An NVMe-based Secure Computing Platform with FPGA-based TFHE Accelerator

Yoshihiro Ohba[1], Tomoya Sanuki[1], Claude Gravel[2,3], Kentaro Mihara[2], Asuka Wakasugi[2], Kenta Adachi[2]
[1]KIOXIA Corporation, [2]EAGLYS Inc., [3]Toronto Metropolitan University (formerly Ryerson University)

***Abstract*—In this study, we introduce a new approach to secure computing by implementing a platform that utilizes a non-volatile memory express (NVMe)-based system with an FPGA-based Torus fully homomorphic encryption (TFHE) accelerator, solid state drive (SSD), and middleware on the host-side. Our platform is the first to offer completely secure computing capabilities for TFHE by using an FPGA-based accelerator. We defined secure computing instructions to evaluate 14-bit to 14-bit functions using TFHE. Our middleware allows for the communication of ciphertexts, keys, and secure computing programs while invoking secure computing programs through NVMe commands with metadata. Our performance evaluation demonstrates that our secure computing platform outperforms CPU-based and GPU-based platforms by 15 to 120 times and 2.5 to 3 times, respectively, in gate bootstrapping execution time. Additionally, our platform uses 7 to 12 times less electric energy consumption during the gate bootstrapping execution time than CPU-based platforms and 4.95 times less than a GPU-based platform. The performance of a machine learning application running on our platform shows that bootstrapping accounts for more than 80% of ciphertext learning time.**

I. Introduction

Securing data is vital because public and private organizations recognize it as an asset when collecting, using, and sharing information. Therefore, data protection regulations are growing worldwide, and the demand for global privacy and requirements is also increasing.

Along with these trends, there is a highly increasing demand for secure computation, also known as privacy-preserving methods. Gentry introduced a class of cryptographic methods known as Fully Homomorphic Encryption (FHE) [28], which is considered as one of the most compelling technologies in secure computing. FHE-based privacy-preserving methods do not require computing nodes to decrypt encrypted data to perform secure computation.

Since its inception, FHE has sparked significant interest, leading to the emergence of novel constructions following Gentry's idea. This evolution has culminated in the development of four FHE schemes, namely, BGV [12], BFV [11,25], CGGI (also known as TFHE (Torus FHE)) [20], and CKKS [10,18], which are considered the most representative and are currently undergoing international standardization under ISO [41]. This progression shows growing interests in FHE and continuous advancements in the field, making it an exciting area of research.

Bootstrapping is a critical procedure for FHE to decrease ciphertext errors by homomorphically evaluating a decryption circuit [28]. The usual computation in bootstrapping is the inner product of two vectors encoding polynomials. One vector is a transformation of a ciphertext, and the other is the encryption of the secret key used to generate the ciphertext. The computational complexity of polynomial multiplications is N times larger than that of decrypting the ciphertext, where N is the ideal degree of the polynomial ring. Because N typically ranges from 2^{10} to 2^{16} depending on the FHE mechanism and security parameters [20,30], there is a strong demand for speeding up the bootstrapping procedure. Some FHE accelerators have been built for this purpose using graphical processing units (GPU), field-programmable gate arrays (FPGA), or application-specific integrated circuits (ASIC). All the FHE accelerators listed in [30] implement number theoretic transform (NTT) into hardware. However, further work must be conducted on FHE-based secure computing platforms that integrate TFHE accelerators.

We designed and implemented a secure computing platform based on non-volatile memory express (NVMe) with an FPGA-based TFHE accelerator, solid state drive (SSD), and host-sided middleware to speed up non-linear or bit-wise operations. Our secure computing platform uses NVMe commands to read, write, and execute secure computing programs containing a sequence of secure computing instructions. The NVMe commands also read and write ciphertexts and keys. We defined a set of secure computing instructions to evaluate any 14-bit to 14-bit function using TFHE.

Our accelerator has an optimized circuit for performing $N-$point NTT or inverse NTT (INTT) with $N = 16384$ operating at 200MHz using the method described in [39] to eliminate bit-reversal operations, pre-processing of multiplying NTT twiddle factors with the inputs and post-processing of multiplying INTT twiddle factors with proper normalization. Section VI shows that our secure computing platform outperforms CPU-based and GPU-based platforms by 15 to 120 times and 2.5 to 3 times, respectively, in gate bootstrapping execution time, and uses 7 to 12 times less electric energy consumption during the gate bootstrapping execution time than CPU-based platforms and 4.95 times less than a GPU-based platform. Finally, the performance of a machine learning application running on our platform shows that bootstrapping accounts for more than 80% of the ciphertext learning time for more than 11-bit precision.

The remainder of this paper is organized as follows. Section II discusses existing work related to this study and clarifies our contributions to FHE-based secure computing. Section III describes the basic architecture of the proposed secure computing platform. Sections IV and V describe the

design and implementation of the proposed accelerator and middleware, respectively. Section VI presents the performance evaluation of our secure computing platform implementation. Finally, Section VII summarizes the study and discusses future work.

II. Related Work

A detailed survey of FHE was presented in [34]. A survey of FHE accelerators was also conducted in [30]. Three studies have been conducted on FHE-based secure computing platforms. In [47], an FPGA-based accelerator called SmartSSD [31] was used to implement the basic operations required for CKKS. In [23], a secure computing platform was implemented with an FPGA-based accelerator for NTT, focusing on accelerating the Chinese Remainder Transform (CRT) and using Direct Memory Access (DMA) to use host DRAM to communicate commands and data between the CPU and FPGA. In contrast to [23], our platform focuses on speeding up NTT in a TFHE-specific manner. However, a reduced instruction set computer (RISC)-based secure computing platform for TFHE was developed in [36], mainly focusing on accelerating CMux-tree operations without accelerating the bootstrapping procedure.

Among the FHE accelerators based on GPU [6, 7, 38, 49], ASIC [3, 43], and FPGA [2, 8, 23, 27, 40, 42, 45, 47], we chose FPGA as our initial hardware target prior to ASIC because bootstrapping is known as a memory-bandwidth-bound workload for a CPU with an arithmetic intensity (the ratio between the number of executed operations and the number of bytes transferred between the CPU and main memory) of less than 1 (see [13]). It is believed that FPGA is more suitable for such a workload than GPU because it allows multiple arithmetic logics to access different internal memory blocks or caches simultaneously. However, there has been no work on detailed TFHE bootstrapping performance comparison between GPU and FPGA.

Several studies have been conducted on FPGA-based accelerators that implement NTT for TFHE [8, 27], CKKS [2, 40], and BGV/BFV [42, 45]. For instance, an FPGA-based TFHE accelerator [8] uses a fixed-point Fast Fourier Transform (FFT) to achieve a high throughput and low control overhead. Another FPGA-based TFHE accelerator [29] uses an approximate multiplication-less integer FFT. Another FPGA-based TFHE accelerator [48] introduced an optimization technique called bootstrapping key unrolling which was designed based on the tradeoff between bootstrapping performance and FPGA resource consumption. Another FPGA-based TFHE accelerator [50] implements TFHE on ZYNQ ZCU102 FPGA board. These four FPGA-based TFHE accelerators were implemented for Small-Degree Polynomials (SDP) with $N = 1024$. In contrast, our FPGA-based TFHE accelerator has a different design goal of speeding up the multiplication of polynomials without losing precision for a large value of N. Supporting Large-Degree Polynomials (LDP) with $N = 16384$ or greater has the following advantages. First, LDP can provide higher plaintext precision in Programmable Bootstrapping (PBS) [21] than SDP. Second, given the same plaintext precision in PBS, LDP can encode more outputs of single-value or multi-value functions than SDP.

In [27], an FPGA-based programmable vector engine that supports the processing of an application-specific instruction set was designed without accelerating the bootstrapping procedure. In [37], a TFHE accelerator on a commodity CPU-FPGA hybrid machine was designed for the parallel execution of multiple homomorphic Boolean gates to increase the processing throughput without reducing latency. We focus on reducing the latency in executing bootstrapping ciphertexts encoded in LDPs.

The study in [29] extensively evaluated the performance of an FPGA-based accelerator during bootstrapping, focusing on the latency, throughput, and power consumption. However, a significant gap remains in the literature, as no study has yet comprehensively assessed an FHE-based secure computing platform that includes an accelerator and a host CPU in terms of these performance metrics.

In terms of security of FHE, existing attacks and countermeasures for the attacks on TFHE also apply to our architecture including IND-CCA/IND-CPA/IND-CPA^{D} attacks [17, 26, 33], lattice-based attacks [9, 22, 44, 46], side-channel attacks [5, 19], key-recovery attacks [14, 16] and application-level attacks [15].

Our major contributions are as follows.

- We are the first to provide a full-fledged secure computing platform for TFHE using an FPGA-based accelerator supporting LDP. The platform defines virtual registers for manipulating secure computing instructions designed for TFHE and host-sided middleware to communicate ciphertexts, keys, and programs to compute. We can invoke a secure computing program over the accelerator through middleware using NVMe commands with metadata.
- We are also the first to provide a platform-level comparison instead of a processor-level comparison among different processors such as CPU, GPU, and FPGA, showing that a secure computing platform with an FPGA-based accelerator can outperform software-based and GPU-based secure computing platforms in terms of speed and energy consumption for executing bootstrapping operations for LDP. We have shown that NTT and Inverse NTT (INTT) with $N = 16384$ are memory-I/O-bound workloads for the GPU.

III. Architecture

We introduce two architectures for our secure computing platform: Basic architecture and extended architecture. In this paper, we focus on the detailed design, implementation and performance evaluation of the basic architecture. Detailed design and implementation of the extended architecture is left for future work.

A. Basic Architecture

Figure 1 shows the basic architecture of our secure computing platform. This architecture was chosen to process data close to its location. The architecture comprises a Host, an

Accelerator, and NVMe SSDs. NVMe is a family of specifications (https://nvmexpress.org/specifications/) that defines how the host software communicates with non-volatile memory across multiple transports such as PCI Express (PCIe), Remote Direct Memory Access (RDMA) and TCP.

A secure computing application running on the host controls the behavior of the accelerator through middleware and an API by using NVMe Read/Write commands. The accelerator (i) intercepts each NVMe Read/Write command issued to NVMe SSDs, (ii) stores a copy of the data carried by the command in its main memory, and (iii) depending on the command type and properties of the command data, referred to as secure computing metadata, runs a program for homomorphic calculation and returns the result. The main reason for using NVMe as the carrier for secure computing data and metadata is to reduce the overall latency of FHE-based secure computing by performing storage I/O and computing in a single I/O command, thereby avoiding unnecessary data transfer from the host main memory to the accelerator after the host reads the data from the storage or the accelerator to the host main memory before the host writes the data to the storage. In other words, our accelerator is a computing storage accelerator. In addition, unlike SmartSSD [31], in which an FPGA and an SSD are standalone peripheral component interconnect (PCIe) devices connected under a PCIe switch, our architecture further avoids "double transfer" of the same data over the same PCIe segment between the FPGA and the PCIe switch, one for transferring the data between the host and FPGA and another for transferring the data between the FPGA and SSD, which halves the I/O throughput [32]. Instead, our architecture maintains the I/O throughput by physically separating the PCIe segment between the host and FPGA and the downstream PCIe segment between the FPGA and SSD. In implementing the basic architecture, PCIe is the NVMe transport between the host and accelerator and between the accelerator and the NVMe SSDs.

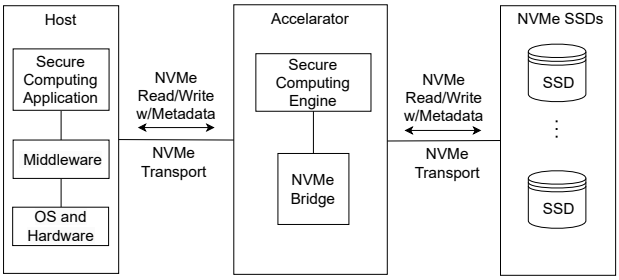


Fig. 1: Basic architecture

B. Extended Architecture

Figure 2 shows an extended architecture of our secure computing platform for providing scalability and robustness. In the extended architecture, groups of hosts, SSD servers and accelerators are inter-connected via a network called inter-cluster network and orchestrated by Kubernetes. A group of accelerators forms an accelerator cluster inside which accelerators can directly communicate via a intra-cluster network. Each host and accelerator is implemented as a Kubernetes worker node controlled by a Kubernetes master node. In the extended architecture, NVMe over fabrics (NVMe-oF) is used for NVMe transport between the hosts and SSD servers, and Kubernetes Container Storage Interface (CSI) is used for orchestrating SSD servers with other Kubernetes components. The use of Kubernetes in the extended architecture provides auto-scaling, load balancing, self-healing and dynamic configurations features to make our architecture scalable and robust.

