

Hypnos: A Hardware–Software Co-Design Framework for Memory-Efficient Homomorphic Processing

Haoxuan Wang[✉], Yinghao Yang[✉], Shangjie Pan[✉], Hang Lu[✉], and Xiaowei Li[✉], *Senior Member, IEEE*

Abstract—Fully homomorphic encryption (FHE) introduces a novel paradigm in privacy-preserving computation, but operating on encrypted data imposes significant challenges, elevating data transmission, memory access demands, and programming complexity. Consequently, developing an efficient and usable system becomes vital. Conventional FHE accelerators often prioritize computational performance, typically assuming abundant encrypted data resides in accelerator memory. However, this perspective frequently overlooks the inefficiencies of the peripheral component interconnect express (PCIe) bus and main memory, alongside the complexities of optimizing for dedicated hardware in real-world deployments. This article proposes Hypnos, a hardware–software codesign framework for memory-efficient homomorphic processing. The Hypnos framework pairs a novel processing unit architecture with a dedicated Hypnos Compiler. In architecture, the heterogeneous processing unit based on homomorphic encryption uses a paging memory management system to reduce memory fragmentation and optimize the PCIe traffic. In software, the Hypnos compiler is able to automatically translate high-level FHE schemes into optimized hardware commands. It schedules the data placement in cooperation with our memory management system to effectively harness the performance of hardware. This codesign of hardware and compiler not only reduces memory access and execution time but also lowers the complexity of deploying FHE applications. Finally, We implement Hypnos on the QianKun field-programmable gate array (FPGA) Card and highlight the following results: 1) outperforms state-of-the-art (SOTA) application-specific integrated circuit (ASIC) and FPGA solutions in data-intensive applications by up to $2.75\times$ and $4.72\times$; 2) the communication overhead is reduced by $4.85\times$ compared to traditional architectures; and 3) up to $30.1\times$ and $20.7\times$ energy efficiency improvement compared to ASIC-based ARK and FPGA-based Poseidon for ResNet-20, respectively.

Index Terms—Field-programmable gate array (FPGA), fully homomorphic encryption (FHE), memory efficient, peripheral component interconnect express (PCIe).

NOMENCLATURE

Parameter	Description
λ	Security parameter, determining the cryptographic strength.

Received 22 September 2025; revised 30 December 2025; accepted 25 January 2026. Date of publication 2 February 2026; date of current version 22 September 2026. This work was supported by the National Natural Science Foundation of China under Grant 62172387. This article was recommended by Associate Editor W. Zhang. (*Corresponding authors: Hang Lu; Yinghao Yang.*)

Haoxuan Wang and Shangjie Pan are with the Institute of Computing Technology, University of Chinese Academy of Sciences, Beijing 100190, China.

Yinghao Yang, Hang Lu, and Xiaowei Li are with the Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100190, China (e-mail: yinghao.Y@foxmail.com; luhang@ict.ac.cn).

Digital Object Identifier 10.1109/TCAD.2026.3660206

N	Degree of the polynomial ring, typically a power of 2.
Q	Ciphertext modulus, a large integer defining the ring R_Q .
P	Auxiliary modulus, used to reduce noise in keyswitching.
R	Ring of integer polynomials $\mathbb{Z}[X]/(X^N + 1)$.
R_Q	Polynomial ring modulo Q , $R_Q = \mathbb{Z}_Q[X]/(X^N + 1)$.
m	Plaintext, the original data to be encrypted.
ct	Ciphertext, the encrypted data.
evk	Evaluation key, used to support specific operations like relinearization and keyswitching.
L	Maximum multiplicative depth supported by the scheme.
Δ	Scaling factor.
d_{num}	Number of digits used for hybrid keyswitching.

I. INTRODUCTION

THE rapid development of cloud computing has introduced significant challenges to privacy protection and information security. In this context, the role of data privacy computation technologies has become increasingly prominent. Fully homomorphic encryption (FHE) is a critical technology in privacy-preserving computation, enabling secure computations on encrypted data [1], [2], [3], [4], [5], [6], [7].

The primary advantage of FHE is its ability to perform unlimited addition and multiplication operations without decrypting the data. However, this characteristic of FHE results in an explosive expansion in data volume and computational complexity, ranging from 10^4 to 10^5 times [8], [9], [10]. This inherent scale and complexity of FHE present a two-pronged challenge for its practical adoption. First, the computational and data load overwhelms general-purpose processors, clearly necessitating the development of specialized hardware accelerators. Second, and no less critically, even with such accelerators, the task of programming these specialized systems is extremely challenging. Managing the intricate and often dynamic FHE data structures and optimizing the overall application flow poses a significant deployment dilemma that can hinder the effective use of even the most powerful hardware.

