

Juliet: A Configurable Processor for Computing on Encrypted Data

Charles Gouert^{id}, *Graduate Student Member, IEEE*, Dimitris Mouris^{id}, *Graduate Student Member, IEEE*,
Nektarios Georgios Tsoutsos^{id}, *Member, IEEE*

Abstract—Fully homomorphic encryption (FHE) has become progressively more viable in the years since its original inception in 2009. At the same time, leveraging state-of-the-art schemes in an efficient way for general computation remains prohibitively difficult for the average programmer. In this work, we introduce a new design for a fully homomorphic processor, dubbed Juliet, to enable faster operations on encrypted data using the state-of-the-art TFHE and cuFHE libraries for both CPU and GPU evaluation. To improve usability, we define an expressive assembly language and instruction set architecture (ISA) judiciously designed for end-to-end encrypted computation. We demonstrate Juliet’s capabilities with a broad range of realistic benchmarks including cryptographic algorithms, such as the lightweight ciphers SIMON and SPECK, as well as logistic regression (LR) inference and matrix multiplication.

Index Terms—Applied Cryptography, Encrypted Processor, Hardware Acceleration, Homomorphic Encryption, Privacy-Enhancing Technologies, Privacy-Preserving ISA, Private Computation, Secure Cloud Computing.

1 INTRODUCTION

As organizations continue to produce more data, it becomes increasingly difficult and costly to maintain the in-house infrastructure needed to analyze and manage this information: whether it be customer information, corporate records, or a plethora of other documents necessary for an organization to function. The advent of cloud computing has garnered the interest of these companies as an attractive alternative. Nevertheless, one glaring problem with this approach is that the cloud service provider can plausibly view sensitive data stored on their servers. In addition, cloud servers have become a lucrative target of cyberattacks, which increases the risk of sensitive data being compromised. In [1], the authors have identified six distinct attack surfaces for cloud services, accounting for the various interactions between the user, the cloud, and the offered service. As a notable example, researchers were able to mount cross-VM side-channel attacks to leak data from Amazon’s EC2 service [2].

Since users have no definitive control over the security of their data in this outsourced context, they often turn to encryption: the most widely employed mechanism to protect data confidentiality. While standard encryption techniques seem effective in this respect, there is a major downside with this approach: “*how can the cloud process user data while being encrypted?*”. Normally, to perform meaningful computations the users have to download their encrypted data, decrypt locally to retrieve and manipulate the plaintexts, re-encrypt,

and finally re-upload to the cloud server. This process must be repeated every time the data needs to be processed in some way, which introduces major time and communication overhead for the users, and effectively defies the goals of outsourcing to the cloud.

A specialized form of cryptography called *Fully Homomorphic Encryption (FHE)* [3] offers a revolutionary solution to the problem of end-to-end encrypted computation, which allows users to privately outsource their data to an untrusted third party. Contrary to other privacy-preserving technologies like zero-knowledge proofs [4], [5], [6] and secure multi-party computation [7], [8], FHE allows clients to completely outsource a computation to a cloud server securely. Using FHE, one can instruct a third party to perform operations on the outsourced data without ever exposing the plaintext information, so that the third party never learns anything about the input data or the intermediate results. In effect, FHE has the potential to mitigate entire families of eavesdropping and leakage attacks to the cloud, as any disclosed information would always be encrypted.

While FHE is quite powerful, it still remains difficult to use for a programmer with little to no experience in cryptography. For instance, a popular open-source FHE library called *HElib* [9] requires programmers to manually handle key generation, determine various parameters, and initiate special noise maintenance procedures when the FHE ciphertext becomes too noisy to use.

Likewise, FHE computations are a particularly unnatural fit for today’s computers, as existing processor designs are not developed with encrypted computation in mind. For example, native word sizes are typically set to 32 or 64 bits, which are ill-suited to efficiently represent FHE ciphertexts that can be on the order of several kilobytes, even for implementations with small parameters. Further, since FHE algorithms are expressed as arithmetic circuits, constructing

- C. Gouert, D. Mouris and N.G. Tsoutsos are with the Department of Electrical and Computer Engineering, University of Delaware, Newark, DE, 19716, USA. E-mail: {cgouert, jimouris, tsoutsos}@udel.edu.
- This work is supported by NSF Award #2239334.
- Manuscript revised March 15, 2024; accepted May 27, 2024.
- Copyright © 2024 IEEE. Personal use of this material is permitted. However, permission to use this material for any other purposes must be obtained from the IEEE by sending an email to pubs-permissions@ieee.org.

and optimizing FHE computations is a non-trivial problem. When working over encrypted bits, however, the same techniques used in digital circuit design do not necessarily translate in the encrypted domain. For instance, employing wider circuits is more desirable for HE circuits since all gates at the same depth can be executed in parallel. In effect, the underlying hardware used to run an HE circuit should also be taken into consideration when defining the FHE computation: an HE adder built for a CPU with a small number of cores should not be as wide as one designed to run on a GPU backend. Instead, it should be fairly narrow and aim to minimize the total number of gates in the circuit.

To address the usability and performance limitations of FHE, we introduce a new encrypted computer architecture that natively supports all FHE operations. Our approach is realized in a novel processor design, called Juliet, that is capable of running end-to-end FHE computations using both CPU and GPU hardware acceleration. We complement our custom Instruction Set Architecture (ISA) with an optimized abstract machine that implements native homomorphic computations. Moreover, to significantly improve the usability of FHE, we introduce a domain-specific high-level programming language, called eJava, along with a compiler to translate high-level code into Juliet’s assembly language (JAL). Overall, our versatile framework enables programmers and embedded system designers to leverage the numerous benefits of homomorphic encryption and painlessly execute arbitrary programs on encrypted data.

Traditional FHE computation requires the target algorithm to be expressed as a large arithmetic circuit [10], where the difficulty of optimizing such circuits brings significant usability restrictions to programmers. Our key observation to resolve this major limitation is to express arbitrary programs as a sequence of FHE instructions on an encrypted computer with special functional units. These custom functional units implement special FHE circuits for these instructions that are rigorously optimized to reduce latency and tailored to the specific hardware platform running the encrypted computation. The JAL assembly developed for this research enables a wide range of plaintext operations along with their equivalent FHE counterparts to support encrypted computation. Likewise, Juliet’s I/O is enabled using two input streams (dubbed *tapes*): A public tape is exclusively used for plaintext inputs, while the private tape is used solely for encrypted inputs.

Our new JAL assembly instructions are evaluated in the encrypted domain using an underlying FHE cryptosystem. In this work, we selected the highly efficient *TFHE* scheme [11] that supports encrypted Boolean operations as logic gates on ciphertexts. In particular, plaintext inputs are first converted to binary and then encrypted bit by bit into ciphertext arrays. Then, the encrypted data are uploaded to a cloud service that implements the Juliet abstract machine, along with a compiled JAL program. Additionally, we observe that incorporating operations between two plaintext operands as well as mixed operations between plaintext and ciphertext data results in substantial latency reductions compared to solely encrypted operations. As such, Juliet supports an array of these types of operations, which are executed by both an encrypted and plaintext ALU. Internally, Juliet invokes the standard *TFHE* library that is capable

of evaluating approximately 76 FHE logic gate operations per second. To support hardware acceleration on a GPU, Juliet employs the *cuFHE* library to further achieve a $25\times$ acceleration of FHE operations compared to the CPU-only *TFHE* library. Notably, both the *TFHE* and *cuFHE* libraries are among the fastest fully homomorphic encryption implementations in existence today. Overall, our contributions in this work can be summarized as follows:

- We introduce a new Instruction Set Architecture and processor design tailored to encrypted computation.
- We propose an assembly language for encrypted processors and implement a compiler to translate high-level programs to our domain-specific assembly.
- We complement our design with a usable end-to-end framework for compiling and executing JAL programs using FHE, leveraging both CPU- and GPU-based hardware acceleration.

The rest of the article is organized as follows: In Section 2, we offer a brief introduction to the theory of homomorphic encryption and related background information, while in Section 3 we present our *full stack* framework for end-to-end encrypted computation. Our eJava programming language tailored for encrypted computation is presented in Section 4, and our experimental evaluation and analysis are discussed in Section 6. Finally, Section 7 offers a discussion and comparison of related works, and our concluding remarks are presented in Section 8.

2 PRELIMINARIES

2.1 Fully Homomorphic Encryption

Bootstrapping is the mechanism that allows transforming leveled HE schemes into fully homomorphic schemes. With the invention of bootstrapping, FHE was made possible in 2009 by Craig Gentry [3], allowing the homomorphic evaluation of unlimited addition and multiplication operations on encrypted values. Because these two operations form a *functionally complete set*, it is theoretically possible to express any algorithm as an arithmetic circuit of multiplications and additions [3]. This observation is incredibly important for FHE, as it offers a blueprint on how to evaluate any program in the encrypted domain. Moreover, bootstrapping can be used to refresh and reduce the ciphertext noise an unlimited number of times, which enables the evaluation of arbitrarily deep arithmetic circuits (e.g., iterative programs on encrypted data). Internally, bootstrapping works by instructing the cloud to evaluate the FHE decryption circuit *homomorphically* using an encryption of the secret key. This counter-intuitive operation creates a new ciphertext encrypted under the user’s secret key with a significantly reduced noise magnitude [3].

It is important to note that the bootstrapping procedure itself is, somewhat paradoxically, another source of noise, albeit introducing much less noise than it removes. For the user, this means that bootstrapping must be invoked before the entire noise budget required for decryption is depleted. The noise budget consumed by the decryption circuit depends on a number of factors, where the most important is the plaintext modulus, which defines the range of values that a plaintext can take on. Larger plaintext moduli

require more noise budget in order to successfully evaluate the bootstrapping procedure [12]. It is generally difficult to predict the noise budget constraints of bootstrapping with a given set of parameters, so this process requires significant trial and error. Hence, this remains an essential consideration given that several FHE libraries require users to invoke the bootstrapping procedure manually. If the noise budget remaining in a ciphertext is lower than what is required by the bootstrapping procedure, the ciphertext will not decrypt properly and is rendered useless.

Notably, the complexity and computational overhead of bootstrapping depends on the underlying encryption scheme and its parameters, such as the plaintext modulus, the polynomial degrees, and prime chain size [13]. Likewise, the bootstrapping execution time across different FHE constructions and parameter sets can vary from a fraction of a second to upwards of 30 minutes. Thus, similar to determining the minimum noise budget needed before bootstrapping, optimizing for bootstrapping execution time is a challenge. For example, FHE libraries like HELib provide a list of pre-computed parameters for bootstrapping, but these sets are not optimal for all applications [12].