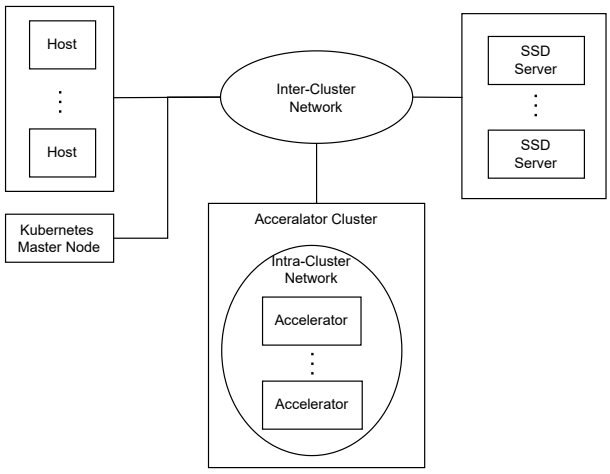


Fig. 2: Extended architecture

IV. Accelerator

A block diagram of the accelerator implementation is shown in Figure 3. Our accelerator was implemented on a HiTech Global HTG-937 board equipped with a Xilinx XCVU47P FPGA with three super logic regions (SLRs) and 16GB of high bandwidth memory (HBM). The accelerator consists of a Secure Computing Engine (hereafter referred to as the computing engine) and an NVMe Bridge. The computing engine first inputs an NVMe command or command completion with its associated data to the NVMe bridge; it extracts secure computing data and metadata from the input and stores the secure computing data in a Virtual Register (VR). Second, the engine executes a program containing a sequence of secure computing instructions, depending on the type of secure computing data indicated in the secure computing metadata and the type of NVMe Command. Third, the engine outputs the NVMe command or command completion and its associated data containing either the input or computed data to the NVMe bridge.

The computing engine has the following components.

- The *main memory* stores VRs, VR tables, and page tables. It also has a stack region used by the `push` and `pop` instructions, defined in Section IV-D. An HBM is a memory device that provides sufficient memory access bandwidth. The main memory is partitioned into a persistent area for which paging operations are not applied and a non-persistent area for which the paging

operations are used. See Section IV-C for details on the paging operations.

- The *cache memory* consists of many block RAM (BRAM) blocks for high-speed distributed memory access.
- The *data movers* move secure computing data and metadata between the main memory, cache memory, central processor, and module processor. Data movers have first-in first-out (FIFO) buffers.
- The *module processor* provides ring or vector operations. Multiple logic blocks in the module processor can simultaneously access different BRAM blocks in the cache memory. The module operations supported by the module processor are presented in Table I. All module operations were implemented as high-level synthesis (HLS) modules, except for NTT and INTT. The NTT/INTT circuit was implemented as a register transfer level (RTL) module as described in Section IV-E. Module operations are performed element-wise except for NTT, RING_ROT, VECTOR_ROT, and SAMPLE_EXT. The module processor has one MicroBlaze soft-core microprocessor to process the module operations.
- The *central processor* executes microprograms to control the data movers and module processor and manage the main memory. A MicroBlaze soft-core microprocessor is used as the central processor.
- The *multiplexers* and *demultiplexsers* exchange NVMe commands and command completions with their associated data input from the NVMe bridge among the soft-core microprocessor, data movers, and NVMe bridge.

The NVMe bridge provides a bridging function for NVMe commands; that is, it forwards an NVMe command from the host to the NVMe SSDs, either forwards Write Data from the host to the NVMe SSDs or forwards Read Data from the NVMe SSDs to the host, and forwards an NVMe command completion received from the NMVe SSDs to the host. Before forwarding an NVMe command or command completion, the NVMe bridge passes NVMe Read/Write data to the computing engine for copying and data computation. The NVMe bridge has a MicroBlaze soft-core microprocessor. We use IntelliProp's NVMe bridge IP core licensed for Xilinx XCVU47P FPGA.

A. Secure computing metadata

Each piece of secure computing data or `sc_data` accompanies secure computing metadata or `sc_metadata` containing the type of `sc_data`, Key Identifier identifying the set of keys associated with `sc_data`, data identifier of `sc_data`, and size of `sc_data`. The domain of the type field is listed in Table II.

Any `sc_data` carried in the NVMe Read/Write command data are stored in the VR corresponding to the associated `sc_metadata`. In addition, `sc_data` of Type 3 (TLWE-CoR) carried with an NVMe Read Command or Type 4 (TLWE-CoW) carried with an NVMe Write Command invokes the execution of a secure computing program stored in the VR register of Type 0, or the program register (see Figure 4).

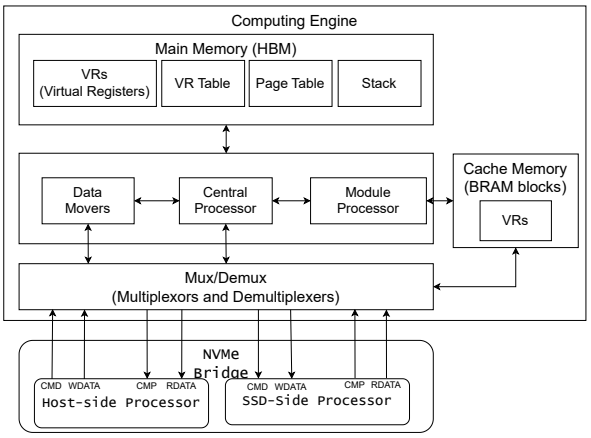


Fig. 3: Accelerator block diagram

TABLE I: Module processor operations. NTT operation is implemented as Register Transfer Level (RTL) modules. All other operations are implemented as High Level Synthesis (HLS) modules

Name of Internal Operation	Module Type	Description
`NTT`	Ring	NTT and INTT
`MULMOD64`	Vector	64-bit element-wise multiplication modulo prime $p = 2^{64} - 2^{32} + 1$ for CMux
`ADDMOD64`	Vector	64-bit element-wise addition modulo prime $p = 2^{64} - 2^{32} + 1$ for CMux
`KEY_SWITCH`	Vector	Public functional Key Switching
`DECOMP`	Vector	Gadget Decomposition
`ADD32_ACC`	Vector	32-bit element-wise addition to ACC
`SUB32_ACC`	Vector	32-bit element-wise subtraction to ACC
`ADD32_VR`	Vector	32-bit element-wise addition to VR
`SUB32_VR`	Vector	32-bit element-wise subtraction to VR
`INT_MULT32`	Vector	32-bit element-wise scalar multiplication to VR
`RING_ROT`	Ring	Circular rotation of ring coefficients
`VECTOR_ROT`	Vector	Circular rotation of Vector elements
`SAMPLE_EXT`	Ring	Sample Extract

ACC: Accumulator

TABLE II: Secure computing metadata types

Type	Type Name	Description
0	PRG	Secure computing program
1	TV	Test vector
2	KEY	Key used for TFHE bootstrapping operations
3	TLWE-CoR	TLWE ciphertext invoking Compute-on-Read (CoR) operation
4	TLWE-CoW	TLWE ciphertext invoking Compute-on-Write (CoW) operation

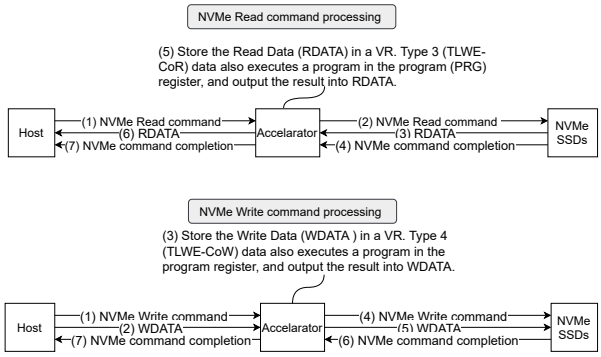


Fig. 4: NVMe Read/Write command processing

B. Virtual registers and virtual addressing

Virtual Registers (VRs) are variable-length data structures maintained inside an accelerator and are manipulated using NVMe Read/Write commands. They are distinguished from soft-core microprocessor registers in that a VR can be so large that its entire part does not fit into FPGA logical elements. For example, a TFHE bootstrapping key can be a few gigabytes in size.

A VR number identifies each VR and is uniquely calculated from the Type, Key Identifier, and Data Identifier contained in `sc_metadata` associated with the corresponding `sc_data`. The VR table stores the pairs of VR numbers and virtual addresses for each VR.

The type and data identifier specified in Table III determines the size of the VR. We use a 32-bit integer to encode an element in $\mathbb{T} = \mathbb{R}/\mathbb{Z}$ for all data types other than the NTT-applied keys, which used 64-bit integers to multiply ring polynomials using NTT.

The accelerator uses a 38-bit virtual address with a 26-bit page number and a 12-bit offset. BKNTT, KSK, and PrvKSKNTT denote keys. BKNTT is a bootstrapping key that is transformed linearly. Specifically, NTT is applied to the bootstrapping key. KSK is a key-switching key used in the public functional key-switching mechanism of TFHE. PrvKSKNTT is a key-switching key used in the private-functional key-switching mechanism of TFHE and is transformed linearly in the same manner as BKNTT. Parameters N, k, n, ℓ, and t denote the degree of the polynomial representing the ideal, the number of polynomials encoding a secret key in a Torus Ring Learning with Errors (TRLWE) sample, the bit length of the Torus LWE (TLWE) secret key, the number of digits in the radix B_g of the TFHE Gadget Decomposition algoritym [20], and the number of digits in the binary-decomposed TLWE samples.

C. Paging

Similar to legacy computers, the proposed accelerator invokes a paging function when the main memory is full. The paging algorithm implemented on the accelerator uses NVMe SSDs as the swap area. It maintains the contents of the received VRs to be stored in the main memory or swap area

TABLE III: Virtual register sizes (BKNTT: NTT-applied bootstrapping key, KSK: key-switching key, PrvKSKNTT: NTT-applied private-functional key-switching key)

Type	Data Identifier	VR Size in bytes
0 (PRG)	0	configurable
1 (TV)	any	$(k+1)\cdot N\cdot 4$
	1 (BKNTT)	$n\cdot N\cdot \ell\cdot (k+1)^2\cdot 8$
2 (KEY)	2 (KSK)	$(n+1)\cdot t\cdot N\cdot 4$
	3 (PrvKSKNTT)	$(k+1)\cdot (n+1)\cdot t\cdot N\cdot 8$
3 (TLWE-CoR)	any	$(n+1)\cdot 4$
4 (TLWE-CoW)	any	$(n+1)\cdot 4$

in the following manner. Each VR content initially sent to the accelerator through an NVMe command is temporarily held in a FIFO queue in the cache memory and then moved to the main memory. Suppose that the central processor of an accelerator attempts to access a VR, say x. Suppose that neither x is in the main memory nor sufficient space is available to store x. In this case, the paging algorithm (i) selects another VR, say y, stored on a page of the main memory, copies the page's content to the swap area, and (ii) copies x to the page. Paging operations (i) and (ii) are *page-out* and *page-in* operations, respectively. The copy source of a page-in operation is either the swap area for the previously received VR or the cache memory for the newly received VR.

The accelerator includes a page table dedicated to Type 3 (TLWE-CoR) and Type 4 (TLWE-CoW) VRs, with a page size that matches the VR size. All other VRs are stored in the persistent area of the main memory, for which paging operations are not required. Each entry in the page table contains a flag and physical address for the corresponding page. If the flag is unset, the physical address field contains the physical address of the main memory. Otherwise, it contains the LBA of a logical block in the swap area. The page-in and page-out operations rely on NVMe Read/Write commands to transfer pages between the accelerator and NVMe SSDs.

D. Secure computing instruction set

The accelerator supports the following secure computing instructions, summarized in Table IV.