In recent years, several FHE-specific accelerator architectures have been proposed for FHE acceleration based on field-programmable gate array (FPGA) [11], [12] and application-specific integrated circuit (ASIC) [13], [14], [15], [16], [17]. The advantage of using FPGAs lies in their lower cost and programmable nature. In contrast, ASICs deliver

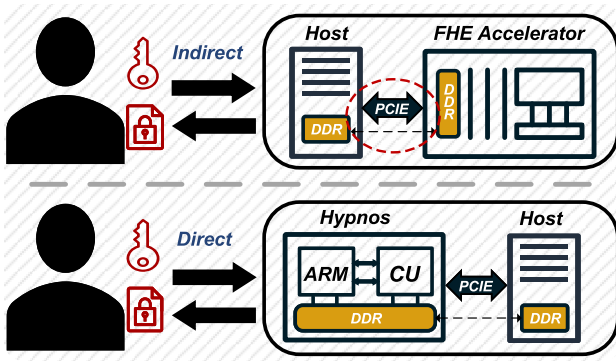


Fig. 1. Comparison with Traditional FHE Accelerators: In Hypnos, data processing is conducted directly by the CPU within the device, eliminating the dependency on the host CPU and the congested system peripheral component interconnect express (PCIe) bus.

higher performance and lower power consumption by allocating more substantial hardware resources to address the bottlenecks in FHE. For example, some designs employ up to 180–512 MB of on-chip scratchpad memory to mitigate the overhead caused by data transfers between off-chip memory and compute units (CUs) [14], [15], [16].

While these hardware architectures are crucial in advancing the application of FHE, they also exhibit significant limitations that cannot be overlooked.

Prior works [13], [14], [16] have typically assumed that substantial volumes of ciphertext and keys are always stored within the double data rate (DDR)/high bandwidth memory (HBM) of the accelerator. However, this assumption is overly idealistic, as handling intricate FHE applications often exceeds the capacity of on-chip memory. This indicates that communication between the host and the accelerator via the PCIe bus will be frequent. As depicted in the upper section of Fig. 1, communication between the user and the accelerator is managed by host memory, necessitating data to traverse the PCIe bus after initial storage in host memory. This bottleneck can potentially undermine the performance gains reported in earlier works.

Furthermore, current FHE accelerator designs, while focusing on computational enhancements and on-chip cache optimization [18], often overlook the efficient utilization of the accelerator’s larger off-chip memory (DDR/HBM). This neglect is particularly problematic for FHE schemes based on the residue number system (RNS), where ciphertext and key sizes can vary dynamically throughout the computation. Conventional memory management with fixed-size allocations for these dynamic data structures leads to significant internal fragmentation and the storage of redundant RNS components, drastically reducing memory utilization (MU) efficiency. This inefficiency further exacerbates the PCIe communication overhead. There is a substantial amount of currently unused information that needs to be swapped between the host and the accelerator.

As FHE accelerator architectures evolve, they often feature unique, highly specialized designs. Manually tailoring FHE algorithms to these specific architectures, optimizing data movement, and managing resources like RNS levels is an exceedingly complex, error-prone, and time-intensive endeavor. This requires developers to possess a rare combination of deep expertise in FHE cryptography and the

specific target hardware. The absence of a sophisticated compiler means that much of the potential performance gain from hardware acceleration remains locked behind a steep programmability barrier, hindering rapid deployment and broader adoption of FHE technologies.

To address these deeply intertwined hardware bottlenecks and critical software deployment challenges, which collectively hinder the practical adoption of FHE. We propose Hypnos, which is a hardware–software codesign framework for memory-efficient homomorphic processing. Our approach recognizes that neither hardware optimization nor software abstraction alone is sufficient. Instead, a synergistic codesign is paramount.

We summarize the contributions of Hypnos as follows.

- 1) We propose a practical, memory-efficient homomorphic processing unit. It aims to mitigate the performance overhead caused by inefficient PCIe communication and improve off-chip MU.
- 2) We propose a novel memory management system based on “HE page” to manage memory at the RNS component level, significantly increase MU, and further reduce the data transfer volumes during the execution of FHE applications.
- 3) We introduce the Hypnos Compiler, which automates the abstraction of FHE operations into optimized hardware instructions and orchestrates data management, bridging the gap between high-level applications and the underlying hardware to enable rapid deployment.
- 4) We implement a prototype of Hypnos on the commercial Qiankun, and evaluate its performance in practical FHE applications. We highlight the following results: 1) in data-intensive FHE applications, Hypnos outperforms state-of-the-art (SOTA) ASIC and FPGA solutions by $2.75\times$ and $4.72\times$, respectively; 2) communication overhead is reduced by $4.85\times$ compared to traditional accelerator architectures; and 3) In terms of energy efficiency, Hypnos also surpasses existing ASIC and FPGA accelerators by a factor of $30.1\times$ and $20.7\times$, respectively.

Note: This article is an extended version of our previous work [19]. Compared to the conference version, we have substantially expanded the manuscript in three major aspects: we provide a detailed design of the Hypnos compiler, conduct a comprehensive evaluation with new compiler-driven optimizations, and offer an in-depth analysis of the challenges in deploying high-performance FHE accelerators.