Overall, the bottleneck for unlimited FHE operations is the bootstrapping cost. Several works have focused on improving the efficiency of bootstrapping (e.g., [13], [14]), and have made great strides to enable FHE adoption outside of research circles. As a result, a diverse set of innovative FHE applications have been proposed, ranging from machine learning [15] to biometrics [16].

2.2 The GSW Cryptosystem

Unlike other HE schemes that require key-switching (which are needed for rotations and multiplications), GSW removes the need for large key-switching matrices. Notably, the only information shared with the cloud is a set of public parameters. Internally, GSW uses a different approach compared to previous FHE cryptosystems, by representing ciphertexts as $N \times N$ matrices over \mathbb{Z}_q , where N is a dimension parameter and q is a ciphertext modulus. Likewise, the secret key in GSW is an N dimensional vector over \mathbb{Z}_q whereas the public key is a uniformly random $m \times (N + 1)$ matrix over \mathbb{Z}_q . Formally, a ciphertext c encrypts a plaintext p if the following relationship holds: $c \cdot sk = p \cdot sk + e$, where sk is the secret key vector and e is an error vector [17]. In this case, decryption of c entails extracting the i -th row of c , computing $x = \langle c_i, sk \rangle$ and finally evaluating $p = \lfloor x / sk_i \rfloor$.

In GSW, homomorphic addition and multiplication of ciphertexts c_1, c_2 mirrors matrix addition and multiplication on plaintexts p_1, p_2 respectively. More formally, we have the following relationships:

$$\begin{aligned} c_{sum} \cdot sk &= (p_1 + p_2) \cdot sk + (e_1 + e_2), \\ c_{prod} \cdot sk &= (p_1 \cdot p_2) \cdot sk + p_2 \cdot e_1 + c_1 \cdot e_2, \end{aligned}$$

where $c_{sum} = c_1 + c_2$ and $c_{prod} = c_1 \cdot c_2$. These equations show that the noise accumulates much faster for multiplication versus addition. In fact, if the plaintext values and elements of the ciphertext matrices are not kept small relative to q , the noise explodes. To enforce this and allow for more levels and evaluations within the noise budget, GSW introduces a new noise mitigation technique called

flattening. The latter consists of a series of operations that modify vectors without changing the dot product and is used to scale the ciphertext matrices after every encryption, addition, or multiplication operation. Finally, we observe that this scheme becomes fully homomorphic by incorporating a bootstrapping procedure. In this case, the GSW evaluation key holds a series of encryptions of the private key and does not require key switching matrices (i.e., its size is relatively small compared to other HE cryptosystems).

2.3 Homomorphic Encryption Libraries

There are a variety of libraries that implement HE developed by companies such as Microsoft and IBM, as well as groups of academic researchers. All libraries have advantages and disadvantages arising both from the underlying HE scheme as well as the implementation itself. One of the first was IBM HELib, which supports both the BGV [18] and CKKS [19] schemes. HELib exposes addition and multiplication operations over ciphertexts and can pack multiple plaintexts into a single ciphertext (due to the batching properties of both cryptosystems). The user is responsible for setting most parameters, which offers a high degree of control at the cost of usability. Notably, when using BGV in an FHE context, bootstrapping must be manually invoked in most cases, so ciphertext noise must be closely monitored. Depending on parameter choices, a single bootstrap can take from several seconds to several minutes, which makes the library impractical for deep arithmetic circuits.

Likewise, Microsoft SEAL implements the BFV [20], BGV, and CKKS HE schemes, and provides an intuitive API for developers [21]. Nevertheless, SEAL does not support FHE as bootstrapping is not supported. Thus, it is only suitable for applications that have a small depth, which excludes most computationally complex algorithms.

Conversely, the FHEW library which implements a ring-variant of GSW, only supports an FHE mode of operation and is oriented towards Boolean gates [22], as opposed to adds and multiplies on encrypted data. Each ciphertext encrypts a single bit of plaintext and the supported HE gates consume two ciphertexts and output an encrypted result (except for the univariate NOT gate). Since FHEW supports all standard logic gates, it is functionally complete and can orchestrate any arbitrary algorithm as a sequence of gates. Additionally, FHEW achieves faster bootstrapping speeds than BGV, CKKS, and BFV, while its noise reduction operation is invoked automatically during every gate evaluation, as it is integral to the actual computation. The latency cost of a single bootstrap is on the order of hundreds of milliseconds on a CPU as opposed to seconds or minutes for the previously discussed schemes. Nevertheless, one disadvantage of FHEW is that it does not support ciphertext batching, which would allow many plaintext bits to be encrypted into one ciphertext for SIMD-style computation.

Lastly, the TFHE library is also based on a ring variant of GSW and improves upon FHEW by exhibiting even faster bootstrapping speeds [14]. The TFHE bootstrapping procedure exhibits low latency, a few milliseconds on a CPU, for a variety of reasons. First, the parameters employed by both TFHE and FHEW are typically much smaller than those used by the other cryptosystems. As a case in

point, most FHE-CKKS implementations employ a ciphertext polynomial degree of 2^{16} or 2^{17} and each coefficient can be thousands of bits in length [23]. The reason large parameters must be used is to accommodate for the noise introduced by the CKKS bootstrap before it can be refreshed. Conversely, TFHE employs polynomial degrees of 2^{10} , with each coefficient being no more than 32 bits in length as the bootstrapping consumes far less noise budget than that required by schemes like CKKS. Another powerful feature of the bootstrapping of this cryptosystem is that it allows for non-linear logic gate computation is its programmability; while refreshing the noise, it is possible to evaluate a lookup table encoded as a polynomial for no added cost. In the case of Boolean gates, most gates are implemented as a series of linear operations over ciphertext data followed by a bootstrap that evaluates a lookup table dependent on the gate type to generate the final encrypted output.

Additionally, TFHE incorporates an encrypted multiplexer gate, which allows for oblivious selection between two ciphertext inputs using an encrypted select bit. Internally, TFHE treats a 2-to-1 multiplexer as a single gate operation that requires two inherent bootstraps. This is an important improvement over FHEW, which requires three bootstraps to implement multiplexing using four gates (i.e., a NOT gate, two AND gates, and one OR gate). Another benefit of TFHE is that only one settable parameter is available to users with the standard API. This parameter indicates the desired security level and TFHE will choose pre-configured parameters to satisfy the given security requirement. Like FHEW, ciphertext packing is not possible in TFHE and all algorithms must be expressed in terms of Boolean circuits, which can be a limitation for programmers without knowledge of logic synthesis. Nevertheless, TFHE boasts the fastest bootstrapping speed compared to all other FHE libraries, and thus it is selected as the target FHE scheme for Juliet. In addition, we employ cuFHE [23] that implements the TFHE cryptosystem using GPU acceleration and is capable of achieving approximately $25\times$ higher encrypted logic gate throughput compared to the CPU-only TFHE library.

3 FRAMEWORK OVERVIEW

The Juliet computer is designed to run on the cloud to enable secure outsourcing with FHE. A high-level overview of the interactions between the client and cloud is presented in Figure 1: A client generates the necessary cryptographic keys, compiles an encrypted program developed using our high-level eJava language or the Juliet Assembly Language, and loads both encrypted and unencrypted data into two files representing the private and public tapes respectively. These steps are automated using one of Juliet’s supporting tools: the key generation module, the compiler, and the preprocessor that reads sensitive data, encrypts the individual bits of these inputs, and loads the ciphertexts into the private tape file. Once the program, evaluation key, and two tape files are uploaded to the cloud, the Juliet execution engine can begin evaluation.

3.1 Evaluating Juliet Instructions

The cloud runs two concurrent modules to execute an encrypted program with Juliet: an execution engine and

an encrypted ALU. When the execution engine fetches an encrypted instruction, it signals the encrypted ALU which implements various *homomorphic functional units* as TFHE Boolean circuits. The ALU will execute the algorithm on the data indicated by the execution engine, store the result in a heterogeneous memory unit, and signal the execution engine to proceed. For instance, if an encrypted add operation is encountered by the execution engine, it will send a signal to the ALU, which will run an implementation of an encrypted adder module using the inputs provided by the main program. Similarly, to evaluate instructions on plaintexts, the execution engine employs a standard plaintext ALU and stores the result in the memory unit.

From a programmer’s perspective, there are two main data types supported by Juliet: encrypted and unencrypted integers, which can both live in the heterogeneous memory. In both cases, the size of the encoded values is always aligned to Juliet’s word size. The latter can be configured since it informs a trade-off between the range of supported plaintexts versus performance and memory overheads.

Operations on encrypted integers map to a sequence of homomorphic logic gate operations in the TFHE cryptosystem. Under the hood, each logic gate consists of a series of polynomial and primitive HE operations such as bootstrapping and key switching. Through profiling with the `callgrind` tool, we observe that the execution time of all 2-input encrypted logic gates is completely dominated by bootstrapping, which is responsible for 99.99% of the latency of the gate operation. The remaining 0.01% percent consists of linear operations between ciphertexts, such as polynomial addition and subtraction. On the other hand, the NOT gate is composed entirely of linear operations as it does not require bootstrapping. Lastly, bootstrapping composes 98.6% of the execution time of the homomorphic 2:1 MUX, with the remaining time dedicated to linear operations.

For the bootstrapping operation itself, we observe that 70% of the runtime consists of forward and inverse FFT operations, which are utilized for efficient, asymptotically faster polynomial multiplication. Likewise, 12.5% of the execution time is spent doing linear operations, and the remaining time is dedicated to miscellaneous functions, such as memory-related operations. Notably, Juliet supports encrypted computation on GPU platforms, which can greatly improve the speed of certain homomorphic operations. We utilize functional units constructed using the cuFHE library [23]. Since cuFHE lacks native support for encrypted multiplexer gates, we have also expanded the library to incorporate this functionality. The two primary challenges related to GPU acceleration are: (a) exploiting the parallelism in each functional unit to achieve high GPU utilization, and (b) optimizing ciphertext transfers between the host running Juliet’s execution engine and the GPU accelerator. As such, our design considerations for GPU functional units are substantially different than for CPU implementations, and the construction of both our GPU and CPU functional units is presented in Section 4.