- `return` sends the content of VR n containing a TLWE sample to the host or the SSD.
- `move` moves the content of VR n_2 containing a TLWE sample to VR n_1.
- `push` moves the content of VR n containing a TLWE sample to the top of the stack and increments the stack pointer.
- `pop` moves the content of the stack top to VR n and decrements the stack pointer.
- `bootstrap` uses the content of VR tv containing a TFHE test vector, performs Gate Bootstrapping (GBS) [20] for VR n containing a TLWE sample, and stores the output to VR n. Note that GBS also realizes Programmable Bootstrapping (PBS) [21] for a function $f(x)$ provided by the TFHE test vector, in which case the default TFHE test vector for the identity function is

replaced with the provided one. Note that a pair of a bootstrapping key and key-switching key used for GBS is identified by the Key ID field of `sc_metadata` associated with VR register n.

- `homadd` instruction adds the content of VR n_2 containing a TLWE sample and VR n_1 containing another TLWE sample and stores the result to VR n_1. `homadd` internally executes `ADD32_VR`.
- `homsub` subtracts the content of VR n_2 containing a TLWE sample from VR n_1 containing another TLWE sample and stores the result to VR n_1. `homsub` internally executes `SUB32_VR`.
- `homintmult` multiplies the content of VR n containing a TLWE sample by value v and stores the result in VR n. `homintmult` internally executes `INT_MULT32`.

TABLE IV: Secure computing instruction set (VRs n, n_1, and n_2, each containing a TLWE sample. VR tv contains a TFHE test vector)

Name	Type	Arg. 1	Arg. 2	Description
`return`	0	n	none	return n
`move`	1	n_1	n_2	$n_1 \leftarrow n_2$
`push`	2	n	none	`++stackptr` $\leftarrow n$
`pop`	3	n	none	$n \leftarrow$ `stackptr--`
`bootstrap`	4	tv	n	perform GBS or PBS for n with tv
`homadd`	5	n_1	n_2	$n_1 \leftarrow n_1 + n_2$
`homsub`	6	n_1	n_2	$n_1 \leftarrow n_1 - n_2$
`homintmult`	7	n	v	$n \leftarrow n \cdot v$

Table V shows the sequence of instructions of a secure computing program that homomorphically performs a multiplication of two integers, x and y. Also, Table VI shows an example sequence of NVMe commands used for running the example secure computing program. Note that once the VRs are loaded into the accelerator via NVMe Read/Write Commands, they only need to be reloaded once they are updated. For example, Steps 1-6 in Table VI are not required for the next run of another secure computing program that uses the same BK, KSK, and TV1. In addition, for NVMe SSDs supporting the NVMe Metadata feature, the number of NVMe Commands in the sequence is reduced by half using `sc_metadata` and its associated `sc_data` contained in the same NVMe Read/Write Command.

TABLE V: An example of a secure computing program for computing $xy = \{(x+y)^2/4 - (x-y)^2/4\}$ homomorphically (r_i is the VR number for TLWE sample s_i. tv is the VR number for the test vector representing the function $f(z) = z^2/4$. TLWE samples s_2 and s_3 contain encrypted data for x and y, respectively. VR r_1 stores a temporal result and a final result to return)

No.	Instruction	Argument(s)
1	`mov`	r_1, r_2
2	`homadd`	r_1, r_3
3	`bootstrap`	tv, r_1
4	`homsub`	r_2, r_3
5	`bootstrap`	tv, r_2
6	`homsub`	r_1, r_2
7	`return`	r_1

TABLE VI: An example sequence of NVMe commands (Write(d) represents an NVMe Write command with data d. Read(d) represents an NVMe Read command with data d. MD(x, y, z) represents the NVMe metadata of Type x, Key Identifier y, and Data Identifier z. An NVMe command completion (not shown in the figure) is returned for each NVMe command. Abbreviations: BK: bootstrapping key. KSK: key-switching key. PRG: secure computing program)

No.	NVMe Command	Comment
1	Write(MD(2,0,1))	BKNTT
2	Write(k_0)	BKNTT data
3	Write(MD(2,0,2))	KSK
4	Write(k_1)	KSK data
5	Write(MD(1,0,0))	TV1
6	Write(v_1)	TV1 data
7	Write(MD(0,0,0))	PRG
8	Write(p)	PRG data
9	Write(MD(3,0,1))	TLWE-CoR
10	Write(s_1)	TLWE-CoR data for TLWE sample s_1 encrypting value x
11	Write(MD(4,0,2))	TLWE-CoW
12	Write(s_2)	TLWE-CoW data for TLWE sample s_2 to be used for encrypting $f(x)$. Secure computing program p is invoked here.
13	Read(MD(4,0,2))	TLWE-CoW
14	Read(s_2)	TLWE-CoW data for TLWE sample s_2 encrypting value $f(x)$

E. NTT implementation and optimized CMux

Because bootstrapping is the most time-consuming operation in TFHE, the NTT/INTT circuit in the module processor was implemented as an RTL module to optimize its circuit design. The NTT/INTT circuit supports N-point NTT/INTT with $N = 16384$.

The NTT/INTT circuit implements an optimized scheme described in [39], which eliminates pre-FFT (Fast Fourier Transform) processing and post-IFFT (Inverse FFT) processing (including bit reversing) by merging NTT twiddle factors $\{\psi^i \mid 0 \leq i < N\}$ and FFT twiddle factors $\{\omega^i | 0 \leq i < N\}$ where ω is a primitive Nth root of unity and ψ is a primitive $2N$-th root of unity, and thus $\psi = \omega^2$. For readers convenience, the NTT and INTT constructions for $N = 8$ are shown in Figure 5. The NTT and INTT constructions are the same as those in Figure 1 in [39]. In the proposed implementation, $\omega = 10930245224889659871$ and $\psi = 3333600369887534767$. Appendix A provides mathematical derivations for the NTT and INTT constructions using the two types of butterfly elements. There is another optimization scheme described in [24] in which a specific Nth root of unity for which the 64-th root of unity is 8, but the scheme is not used for our accelerator because not all twiddles derived from the specific Nth root of unity have a power of two. Thus, more logic resources are required to implement the two types of butterfly circuits (one for power-of-2 twiddles and another

for other twiddles).

For a complete INTT operation, a normalization factor of $1/N$ is required for each output element of the INTT as shown in Figure 5. This scaling can be precomputed [39] for the input of an NTT operation depending on the arithmetic operation that uses the NTT in its implementation. In TFHE, the CMux gate [20], as defined below, is an arithmetic operation in which $1/N$ scaling can be precomputed.

$$\text{CMux}(C, d_0, d_1) = \sum_{i=1}^{(k+1)\ell} u_i \cdot C_i + d_0 = \langle u, C \rangle + d_0,$$

where $C = (C_i)_{1 \le i < (k+1)\ell}$ is a Torus Ring GSW (TRGSW) sample, d_0 and d_1 are TRLWE samples, and $u = (u_1, u_2, ..., u_{(k+1)\ell}) \in (\mathbb{Z}[X]/(X^N+1))^{(k+1)\ell}$ is the output of the Gadget Decomposition for $d_1 - d_0$ and $\langle x, y \rangle$ denotes the inner product of two vectors x and y.

In [20], CMux gates were used inside the TFHE Blind Rotate algorithm, which is invoked from the TFHE Gate Bootstrapping algorithm (Case 1) or the TFHE Vertical Packing algorithm used together with the TFHE Circuit Bootstrapping [20] (Case 2). In Case 1, C_i is the ith TRGSW sample of bootstrapping key BK. In Case 2, C_i is the ith TRGSW sample of the output of Circuit Bootstrapping, calculated as a linear sum of the elements of private functional key-switching key PrvKSK. Because C_i is generally a linear sum of constant polynomials in both cases and takes advantage of the linearity property of NTT, our optimized CMux gate uses pre-scaled and pre-transformed keyntt $= (\text{keyntt}_i)_i = (\text{NTT}^{(N)}(\text{key}_i/N))_i$, where key_i is the ith element of BK or PrvKSK, as follows:

$$\begin{aligned}
\langle u, C \rangle &= \frac{1}{N} \sum_{i=1}^{(k+1)\ell} \text{INTT}^{(N)} \left(\text{NTT}^{(N)}(u_i) \odot \text{NTT}^{(N)}(C_i) \right) \\
&= \text{INTT}^{(N)} \left(\sum_{i=1}^{(k+1)\ell} \text{NTT}^{(N)}(u_i) \odot \text{NTT}^{(N)}(C_i/N) \right) \\
&= \text{INTT}^{(N)} \left(\sum_{i=1}^{(k+1)\ell} \text{NTT}^{(N)}(u_i) \odot \langle c'_i, \text{keyntt} \rangle \right) \quad (1)
\end{aligned}$$

where $\text{NTT}^{(N)}(\cdot)$ and $\text{INTT}^{(N)}(\cdot)$ are N-point NTT and INTT functions, respectively. $\odot$ denotes the Hadamard product. $c'_i = (c'_{i,j})_{1 \le j \le (n+1)t}$ is an integer vector with $c'_{i,j} = \delta_{ij}$ for Case 1 where δ_{ij} is the Kronecker delta function, and $c'_{i,j} = -\tilde{c}_{i,j-1 \text{ div } t, j-1 \bmod t}$ for Case 2 where $(\tilde{c}_{i,j,k})_{1 \le j \le n+1, 1 \le k \le t}$ are t bit-decomposed TLWE samples generated from the i-th TLWE sample in private-functional key-switching [20]. This optimization halves the number of NTT operations performed in CMux. To date, we have implemented only Case 1.

Figure 6 shows two types of radix-2 butterfly calculation elements: one for NTT and the other for INTT. Our implementation integrates these two types of butterfly calculation elements into a single integrated butterfly circuit, as shown in Figure 7. The integrated butterfly circuit has the same construction as the NTT and INTT parts of the unified butterfly circuit described in Ref. [51].

Figure 8 shows the pipeline and parallel processing model for computing NTT and INTT in the module processor. The NTT/INTT circuit in the module processor has 32 integrated butterfly circuits operating in parallel at 200MHz, and is partitioned into two sub-circuits of 16 integrated butterfly circuits, where each sub-circuit processes one of the two polynomials in a sample $(a, b) \in \{\mathbb{Z}[X]/(X^N+1)\}^2$. Data processing within each integrated butterfly circuit is pipelined such that the subsequent coefficients and twiddles are read from the BRAM during the butterfly calculation for the current coefficients and twiddles. Each integrated butterfly circuit performs 512 butterfly calculations in the pipeline to compute one row of NTT or INTT. Note that the transfer of coefficients between the HBM and BRAM is performed by the data mover in the background of the module processor pipeline, which never causes pipeline stall owing to the high bandwidth of the HBM.

Figure 9 shows the die layout of the accelerator. Two design policies were applied to reduce data transfer among the SLRs. First, each sub-circuit of the module processor is laid out in a different SLR (i.e., SLR#2 and SLR#3 in Figure 9). Second, the data movers are placed in SLR#1, which is the closest SLR to the HBM, as the data movers are the interface between the HBM and other FPGA logic.

F. Computing modulo prime $p = 2^{64} - 2^{32} + 1$

NTT for a 32-bit torus can use any prime number greater than $(2^{32}-1)^2 = 2^{64} - 2^{33} + 1$. We chose a Proth prime $p = 2^{64} - 2^{32} + 1 = 18446744069414584321$ as used in existing open-source TFHE implementations because modulo calculation for a Proth prime requires no integer multiplication calculation. For two integers $x, y \in [0, p)$, $z = xy$ can be decomposed into three parameters $a, b \in [0, 2^{32})$ and $c \in [0, 2^{64})$ as $z = a \cdot 2^{96} + b \cdot 2^{64} + c$, $z \bmod p$ for z is calculated using addition, subtraction, shift, and comparison operations as follows.

$$z \bmod p = \begin{cases} m(z), & \text{if } z < 2p \\ m\big(m(b \cdot 2^{32}) + \\ \quad m\big(m(c) + p - m(a+b)\big)\big), & \text{otherwise} \end{cases}$$

where

$$m(j) = \begin{cases} j, & \text{if } j < p \\ j + \texttt{uint32}(-1), & \text{if } j \in [p, 2p). \end{cases}$$

Although a 32-bit polynomial ring multiplication via NTT requires 64-bit to 64-bit integer multipliers compared with naive 32-bit polynomial ring multiplication without NTT using 32-bit to 32-bit integer multipliers, the former requires only $4(N/2)\log_2 N = 2Nlog_2N$ cycles of 32-bit to 32-bit integer multiplications when a 64-bit to 64-bit multiplier is implemented by four 32-bit to 32-bit multipliers. In contrast, the latter requires N^2 cycles of 32-bit to 32-bit integer multiplications.