II. BACKGROUND

In this section, we first introduce the concepts of FHE. Next, we present the basic principles of the CKKS scheme [20], which is the main target scheme of this work. Finally, we will detail the RNS representation, which is widely adopted in current FHE schemes. This background knowledge is essential for understanding the design of the Hypnos memory-efficient homomorphic processing unit proposed in this article. Nomenclature summarizes commonly used notations in this article.

A. FHE

A FHE scheme typically consists of the following four core algorithms.

- 1) **KeyGen**(1^λ) $\rightarrow (sk, pk, evk)$: Takes a security parameter λ as input and outputs a secret key sk , a public key pk , and an evaluation key evk . The evaluation key is used to support homomorphic operations between ciphertexts.
- 2) **Enc**(pk, m) $\rightarrow ct$: Encrypts a plaintext m using the public key pk to produce a ciphertext ct .
- 3) **Dec**(sk, ct) $\rightarrow m'$: Decrypts a ciphertext ct using the secret key sk to recover a plaintext m' . For an ideal FHE scheme, $m' = m$; for approximate FHE schemes, m' is an approximation of m .
- 4) **Eval**($evk, f, ct_1, \dots, ct_k$) $\rightarrow ct_f$: Using the evaluation key evk , performs homomorphic operations corresponding to a function f on ciphertexts $ct_1, \dots, ct_k$, and outputs a resultant ciphertext ct_f such that **Dec**(sk, ct_f) is (approximately) equal to $f(m_1, \dots, m_k)$.

Most modern FHE schemes are based on hard problems in lattice-based cryptography, such as the ring learning with errors (RLWEs) problem [21]. In the RLWE problem, given a ring $R_q = \mathbb{Z}_q[X]/(X^N + 1)$ (where N is typically a power of 2, and q is a large modulus), polynomially many samples $(a_i, b_i = a_i \cdot s + e_i)$, where a_i are uniformly random in R_q , and s, e_i are sampled from some narrow error distribution χ , it is computationally hard to distinguish these samples from (a_i, u_i) where u_i are uniformly random in R_q .

B. CKKS Basic

The Cheon–Kim–Kim–Song (CKKS) scheme [20], [22] is an FHE scheme designed for approximate arithmetic, enabling efficient operations on encrypted real or complex number vectors. The core idea of CKKS is to treat the inherent noise in cryptographic operations as part of the approximation, thus allowing homomorphic computation on floating-point numbers.

1) *Encoding and Encryption*: The CKKS scheme first encodes a vector of complex numbers $\mathbf{z} = (z_0, \dots, z_{N/2-1}) \in \mathbb{C}^{N/2}$ into a plaintext polynomial $m(X) \in R$. The encoding process typically involves an isomorphism $\phi : \mathbb{C}^{N/2} \rightarrow R_{\mathbb{C}} = \mathbb{C}[X]/(X^N + 1)$, followed by scaling by a factor Δ and rounding to convert floating-point coefficients to integers:

$$m(X) = \text{round}(\Delta \cdot \phi(\mathbf{z})) \in R. \quad (1)$$

The encryption process is similar to standard RLWE encryption, producing a ciphertext $ct \in R_q^2$.

2) *Homomorphic Operations*: CKKS supports element-wise addition and multiplication on encrypted vectors.

- 1) *Homomorphic Addition*: $ct_{\text{add}} = ct_1 + ct_2$. The corresponding plaintext is $m_{\text{add}} \approx \Delta(m_1/\Delta + m_2/\Delta)$.
- 2) *Homomorphic Multiplication*: $ct'_{\text{mult}} = ct_1 \otimes ct_2$. This is a tensor product, which increases the number of ciphertext elements. A **Relinearization** [23] step is typically required, using an evaluation key, to convert ct'_{mult} back to the standard R_q^2 form, ct_{mult} . The corresponding plaintext is $m_{\text{mult}} \approx \Delta^2(m_1/\Delta \cdot m_2/\Delta)$.

Homomorphic multiplication causes the plaintext's scaling factor to become Δ^2 , and the noise also grows significantly. To manage the scaling factor and noise, CKKS introduces a **Rescaling** operation

$$ct' = \text{round}(ct/q_0) \pmod{q/q_0} \quad (2)$$

where q_0 is a modulus close to Δ . Rescaling reduces the ciphertext modulus from q to $q' = q/q_0$ and approximately changes the effective scaling factor of the plaintext from Δ^2 back to Δ .

3) *Modulus Chains and Levels*: To support multiple multiplications, CKKS employs a **modulus chains** technique. An initial ciphertext uses a large modulus $Q = q_L \cdot q_{L-1} \cdot \dots \cdot q_0$ (often represented using RNS). After each multiplication and rescaling, the modulus of the ciphertext is reduced by a factor q_i . Thus, the multiplicative depth L of the scheme depends on the size of the initial modulus and the number of bits removed by each rescaling.

Precision analysis in CKKS is a complex issue, depending on the initial encoding precision, rounding errors introduced by rescaling, and noise accumulated from homomorphic operations. Detailed expositions of CKKS noise management and precision analysis, which are beyond the scope of this background summary, are provided in the seminal works [20], [22].