3.2 Juliet I/O: Communicating with Users

Users can upload inputs to the execution engine in two ways: the public tape and the private tape. The public

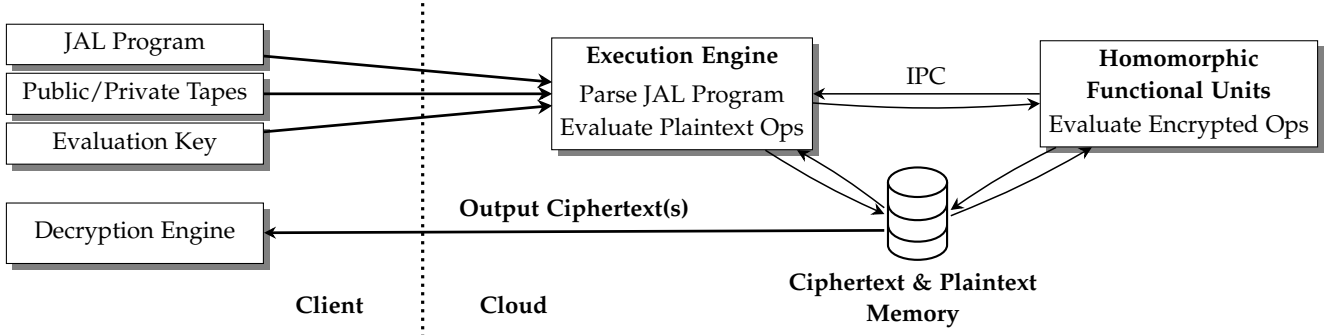


Fig. 1. **Juliet Framework:** The user supplies the cloud with a set of instructions written in the Juliet Assembly Language (JAL) along with an evaluation key (which is necessary to carry out the fully homomorphic operations) and two tapes containing non-sensitive data as well as encrypted data for use with the uploaded program. After successful evaluation, the cloud sends the resulting ciphertexts back to the user for decryption. The thick arrows represent transferring ciphertext while the thin arrows represent control signals and small data transfers (in the order of tens of bytes).

tape consists of plaintext constants that will be used in the Juliet program and each value is read sequentially from the beginning of the tape to the end. In effect, when a *load from the public tape* instruction is issued, the next tape value is popped into a processing queue. To use these constants in computations involving encrypted data, they must be encoded first. Unlike regular encryption of sensitive data, instead, constants can be encoded with a “constant gate” operation and are treated as “trivial” ciphertexts. The underlying constants encoded by these special ciphertexts are not private and no noise is injected during the encoding process. Thus, all data placed in the public tape should be assumed to be readily accessible by all parties. We remark that non-sensitive data should always be loaded in the public tape, as mixed operations between public data and secure ciphertexts are slightly cheaper than operations only between ciphertexts and result in far less communication overhead as these values can be transmitted to the cloud in plaintext form. Additionally, this mechanism allows the cloud to load its own plaintext inputs into the program.

The private tape is read in the same way as the public tape (i.e., it behaves like a read-only queue). The key difference here is that the data stored in the private tape are not the encrypted values themselves, but pointers to the actual encryptions stored in Juliet’s ciphertext memory. In this case, each memory location corresponds to a ciphertext vector whose size is aligned to Juliet’s configured wordsize. For instance, if the processor wordsize is 16, the ciphertext memory unit comprises vectors of 16 TFHE ciphertexts (each encrypting one bit of the word), which amounts to a storage size of approximately $2.2 \times 16 = 35.2$ kilobytes per encrypted word at 80 bits of FHE security. Then, as soon as the computation terminates, the cloud returns the output ciphertexts to the client for decryption.

3.3 Juliet’s Heterogeneous Processor Design

A design for an encrypted processor poses unique challenges and requires additional considerations such as memory systems for large, kilobyte-sized ciphertexts, merging both plaintext and encrypted operations for efficient evaluation, and dealing with the termination problem that is unique to encrypted computation [24]. Contrary to prior works, we propose the first dedicated heterogeneous processor architecture (Figure 2) that can support both FHE computation as well as computation on plaintext values while running efficiently on modern hardware.

The intuition behind supporting plaintext operations is that, in many cases, non-sensitive values will eventually be mixed with sensitive encrypted values. Without plaintext support, all operations will need to be carried out in the encrypted domain, regardless of the sensitivity of the data. Conducting expensive HE operations on public data can increase latency and communication overhead. To increase efficiency, these values should only be encoded as ciphertexts when they are scheduled in a computation with an encrypted value. Hence, non-sensitive values can be processed and modified with fast plaintext operations before encryption. Therefore, we incorporate two ALUs in our design, where the operands of the plaintext ALU are aligned to the chosen wordsize K , and the operands of the encrypted ALU are vectors composed of K ciphertexts.

Our processor adopts a Harvard architecture where data and instruction memories use different address spaces and features two distinct data memory regions: a large encrypted memory bank and a memory for plaintext values. The encrypted region is used to store ciphertext vectors of size K while the plaintext region stores integer entries of K bits. Juliet also features a register file that can store plaintext values directly, as well as ciphertexts through indirection. Specifically, indirection is beneficial so that our registers are aligned to K bits in all cases; when needed to refer to ciphertext data, the registers store a pointer to a ciphertext vector stored in the encrypted memory region.

Memory operations are facilitated by two load-store units (LSUs): a plaintext LSU and an encrypted vector LSU. The plaintext LSU stores and loads K -bit data to and from the plaintext memory. On the other hand, the encrypted vector LSU receives pointers to ciphertext vectors from the registers, and processes inputs and outputs of the encrypted ALU. The input addresses are used to locate and then load ciphertext vectors directly into an encrypted vector bank whose entries correspond to the ciphertext inputs of the ALU operations (Figure 2). When the encrypted ALU finishes an operation, the output ciphertext vector is sent back to the encrypted vector LSU, which stores the output as a new entry in the encrypted memory and also stores the pointer to the entry in the destination register.

Next, we present the instructions supported by Juliet, along with the corresponding ALU functional units for each instruction, targeting CPU and GPU backends.

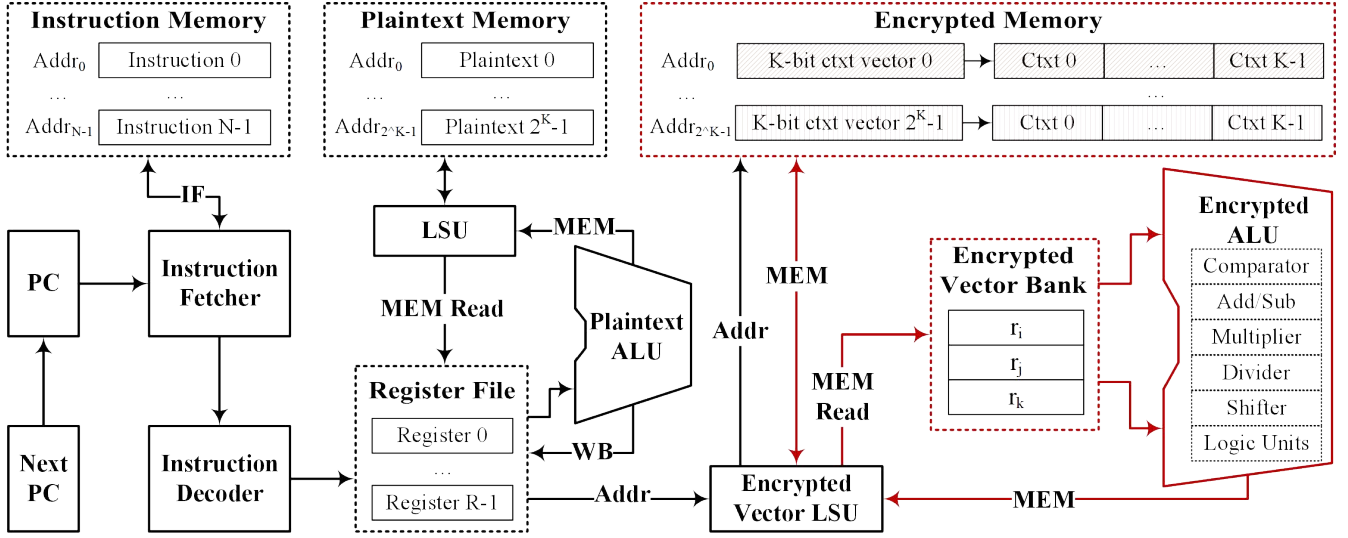


Fig. 2. **Processor Design:** Juliet incorporates a dedicated ALU and memory system for encrypted data. Plaintext operations are fully supported for computation on non-sensitive values and can interface seamlessly with encrypted values after trivial encryption with an evaluation key.

4 JULIET ASSEMBLY LANGUAGE

The Juliet Assembly Language (JAL) instructions are divided into two categories: operations on plaintext data and operations on encrypted data. Juliet supports a robust set of functionally complete operations in both domains as shown in Table 1. An e/p in the **Domain** column means that the instruction has both an encrypted and plaintext variant, whereas a p or e indicates if the instruction is only possible on plaintext data or encrypted data, respectively. Logical and arithmetic operations on plaintext data are carried out in the plaintext ALU, while store and load operations use the LSU to interface with plaintext memory; due to the small size of the values (relative to encrypted values), the data itself can be held directly in the registers. In the next subsections, we discuss the more complex case of encrypted logical, relational and arithmetic operations, followed by the encrypted multiplexer operation that helps resolve runtime decisions, as well as the intricacies of GPU acceleration.

4.1 Encrypted Logical and Relational Operations

Our encrypted ALU implements operations as a netlist of homomorphic gates using the TFHE library on a target device (i.e., CPU or GPU). For encrypted bitwise operations in Table 1, Juliet invokes the homomorphic gate operations directly to operate over each encrypted bit of the operands. With word sizes greater than a single bit, this entails executing K gate evaluations in parallel. We remark that shifts and rotations are notoriously noisy operations when using other FHE libraries that work on encrypted integers instead of bits; for example, HElib’s standard shifting operations require the use of several automorphisms and key-switching operations [25], and can accumulate more noise than a multiplication between two ciphertexts (depending on the FHE parameters used). Conversely, shifts and rotations on encrypted data are *essentially free* with TFHE because it represents integers as vectors of bit encryptions. Since these ciphertexts are independent, shifting or rotating them is just a matter of transforming their indices in the vector, just

like one would transform a plaintext bit array. Therefore, no computationally expensive operations are required.