V. Middleware

Set/get `sc_metadata` and `sc_data` to/from the accelerator and allow the accelerator to execute secure computing programs via NVMe commands, the middleware of our

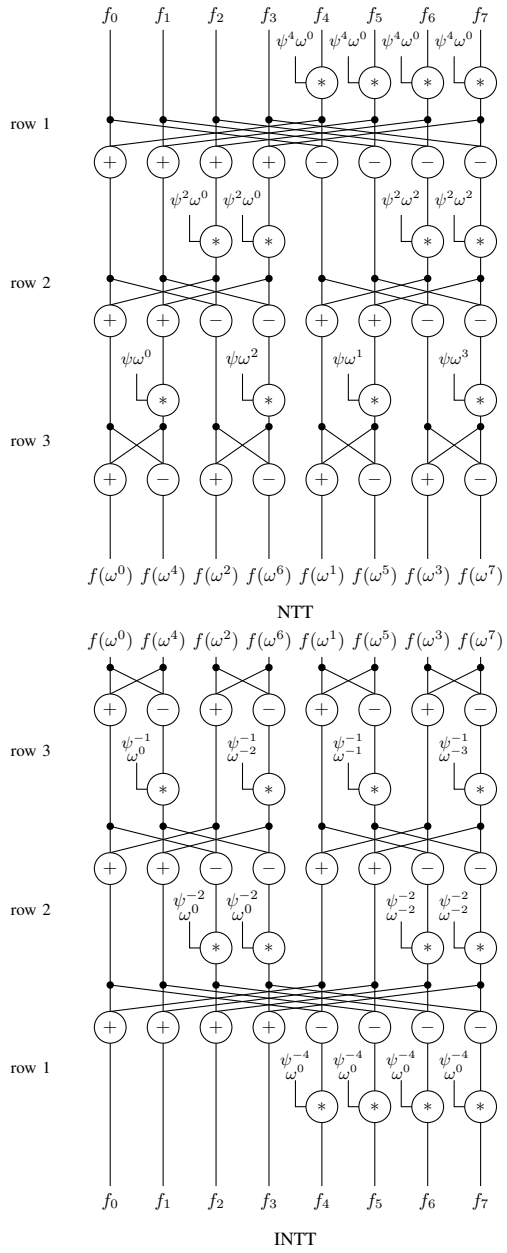


Fig. 5: N-point NTT and INTT constructions for $N = 8$ (ω and ψ are Nth and $2N$th roots of unity, respectively. $\{\omega^i : 0 \leq i < N\}$ and $\{\psi^i : 0 \leq i < N\}$ are FFT twiddle and NTT twiddle factors, respectively. f_i is the ith input element in the time domain. $f(\omega^i)$ is the value in the frequency domain for f_i. INTT's row number is in reverse order of NTT's row number)

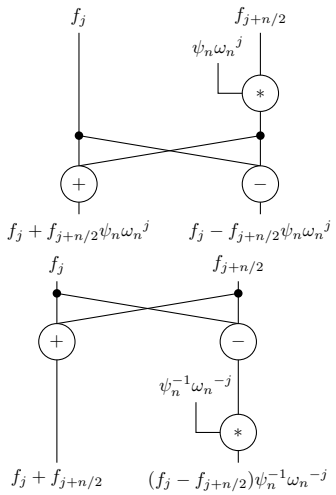


Fig. 6: Two types of radix-2 butterfly circuits at row $\log_2 n$ (Left: NTT butterfly, Right: INTT butterfly, $n = 2, 4, \ldots, 2^i, \ldots, N, 0 \leq j < n/2$, $\omega_n = \omega^{N/n}, \psi_n = \psi^{N/n}$. N is a power of 2)

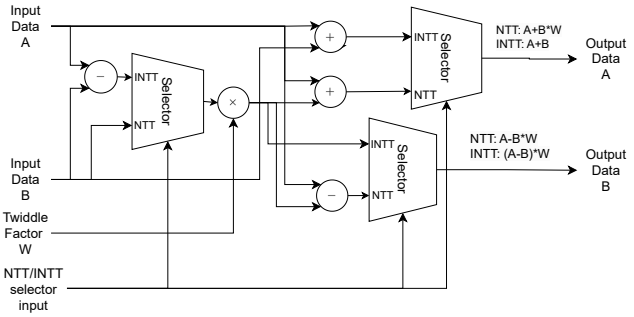


Fig. 7: Integrated Butterfly Circuit

platform uses the Blobstore feature of Storage Performance Development Kit (SPDK) (https://spdk.io/). Figure 10 shows the middleware architecture. The middleware API functions and the internal API functions from SPDK and SPDK Blobstore (hereafter the blobstore) are callback-based functions to achieve high-performance and nonblocking NVMe storage access; functions directly or indirectly interact with an abstraction thread library primarily based on the Portable Operating System Interface (POSIX).

The blobstore manages user data as blobs. A blob consists of blob data that contains user data and blob metadata describing attributes, such as the size of the blob data. Blob data and metadata are stored as clusters, each consisting of one or more pages stored in consecutive logical blocks. The first cluster stores pieces of blob metadata in its corresponding region, and the remaining clusters store pieces of blob data. The host RAM maintains a copy of the blob metadata region.

The accelerator can assemble VRs without overhead because it accesses the blob metadata region of the disk or maintains a copy of the blob metadata region in the HBM or BRAM (1) by using the extended portion of blob metadata to pass secure computing metadata between the middleware API and the Blobstore API, and (2) by mapping between the blob metadata and `sc_metadata` in one of the following ways.

The mapping is straightforward for NVMe SSDs supporting NVMe Metadata; the middleware places `sc_metadata` into the NVMe Metadata part of an NVMe Read/Write command for reading or writing a page or pages of a cluster. For

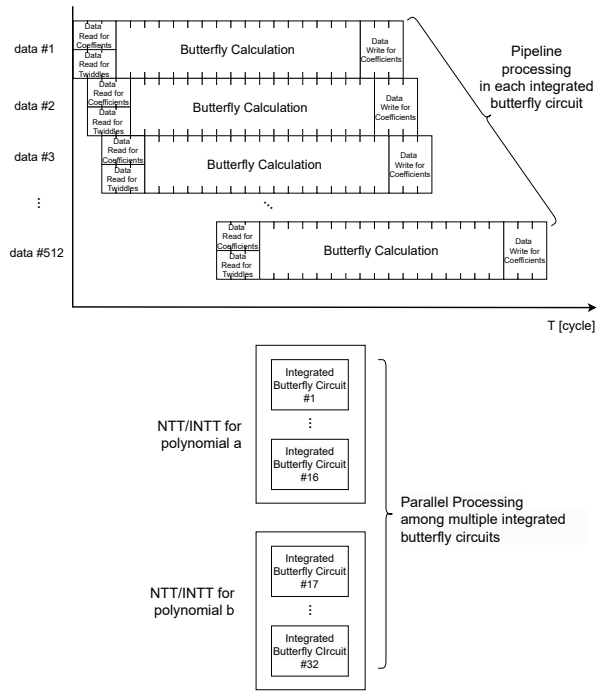


Fig. 8: Pipeline and Parallel Processing Model for NTT and INTT (Upper: pipelining within each integrated butterfly circuit, Lower: parallel processing among multiple integrated butterfly circuits)

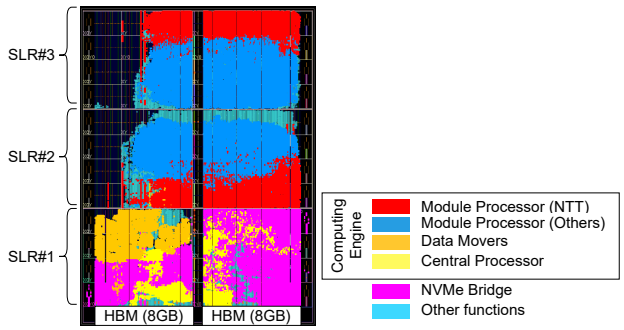


Fig. 9: Accelerator die layout

other NVMe SSDs that do not support NVMe Metadata, the middleware partitions the entire NVMe LBA space into two equally-sized LBA subspaces, using the first LBA subspace to store all clusters and the second LBA subspace to store `sc_metadata`. Let $L = \log_2(S_{\max})$ where $S_{\max}$ is the maximum size of the blob storage, $p(i, j, k)$ be the kth page of the jth cluster of the ith blob, $a(i, j, k)$ be the LBA of $p(i, j, k)$, respectively. Then, the LBA of the `sc_metadata` for the jth cluster of the ith blob is calculated as $a(i, j, 0) + 2^{L-1}$, as shown in Figure 11. The current middleware and accelerator implementations are based on the latter scheme. Note that a more space-efficient subspace management is possible for the latter scheme by packing pieces of `sc_metadata` into consecutive logical blocks in the second LBA subspace to provide more room for the first LBA subspace.

The middleware API functions were written in Rust (https://www.rust-lang.org/) and are listed in Table VII. Because the blob sizes for some VRs, such as BKNTT, can be large, two methods were defined for blob read and write commands. One method specifies the file name as the source and destination of the blob data in `write_blob1` and `read_blob1`, respectively. The second method specifies the memory address as the source and destination of blob data in `write_blob2` and `write_blob2`, respectively.

TABLE VII: Middleware API functions (The last two arguments of each API function are a callback function and its argument)

Function	Arguments
create_blob	- `md: Metadata` - `cb_fn: spdk_blob_op_with_id_complete` - `cb_arg: *mut c_void`
delete_blob	- `blobid: BlobId` - `cb_fn: spdk_blob_op_complete` - `cb_arg: *mut c_void`
write_blob1	- `blobid: BlobId` - `data_file: String` - `cb_fn: spdk_blob_op_complete` - `cb_arg: *mut c_void`
write_blob2	- `blobid: BlobId` - `data: &mut Vec<u8>` - `cb_fn: spdk_blob_op_complete` - `cb_arg: *mut c_void`
read_blob1	- `blobid: BlobId` - `data_file: String` - `cb_fn: spdk_blob_op_complete` - `cb_arg: *mut c_void`
read_blob2	- `blobid: BlobId` - `data: &mut Vec<u8>` - `cb_fn: spdk_blob_op_complete` - `cb_arg: *mut c_void`

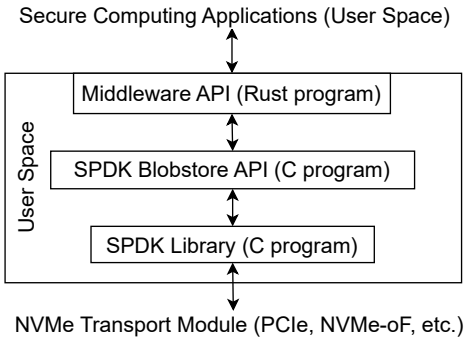


Fig. 10: Middleware architecture

VI. Performance Evaluation

In addition to developing a full-fledged secure computing platform for TFHE using an FPGA-based accelerator, we provide a system-level comparison of FPGA-based, GPU-based and CPU-based secure computing platforms. We then

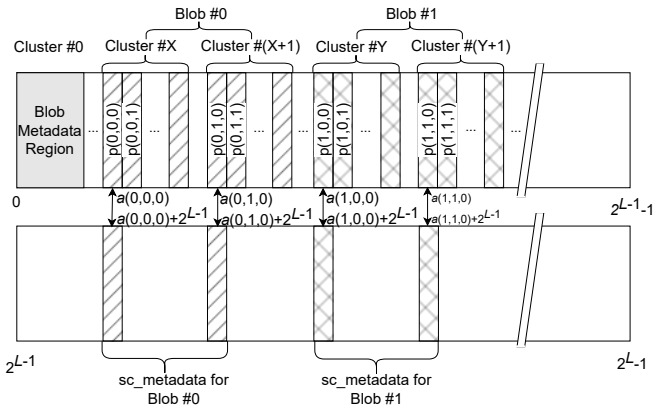


Fig. 11: NVMe logical address space map

performed a simple linear regression with ciphertexts on our platform and compared the learning results when the computational accuracies were 10 and 14 bits. This section uses the following TFHE parameters to evaluate our secure computing platform with an FPGA-based accelerator: $n = 800$, $\alpha = 2^{-19}$, $N = 16384$, $k = 1$, $B_g = 2^6$, $l = 5$, $t = 7$ where α is the standard deviation of the noise. This parameter set provides 128-bit classical security [4] using a lattice parameter estimator tool (https://github.com/malb/lattice-estimator). Table VIII shows the sizes of fixed-length VRs with this parameter set.