C. Residue Number System

RNS is a technique to accelerate polynomial arithmetic in FHE schemes by representing large integers as vectors of residues modulo smaller coprime moduli [24].

1) *RNS Representation*: Given a set of pairwise coprime moduli $\{q_0, q_1, q_2, \dots, q_{k-1}\}$, any integer $a < Q = \prod_{i=0}^{k-1} q_i$ can be uniquely represented as

$$a \mapsto (a_0, a_1, a_2, \dots, a_{k-1}), \quad a_i = a \pmod{q_i}. \quad (3)$$

Based on the Chinese remainder theorem (CRT) [25], the integer $a \pmod{Q}$ can be uniquely reconstructed from the residue vector $\{a_1, a_2, \dots, a_k\}$.

2) *Advantages in FHE*: In FHE, the ciphertext modulus Q is typically very large (e.g., hundreds or even thousands of bits). Performing polynomial arithmetic directly in R_Q is inefficient [22], [26]. By choosing an RNS basis $\mathcal{B}_q = \{q_0, \dots, q_{k-1}\}$ such that $q = \prod q_i$, operations in R_q can be decomposed into parallel operations in each R_{q_i} . The difference between polynomial arithmetic with large moduli and polynomial arithmetic with RNS representations is shown in Table I. A polynomial $a(X) \in R_Q$ can be represented as $(a_0(X), \dots, a_{k-1}(X))$, where $a_i(X) = a(X) \pmod{q_i}$. The introduction of RNS significantly improves the performance of FHE schemes, making them more practical [26], [27]. Here we briefly summarize the following advantages.

- 1) *Parallelism*: Arithmetic operations can be performed independently on each residue, enabling SIMD-style parallelism.
- 2) *Efficiency*: Reduces the need for costly large-integer arithmetic.
- 3) *Memory Management*: In RNS-based FHE, ciphertexts and keys are stored as collections of RNS components, facilitating fine-grained ciphertext management and efficient data movement.

It is exactly due to the advantages of RNS in FHE applications and the technique of noise management via modulus chains that our proposed design of fine-grained ciphertext management becomes possible. Realizing such an efficient management system is a primary motivation for the Hypnos framework.

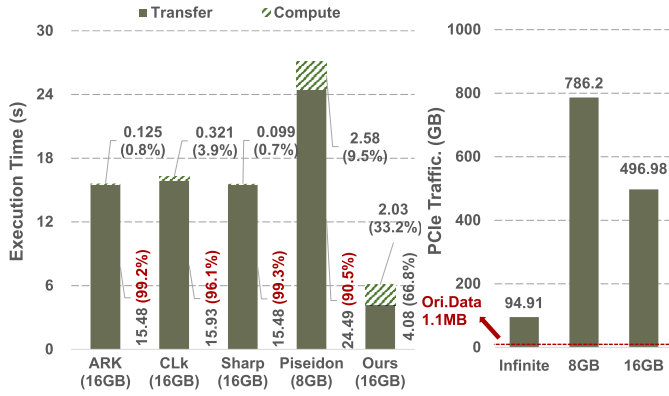


Fig. 2. Execution time breakdown of ResNet-20. The data size under FHE encryption is severely inflated compared to plaintext (1.1 MB marked in red), introducing significant PCIe communication overheads. PCIe transfer overhead is high for existing accelerator architectures, over 96% for ASIC accelerators, and up to 90% for FPGA-based Poseidon.

However, the use of RNS is not without its challenges; for instance, it introduces the complexity of managing conversions between different RNS bases. The design of Hypnos carefully considers these characteristics to optimize both memory access and computation pipelines. While the CKKS scheme [20], [22] is the primary focus of this work and one of the most widely used schemes today, the principles of the Hypnos design are broadly applicable to any FHE scheme based on the RLWE problem, including BGV [28] and BFV [27], giving it significant potential for future applications. The specific domain challenges and further motivations for our design will be detailed in Section III.

III. MOTIVATION

The unique characteristics of homomorphic encryption have attracted increasing attention. However, due to its huge computational overhead, a large number of previous hardware acceleration works have focused on improving the computing performance of FHE accelerators, which has greatly narrowed the performance gap between FHE ciphertext and plaintext computation in recent years. However, from the perspective of the overall system, the power of a system depends on more than improvements in computing performance. After analysis, we think that the previous designs have the following three common shortcomings.