A more complex case involves relational operations such as less-than, greater-than, and equality. All of Juliet’s relational operations utilize the same area-efficient comparator circuit [26], which supports arbitrary word sizes through the use of cascading. The comparator circuit has three output wires: a less-than output, an equality output, and a greater-than output. Only one of these wires will be asserted at a time, indicating the relation of the magnitude of one ciphertext to another. “Less than or equal” or “greater than or equal” can be intuitively achieved by ORing the less than or greater than signal with the equality signal. Lastly, to align the desired output signal to the word size, Juliet replicates the encrypted single bit value K times (i.e., with a wordsize of $K = 8$, an output of encrypted “1” becomes ciphertext vector “1111111”).

4.2 Encrypted Arithmetic Operations

The homomorphic arithmetic operations provided by Juliet for a CPU target include multiplication, addition, and subtraction circuits implemented using TFHE gates, and all circuits employ cascading techniques to support arbitrary word sizes. It is crucial that the underlying building blocks of these circuits (full adders, full subtractors, and 1-bit comparators) are optimized in order to get good performance out of the larger compound circuits. Towards that end, we opted for the designs depicted in Figure 3, and we remark that the motivation behind our full adder design stems from the fact that TFHE can execute a multiplexer circuit with the cost of only two bootstraps, as mentioned in Section 2. Therefore, this full adder design requires only four bootstrap operations total, as opposed to a standard full adder design (with two XOR gates, two AND gates, and an OR gate) that requires five bootstraps. Since bootstrapping remains the primary computational bottleneck in FHE operations, our MUX-based full adder evaluates faster in the encrypted domain, especially when used within larger, multi-bit adders. Specifically, the MUX-based full

TABLE 1

Juliet's ISA encompasses a set of operations on both encrypted and unencrypted data and can be used to develop any algorithm. Here, r_i, r_j, r_k, r_l represent registers, A represents either a register or an immediate which can hold data in the plaintext domain and a register holding a pointer to ciphertext data in the encrypted domain, while I represents an immediate that cannot be encrypted. The Domain column specifies whether the operation is supported in the plaintext domain (p), in the encrypted domain (e) or both (e/p).

T	Domain	Op.	Registers		Description	
Bitwise	e/p	and	r_i	r_j	A	$r_i = r_j \& A$
	e/p	nand	r_i	r_j	A	$r_i = \sim(r_j \& A)$
	e/p	or	r_i	r_j	A	$r_i = r_j A$
	e/p	nor	r_i	r_j	A	$r_i = \sim(r_j A)$
	e/p	xor	r_i	r_j	A	$r_i = r_j \oplus A$
	e/p	xnor	r_i	r_j	A	$r_i = \sim(r_j \oplus A)$
	e/p	not	r_i	A		$r_i = \sim A$
	e	mux	r_i	r_j	r_k r_l	$r_i = r_j \times r_l + r_k \times \sim r_l$
Arithmetic	e/p	add	r_i	r_j	A	$r_i = r_j + A$
	e/p	sub	r_i	r_j	A	$r_i = r_j - A$
	e/p	mul	r_i	r_j	A	$r_i = r_j \times A$, keep LSB
	p	div	r_i	r_j	A	$r_i = r_j \div A$ (unsigned)
	e/p	mod	r_i	r_j	I	$r_i = r_j \% I$ (unsigned)
	e	const	r_i	A		$r_i = \text{Encrypt}(A)$
	e/p	sll	r_i	r_j	I	$r_i = r_j \ll I$
	e/p	srl	r_i	r_j	I	$r_i = r_j \gg I$
	p	mov	r_i	A		$r_i = A$
	p	cmov	r_i	r_j	A	$r_i = A$ (if $r_j == 1$)
CMP & Jump	p	jmp	I			Set PC to I
	p	cjmp	r_i	I		Set PC to I (if $r_i == 1$)
	e/p	seq	r_i	r_j	A	$r_i = (r_j == A)$
	e/p	sgt	r_i	r_j	A	$r_i = (r_j > A)$ (unsigned)
	e/p	slt	r_i	r_j	A	$r_i = (r_j < A)$ (unsigned)
	e/p	sge	r_i	r_j	A	$r_i = (r_j \geq A)$ (unsigned)
e/p	sle	r_i	r_j	A	$r_i = (r_j \leq A)$ (unsigned)	
L/S	p	sw	r_i	$I(r_j)$		$\text{mem}[I + r_j] = r_i$
	p	lw	r_i	$I(r_j)$		$r_i = \text{mem}[I + r_j]$
I/O	e/p	read	r_i			Consume word from tape
	p	print	r_i			Output r_i to stream
	e/p	ret	r_i			Return r_i
	e/p	cret	r_i	r_j		Return r_i (if $r_j == 1$)

adder exhibits a latency reduction of 15% relative to the baseline. When used as the building block of a 64-bit adder, it outperforms the baseline approach by 14% and a 64-bit multiplier utilizing the MUX-based variant is 18% faster than the multiplier using a standard full adder. Our full subtractor follows the same paradigm, with the inclusion of an inverter, which can be evaluated without bootstrapping and introduces negligible overhead.

Addition and subtraction use the same circuit which takes a ripple carry approach. We chose a ripple carry design for the CPU backend, as opposed to carry look-ahead or other fast adders, because it uses the least number of gate evaluations and therefore results in the lowest latency in the encrypted domain without parallelism. Multiplication involves computing partial products by left shifting the bits of the first operand K times to produce K individual ciphertext arrays and then ANDING each with its corresponding bit of the second operand. This way, the result is K arrays (each of size K) that encode partial products where the corresponding bit of the second operand is encrypted "1" and encryptions of zero otherwise. Lastly, the partial products are summed using K invocations of the adder.

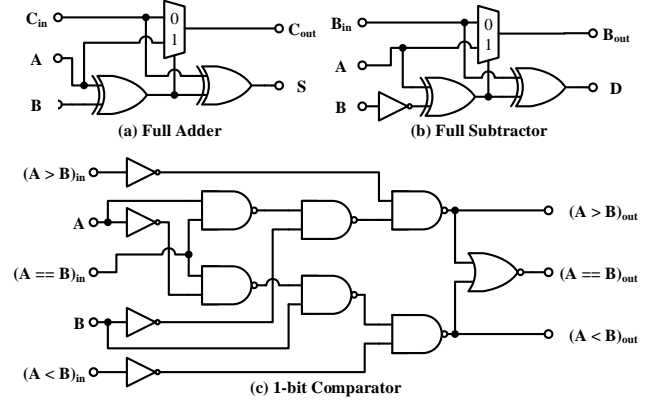


Fig. 3. **Indivisible Circuits:** By substituting logic gates with a MUX, the full adder and subtractor can successfully evaluate with only four bootstraps. Compared to a standard adder/subtractor design, these designs execute faster homomorphically because they require one fewer bootstrap. Together with the 1-bit comparator, these units serve as the basic building blocks for Juliet's functional units.

4.3 JAL Encrypted Multiplexing

Making runtime decisions using ciphertexts is an important constraint when computing on encrypted data [24]. Specifically, branching on encrypted data is not possible as it would reveal information about the control value encoded in a ciphertext if the cloud was permitted to actually resolve the branch outcome. As it stands, the best solution to this problem is to use multiplexing techniques to evaluate both branches and finally choose the output of the correct branch with an encrypted selection bit [24]. While this approach incurs higher complexity for applications such as sorting and searching over encrypted data, it is quite powerful and remains the only viable solution to making decisions in the encrypted domain without leaking information or involving the user. The encrypted multiplexing operation in JAL takes four arguments: the destination register, two source registers pointing to the encrypted data inputs of the multiplexer, and a third source register pointing to the encrypted selection bit. Depending on the value of the selection bit, the corresponding source ciphertext will be propagated to the destination.

4.4 GPU-accelerated Functional Units

In terms of logical units, our GPU implementation follows the same methodology as the CPU: TFHE logic gate operations computed across all operand bits. In this case, however, these gates are evaluated concurrently using all available GPU streaming multiprocessors (SMs). Relational functional units remain identical to the CPU counterparts with the exception of the equality circuit. Instead of evaluating equality using a sequence of cascaded comparators (that limit parallelism due to data dependencies), we adopt K 1-bit comparators that evaluate *in parallel* only the necessary logic to determine the equality signal, and then perform encrypted AND operations between the K outputs to get the final result. This circuit design allows Juliet to exploit as many SMs as possible to carry out this instruction.

The circuit that deviates the most from its CPU equivalent is our adder design. To effectively exploit GPU parallelism, it is more beneficial to employ wider circuits with a shorter critical path instead of the narrow carry ripple adder, which contains the minimum number of gate evaluations,

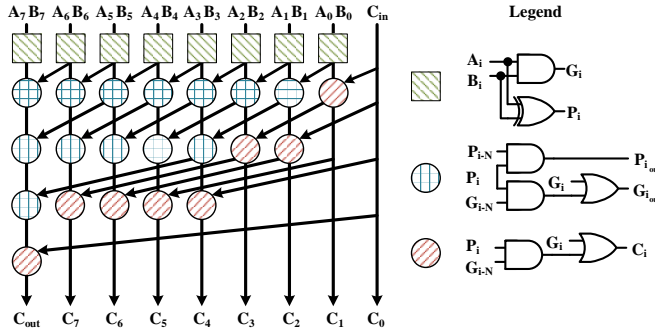


Fig. 4. **Kogge-Stone Adder**: This design for an 8-bit adder is used to compute the carry for each step of the adder. To retrieve the sum (not depicted), one simply needs to compute an XOR operation with the generated output carry and the propagate signal generated in the first stage of the adder.