Note that $N = 16384 (= 2^{14})$ was chosen for providing 14-bit plaintext accuracy via PBS and $k = 1$ was chosen following all known TFHE implementations. $(B_g, l) = (2^6, 5)$ was chosen to satisfy that $l \cdot \log_2 B_g$ is as close to as but no more than the bit length of Torus. In our architecture, B_g is the parameter that controls tradeoff between reliability of decryption and processing speed. Smaller B_g makes the amount of noise accumulated in the ciphertext lower, and hence increases the reliability of decryption. On the other hand, smaller B_g makes l larger, which in turn makes the size of BKNTT and bootstrapping processing time larger. For example $(B_g, l) = (2^4, 8)$ makes BKNTT size and bootstrapping time $8/6 = 1.3$ times larger than those for $(B_g, l) = (2^6, 5)$ at the cost of increasing decryption reliability.

TABLE VIII: Evaluated virtual register sizes (BK: bootstrapping key, BKNTT: NTT-applied bootstrapping key, KSK: key-switching key)

Type	Data Identifier	VR Size in bytes
2 (KEY)	1 (BKNTT) 2 (KSK)	2.10GB 367MB
3 (TLWE-CoR)	any	3.2KB
4 (TLWE-CoW)	any	3.2KB

A. Amount of FPGA resources

Table IX lists the number of FPGA resources used. BRAM resources are the most utilized resource in FPGA. Table X lists the number of FPGA logic resources for each function. The computing engine uses logic resources three times more than the NVMe bridge does.

TABLE IX: FPGA resources (LUT: Look-Up Table, FF: Flip Flop, BRAM: Block RAM, URAM: Ultra RAM, DSP: Digital Signal Processor)

Resource	Utilization	Available	Utilization (%)
LUTs	625520	1303680	47.98
FFs	763718	2607360	29.29
BRAM Blocks	1265.50	2016	62.72
URAM Blocks	96	960	10.00
DSP Slices	1564	9024	17.33

TABLE X: Breakdown of FPGA resources (NB: NVMe Bridge, CE: Computing Engine. Registers are constructed from FFs)

Name	LUTs	Registers	BRAM Blocks	URAM Blocks	DSP Slices
NB	134504	115600	233	0	9
CE	461123	615303	1023.5	64	1549
Other	29893	32818	8	32	6

B. Secure computing instruction execution time

Table XI lists the average, minimum, and maximum execution times of each secure computing instruction by our accelerator. The minimum and maximum values for Bootstrap execution time are within ±10us of the average value. Table XII shows the average execution times of GBS for comparing a software-based platform, a GPU-based platform, and our FPGA-based platform.

Our FPGA-based platform uses an AMD Ryzen 9 5950X (3.4-4.9GHz/16-core/32-thread/ 64MB cache) CPU with 128GB RAM as its host-side CPU. We took ten runs for each secure computing instruction on the accelerator.

The software-based platform uses TFHEpp, an open-source TFHE implementation [35], running on two CPU architectures: AMD Ryzen 9 5950X (the same CPU as the host-side CPU of our FPGA-based platform) and Apple M1 (an ARM-based system-on-a-chip (SoC) processor) with 16GB RAM. As an open-source software, we used the *gatebootstrappingntt* test suite from the TFHPpp [35] with commit c6c5a38, using the same parameter set as our accelerator. Ten measurements were taken for the `gatebootstrappingntt` test suite on the CPU.

For GPUs, NVIDIA Tesla T4 on AWS EC2 `g4dn.2xlarge` and NVIDIA A100 with 40GB HBM on a local PC with AMD Ryzen 7 5800X (3.8-4.7GHz/8-core/16-thread/ 32MB cache) CPU with 128GB RAM were used, both running a modified version of the cuFHE library [1] to add support for $N = 16384$. The source code of the modified cuFHE library is available at https://github.com/eaglys-platform/cuFHE16384.git. The original cuFHE library only supports GBS for N = 1024

and contains several flaws in its NTT implementation, such as the lack of multiplication by a twiddle factor inside the radix-2 butterfly. We also addressed these flaws for a fair comparison. We implemented two GPU schemes: Schemes 1 and 2. In Scheme 1, each thread performs one butterfly calculation at each butterfly stage of NTT/INTT by allocating eight streaming multiprocessors (SMs) for each NTT and INTT, and at most $8(k+1)(=16)$ NTT or INTT operations run in parallel on 16 SMs. In Scheme 2, each thread sequentially performs eight butterfly calculations at each stage of NTT/INTT by allocating one SM for each NTT, and at most $(k+1)\ell(=10)$ NTT or INTT operations run in parallel on 10 SMs. While both Schemes 1 and 2 worked with T4, Scheme 2 did not work with A100 due to a runtime resource shortage error. With T4, Scheme 1 achieves a higher parallelism than Scheme 2, whereas Scheme 2 avoids device-level thread synchronization during GBS processing, including NTT and INTT. In both schemes, there are 1024 threads per SM. Both schemes were implemented to generate less than $1024 \cdot M$ instantaneous threads where M is the maximum number of SMs and $M = 40$ for T4 and $M = 108$ for A100. For the GPUs, 100 GBS measurements were taken.

Note that device-level thread synchronization is required among threads across all SMs. We also note that the entire NTT or INTT input or output data for $N = 16384$ coefficients of a polynomial fit into the L2 cache of the GPU device, whereas the data do not fit into the L1 cache of a single SM. Regarding the GBS processing time for $N = 16384$, our accelerator outperformed the CPU-based and GPU-based platforms by 15 to 120 times and by 2.5 to 3 times, respectively.

Figure 12 shows the GPU and FPGA processing breakdowns of GBS. The GPU is three to four times slower than the FPGA in processing NTT and INTT, whereas there is no significant difference for non-NTT/INTT operations. Figure 13 shows the GPU processing breakdown of the NTT and INTT. A comparison of Schemes 1 and 2 of Tesla T4 in Figure 13 shows the tradeoff between parallelism and synchronization in the GPU. Figure 13 also shows that NTT and INTT are memory-bandwidth-bound workloads for the GPU. Comparing T4 and A100 on Scheme 1 NTT/INTT performance in Figure 12, A100 shows higher NTT/INTT processing time than T4. This is because A100 has more SMs than T4, and hence it takes longer time for device-level thread synchronization.

Our FPGA outperforms the GPU in terms of GBS processing time for large degree (such as $N = 16384$) polynomials because (i) our FPGA allows multiple integrated butterfly circuits to access different BRAM blocks in parallel, (ii) our FPGA pipelines butterfly calculation and memory access, and (iii) our FPGA does not require device-level thread synchronization.

C. Secure computing program execution time

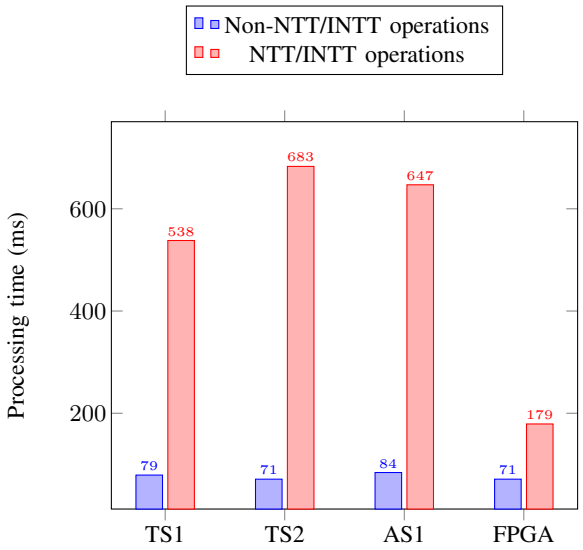
Table XIII lists the execution time of a secure computation program on our platform. We use the secure computing program listed in Table V. Correctness of the program is validated by comparing the decrypted and decoded return value z with the multiplication of the two cleartext input values x and y for

TABLE XI: Secure computation instruction execution time on our accelerator. The minimum and maximum values for Bootstrap execution time are within $\pm$10us of the average value

Instruction	Average	Minimum	Maximum
Bootstrap	249.96ms	249.96ms	249.97ms
HomAdd	124us	124us	125us
HomSub	124us	124us	125us
HomIntMult	90us	90us	90us

TABLE XII: GBS execution time comparison

CPU-based Platform	Ryzen 9	Apple M1
	3.97s	30.8s
GPU-based Platform	Tesla T4	A100
	617ms (Scheme 1) 754ms (Scheme 2)	731ms (Scheme 1)
FPGA-based Platform	250ms	



TSi: T4 Scheme $i(i = 1, 2)$, AS1: A100 Scheme 1

Fig. 12: GPU and FPGA Processing Breakdown for GBS

$x, y, z \in [a, b)$, and the validation succeeds if $|z - xy|/b \leq h$ where $a = -10.0, b = 10.0, h = 0.3$. According to Table XIII, since the program contains two `bootstrap` instructions, each taking 249.96ms, bootstrapping dominates the overall performance of the execution time of a secure computing program compared to the execution time of other instructions and the processing time of the NVMe Write command for invoking the program and writing the program's output to the SSD.

D. Power and energy consumption

Tables XIV, XV, and XVI show the electric power and energy consumption, and GBS throughput per USD of the software, A100 GPU, and FPGA-based platforms, respectively.

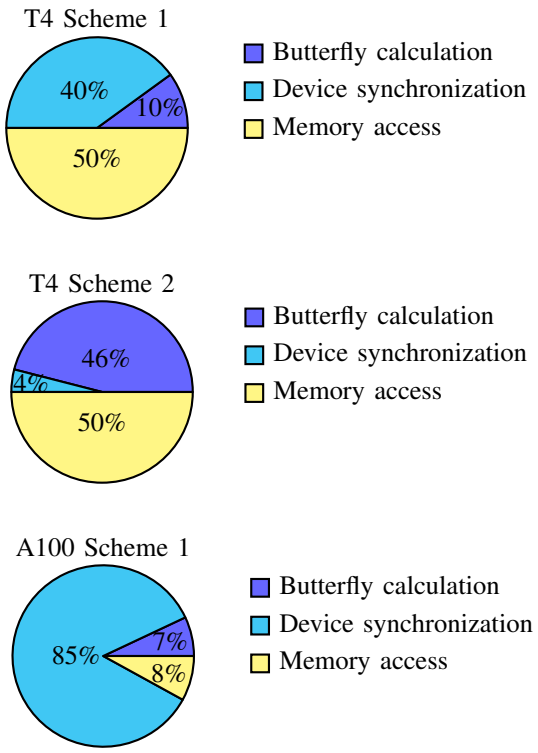


Fig. 13: GPU Processing Breakdown for NTT and INTT

TABLE XIII: Secure computation execution time on our platform for a program homomorphically computing xy. The total includes the computing time and the processing time of the NVMe Write command for invoking the program and writing the TLWE sample carrying the return value to the SSD.

Computing time	Total
500.39ms	502.28ms

The power consumption of the Ryzen 9 CPU was measured using the AMD μProf tool (https://www.amd.com/en/developer/uprof.html). The power consumption of the Apple M1 CPU was measured using the Mx Power Gadget tool (https://www.seense.com/menubarstats/mxpg/). The power consumption of A100 GPU was measured using the `nvidia-smi` command. Because the virtual performance monitoring unit (vPMU) feature is disabled in the AWS hypervisor, power data for CPU hositng T4 is unavailable. The power consumption of our FPGA accelerator board is measured using a Tektronics A622 current probe and PicoScope 3206A oscilloscope with the current probe attached to the 12V PCIe power connector of the FPGA board. The probe measures current every 500ms and provides 10mV outputs for each Ampere. For example, if the probe outputs 20mV, the measured current is 20(mv)/10(mV/A)=2(A), and hence the measured power is 2(A)$\times$12(V)=24(W). Energy consumption was calculated using the power consumption and GBS execution time, as described in Section VI-B. The GBS throughput per USD is $3600 \cdot 1000/(CE)$, where C is the electricity price in USD/kWh, and E is the energy consumption per GBS in joules. We use $C = 0.086$ which is the average price for industrial electricity consumers in the United States for 2023 (https://www.statista.com/statistics/).