A. Platform Limitations for FHE Accelerators

Due to the off-chip memory limitation, accelerators must frequently transmit large volumes of ciphertext and keys when processing FHE applications. This extensive data movement between the host DDR and the accelerator's local memory can severely impact performance. Prior works have often underestimated this issue. In our evaluation of the ResNet-20 application based on the SEAL library [29], as depicted in the left half of Fig. 2, despite the superior computational efficiency of ASIC accelerators, the overhead associated with PCIe transfers can account for up to 96% of the total execution time. It is natural to encounter such situations. For example, in the ResNet-20 application, there are 76.8 GB of keys needed [30]. The execution time reported by existing accelerators is usually in milliseconds. While even using the common PCIe

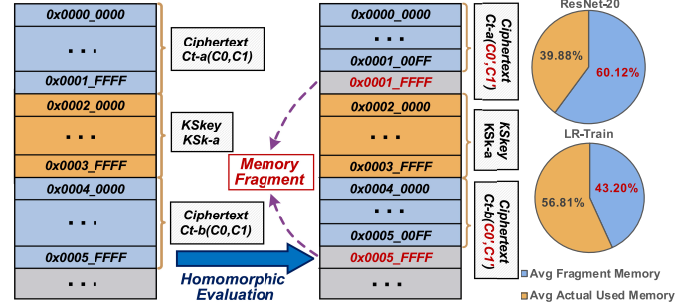


Fig. 3. Memory fragmentation during homomorphic computation. Changes in the number of RNS components of the ciphertext lead to memory fragmentation. The right side quantitatively analyzes the memory fragmentation under different FHE workloads.

Gen4 $\times$ 16 mentioned in those papers to transfer all keys at once would take seconds. Not to mention that currently it's impossible to load all data at once into the common 16 GB off-chip memory. This is why the communication time accounts for a large proportion. Similarly, FPGA accelerators confront analogous issues attributable to the limited off-chip memory. The right side of Fig. 2 shows the increase in PCIe communication overhead under different off-chip memory sizes, where the "infinite" item represents the communication scenario when all data could be loaded into the off-chip memory at once. The results show that, under memory constraints, PCIe communication volume can surge dramatically. Therefore, reducing PCIe communication overhead is a critical consideration for FHE accelerators. Otherwise, continuously optimizing the performance of the accelerators alone will not eliminate the PCIe communication bottleneck within the overall system.

B. Neglected Memory Fragmentation

Another significant factor contributing to excessive PCIe traffic and inefficient MU is memory fragmentation within off-chip memory. This factor has also been overlooked in prior FHE accelerator designs. In RNS-based FHE schemes [20], [22], [28], ciphertexts and keys are represented as collections of polynomials. As illustrated in Fig. 3, during homomorphic computations, the number of RNS components constituting a ciphertext changes (e.g., due to modulus switching or rescaling operations). This dynamic variation in ciphertext size, along with changes in the number of RNS components required for keys, poses a substantial challenge for fixed-size memory management strategies. Inefficient use of memory capacity leads to more frequent data transfers to and from the host, effectively reducing the accelerator's available working set and potentially stalling computational units. This issue is particularly evident in applications like ResNet-20 [30] and logistic regression training (LR-Train) [2] employing the CKKS scheme, where memory efficiency can be notably reduced. We conducted a quantitative evaluation of memory fragmentation under the CKKS scheme for the two applications illustrated on the right-hand side of Fig. 3. For LR-Train, which comprises three iterations and one Bootstrapping operation, the average memory fragmentation rate reaches 43.20%. ResNet-20 employs a larger data volume and adopts distinct network structures, resulting in an average memory fragmentation rate of up

to 60.12%. Further detailed hierarchical memory analysis is provided in Section VII-D. This, in turn, a substantial portion of the data transferred over PCIe may consist of redundant or unused RNS components resulting from fragmentation, unnecessarily consuming precious bus bandwidth and energy.

Existing FHE accelerator architectures, whether FPGA-based or ASIC-based, are generally not well-suited to address the challenge of memory fragmentation. They rely on conventional Host $\rightarrow$ PCIe $\rightarrow$ ACC configurations. Consequently, they are unable to implement the necessary fine-grained, dynamic memory management for the following reasons:

First, such management demands complex control logic that is often infeasible or excessively costly for hardware to implement autonomously. Second, given that the PCIe bus is typically a shared resource, relying on the host CPU for fine-grained scheduling and accelerator memory management may introduce unpredictable latency and contention. Such conditions are detrimental to the stringent timing requirements and performance consistency of the accelerator. Critically, the host and the accelerator operate in distinct physical address spaces. This separation prevents FHE applications running on the host from having direct, low-latency awareness or control over the precise data layout and status within the accelerator's memory. Consequently, any host-driven management would necessitate high-overhead data structure mirroring, synchronization, and explicit data copying operations.

C. Deployment Dilemma

Beyond hardware limitations, the practical deployment of FHE accelerators faces a significant programmability hurdle. Specialized architectures, while powerful, introduce new complexities in mapping high-level FHE algorithms to the underlying hardware. Manually optimizing the execution sequence for such complex hardware is an extremely challenging, error-prone, and time-consuming task, requiring deep expertise in both FHE algorithms and the specific hardware architecture. This software complexity creates a major bottleneck, hindering the rapid deployment and efficient utilization of advanced FHE accelerators [31]. A holistic solution must therefore include a sophisticated compilation framework that can automate these complex optimization tasks, effectively bridging the gap between the FHE application developer and the specialized hardware. This underscores the necessity for advanced compiler support that can automate such scheduling optimizations, transforming Hypnos from a powerful hardware platform into a more efficiently programmable and higher-performing system.

