life applications. Furthermore, our new instruction set can easily be adopted by high-level languages, enabling a seamless and effortless coding experience for users. In terms of performance, PulpFHE has demonstrated notable improvements, compared to the original Juliet processor.

Acknowledgments

This work has been supported by NSF Award #2239334.

References

1. Alouffi et al. A systematic literature review on cloud computing security: threats and mitigation strategies. *IEEE Access*, 9:57792–57807, 2021.
2. Ahmad Al Badawi et al. Openfhe: Open-source fully homomorphic encryption library. Cryptology ePrint Archive, Paper 2022/915, 2022. https://eprint.iacr.org/2022/915.

3. Brakerski et al. (leveled) fully homomorphic encryption without bootstrapping. *ACM Transactions on Computation Theory (TOCT)*, 6(3):1–36, 2014.
4. Cheon et al. Homomorphic encryption for arithmetic of approximate numbers. In *Advances in Cryptology–ASIACRYPT 2017: 23rd International Conference on the Theory and Applications of Cryptology and Information Security, Hong Kong, China, December 3-7, 2017, Proceedings, Part I 23*, pages 409–437. Springer, 2017.
5. Po-Wen Chi, Yun-Hsiu Lu, and Albert Guan. A privacy-preserving zero-knowledge proof for blockchain. *IEEE Access*, 2023.
6. Chillotti et al. Tfhe: fast fully homomorphic encryption over the torus. *Journal of Cryptology*, 33(1):34–91, 2020.
7. W Dai and B Sunar. Cuda-accelerated fully homomorphic encryption library, 2018.
8. Dathathri et al. Chet: an optimizing compiler for fully-homomorphic neural-network inferencing. In *Proceedings of the 40th ACM SIGPLAN conference on programming language design and implementation*, pages 142–156, 2019.
9. Junfeng Fan and Frederik Vercauteren. Somewhat practical fully homomorphic encryption. Cryptology ePrint Archive, Paper 2012/144, 2012. https://eprint.iacr.org/2012/144.
10. Craig Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the forty-first annual ACM symposium on Theory of computing*, pages 169–178, 2009.
11. Gouert et al. Juliet: A configurable processor for computing on encrypted data. *IEEE Transactions on Computers*, pages 1–14, 2024.
12. Charles Gouert et al. SoK: New Insights into Fully Homomorphic Encryption Libraries via Standardized Benchmarks. *Proceedings on Privacy Enhancing Technologies*, 2023(3):154–172, July 2023.
13. Charles Gouert, Dimitris Mouris, and Nektarios Georgios Tsoutsos. Helm: Navigating homomorphic encryption through gates and lookup tables. Cryptology ePrint Archive, Paper 2023/1382, 2023. https://eprint.iacr.org/2023/1382.
14. Saransh Gupta, Rosario Cammarota, and Tajana Šimunić. Memfhe: End-to-end computing with fully homomorphic encryption in memory. *ACM Transactions on Embedded Computing Systems*, 23(2):1–23, 2024.
15. Lee et al. Hecate: Performance-aware scale optimization for homomorphic encryption compiler. In *2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 193–204. IEEE, 2022.
16. Morimura et al. Improved integer-wise homomorphic comparison and division based on polynomial evaluation. In *Proceedings of the 17th International Conference on Availability, Reliability and Security*, pages 1–10, 2022.
17. Okada et al. Linear depth integer-wise homomorphic division. In *IFIP International Conference on Information Security Theory and Practice*, pages 91–106. Springer, 2018.
18. Rogério Paludo and Leonel Sousa. Ntt architecture for a linux-ready risc-v fully-homomorphic encryption accelerator. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 69(7):2669–2682, 2022.
19. Samardzic et al. F1: A fast and programmable accelerator for fully homomorphic encryption. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 238–252, 2021.
20. Microsoft SEAL (release 4.1). https://github.com/Microsoft/SEAL, January 2023. Microsoft Research, Redmond, WA.
21. Hamed Tabrizchi and Marjan Kuchaki Rafsanjani. A survey on security challenges in cloud computing: issues, threats, and solutions. *The journal of supercomputing*, 76(12):9493–9532, 2020.