Table XVII shows the breakdown of the power consumption on the FPGA chip (XCVU47P) of our accelerator estimated using the Xilinx Vivado tool (https://www.xilinx.com/products/design-tools/vivado.html), with a default toggle rate of 12.5%, where the toggle rate reflects how often the outputs of the gates change per clock cycle on average. The estimated total on-chip power in Table XVII(a) is less than the measured power of the FPGA board during GBS in Table XIV because Table XVII(a) is calculated based on the reference clock frequency of 150MHz. We use the dynamic clock reconfiguration feature of the phase-locked loop (PLL) to increase the operating frequency to 200MHz.

Our FPGA-based platform may consume more power when idle or processing GBS than software and GPU-based platforms. However, during GBS execution, our platform consumes less energy than other platforms. Our platform uses 12 times less energy than Ryzen 9 and seven times less energy than Apple M1. It also uses 4.95 times less energy than A100-based platform. Our platform offers higher GBS throughput per watt and GBS throughput per USD than any other platform. As shown in Table XVII, the HBM consumes more than 50% of the dynamic power of the FPGA chip. Because NTT and INTT are memory-bandwidth-bound workloads, our accelerator is optimally designed to utilize the most needed power. In future work, we plan to explore the implementation of a power-saving scheme to reduce the energy consumption during idle states.

TABLE XIV: Comparison of power consumption (Upper: During idle state, Lower: During GBS)

	Ryzen 9	Apple M1
CPU-based Platform	18.61W	0.082W
	63.08W	4.64W
	CPU+GPU (GPU-only)	
GPU(A100)-based Platform	52.32W (33.61W)	
	131.80W (72.42W)	
	CPU+FPGA (FPGA-only)	
FPGA-based Platform	60.84W (42.23W)	
	77.79W (59.18W)	

TABLE XV: Comparison on energy consumption per GBS

	Ryzen 9	Apple M1
CPU-based Platform	250.42J	142.91J
	CPU+GPU (GPU-only)	
GPU(A100)-based Platform	96.21J (52.87J)	
	CPU+FPGA (FPGA-only)	
FPGA-based Platform	19.44J (14.80J)	

TABLE XVI: Comparison on GBS throughput per USD

CPU-based Platform	Ryzen 9	Apple M1
	167161	29291
GPU(A100)-based Platform	435095	
FPGA-based Platform	2153316	

TABLE XVII: Breakdown of the power consumption on FPGA chip

(a) On-Chip Power

Element	Power	%
Hard IP	0.59W	1%
Dynamic	45.63W	88%
Static	5.89W	11%
Total	52.11W	100%

(b) Dynamic Power

Element	Power	%
Clocks	3.71W	8%
Signals	5.75W	13%
Logic	4.47W	10%
BRAM	2.07W	5%
URAM	0.24W	1%
DSP	0.73W	2%
I/O	0.03W	$< 1\%$
HBM	25.27W	52%
Other	3.36W	9%
Total	45.63W	100%

E. Application performance

Finally, we demonstrate the performance of a machine learning application running on our secure computing platform.

Auto-MPG dataset (https://www.tensorflow.org/tutorials/keras/regression?hl=en) was used for a simple linear regression application and choose the column "Horsepower" and "MPG" as the explanatory variable and the objective variable, respectively. The total number of data samples in the dataset is 392. The dataset was pre-processed and then a randomly chosen subset of D samples of the pre-processed dataset was input to the linear regression application. In the pre-processing stage, the dataset was normalized by dividing each data sample by the scaled_size parameter, where scaled_size is a positive integer with an upper bound of $(b-a)/2^w$ for a given precision of w bits. Table XVIII shows the upper bound of scaled_size versus w for the simple linear regression model given by $y = Ax + B$.

We iteratively run *ciphertext learning*, or the simple linear regression learning model calculated homomorphically using our secure computing platform for pre-processed D data samples that are encoded in w bits in cleartext and encrypted as TLWE samples with increasing D, where scaled_size for w is chosen from Table XVIII.

The iteration started from $D = 2$ until an overflow was detected. We consider that an overflow occurs when the difference between the decrypted result of ciphertext learning and the result of *cleartext learning*, or the cleartext calculation of the simple linear regression model for the same pre-processed D data samples exceeds a certain threshold.

Tables XIX and XX present a comparison of the model training results between cleartext learning and ciphertext learning for $w = 14$ and $D = 15$, where the relative error represents the absolute value of the difference between the cleartext value and the value of the decrypted and decoded ciphertext. Table XIX shows that the relative error of ciphertext learning compared with cleartext learning was less than 0.5% for $w = 14$ and $D = 15$.

Table XXI shows D_{max}, or the maximum number of D, so an overflow does not occur in cleartext learning. We observed an overflow even at $D = 2$ for $w = 10$. Therefore, using SDP with $N = 2^{10}$ for ciphertext learning on this dataset is impossible. Additionally, to perform ciphertext learning over the entire dataset, at least 18-bit precision is required.

Table XXII presents a comparison of the model training results between cleartext learning and ciphertext learning for $(w, D_{max}) \in \{(11, 2),\ (12, 4),\ (13, 7),\ (14, 15)\}$. For both cleartext learning and cipertext learning the relative error decreases with a larger w. Finally, Table XXIII shows the total execution time (t_l) of ciphertext learning for (w, D_{max}) mentioned above and the total bootstrapping time (t_b) during ciphertext learning, where t_b is calculated as the product of the total number of `bootstrap` instructions executed and the mean `bootstrap` instruction execution time shown in Table XI.

Tables XXI and XXII show that as w increases, more data samples can be analyzed with greater precision. Moreover, Table XXIII shows that bootstrapping accounted for more than 80% of the ciphertext learning time and the percentage increased with D_{max} determined by precision w. These tables indicate that bootstrapping significantly dominates ciphertext learning time and further improvements in both precision and bootstrapping speed are expected.

Note that the purpose of showing the application performance is to show that the proposed platform with providing 14-bit plaintext accuracy works for a small machine learing application. The application performance also shows that 14-bit plaintext accuracy is still insufficient for applying our platform to wider range of applications, and thus extending our platform to support 16-bit or more plaintext accuracy is needed for real-world deployment.

TABLE XVIII: Upper bound of scaled_size versus required precision (w)

w	Upper Bound of scaled_size
10	51
11	102
12	204
13	409
14	819
15	1638
16	3276
17	6553
18	13107

TABLE XIX: Comparison on the model's training results for $w = 14$ and $D = 15$

model	Cleartext Learning	Ciphertext Learning	Relative Error
A	-0.05481	-0.05005	0.0048
B	0.03086	0.02930	0.0016

TABLE XX: Comparison on the model's training progress for $w = 14$ and $D = 15$

Term	Cleartext Learning	Ciphertext Learning	Relative Error
$\sum x_i y_i$	0.05923	0.05981	0.0006
$\sum x_i$	3.15000	3.15308	0.0031
$\sum y_i$	0.29024	0.29053	0.0003
$\sum x_i \sum y_i$	0.91427	0.91553	0.0013
$\sum x_i \sum y_i / D$	0.06095	0.06104	0.0001
$\sum x_i y_i - \sum x_i \sum y_i / D$	-0.00172	-0.00122	0.0005
$\sum x_i^2$	0.69293	0.68604	0.0069
$(\sum x_i)^2$	9.92250	9.94141	0.0189
$(\sum x_i)^2 / D$	0.66150	0.66162	0.0001
$\sum x_i^2 - (\sum x_i)^2 / D$	0.03143	0.02441	0.0070
$(\sum x_i^2 - (\sum x_i)^2 / D)^{-1}$	0.31812	0.40894	0.0908
A	-0.05481	-0.05005	0.0048
$\sum (y - ax)$	0.46291	0.44922	0.0137
B	0.03086	0.02930	0.0016

TABLE XXI: Maximum number of data samples without overflow (D_{max}) versus required precision (w)

w	D_{max}
11	2
12	4
13	7
14	15
15	37
16	81
17	174
18	392

TABLE XXII: Comparison on the model's training results versus (w, D_{max})

(w, D_{max})	Model	Cleartext Learning	Ciphertext Learning	Relative Error
(11,2)	A	-0.08572	-0.10620	0.0205
	B	0.28572	0.31250	0.0268
(12,4)	A	-0.07477	-0.09033	0.0156
	B	0.13694	0.14771	0.0108
(13,7)	A	-0.04174	-0.03540	0.0063
	B	0.05628	0.05371	0.0026
(14,15)	A	-0.05481	-0.05005	0.0048
	B	0.03086	0.02930	0.0016

TABLE XXIII: Bootstrap and ciphertext learning time (t_b and t_ℓ) versus (w, D_{max})

(w, D_{max})	t_ℓ	t_b	t_b/t_ℓ
(11,2)	6.151s	4.999s	81.27%
(12,4)	9.394s	7.999s	85.15%
(13,7)	14.23s	12.50s	87.84%
(14,15)	27.26s	24.50s	89.88%

VII. Conclusion

We successfully developed and implemented an exceptionally secure computing platform that utilizes NVMe technology, an FPGA-based TFHE accelerator, an SSD, and middleware on the host side. Our platform stands out from the crowd as it supports a set of secure computing instructions that enable the evaluation of any 14-bit to 14-bit function using TFHE and virtual registers. Our performance evaluations demonstrated that our platform outperformed the CPU-based and GPU-based platforms by 15 to 120 times and 2.5 to 3 times, respectively, in gate bootstrapping execution time. Furthermore, our platform has lower electric energy consumption during the gate bootstrapping execution time, outperforming the CPU-based one by 7 to 12 times and a GPU-based one by 4.95 times, respectively. We also demonstrated the application performance of a simple linear regression model running on our platform.

Moving forward, we are confident of our ability to develop a compiler and assembler to convert applications into instructions that can be executed on our secure computing platform by using our middleware API. We also plan to implement and evaluate the extended architecture to include clusters of FPGA-based accelerators and NVMe SSDs interconnected through a high-speed network by using Kubernetes and NVMe-oF to increase the scalability and robustness of our platform. Finally, we can confidently extend the platform's capabilities to support secure computing of 16-bit to 16-bit or higher precision functions.

Appendix

This section provides the equations and algorithms that lead to the NTT and INTT constructions described in Section IV-E.

A. Equations

We derived equations used as the basis for NTT and INTT butterflies, as shown in Figures 5, 6, and 7. Let $\omega_N = \omega$ and $\psi_N = \psi$. The following equations were used: $\omega_N^2 = \omega_{N/2}$, $\psi_N^2 = \psi_{N/2}$, $\omega_N^N = 1$, and $\omega_N^{N/2} = -1$.

1) Equations for NTT butterfly: We denote $\mathrm{NTT}_i^{(N)}(f)$ and $\mathrm{DFT}_i^{(N)}(f)$ as the ith output of N-point NTT and Discrete Fourier Transform (DFT) for the time-domain input vector $f = (f_0, \ldots, f_{N-1}) \in (\mathbb{Z}/p\mathbb{Z})^N$, respectively.

An NTT butterfly is composed using a Cooley-Tukey (CT) butterfly by partitioning the input vector F into two subvectors F_{ev} and F_{od} where F_{ev} contains even elements from the starting index 0 of F and F_{od} contains odd elements from the starting index 0 of F.

Let $\Psi_N = (1, \psi_N, \ldots, \psi_N^{N-1})$. Let

$$A_i = \begin{cases} \mathrm{NTT}_i^{(N/2)}(F_{\mathrm{ev}}) & \text{if} \quad 0 \le i < N/2 \\ \mathrm{NTT}_{i-N/2}^{(N/2)}(F_{\mathrm{ev}}) & \text{o.w.} \end{cases},$$

$$B_i = \begin{cases} \mathrm{NTT}_i^{(N/2)}(F_{\mathrm{od}}) & \text{if} \quad 0 \le i < N/2 \\ \mathrm{NTT}_{i-N/2}^{(N/2)}(F_{\mathrm{od}}) & \text{o.w.} \end{cases}.$$

Then, equations used for NTT butterfly are derived as follows:

$$\mathrm{NTT}_i^{(N)}(f) = \mathrm{DFT}_i^{(N)}(\psi_N \odot f) = \sum_{j=0}^{N-1} \psi_N^j f_j \omega_N^{ij}.$$

where $\odot$ denotes the Hadamard product.