Based on the preceding analysis, we propose the Hypnos framework, which consists of three major components. The tightly-coupled CPU-accelerator architecture wherein the CPU is integrated into the accelerator. Unlike external scheduling strategies that depend on the host CPU and PCIe bus, this tight coupling provides distinct advantages. It enhances coordination efficiency between the managing CPU and the accelerator hardware, and concurrently achieves lower communication latency. Such capabilities are beyond the reach of conventional Host $\rightarrow$ PCIe $\rightarrow$ ACC architectures.

In terms of memory, we designed the homomorphic paged memory management system. The structural characteristics of RNS-based FHE schemes present a notable advantage.

Specifically, the size of individual RNS components is typically established and remains constant for a given parameter set. Consequently, all ciphertexts and keys can be represented as compositions of multiple, fixed-size RNS data segments. This inherent regularity, particularly when coupled with the predictable data flow often observed in FHE algorithms, renders RNS components a suitable level of granularity for implementing a paged memory management strategy.

Finally, the Hypnos compiler analyzes the optimization space in the program and memory management based on RNS components, generating hybrid executable files to drive the efficient execution of the entire Hypnos system.

IV. HYPNOS ARCHITECTURE

A. Architecture Overview

Hypnos is a novel memory-efficient heterogeneous CPU-accelerator architecture. First, we integrate a CPU in a tightly-coupled manner to assist hardware in efficient memory management, unlike previous designs where CPUs merely serve as accelerator schedulers. In Hypnos, the CPU is embedded within the hardware acceleration unit to collaboratively manage private data efficiently. Second, we introduce a customized homomorphic paged memory management system inspired by CPU paging concepts [32] but tailored for FHE accelerators. Compared to traditional CPU paging systems, Hypnos's customized homomorphic paging just introduces lower overhead. For instance, while CPU page sizes are typically 4 KB, RNS components in Hypnos are substantially larger (e.g., 2 MB), resulting in minimal page table overhead under typical DDR/HBM sizes (16 GB).

As shown in Fig. 4, the Hypnos architecture comprises three distinct regions.

- 1) *Processing System*: The processing system (PS) region is equipped with a high-performance ARM Cortex-A72 CPU that runs the operating system. This CPU is responsible not only for initializing the state of the CUs but also for orchestrating the execution of FHE computation applications.
- 2) *Programmable Logic*: The programmable logic (PL) region provides reconfigurable hardware resources serving as CU. The CU integrates an FHE controller for command scheduling, a dedicated register file for fast data access, and a homomorphic encryption page management unit (HEPMU) for efficient memory management.
- 3) *Network-On-Chip*: The network-on-chip (NOC) region acts as the high-bandwidth, low-latency point-to-point interconnection network, leveraging the on-chip buses of the AMD Xilinx Versal ACAP series. It facilitates seamless communication between the PS and PL regions.

B. FHE Execution and Controller Unit

Hypnos adopts an abstracted methodology utilizing operational-level operators, as opposed to basic operators (such as NTT, MM, MA, etc.), providing 64 parallel processing units. Within these operational-level constructs, basic operators function in a cascaded arrangement, with each unit possessing its own cache, thereby mitigating accesses to main memory and enhancing the efficiency of request processing