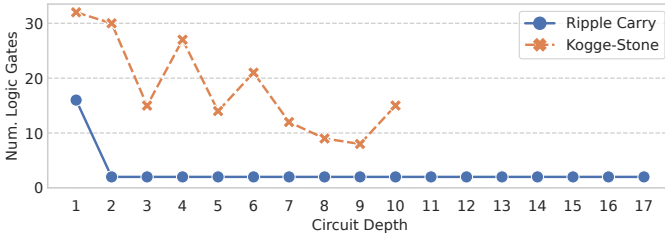


Fig. 5. **16-bit Adder Circuit Topologies**: The ripple carry adder employed as a CPU-encrypted functional unit exhibits a low number of gates but a much longer critical path. On the other hand, the Kogge-Stone adder consists of more gates, but wider circuit levels (and thus more parallelism) and a shorter critical path. For massively parallel devices like GPUs, the Kogge-Stone adder is a better choice.

yet is not readily parallelizable. Instead, we opt for an adder that will minimize the critical path as well as allow for maximal parallel execution, and we adopt a Kogge-Stone carry look-ahead design (Figure 4). A comparison of the circuit topologies of the ripple carry versus the Kogge-Stone adder is depicted in Figure 5. The critical path is nearly $2\times$ shorter for the Kogge-Stone adder and more gates can be evaluated concurrently compared to the ripple carry adder, which has many thin circuit levels with few opportunities for parallelization. Indeed, the Kogge-Stone adder outperforms a GPU implementation of the ripple carry design by approximately 38%. For a wordsize of 16 bits, this adder outperforms the CPU ripple carry design by nearly $5\times$ even though the total number of gates is greater; this is due to the large width of the early stages of the adder, which can achieve high utilization on a GPU. This design is also employed for subtraction by adding one operand to the two’s complement of the other, which entails a simple inversion followed by an increment operation. Lastly, our GPU multiplier generates partial products in parallel through the use of shifts and AND operations and then sums the partial products using multiple invocations of the Kogge-Stone adder.

5 JULIET FRONT-END

In this section, we describe the front-end of the Juliet framework that comprises a Java-like high-level language called *eJava*, and a compiler to translate *eJava* code to JAL instructions. The primary motivation for *eJava* is two-fold: allow programmers to clearly express intent with respect to the sensitivity of data and omit language features that are

non-compliant with encrypted data. For example, a program written in the standard Java language does not indicate which variables should be private, resulting in ambiguity as to whether or not it needs to be encrypted. Encrypting non-sensitive data will result in sub-optimal FHE evaluation. Additionally, language features such as loops with an encrypted index and hash maps with encrypted keys are incompatible with encrypted computation, as the decryption key is not shared. Overall, *eJava* is a self-contained language that is tailored to encrypted computation and enables expressing arbitrary programs that involve both encrypted and unencrypted data types. In the following sections, we elaborate on our design choices for *eJava* and the JAL compiler.

5.1 Design of *eJava* Language

The primary goal of introducing a high-level language is to simplify the development of JAL programs. Therefore, we created a strongly typed object-oriented language like Java that is judiciously designed for encrypted computation. In *eJava*, classes may contain fields and methods that have arguments and return types of basic or other class types. The `new` operator calls a default void constructor. Moreover, *eJava* supports inheritance and its methods can be defined in a subclass if that subclass has the same return type and arguments. The fields in the derived class and in the base class are different fields, even if they have the same names; however, *eJava* does not offer support for inner classes and static methods or fields. The `this` identifier can be used to access the methods of a class from another method of the same class. By default, all methods are public and all fields are protected. Thus, a method of a class cannot access fields of another class, except if it is inherited from that class. Local variables can be defined at the beginning of a method, and if they have the same name as a class field then they shadow that field. Finally, a *eJava* program begins with a special main class that only contains the main method (i.e., `public static void main(String[] args)`), and afterward it may contain other classes that have fields and methods.

5.1.1 Operations in the Encrypted Domain

The basic types of *eJava* are `int` for K -bit integers, where K is the word size, `int[]` for arrays of integers, and `boolean` for logical values. For the encrypted domain, *eJava* supports `EncInt` for encrypted integers and `EncInt[]` for arrays of encrypted integers. The syntax and functionality of encrypted operations closely mimics that of the plaintext equivalent, simplifying development. Operations between the encrypted and unencrypted domains are possible by first encoding the variables from the unencrypted domain. Since the computation is outsourced to a third party that only possesses the evaluation key (and not the decryption key), such operations involve encoding variables on the fly as *trivial ciphertexts without noise* using the evaluation key. However, initializing encrypted variables with plaintext data, such as in Fig. 6(a), and resorting to encrypting with the evaluation key by the untrusted third party raises numerous concerns.

First, encrypting a sensitive value with the evaluation key adds no noise to the generated ciphertext, so the underlying LWE problem becomes trivial to solve. Nevertheless,


```

1  EncInt x = 7;    // x = enc(7)
2  x *= 2;        // x = x * enc(2)
3  return x;      // x = enc(14)

```

(a) Illegal Initialization.

```

1  EncInt x = PrivateTape(); // x = enc(?)
2  x *= 2;                // x = x * enc(2)
3  return x;              // x = enc(?)

```

(b) Correct way to initialize EncInt variables.

Fig. 6. Initialization of encrypted integers.

if an FHE operation combines such a ciphertext without noise with a noisy ciphertext (i.e., one that was encrypted with the user’s key), the resulting ciphertext has noise and is secure. For example, if we add a secure ciphertext (with sufficient noise) to a trivial ciphertext (with negligible noise), then we will end up with a secure ciphertext: $N_{noisy} + N_{negligible} \rightarrow N_{noisy}$.

Second, the cloud can read constants initialized in the outsourced JAL program and perform static analysis to trace how they affect the output of the program. For example, in line 1 in Fig. 6(a) encrypted variable x is assigned the encryption of constant 7; however, this is an illegal assignment and a violation of our threat model since the contents of x will be encrypted on-the-fly without noise, and the cloud can learn meaningful information about a ciphertext. Thus, the only way to initialize the `EncInt` and `EncInt[]` data types in eJava is through the private tape with secure ciphertexts that were generated offline. For instance, in line 1 Fig. 6(b), the user directly assigns a ciphertext to the encrypted variable x through the private tape. Since this ciphertext has been generated by the user, the cloud server has no way of inferring any information about it.

5.1.2 Inputs from Public and Private Tapes

In more detail, eJava supports both public and private inputs via two read-only *tapes*. These built-in methods have a one-to-one correspondence with JAL instructions: the public tape can be read sequentially using the `pread` instruction, while for the private tape, Juliet uses the `eread` instruction. In the public tape case, the next word (K -bit integer) from the tape is consumed, while when the private tape is used as input, a pointer to the next ciphertext array is read. We note that the input ciphertext arrays pointed to by the private tape pointers are pre-loaded into the ciphertext memory by the user, who uploads the memory structure to the cloud for further evaluation.

5.2 eJava Compiler for Juliet

We complement Juliet with a custom compiler to translate eJava code into optimized JAL assembly instructions.

5.2.1 Type Checking

The first step in the compilation process is type checking by statically analyzing the input program and verifying its type safety. Our compiler performs one first pass of the source file to gather the defined classes and then another pass to generate a symbol table with all the classes, fields, methods, and local variables [27]. In a third pass, the compiler confirms that all data types of expressions and variables are consistent. As already mentioned, due to threat model constraints (Section 5.1.1), our compiler does not allow direct

```

1  EncInt x = PrivateTape(); // x = enc(?)
2  EncInt res = PrivateTape(); // res = enc(?)
3  if (x == 0) { // Cannot branch
4      res += 7; // res = res + enc(7);
5  } else {
6      res += 13; // res = res + enc(7);
7  }
8  return res; // x = enc(?)

```

(a) Type-checking error: Illegal branch on encrypted data.

```

1  EncInt x = PrivateTape(); // x = enc(?)
2  EncInt res = PrivateTape(); // res = enc(?)
3  EncInt sel = x == 0; // enc(0)/enc(1)
4  res = sel * (res+7) + (1-sel) * (res+13);
5  return res; // x = enc(?)

```

(b) Correct way to check for encrypted equality in eJava.

```

1  EncInt x = PrivateTape(); // x = enc(?)
2  EncInt res = PrivateTape(); // res = enc(?)
3  res = (x == 0) ? (res + 7) : (res + 13);
4  return res; // x = enc(?)

```

(c) Encrypted equality using ternary operators.

```

1  eread t0 // x
2  eread t1 // res
3  econst t2, 0
4  eseq t2, t0, t2 // t2 = x == enc(0)
5  econst t3, 7
6  eadd t3, t1, t3 // t3 = res + enc(7)
7  econst t4, 13
8  eadd t4, t1, t4 // t4 = res + enc(14)
9  emux res, t2, t3, t4
10 eret t1 // res

```

(d) JAL assembly for (c).

Fig. 7. eJava example for multiplexing based on encrypted data. (a) Example of illegal branching on encrypted data; (b) Same functionality as in (a) but without branching on encrypted data; (c) Same functionality as in (b) using a ternary operator; (d) JAL assembly code for (c).

assignment of constants in variables of encrypted type (like in Fig. 6(a)). Additionally, operations between encrypted and unencrypted variables are permitted if the destination is also declared as encrypted, as in Fig. 6(b). Lastly, our compiler also throws an error if an `answer` function is missing, as this is required to halt the abstract machine.

An important consideration that our type-checking system has to take into account is the inability to make runtime decisions based on encrypted data. For instance, our compiler throws an exception if the source code includes branches based on encrypted values, like in Fig. 7(a). It is the programmers’ responsibility to express such statements obliviously (as in Fig. 7(b) lines 3-4), or using a ternary operator that directly invokes the `MUX` JAL instruction (as in Fig. 7(c)). The corresponding JAL assembly for Fig. 7(c) is presented in Fig. 7(d). This aspect of eJava is the most notable deviation from standard high-level languages, but is an inherent feature of encrypted computation. We note that most branches can be readily resolved with a series of multiplexers.

5.2.2 JAL Assembly Generation

After the type safety of the eJava source program has been verified, our compiler parses the high-level code, generates an intermediate representation (IR), and performs various optimizations. First, the eJava compiler runs a static analysis phase to identify constant- and copy-propagation optimizations, as well as live range analysis and dead code

elimination [27]. These optimizations significantly reduce the size of the IR as they remove intermediate variables and unused code blocks. Our compiler continues performing static analysis and other optimizations until a threshold is reached so that two different iterations result in the same output (i.e., no further optimizations are possible). Finally, the eJava compiler maps the virtual registers to a predefined number of registers that the Juliet abstract machine supports by utilizing graph coloring techniques [28], and the optimized JAL assembly is generated.