(i) For $0 \le i < N/2$,

$$\begin{aligned}
\mathrm{NTT}_i^{(N)}(f) &= \sum_{j=0}^{N/2-1} (f_{2j}\psi_N^{2j})\omega_N^{i(2j)} \\
&\quad + \sum_{j=0}^{N/2-1} f_{2j+1}\psi_N^{2j+1}\omega_N^{i(2j+1)} \\
&= \sum_{j=0}^{N/2-1} f_{2j}\psi_{N/2}^{j}\omega_{N/2}^{ij} \\
&\quad + \psi_N\omega_N^{i} \sum_{j=0}^{N/2-1} f_{2j+1}\psi_{N/2}^{j}\omega_{N/2}^{ij} \\
&= \mathrm{NTT}_i^{(N/2)}(f_{\mathrm{ev}}) + \psi_N\omega_N^{i}\mathrm{NTT}_i^{(N/2)}(f_{\mathrm{od}}) \\
&= A_i + \psi_N\omega_N^{i}B_i.
\end{aligned}$$

(ii) For $N/2 \le i < N$,

$$\begin{aligned}
\mathrm{NTT}_i^{(N)}(f) &= \sum_{j=0}^{N/2-1} (f_{2j}\psi_N^{2j})\omega_N^{i(2j)} \\
&\quad + \sum_{j=0}^{N/2-1} f_{2j+1}\psi_N^{(2j+1)}\omega_N^{i(2j+1)} \\
&= \omega_N^{N/2\cdot 2j} \sum_{j=0}^{N/2-1} (f_{2j}\psi_N^{2j})\omega_N^{(i-N/2)(2j)} \\
&\quad + \omega_N^{N/2\cdot(2j+1)} \\
&\quad \cdot \sum_{j=0}^{N/2-1} (f_{2j+1}\psi_N^{2j+1})\omega_N^{(i-N/2)(2j+1)} \\
&= \sum_{j=0}^{N/2-1} (f_{2j}\psi_{N/2}^{j})\omega_{N/2}^{(i-N/2)j} \\
&\quad - \psi_N\omega_N^{(i-N/2)} \\
&\quad \sum_{j=0}^{N/2-1} (f_{2j+1}\psi_{N/2}^{j})\omega_{N/2}^{(i-N/2)j} \\
&= \mathrm{NTT}_{i-N/2}^{(N/2)}(f_{\mathrm{ev}}) \\
&\quad - \psi_N^{-1}\omega_N^{-(i-N/2)}\mathrm{NTT}_{i-N/2}^{(N/2)}(f_{\mathrm{od}}) \\
&= A_i - \psi_N\omega_N^{(i-N/2)}B_i.
\end{aligned}$$

We note that $A_{i-N/2} = A_i$ and $B_{i-N/2} = B_i$ for $N/2 \le i < N$ by $\omega_{N/2}^{N/2} = 1$. So,

$$\begin{aligned}
\mathrm{NTT}_i^{(N)}(f) &= A_i - \psi_N\omega_N^{(i-N/2)}B_i \\
&= A_{i-N/2} - \psi_N\omega_N^{(i-N/2)}B_{i-N/2}.
\end{aligned}$$

Each pair of A_i and B_i is the input of an NTT butterfly with a pair of $(A_i + \psi_N^{-1}\omega_N^{-i}B_i)$ and $(A_{i-N/2} - \psi_N^{-1}\omega_N^{-(i-N/2)}B_{i-N/2})$ as its outputs for $0 \le i < N/2$ and $N/2 \le i < N$, respectively.

2) Equations for INTT butterfly: We denote $\mathrm{INTT}_i^{(N)}(F)$ and $\mathrm{uIDFT}_i^{(N)}(F)$ as the ith output of N-point, unnormalized Inverse NTT, and unnormalized Inverse DFT, respectively, for the frequency-domain input vector $F = (F_0, \ldots, F_{N-1}) \in (\mathbb{Z}/p\mathbb{Z})^N$, respectively.

Let $\Psi_N^{-1} = (1, \psi_N^{-1}, \ldots, \psi_N^{-(N-1)})$. Then, equations used for INTT butterfly are derived as follows:

$$\mathrm{INTT}_i^{(N)}(F) = \mathrm{uIDFT}_i^{(N)}(F) \odot \Psi_N^{-i} = \psi_N^{-i}\sum_{j=0}^{N-1}(F_j)\omega_N^{-ij},$$

An INTT butterfly is composed of a Gentleman-Sande (GS) butterfly whicn partitions the output vector $\mathrm{INTT}^{(N)}(F)$ into two subvectors, one containing even elements of $\mathrm{INTT}^{(N)}(F)$ and the other containing odd elements of $\mathrm{INTT}^{(N)}(F)$.

Let

$$\begin{aligned}
g &= (g_j)_{0\le j<N/2} = (F_j + f_{j+N/2})_{0\le j<N/2}, \\
h &= (h_j)_{0\le j<N/2} = ((F_j - F_{j+N/2})\psi_N^{-1}\omega_N^{-j})_{0\le j<N/2}.
\end{aligned}$$

(i) For $i = 2r$ such that $0 \le r < N/2$,

$$\begin{aligned}
\mathrm{INTT}_{2r}^{(N)}(F) &= \psi_N^{-2r}\left(\sum_{j=0}^{N/2-1} F_j\omega_N^{-2rj} \right. \\
&\quad \left. + \sum_{j=0}^{N/2-1} F_{j+N/2}\omega_N^{-2r(j+N/2)}\right) \\
&= \psi_{N/2}^{-r}\left(\sum_{j=0}^{N/2-1} F_j\omega_{N/2}^{-rj} \right. \\
&\quad \left. + \sum_{j=0}^{N/2-1} F_{j+N/2}\omega_{N/2}^{-rj}\right) \\
&= \psi_{N/2}^{-r}\sum_{j=0}^{N/2-1}\left(F_j + F_{j+N/2}\right)\omega_{N/2}^{-rj} \\
&= \mathrm{INTT}_r^{(N/2)}(g).
\end{aligned}$$

(ii) For $i = 2r+1$ such that $0 \le r < N/2$,

$$\begin{aligned}
\mathrm{INTT}_{2r+1}^{(N)}(F) &= \psi_N^{-(2r+1)}\left(\sum_{j=0}^{N/2-1} F_j\omega_N^{-(2r+1)j} \right. \\
&\quad \left. + \sum_{j=0}^{N/2-1} F_{j+N/2}\omega_N^{-(2r+1)(j+N/2)}\right) \\
&= \psi_{N/2}^{-r}\psi_N\left(\sum_{j=0}^{N/2-1} F_j\omega_N^{-j}\omega_{N/2}^{-rj} \right. \\
&\quad \left. - \sum_{j=0}^{N/2-1} F_{j+N/2}\omega_N^{-j}\omega_{N/2}^{-rj}\right) \\
&= \psi_{N/2}^{-r} \\
&\quad \cdot \sum_{j=0}^{N/2-1}\left((F_j - F_{j+N/2})\psi_N^{-1}\omega_N^{-j}\right)\omega_{N/2}^{-rj} \\
&= \mathrm{INTT}_r^{(N/2)}(h).
\end{aligned}$$

Each pair of F_j and $F_{j+N/2}$ $(0 \leq j < N/2)$ is the input of an INTT butterfly with the pair of $(F_j + F_{j+N/2})$ and $(F_j - F_{j+N/2})\psi_N^{-1}\omega_N^{-j}$ as its outputs.

B. Algorithms

Algorithm 1 computes the NTT of a vector of length N. Algorithm 1 computes in-place. The outputs from Algorithm 1 remain in bit-reversed order. We remind the reader that our goal is to compute the convolution of polynomials represented as vectors. Readers can refer to Equation (1) from Section IV-E for the definition of the convolution. If the bits of the output from Algorithm 1 are in canonical order, an additional cost is incurred when computing the convolutions. Algorithms 1 and 2 do not output normalized transforms to save time when calculating convolutions.

Algorithm 1 Number Theoretic Transform based on Cooley-Tukey

Input: N, length of transform (N is a power of 2.)
Input: $\Phi = (\psi^{N/2^{r+1}}\omega^{j2^r})_{0 \leq r < \log_2 N, 0 \leq j < 2^r}$, two-dimensional list of pre-computed twiddles with the second dimension listed in bit-reversed order.
Input: **a**, data vector of length N in bit-canonical order
Output: NTT(a) in bit-reversed order

for $0 \leq r < \log_2 N$ **do** // NTT Row number minus 1
  $m \leftarrow 2^r$
  $k \leftarrow N/2^{r+1}$
  for $0 \leq i < m$ **do**
    $j_1 \leftarrow 2ik$
    $j_2 \leftarrow j_1 + k$ // Interval length is $k-1$
    for $j_1 \leq j < j_2$ **do** // Butterfly operations here
      $t \leftarrow a_j$
      $u \leftarrow a_{j+k}\Phi_{r,i}$
      $a_j \leftarrow t + u$
      $a_{j+k} \leftarrow t - u$
    end for
  end for
end for

Algorithm 2 Inverse Number Theoretic Transform based on Gentleman-Sande

Input: N, length of transform (N is a power of 2.)
Input: $\Phi^* = (\psi^{-N/2^{r+1}}\omega^{-j2^r})_{0 \leq r < \log_2 N, 0 \leq j < 2^r}$, two-dimensional list of precomputed twiddles with the second dimension listed in bit-reversed order.
Input: **a**, data vector of length N in bit-reversed order
Output: INTT(a) in bit-canonical order

for $0 \leq r < \log_2 N$ **do** // INTT Row number minus 1
  $m \leftarrow N/2^{r+1}$
  $k \leftarrow 2^r$
  for $0 \leq i < m$ **do**
    $j_1 \leftarrow 2ik$
    $j_2 \leftarrow j_1 + k$ // Interval length is $k-1$
    for $j_1 \leq j < j_2$ **do** // Butterfly operations here
      $t \leftarrow a_j$
      $u \leftarrow a_{j+k}$
      $a_j \leftarrow t + u$
      $a_{j+k} \leftarrow (t - u)\Phi^*_{\log_2 N - r - 1, i}$
    end for
  end for
end for

Algorithms 1 and 2 are similar to Algorithms 7 and 8 in Ref. [39]. However, we note that Algorithms 7 and 8 from [39] contain errors corrected here.

ACKNOWLEDGMENT

We thank all the members of the joint project on secure computing between KIOXIA Corporation and EAGLYS Inc. for their technical inputs.

REFERENCES

[1] V. Group, *Cuda-accelerated fully homomorphic encryption library*, 2019.
[2] Agrawal, R., de Castro, L., Yang, G., Juvekar, C., Yazicigil, R., Chandrakasan, A., Vaikuntanathan, V., and Joshi, A., *FAB: An FPGA-based accelerator for bootstrappable fully homomorphic encryption*, 2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA), 2023, pp. 882–895.
[3] Aikata, A., Mert, C. A., Kwon, S., Deryabin, M., and Roy, S. S., *Reed: Chiplet-based scalable hardware accelerator for fully homomorphic encryption*, 2023.
[4] Albrecht, M., Chase, M., Chen, H., and Ding, J., Goldwasser, S., Gorbunov, S., Halevi, S., Hoffstein, J., Laine, K., Lauter, K., Lokam, S., Micciancio, D., Moody. D, Morrison, T, Sahai, A, and Vaikuntanathan, V., *Homomorphic encryption security standard*, HomomorphicEncryption.org, 2018.
[5] Aydin, F. and Aysu, A., *Leaking secrets in homomorphic encryption with side-channel attacks*, 2023.
[6] Al Badawi, A., Veeravalli, B., Lin, J., Xiao, N., Kazuaki, M., and Mi, A. K. M., *Multi-GPU design and performance evaluation of homomorphic encryption on GPU clusters*, IEEE Transactions on Parallel and Distributed Systems **32** (2020), no. 2, 379–391.
[7] Al Badawi, A., Veeravalli, B., Mun, C. F., and Aung, K. M. M., *High-performance FV somewhat homomorphic encryption on GPUs: An implementation using CUDA*, IACR Transactions on Cryptographic Hardware and Embedded Systems (2018), 70–95.
[8] Beirendonck M. V., D'Anvers, J., and Verbauwhede, I., *FPT: A fixed-point accelerator for torus fully homomorphic encryption*, Cryptology ePrint Archive **2022** (2022), 1635.
[9] Bi, L., Lu, X., Luo, J and Wang, K., *Hybrid dual and meet-LWE attack*, 2022.
[10] Bossuat,J.-P., Mouchet, C., Troncoso-Pastoriza, J., and Hubaux, J.-P., *Efficient bootstrapping for approximate homomorphic encryption with non-sparse keys*, Cryptology ePrint Archive **2020** (2020), 1203.
[11] Brakerski, Z., *Fully homomorphic encryption without modulus switching from classical gapsvp*, Advances in cryptology-crypto 2012, 2012, pp. 868–886.