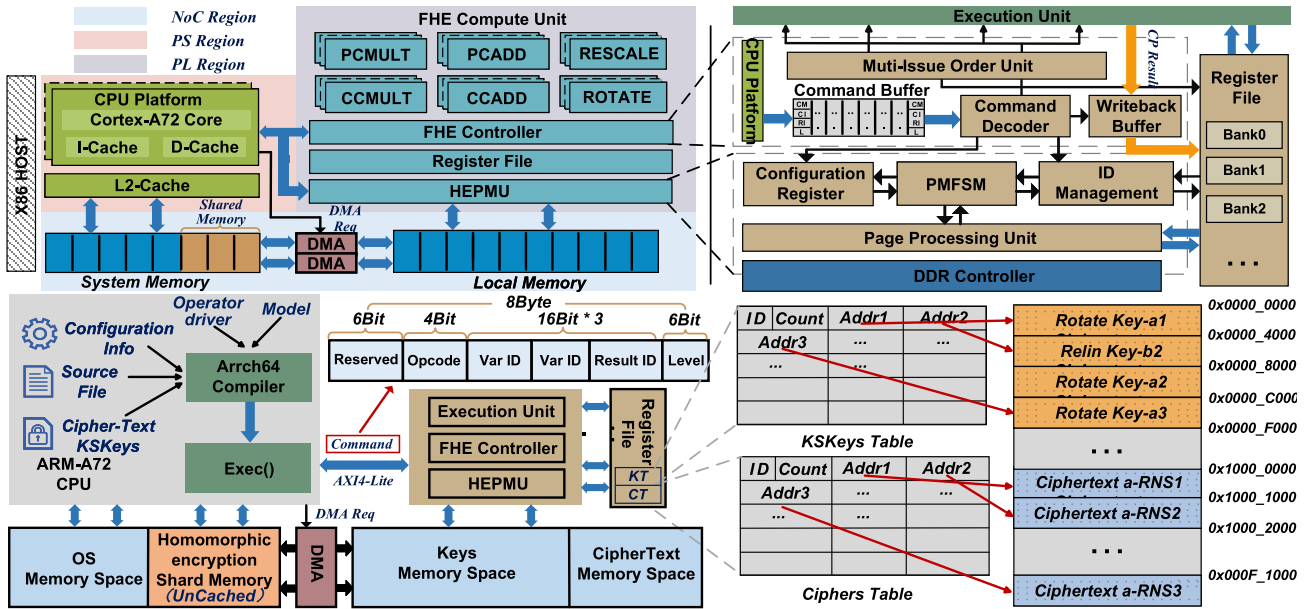


Fig. 4. Hypnos overall architecture. The heterogeneous ARM + FPGA computing architecture features a Cortex-A72 CPU, connected through a NoC to various components, including the FHE CUs, HEPMU, and memory subsystems. This setup highlights the multilayered interaction between processing units, local and system memories, DMA channels, and specialized encryption hardware, illustrating a memory-efficient system for FHE acceleration.

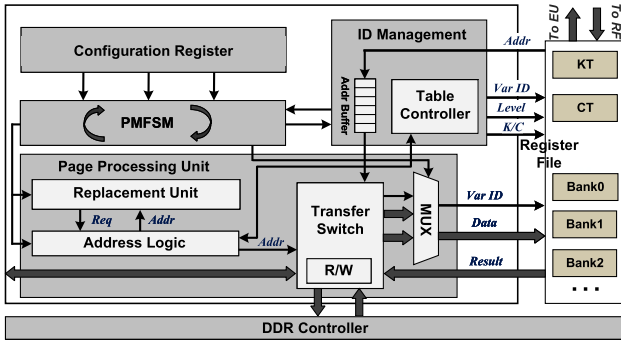


Fig. 5. Micro-architecture of HEPMU.

alongside the overall system performance. Internally, the Barrett Reduction algorithm [33], [34] is employed for modular multiplication (MM), while the number theoretic transform (NTT) [35] utilizes a radix-8 design and incorporates optimizations derived from Poseidon [12].

The FHE controller is responsible for scheduling, whereas the command decoder assigns tasks to the functional units. The role of the Multi-Issue Order Unit is to disseminate processed command information to the CUs. Computed results are subsequently stored in registers via a write-back buffer. The HEPMU manages memory operations and returns results either to the CPU or for further computations.

C. HEPMU

The cornerstone of Hypnos is the HEPMU, which implements paged memory management at the granularity of RNS components. The microarchitecture of the HEPMU is illustrated in Fig. 5. The detailed workflow of the HEPMU will be elaborated upon in Section VI. This section is dedicated to the analysis of hardware architecture design.

1) *Configuration Register*: Provides scalable memory management granularity and table entries of varying sizes based on different applications and security parameters. Configuration details are written via the AXI4-Lite bus and are utilized during the execution of FHE applications. Additionally, the CPU can read this configuration information through the same bus to determine the status of Hypnos.

2) *Page Management Finite State Machine*: Acts as the central control logic, switching between idle, write, eviction, and read states according to the sequence of commands, coordinating task scheduling within the HEPMU.

3) *ID Management*: Performs ID-to-address translation, querying addresses from the Key Table (KT) and Ciphertext Table (CT) based on the type of ciphertext variable, and writing addresses to the buffer. Notably, the CT and KT are stored in a register file to accommodate dynamic changes in table sizes due to varying parameter configurations. For example, when the modulo chain length $L = 32$ and the memory size is 16 GB, the sizes of CT and KT are 69 and 17.25 KB, respectively.

4) *Page Processing Unit*: Executes operations related to the RNS page table, retrieving RNS components based on address information and placing them in the register file for computation. When page replacement is required, it writes page information back to the KT or CT.

It is noteworthy that, as shown in the lower-left portion of Fig. 4, a portion of the system memory is designated as shared memory, specifically reserved for storing ciphertexts and keys, and configured as uncacheable. This design choice is because, during homomorphic computations, the CPU processes address information rather than the ciphertexts and keys themselves. Moreover, the uncacheable attribute of the shared memory eliminates the computational overhead associated with cache coherence checks and cache flushing. Given that different applications have varying requirements for