6 EXPERIMENTAL EVALUATIONS

To encompass a wide range of use cases for encrypted computation, we employ the TERMinator suite [24], [29], which is a set of benchmarks designed for encrypted architectures. In order to integrate TERMinator with Juliet, we made several modifications. First, the TERMinator benchmarks were designed for encrypted computation based on partially homomorphic encryption and employed an oblivious multiplexing function (dubbed \mathbb{G}); the latter used a control input to select between an encrypted input or an encryption of zero. Therefore, in this work, we employ FHE multiplexers instead of \mathbb{G} function calls and ported selected microbenchmarks to eJava to evaluate core operations on encrypted data with Juliet.

Moreover, we performed experiments by evaluating two block ciphers in the encrypted domain, namely SIMON and SPECK [30]. These algorithms are packed with binary and bit manipulation operations that help demonstrate important aspects of our abstract machine and the benefits of the underlying TFHE scheme. In addition, we evaluate Juliet using the multiplication-heavy Sieve of Eratosthenes (SoE) algorithm, as well as the computationally intensive logistic regression (LR) inference, which corresponds to realistic machine learning. We remark that while many prior HE frameworks include proof-of-concept implementations of basic machine learning applications, they almost exclusively employ leveled HE. Even though this approach can help the evaluation of *shallow circuits*, it quickly becomes infeasible for the deeper circuits that need complex applications and high accuracy. Therefore, in the case of LR, the use of FHE (instead of leveled HE) allows us to compute a more accurate approximation of the sigmoid function.

All CPU-based experiments were executed on an Amazon AWS r5.12xlarge instance, which assumed the role of the cloud server (Fig. 1), and was used to exclusively run the FHE operations using our execution engine; all key generation, encryption, and decryption operations were done offline on a laptop (i.e., the client machine). For our GPU-based evaluation, the cloud server used an NVIDIA Tesla M60 GPU hosted on an AWS g3.8xlarge instance.¹

6.1 Evaluation using Microbenchmarks

The class of microbenchmarks in TERMinator comprises a set of applications that involve primarily encrypted additions or multiplications (or both). The Fibonacci benchmark

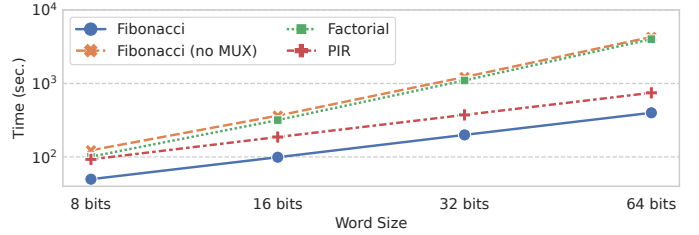


Fig. 8. **Microbenchmarks:** This set of programs evaluates the performance of core FHE operations in Juliet, namely addition, multiplication, multiplexing, and encrypted equality. To showcase the power of native MUX gates, we also include a Fibonacci benchmark that implements multiplexing via multiplication and addition (instead of a MUX).

consists of many addition operations and computes the N^{th} Fibonacci number where N is an encrypted value. Likewise, the factorial benchmark computes $N!$ where N is an encrypted input. In both programs, the cloud has no knowledge of the returned output since a larger range of values is computed first, and then the N^{th} value is selected homomorphically through either multiplication or multiplexing. For our experiments, the eJava program computes the first 20 factorials and Fibonacci numbers, and the encrypted input N selects a random value from 1 to 20. Our third microbenchmark is private information retrieval (PIR), which simulates an FHE database with 50 entries and performs a search operation using encrypted equality checks. Due to the termination problem, the PIR program cannot exit early and must visit every entry in the database while searching for a match. After all equality checks are computed, a multiplexing operation selects the final result.

In Fig. 8 we report the evaluation times of our microbenchmarks on a CPU using a wordsize from 8 to 64 bits. As expected, the factorial benchmark is slower and scales worse than the Fibonacci benchmark due to the large multiplication circuits that require far more gates for increasing word sizes, while our PIR benchmark scales linearly with increasing word sizes. Moreover, a variant of the Fibonacci benchmark that relies on an arithmetic approximation of a multiplexer (like Figure 7(b), line 4) is far slower than its MUX-based counterpart. This approximation requires a large number of multiplication operations on secure ciphertexts and is therefore computationally expensive. At a 64-bit wordsize, this inefficient Fibonacci implementation takes approximately 57.5 minutes to evaluate, which is over an order of magnitude slower than the implementation that employs MUXes to enable selection. At a high level, this radical difference in latency showcases that binary operations on ciphertext arrays are much faster than arithmetic operations in the TFHE cryptosystem.

6.2 Evaluation using Real-Life Benchmarks

Simon/Speck: Compared to our microbenchmarks, the SIMON and SPECK block ciphers, the Sieve of Eratosthenes, and the logistic regression inference are far more complex programs and constitute realistic workloads on encrypted data. While the block ciphers are predominantly composed of bitwise operations (and modular addition in the case of SPECK), the Sieve of Eratosthenes and the logistic regression inference are multiplication intensive. These benchmarks demonstrate the scalability of Juliet and, in the case of the block ciphers, they also enable an exciting application of

¹ The Juliet framework is open-sourced at <https://github.com/TrustworthyComputing/Juliet>, while our HEJava compiler is available at <https://github.com/TrustworthyComputing/HEJava-compiler>.

TABLE 2

GPU Acceleration: This table outlines benchmarks configured with a wordsize of 16 bits. In each case, the GPU implementation of Juliet’s encrypted ALU is faster than the CPU-based version. Logical operations, which are completely parallel, run efficiently on the GPU, hence the fast evaluation time of the SIMON circuit, which achieves a speedup of $8.6\times$ compared to the CPU implementation.

Benchmark	CPU Time (sec.)	GPU Time (sec.)	Speedup
SPECK	69.8	19.4	$3.6\times$
SIMON	60.9	7.1	$8.58\times$
LR Inference	273.1	260.7	$1.05\times$
Fib.	364	329.5	$1.10\times$
Fib. (MUX)	99.1	77.2	$1.28\times$
Factorial	317	284.7	$1.11\times$
Sieve of Erat.	245.5	230.2	$1.07\times$
PIR	187	153.6	$1.22\times$

outsourced homomorphic computation called “transciphering” or “scheme hopping” [31], [32], which enables efficient compression of ciphertexts and can drastically reduce communication overheads between client and server.

SoE: The Sieve of Eratosthenes is used to find all prime numbers in a range. Our benchmark begins with reading an array of encryptions of zero of length equal to the desired range (in our case, we compute all prime numbers between 1 and 20). At runtime, when a prime is detected at a given index, the corresponding encryption of zero becomes an encryption of one. Finally, to generate the encrypted list of the primes, all array values are multiplied by their index.

LR: The logistic regression algorithm is used to classify inputs, such as images or text data, into different classes. For our implementation, we employ the Iris dataset [33], which classifies flowers into one of three species of irises. For our experiments, we restrict the dataset to two species and perform training with the dataset offline.

For encrypted evaluation, we upload the ciphertexts encoding a flower’s characteristics to the cloud, which performs LR inference and returns ciphertexts corresponding to the probabilities that input belongs to each possible class. The workhorse of LR and a major challenge for FHE computation is the sigmoid function (specifically, the standard logistic function), which is defined as $f(x) = 1/(1 + e^{-x})$. To evaluate this function homomorphically, we employ a Maclaurin series approximation and compute the first three terms as $f(x) = 1/2 + x/4 - x^3/48 + \dots$. In addition, to convert all of the terms into integers, we multiply both sides of the equation by 48. This presents a reasonable trade-off between accuracy and performance, as the execution time scales with the number of terms evaluated. After the inference is completed and the cloud returns two encrypted probabilities (one for each class), the user decrypts each one and divides by 48. Overall, our homomorphic inference program is capable of classifying two flowers at a time.

Matrix Multiplication: This benchmark is a core operation for various machine learning and signal processing applications. Due to the high number of multiplications required (n^3 for a product between two square matrices), this benchmark is expensive to compute over Boolean ciphertexts. Relative to strictly plaintext computation running on Juliet, we find that the average cost of encryption is roughly 3.8 orders of magnitude across the three matrix sizes, which is

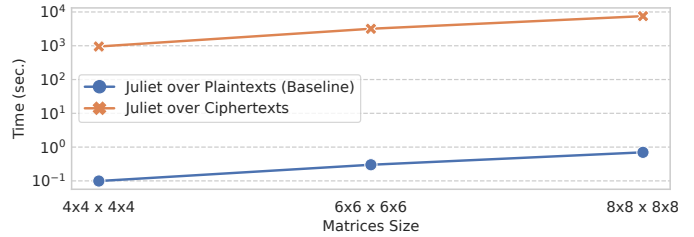


Fig. 9. **Square Matrix Multiplication:** We utilize a word size of 16 and note that this benchmark is multiplication-intensive, resulting in higher latencies due to the large size of 16-bit multiplier circuits.

consistent with what related works report [15], [34]. This performance differential can be attributed to the fact that encrypted operations take the form of high-degree polynomial arithmetic instead of operations across small scalars. Additionally, the functional bootstrapping step in TFHE is measured to consume the vast majority of execution time of the overall matrix multiplication benchmarks ($> 90\%$), while the remaining cost is attributed to interprocess communication and linear polynomial operations.

Discussion: Table 2 provides a comparison of the evaluation of Juliet using a CPU-based and a GPU-based implementation. Both block ciphers were configured with the same key size (64 bits) and block size (32 bits) and were used to “decrypt” a single ciphertext block in FHE. A wordsize of 16 was chosen since both ciphers split the ciphertext block in half during decryption. For our results, we observe that SIMON is faster than SPECK. This is expected as SIMON is composed entirely of fast bitwise operations, while SPECK contains both addition operations (which require Boolean adder circuits) and bitwise operations. Thus, since these ciphers have many logical operations that can be readily parallelized, the GPU implementation excels at both. Overall, we find that the GPU implementation can achieve a much higher gate throughput than the CPU backend. However, in order to take advantage of the heightened throughput, the circuits need to be sufficiently wide. When only a small number of gates can be evaluated in parallel, the GPU utilization drops and the data transfer costs begin to dominate. As such, we note that programs that utilize large word-sizes are well-suited for the GPU backend as the resulting circuits are typically wide. The high speedup of Simon 64/96 (32-bit word size) and Simon 128/128 (64-bit word size) depicted in Figure 10 emphasize this point.