[12] Brakerski, Z., Gentry, C., and Vaikuntanathan, V., *(leveled) fully homomorphic encryption without bootstrapping*, ACM Transactions on Computation Theory (TOCT) **6** (2014), no. 3, 1–36.

[13] de Castro, L., Agrawal, R., Yazicigil, R., Chandrakasan, A., Vaikuntanathan, V., Juvekar, C., and Joshi, A., *Does fully homomorphic encryption need compute acceleration?*, arXiv preprint arXiv:2112.06396 (2021).

[14] Chaturvedi, B., Chakraborty, A., Chatterjee, A. and Mukhopadhyay, D., *A practical full key recovery attack on TFHE and FHEW by inducing decryption errors*, 2022.

[15] _____, *Model stealing attacks on FHE-based privacy-preserving machine learning through adversarial examples*, 2023.

[16] Chaturvedi, B., Chakraborty. A, Chatterjee, A. and Mukhopadhyay, D., *"ask and thou shall receive": Reaction-based full key recovery attacks on FHE*, Computer security – esorics 2024: 29th european symposium on research in computer security, bydgoszcz, poland, september 16–20, 2024, proceedings, part iv, 2024, pp. 457–477.

[17] Cheon, J. H., Choe, H., and Passelègue, A., Stehlé, D. and Suvanto, E., *Attacks against the ind-cpad security of exact fhe schemes*, Proceedings of the 2024 on acm sigsac conference on computer and communications security, 2024, pp. 2505–2519.

[18] Cheon, J. H., Kim, A., Kim, M., and Song, Y., *Homomorphic encryption for arithmetic of approximate numbers*, Advances in Cryptology – ASIACRYPT 2017, 2017, pp. 409–437.

[19] Chillotti, I., Gama, N., and Goubin, L., *Attacking FHE-based applications by software fault injections*, 2016.

[20] Chillotti, I, Gama N., Georgieva M., and Izabachène, M., *TFHE: Fast fully homomorphic encryption over the torus*, Journal of Cryptology, 2020, pp. 34–91.

[21] Chillotti, I., Ligier, D., Orfila, J.-B., and Tap, S., *Improved programmable bootstrapping with larger precision and efficient arithmetic circuits for tfhe*, IACR Cryptology ePrint Archive **2021** (2021), 315.

[22] Chunsheng, G., *Attack on fully homomorphic encryption over the integers*, 2012.

[23] Cousins, D. B., Rohloff, K., and Sumorok, D., *Designing an FPGA-accelerated homomorphic encryption coprocessor*, IEEE Transactions on Emerging Topics in Computing **5** (2017), no. 2, 193–206.

[24] Dai, W., and Sunar, B., *cuHE: A homomorphic encryption accelerator library*, 2015.

[25] Fan, J. and Vercauteren, F., *Somewhat practical fully homomorphic encryption*, IACR Cryptol. ePrint Arch. **2022** (2012), 144.

[26] Fauzi, P., Hovd, M. N., and Raddum, H., *On the IND-CCA1 security of FHE schemes*, 2021.

[27] Gener, S., Newton, P., Tan, D., Richelson, S., Lemieux, G., and Brisk, P., *An FPGA-based programmable vector engine for fast fully homomorphic encryption over the torus*, SPSL: Secure and private systems for machine learning (ISCA workshop), 2021.

[28] Gentry, C., *Fully homomorphic encryption using ideal lattices*, Proceedings of the forty-first annual acm symposium on theory of computing, 2009, pp. 169–178.

[29] Jiang, L., Lou, Q., and Joshi, N., *MATCHA: A fast and energy-efficient accelerator for fully homomorphic encryption over the Torus*, Proceedings of the 59th ACM/IEEE design automation conference, 2022, pp. 235–240.

[30] Latibari, B. S., Gubbi, K. I., Homayoun, H., and Sasan, A., *A survey on FHE acceleration*, 2023 IEEE 16th Dallas circuits and systems conference (DCAS), 2023, pp. 1–6.

[31] Lee, J. H., Zhang, H., Lagrange, V., Krishnamoorthy, P., Zhao, X., and Ki, Y. S., *SmartSSD: FPGA Accelerated Near-Storage Data Analytics on SSD*, IEEE Computer Architecture Letters **19** (2020), no. 2, 110–113.

[32] Lee, Y., Chung, J., and Rhu, M., *SmartSAGE: Training large-scale graph neural networks using in-storage processing architectures*, arXiv preprint arXiv:2205.04711 (2022).

[33] Manulis. M., and Nguyen. J., *Fully homomorphic encryption beyond IND-CCA1 security: Integrity through verifiability*, 2024.

[34] Marcolla, C., Sucasas, V., Manzano, M., Bassoli, R., Fitzek, F. H. P, and Aaraj, N., *Survey on fully homomorphic encryption, theory, and applications*, Cryptology ePrint Archive **2022** (2022), 1602.

[35] Matsuoka, K., *TFHEpp: pure C++ implementation of TFHE cryptosystem*, 2020.

[36] Matsuoka, K., Banno, R., Matsumoto, N., Sato, T., and Bian, S., *Virtual secure platform: A five-stage pipeline processor over TFHE*, 30th USENIX Security Symposium (USENIX Security 21), 2021, pp. 4007–4024.

[37] Nam, K., Oh, H., Moon, H., and Paek, Y., *Accelerating N-bit Operations over TFHE on Commodity CPU-FPGA*, 2022 IEEE/ACM international conference on computer aided design (ICCAD), 2022, pp. 1–9.

[38] Özerk, Ö., Elgezen, C., Mert, A. C., Öztürk, E., and Savaş, E., *Efficient number theoretic transform implementation on GPU for homomorphic encryption*, The Journal of Supercomputing **78** (2022), no. 2, 2840–2872.

[39] Pöppelmann, T., Oder, T., and Güneysu, T., *High-performance ideal lattice-based cryptography on 8-bit atxmega microcontrollers*, Progress in cryptology – latincrypt 2015, 2015, pp. 346–365.

[40] Riazi, M. S., Laine, K., Pelton, B., and Dai, W., *HEAX: An architecture for computing on encrypted data*, Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems, 2020, pp. 1295–1309.

[41] Rohloff, K., *The HomomorphicEncryption.org Community and the Applied Fully Homomorphic Encryption Standardization Efforts*, 2023.

[42] Roy, S. S., Turan, F., Jarvinen, K., Vercauteren, F., and Verbauwhede, I., *FPGA-based high-performance parallel architecture for homomorphic computing on*

encrypted data, 2019 IEEE International Symposium on High Performance Computer Architecture (HPCA), 2019, pp. 387–398.

[43] Samardzic, N., Feldmann, A., Krastev, A., Devadas, S., Dreslinski, R., Peikert, C., and Sanchez, D., *F1: A fast and programmable accelerator for fully homomorphic encryption*, Micro-54: 54th annual ieee/acm international symposium on microarchitecture, 2021, pp. 238–252.

[44] Son. Y. and Cheon., J. H., *Revisiting the hybrid attack on sparse and ternary secret LWE*, 2019.

[45] Turan, F., Roy, S. S., and Verbauwhede, I., *HEAWS: An Accelerator for Homomorphic Encryption on the Amazon AWS FPGA*, IEEE Transactions on Computers **69** (2020), no. 8, 1185–1196.

[46] Wenger, E., Saxena, E., Malhou, M., Thieu, E. and Lauter, K., *Benchmarking attacks on learning with errors*, 2024.

[47] Yang, Y., Lu, H., and Li, X., *Poseidon-NDP: Practical Fully Homomorphic Encryption Accelerator Based on Near Data Processing Architecture*, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (2023), 1–1.

[48] Ye, T., Kannan, R., and Prasanna, V. K., *FPGA acceleration of fully homomorphic encryption over the torus*, 2022 IEEE High Performance Extreme Computing Conference (HPEC), 2022, pp. 1–7.

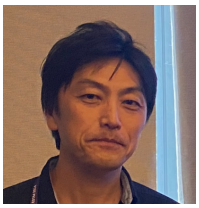
[49] Zhai, Y., Ibrahim, M., Qiu, Y., Boemer, F., Chen, Z., Titov, A., and Lyashevsky, A., *Accelerating encrypted computing on Intel GPUs*, 2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS), 2022, pp. 705–716.

[50] Zhang, J., Cui, A. and Jin, Y., *Acceleration of the bootstrapping in tfhe by fpga*, IEEE Transactions on Emerging Topics in Computing (2024), 1–16.

[51] Yufei Xing and Shuguo Li, *A compact hardware implementation of cca-secure key exchange mechanism crystals-kyber on fpga*, IACR Transactions on Cryptographic Hardware and Embedded Systems **2021** (2021Feb.), no. 2, 328–356.



Yoshihiro Ohba (Fellow, IEEE) Kioxia Corporation. He received B.E., M.E. and Ph.D. degrees in Information and Computer Sciences from Osaka University in 1989, 1991 and 1994, respectively. He joined Toshiba Corporation in 1991. Since then he has been active in standardizing security and mobility protocols for more than 20 years. He served as Chair of IEEE 802.21a and IEEE 802.21d, and also served as Vice Chair and Secretary of ZigBee Alliance Neighborhood Area Network (NAN) WG also known as JupiterMesh. He is one of the main contributors to RFC 5191 (PANA - Protocol for carrying Authentication for Network Access), which is used as the standard network access authentication protocol for B-Route and Home Area Network profiles of Wi-SUN Alliance and ZigBee IP profile of ZigBee Alliance, and has been implemented in all smart meters in Japan supporting B-Route communication with 920MHz band. He received IEEE Region 1 Technology Innovation Award 2008 for Innovative and Exemplary Contributions to the Field of Internet Mobility and Security related Research and Standards. He is currently in charge of developing Fully Homomorphic Encryption hardware accelerator and middleware in Kioxia. His current interest is in secure computing.



Tomoya Sanuki (Member, IEEE) received the M.Sc. degree in physics from Osaka University, Japan. In 1999, he joined Toshiba Corporation, where he was engaged in the research and development of advanced CMOS Logic, embedded DRAM and system LSI at Device Technology Laboratory. In 2015, he became a researcher and device engineer at Institute of Memory Technology R&D, where he was involved in the development of magnetic RAM, 3D flash memory and future device technology. Since 2024, he has been a senior researcher and strategist at Frontier Technology R&D Institute in KIOXIA. He has 95+ U.S. patents issued or pending in the area of new semiconductor devices and authored or co-authored 26+ publications in IEDM, VLSI, EDTM, JXCDC, IMW and J-EDS. He also had been served as technical committees at various conferences including EDTM, VLSI-TSA and IMW.



Claude Gravel has been an Assistant Computer Science Professor at Toronto Metropolitan University (formerly Ryerson University) since January 2024. From 2018 to 2023, he worked as a researcher at Eaglys Inc. and the National Institute of Informatics, Tokyo. From 2015 to 2018, he was a contractor at the Tutte Institute of Mathematics and Computing in the Government of Canada. He completed a Ph.D. in quantum computing from 2012 to 2015 at the University of Montreal and a bachelor's degree in pure mathematics in 2002 at McGill University. Between 2002 and 2012, he held different positions in industry and government.



Kentaro Mihara has been a software engineering manager at Cellid Inc. since May 2024. He has been a Visiting Researcher at the Institute for Advanced Research, Waseda University, since April 2020. Previously, he served as a Research Engineer at EAGLYS Inc. from April 2019 to April 2024. He was also a Research Assistant at the Collider Accelerator Division of Brookhaven National Laboratory, USA, from 2016 to June 2018. He earned a Master's degree in Physics from Stony Brook University, New York, in 2018.



Asuka Wakasugi received Bachelor's and Master's degrees in mathematics and informatics from Chiba University in 2021 and 2023. He is now working for EAGLYS Inc,. His current research work is concerned with post-quantum cryptography and secret computation.



Kenta Adachi has been a Researcher and Engineer at Eaglys Inc. since March 2024. He received a B.S. in Computer Science from University of California, Irvine in 2024.