Aside from matrix multiplication, our LR inference and SoE benchmarks exhibit the longest execution times because they are multiplication-intensive. LR requires 20 homomorphic multiplications (12 for the sigmoid approximation and 8 for multiplying the trained coefficients and inputs), while SoE requires 20 homomorphic multiplications in order to compute all primes up to the number 20 (one for each number from 1 to 20). The small discrepancy in the execution times can be attributed to the fact that LR also requires several homomorphic addition and subtraction operations.

6.3 Experimental Comparisons with Previous Works

We compare Juliet with three FHE frameworks targeting TFHE that are *Turing complete* (i.e., geared towards general computation) and *flexible* so that the whole framework does not have to be recompiled to evaluate new programs. These

TABLE 3

Comparisons of FHE computation frameworks in terms of plaintext type (integer, Boolean, or fixed point), support for mixed operations, whether the implementation is Turing complete, offline computation without user involvement, compilation requirements, bootstrapping efficiency, whether memory accesses are constant-time or scale with memory size, semantic security, and lastly GPU support.

Framework	Type	Fast Bootstrap [†]	Mixed Ops [‡]	Turing Complete	Offline [§]	Flexibility [¶]	GPU Support	Memory Access	Semantic Security
ALCHEMY [35]	Int	○	●	○	●	○	○	Const.	●
Cingulata [36]	Bin	◐	◐	●	●	○	○	Const.	●
E ³ [37]	Int, Bin, FP	◐	◐	●	●	●	○	Const.	●
EVA [38]	FP	○	●	○	●	●	○	Const.	●
Google Transpiler [39]	Bin	◐	○	●	●	○	○	Const.	●
Juliet (this work)	Bin	●	●	●	●	●	●	Const.	●
Porcupine [40]	Int	○	●	○	●	○	○	Const.	●
RAMPARTS [41]	Int	○	●	○	●	●	○	Const.	●
Romeo [32]	Bin	●	○	●	●	○	○	Const.	●
T2 [29]	Int, Bin, FP	◐	●	●	●	●	○	Const.	●
VSP [42]	Bin	●	○	●	○	●	●	Exp.*	○

[†] Fast bootstrap refers to the bootstrapping efficiency of the underlying HE scheme. ○ indicates slow bootstrapping, ◐ indicates that some of the supported schemes have fast bootstrapping, while ● indicates fast bootstrapping.

[‡] Mixed Ops refers to the capability of mixing plaintext and encrypted computation.

[§] Offline indicates whether or not the user needs to participate in the computation in any capacity.

[¶] A framework is flexible if it can support different FHE applications without needing to recompile everything.

* The size of the encrypted memory in the open-source implementation is fixed. The access costs scales exponentially.

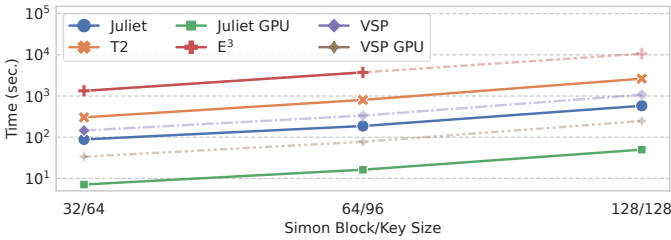


Fig. 10. Comparisons between Juliet, T2, E3, and VSP. All frameworks were configured using TFHE at 80 bits of security. Juliet with a GPU-based ALU is at least an order of magnitude faster than all CPU-based frameworks. The reported VSP timings exclude (a) downstream network delays (incurred when the server sends a copy of the entire encrypted memory to the user), (b) the user’s decryption time of the received ciphertexts, and (c) the upstream network delay for the plaintext termination response to the server. Dashed lines indicate expected timings; for instance, VSP is only configured to use 16-bit registers and can not readily support the word sizes required for larger SIMON variants.

frameworks include T2 [29], E3 [37], and VSP [42]. We selected the SIMON cipher as the underlying application for comparisons since it has strictly bitwise operations and is naturally suited to TFHE. In Fig. 10, we present the performance results for each framework with increasing block and key sizes, which also increases the number of rounds required to evaluate SIMON.

All comparison experiments were run on a g3.8xlarge AWS server. In certain experiments, the transparent lines indicate expected timings; for example, VSP was only able to correctly evaluate SIMON with a 16-bit wordsize (which is amenable to SIMON 32/64). Meanwhile, E3 was unable to compile SIMON 128/128 in less than an hour, so we report its expected performance based on the trend from smaller block sizes. Additionally, we note that while VSP reports GPU support, is only configured to run on two specific GPUs (conversely, Juliet is compatible with any arbitrary NVIDIA GPU). Thus, with the NVIDIA M60 GPU on our AWS server, VSP was unable to evaluate SIMON. Instead, we report the expected timings based on the clock cycle cost of 1.7 seconds, as reported by the authors for one V100

GPU [42]. Moreover, the reported times for VSP do not include the communication overhead incurred after each clock cycle (as required by the VSP protocol). Lastly, we remark that we allowed all frameworks to use the available hardware resources of the host, but we did not modify them to add any parallelization if the framework did not naturally support it (in this case, T2 and E3). Overall, we observe that Juliet is the fastest framework using a CPU backend, while for the GPU backend, it is approximately an order of magnitude faster than other CPU-based frameworks, and nearly 5× faster than VSP on a single GPU. Additionally, for SIMON 32/64, Juliet achieves 62 cycles per second on a single GPU, while VSP achieves 0.8 cycles per second in its fastest configuration with 8 V100 GPUs (which also have a compute capability of 7.0 versus the M60’s 5.2 and 4× as many SMs). We conjecture that this performance differential is mainly attributed to the significant cost of VSP’s encrypted memory accesses, which requires visiting all memory locations for each read/write.

7 RELATED WORK

In recent years, many works have focused on making FHE viable outside of research circles, from both an efficiency and usability standpoint. Previous works focusing on general computation with FHE can be divided into two categories depending on the type of HE circuit the underlying schemes use: arithmetic or Boolean. In Table 3, we present our comparisons of prior works encompassing both approaches.

Frameworks that use arithmetic circuits to evaluate algorithms in the encrypted domain use an integer or fixed point encoding for underlying plaintexts and utilize native HE addition and multiplication operations. This category encompasses works such as ALCHEMY [35], RAMPARTS [41], and Porcupine [40] that introduce their own DSLs that programmers can use to build HE applications leveraging BFV (or a variant of BGV in the case of ALCHEMY). In addition, EVA [38] presents an intermediate representation

that uses the CKKS scheme (which operates over complex numbers) implemented in the SEAL library. These solutions typically leverage batching capabilities to pack multiple plaintexts into a single ciphertext and allow for vector processing. For certain types of applications that can be orchestrated as shallow circuits, this strategy can be beneficial; however, the major drawback of arithmetic-based schemes like BFV and CKKS is that the speed of bootstrapping is prohibitively slow. Therefore, these schemes are typically used in leveled HE mode and suffer from scalability issues when faced with deep, complex algorithms as the parameters required to support high depth will result in significantly slow speeds. Another limitation of these schemes involves the inability to perform practical comparison operations and bitwise shifts in the integer or floating point domains. As such, solutions that use solely arithmetic operations are not Turing-complete. Juliet, on the other hand, is scalable and Turing-complete due to the fast bootstrapping speeds of TFHE, and native support of comparison operations.

The other primary class of HE computation, Boolean circuits, offers key advantages compared to the arithmetic approach. While operations such as multi-bit addition and multiplication require several low-level HE operations, the bootstrapping speeds are orders of magnitude faster for Boolean circuit-based schemes (like TFHE and FHEW), which makes them ideal for general-purpose computation. Juliet leverages this computational model to allow the composition of arbitrary algorithms. Prior works that also incorporate this approach include toolchains that take VHDL or Verilog programs (or even high-level programs converted with High-Level Synthesis), perform logic synthesis to generate a netlist of Boolean gates and perform a direct conversion into an FHE circuit. Both Romeo [32] and Google's Transpiler [39] are frameworks that are capable of converting arbitrary combinational and sequential circuits to equivalent TFHE code. Two potential limitations of these two approaches involve the need to completely recompile the circuit from scratch if any small change needs to be made to the algorithm, as well as the lack of support for plaintext operations. Because the entire program is converted to an HE circuit, plaintext values must be encrypted on the user side and then used for HE logic gate operations.

The VSP framework proposes an encrypted processor design for use with TFHE, which requires the user to be involved in the computation in order to get around the termination problem [42]. The user must decrypt an encrypted signal after a predefined number of cycles to let the cloud know if it has finished the computation or needs to proceed further. This also poses a security risk for users, as the cloud can abuse this mechanism to have the user decrypt any ciphertext (including encrypted key material inside of the evaluation key). Additionally, VSP incorporates an encrypted memory design that does not scale in the encrypted domain: whenever memory is accessed, all memory locations need to be updated or read, followed by costly multiplexing operations to isolate the correct memory region. Juliet, however, is completely offline, it does not rely on expensive memory constructions and requires no interaction from users besides encryption and decryption. Further, we note that VSP and Romeo do not support mixed operations, which prevents the cloud from being able to

supply their own inputs; for instance, in a machine learning setting, the cloud would not be able to provide their own weights to use in an encrypted classification procedure.

Lastly, three prior works support both arithmetic and Boolean operations. In the case of Cingulata [36], either BFV or TFHE is chosen prior to compilation; however, Cingulata does not support GPU backends and must be recompiled for every program. Likewise, E³ [37] is a C++ framework capable of *bridging*, which involves switching from binary to arithmetic ciphertexts. Whereas E³ only works with C++, any arbitrary front-end can be constructed for Juliet. Additionally, we note that Juliet with a CPU backend is an order of magnitude faster than E³ for the SIMON benchmark (Fig. 10). A third work is T2 [29] which incorporates an HE compiler with support for integer, binary, and floating point encodings; however, T2 does not offer GPU support, which is a major benefit of Juliet.

8 CONCLUDING REMARKS

In this work, we propose a novel encrypted processor design and an execution engine for general-purpose encrypted computation on CPUs and GPUs, which enables secure outsourcing to cloud servers. Our processor is complemented by an expressive assembly language targeting encrypted data and a bespoke compiler to convert high-level programs into our encrypted assembly. For validation, we employ benchmarks from TERMinator and realistic algorithms on encrypted data, and demonstrate significant performance and flexibility benefits compared to related works.

REFERENCES

- [1] N. Gruschka and M. Jensen, "Attack surfaces: A taxonomy for attacks on cloud services," in *IEEE 3rd International Conference on Cloud Computing*, 2010, pp. 276–279.
- [2] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage, "Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds," in *Proceedings of the 16th ACM Conference on Computer and Communications Security*, 2009, pp. 199–212.
- [3] C. Gentry, "Fully homomorphic encryption using ideal lattices," in *Proceedings of the 41st Annual ACM Symposium on Theory of Computing*, 2009, pp. 169–178.
- [4] S. Goldwasser, S. Micali, and C. Rackoff, "The knowledge complexity of interactive proof-systems," in *Proceedings of the 17th Annual ACM Symposium on Theory of Computing*, ser. STOC '85, 1985, p. 291–304.
- [5] M. Blum, P. Feldman, and S. Micali, "Non-interactive zero-knowledge and its applications," in *Providing Sound Foundations for Cryptography: On the Work of Shafi Goldwasser and Silvio Micali*. New York, NY, USA: Association for Computing Machinery, 2019, p. 329–349.
- [6] D. Mouris and N. G. Tsoutsos, "Zilch: A Framework for Deploying Transparent Zero-Knowledge Proofs," *IEEE Transactions on Information Forensics and Security*, vol. 16, pp. 3269–3284, 2021.
- [7] Y. Lindell, "Secure multiparty computation," *Commun. ACM*, vol. 64, no. 1, p. 86–96, dec 2020.
- [8] D. Mouris, P. Sarkar, and N. G. Tsoutsos, "PLASMA: Private, Lightweight Aggregated Statistics against Malicious Adversaries," *Cryptology ePrint Archive*, Report 2023/080, 2023, <https://eprint.iacr.org/2023/080>.
- [9] V. Shoup and S. Halevi, "HElib - An Implementation of homomorphic encryption." [Online]. Available: <https://github.com/shaih/HElib>
- [10] A. Acar, H. Aksu, A. S. Uluagac, and M. Conti, "A survey on homomorphic encryption schemes: Theory and implementation," *ACM Computing Surveys (Csur)*, vol. 51, no. 4, pp. 1–35, 2018.
- [11] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène, "TFHE: Fast Fully Homomorphic Encryption Library," August 2016, <https://tfhe.github.io/tfhe/>.

- [12] C. Gouert, R. Khan, and N. G. Tsoutsos, "Optimizing homomorphic encryption parameters for arbitrary applications," *Cryptology ePrint Archive*, Report 2022/575, 2022, <https://ia.cr/2022/575>.
- [13] S. Halevi and V. Shoup, "Bootstrapping for HELib," *Journal of Cryptology*, vol. 34, no. 1, pp. 1–44, 2021.
- [14] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachene, "Faster fully homomorphic encryption: Bootstrapping in less than 0.1 seconds," in *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 2016, pp. 3–33.
- [15] L. Folkerts, C. Gouert, and N. G. Tsoutsos, "REDsec: Running Encrypted Discretized Neural Networks in Seconds," in *Network and Distributed System Security Symposium (NDSS)*, 2023, pp. 1–17.
- [16] M. Steinbauer, M. Indrawan-Santiago, G. Anderst-Kotsis, W. A. A. Torres, N. Bhattacharjee, and B. Srinivasan, "Privacy-preserving biometrics authentication systems using fully homomorphic encryption," *International Journal of Pervasive Computing and Communications*, 2015.
- [17] C. Gentry, A. Sahai, and B. Waters, "Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based," in *Annual Cryptology Conference*. Springer, 2013, pp. 75–92.
- [18] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, "(Leveled) fully homomorphic encryption without bootstrapping," *ACM Transactions on Computation Theory (TOCT)*, vol. 6, no. 3, pp. 1–36, 2014.
- [19] J. H. Cheon, A. Kim, M. Kim, and Y. Song, "Homomorphic encryption for arithmetic of approximate numbers," in *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 2017, pp. 409–437.
- [20] J. Fan and F. Vercauteren, "Somewhat practical fully homomorphic encryption." *IACR Cryptol. ePrint Arch.*, vol. 2012, p. 144, 2012.
- [21] "Microsoft SEAL (release 3.5)," <https://github.com/Microsoft/SEAL>, Apr. 2020, Microsoft Research, Redmond, WA.
- [22] L. Ducas and D. Micciancio, "FHEW: Bootstrapping homomorphic encryption in less than a second," in *Advances in Cryptology – EUROCRYPT 2015*, 2015, pp. 617–640.
- [23] W. Dai and B. Sunar, "cuFHE (v1.0)," 2018. [Online]. Available: <https://github.com/vernamlab/cuFHE>
- [24] D. Mouris, N. G. Tsoutsos, and M. Maniatakos, "TERMinator suite: Benchmarking privacy-preserving architectures," *IEEE Computer Architecture Letters*, vol. 17, no. 2, pp. 122–125, 2018.
- [25] S. Halevi and V. Shoup, "Algorithms in HELib," in *Annual Cryptology Conference*. Springer, 2014, pp. 554–571.
- [26] M. Trikha and R. Mehra, "Area-efficient layout design of comparator using cascaded technique," in *2015 International Conference on Advances in Computer Engineering and Applications*. IEEE, 2015, pp. 619–623.
- [27] A. V. Aho, R. Sethi, and J. D. Ullman, "Compilers, principles, techniques," *Addison Wesley*, vol. 7, no. 8, p. 9, 1986.
- [28] G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins, and P. W. Markstein, "Register allocation via coloring," *Computer Languages*, vol. 6, no. 1, pp. 47–57, 1981.
- [29] C. Gouert, D. Mouris, and N. G. Tsoutsos, "SoK: New Insights into Fully Homomorphic Encryption Libraries via Standardized Benchmarks," *Proceedings on Privacy Enhancing Technologies*, vol. 2023, no. 3, pp. 154–172, July 2023.
- [30] R. Beaulieu, D. Shors, J. Smith, S. Treatman-Clark, B. Weeks, and L. Wingers, "The SIMON and SPECK Families of Lightweight Block Ciphers." *IACR Cryptol. ePrint Arch.*, vol. 2013, p. 404, 2013.
- [31] S. Carpov and R. Sirdey, "Another compression method for homomorphic ciphertexts," in *Proceedings of the 4th ACM International Workshop on Security in Cloud Computing*, 2016, pp. 44–50.
- [32] C. Gouert and N. G. Tsoutsos, "Romeo: conversion and evaluation of HDL designs in the encrypted domain," in *2020 57th ACM/IEEE Design Automation Conference (DAC)*, 2020, pp. 1–6.
- [33] R. A. Fisher, "The use of multiple measurements in taxonomic problems," *Annals of eugenics*, vol. 7, no. 2, pp. 179–188, 1936.
- [34] N. Samardzic, A. Feldmann, A. Krastev, S. Devadas, R. Dreslinski, C. Peikert, and D. Sanchez, "F1: A Fast and Programmable Accelerator for Fully Homomorphic Encryption," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '21, 2021, p. 238–252.
- [35] E. Crockett, C. Peikert, and C. Sharp, "Alchemy: A language and compiler for homomorphic encryption made easy," in *ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 1020–1037.
- [36] CEA-LIST, "Cingulata compiler toolchain," <https://github.com/CEA-LIST/Cingulata>.
- [37] E. Chielle, O. Mazonka, N. G. Tsoutsos, and M. Maniatakos, "E3: A Framework for Compiling C++ Programs with Encrypted Operands." *IACR Cryptology ePrint Archive*, vol. 2018, p. 1013, 2018.
- [38] R. Dathathri, B. Kostova, O. Saarikivi, W. Dai, K. Laine, and M. Musuvathi, "EVA: An encrypted vector arithmetic language and compiler for efficient homomorphic computation," in *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2020, pp. 546–561.
- [39] Google, "Fully Homomorphic Encryption (FHE) C++ Transpiler," <https://github.com/google/fully-homomorphic-encryption/tree/main/transpiler>.
- [40] M. Cowan, D. Dangwal, A. Alaghi, C. Trippel, V. T. Lee, and B. Reagen, "Porcupine: a synthesizing compiler for vectorized homomorphic encryption," in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, 2021, pp. 375–389.
- [41] D. W. Archer, J. M. Calderón Trilla, J. Dagit, A. Malozemoff, Y. Polyakov, K. Rohloff, and G. Ryan, "RAMPARTS: A programmer-friendly system for building homomorphic encryption applications," in *ACM Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, 2019, pp. 57–68.
- [42] K. Matsuoka, R. Banno, N. Matsumoto, T. Sato, and S. Bian, "Virtual secure platform: A five-stage pipeline processor over TFHE," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021.



ally during the Cyber Security Awareness Worldwide (CSAW) event.



He is currently a Senior Cryptography Researcher at Nillion.



He has authored multiple articles in the IEEE Transactions and conference proceedings, and serves in the program committee of several international conferences. He is also the faculty organizer of the international Embedded Security Challenge (ESC) competition that is held annually during the Cyber Security Awareness Worldwide (CSAW) event.

Charles Gouert (S'20) received the B.Sc. degree (2018) in electrical and computer engineering from the University of Delaware, Newark, DE, USA. He is currently pursuing the Ph.D. degree in computer engineering with the Electrical and Computer Engineering department at the University of Delaware. His current research interests include applied cryptography and private outsourcing. He is also the global challenge co-lead of the international Embedded Security Challenge (ESC) competition that is held annually during the Cyber Security Awareness Worldwide (CSAW) event.

Dimitris Mouris (S'19) received his Ph.D. (2024) in computer engineering from the Electrical and Computer Engineering department at the University of Delaware, Newark, DE, USA. Additionally, he holds the B.Sc. (2016) and M.Sc. (2018) degrees in computer science and computer systems respectively from the National and Kapodistrian University of Athens, Greece. His research is in applied cryptography and privacy, particularly in the area of privacy-enhancing technologies using homomorphic encryption, secure multi-party computation, and zero-knowledge proofs.

Nektarios G. Tsoutsos (S'13–M'19) Nektarios Georgios Tsoutsos received the Ph.D. degree in computer science from New York University and the M.Sc. degree in computer engineering from Columbia University. He is currently an Assistant Professor with the Department of Electrical and Computer Engineering, University of Delaware, with a joint appointment in the Department of Computer and Information Sciences. His research interests are in cybersecurity and applied cryptography, with a special focus in hardware security, trustworthy computing, and privacy outsourcing, and holds a patent on encrypted computation using homomorphic encryption.