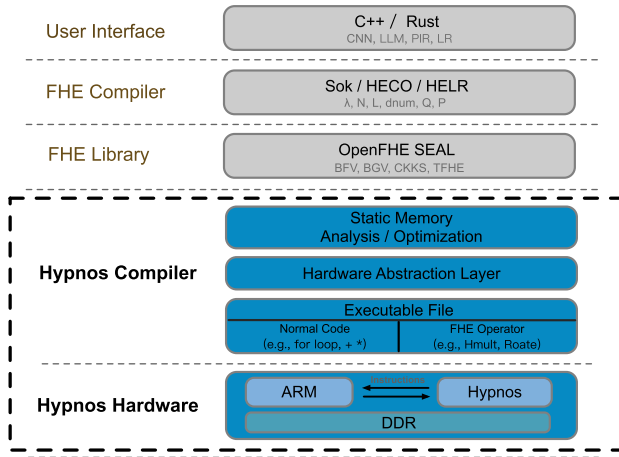


Fig. 6. Overview of the Hypnos Compiler Workflow, illustrating the stages from FHE C++ code input to executable generation for the heterogeneous Hypnos platform.

ciphertext and key memory spaces, the sizes of these spaces are dynamically specified by the FHE program running on the CPU. For instance, in CNN applications that require a large number of keys, a larger key storage space can be allocated to mitigate the need for frequent data exchanges.

Regarding the design of page table replacement strategies in the HEPMU, we evaluated several strategies for ResNet-20 on Hypnos. The performance of ResNet-20 indicated that, both in terms of execution time and the number of replacements, the least recently used (LRU) strategy demonstrated performance closest to the optimal (OPT) [36]. Consequently, we adopted LRU as the page table replacement strategy in the HEPMU.

Under memory constraints, Hypnos's improved MU inherently reduces PCIe data transmission volume, thereby enhancing overall FHE application performance.

V. HYPNOS COMPILER

Building on the hardware innovations described in Section IV, the Hypnos Compiler is a cornerstone of the Hypnos framework, designed to unlock the full potential of the hardware architecture by addressing the programmability and optimization challenges inherent in FHE acceleration. It acts as the crucial bridge between high-level FHE applications and the low-level, memory-efficient operations of the Hypnos processing unit. This section details the design principles and core functionalities of the Hypnos Compiler, with a particular focus on its RNS-level static analysis capabilities that enable the fine-grained memory management realized by the Hypnos framework.

A. Compiler Overview

Traditional FHE compilers typically focus on encapsulating low-level cryptographic details to lower the barrier for developers unfamiliar with cryptography to develop FHE programs [37], [38], [39], [40], [41], [42]. In contrast, the Hypnos Compiler primarily targets backend optimization. The functional design of this compiler is orthogonal in purpose to that of traditional compilers. As illustrated in Fig. 6, the Hypnos Compiler aims to seamlessly integrate with existing

FHE development workflows to fully unlock the performance and efficiency potential of the Hypnos hardware. The compiler is a pivotal element in advancing the Hypnos system from a specialized hardware unit to a more robust and deployable FHE acceleration framework.

B. RNS-Aware Static Analysis for Memory Management

In the CKKS scheme, the number of RNS components dynamically changes during specific operations, particularly during rescaling, thereby posing a significant challenge to memory management. However, the execution of arithmetic FHE algorithms exhibits a high degree of predictability. The program execution sequence of these algorithms can be abstracted as a standard directed acyclic graph (DAG) [17], [43], [44]. Consequently, the usage patterns of RNS components can be accurately predicted. The Hypnos compiler employs an RNS-aware static analysis method for optimization. The static analysis and optimization mainly include the following criteria:

1) *Tracking RNS Levels*: To adapt to the dynamic variation of RNS components, the compiler constructs a DAG through RNS level tracking, where nodes represent FHE operations and edges denote data dependencies. Through static analysis, each operation node is assigned a deterministic RNS level, defining the valid *Level* for each variable ID at that specific program point. For instance, when a rescaling operation occurs (e.g., reducing L from 32 to 16), the compiler sets the specified *Level* for the variable ID in the metadata. At runtime, as described in Section VI, this level of information guides the HEPMU. The HEPMU utilizes this value to selectively load and allocate memory pages only for the active RNS indices (e.g., 0 to 15), while effectively pruning the higher-order components (e.g., 16 to 31). This mechanism prevents redundant RNS components from being allocated or fetched into the local memory, thereby automatically adjusting the working set size.

2) *Command Optimization*: The core of instruction optimization lies in analyzing data dependencies in the DAG to identify operations that can be executed in parallel or reordered without violating program correctness. It uses scheduling algorithms to prioritize operations on the critical path and alleviate resource contention to determine an effective execution order for FHE operators. Meanwhile, to optimize memory usage, the compiler attempts to minimize the lifespan of intermediate RNS components and enhance the temporal locality of RNS component accesses by arranging commands sequentially, which may improve page hit rates and reduce data exchanges. Although physical data transfers are managed by HEPMU, the compiler's scheduling decisions directly affect which RNS components are loaded into the local memory of the CU and when. By grouping operations that use the same data together or scheduling operations to reuse recently fetched RNS components, the compiler aims to reduce the total amount of data moved between shared memory and CU local memory. This includes ensuring that only RNS components relevant to the current RNS level are loaded and computed. It can be seen that although HEPM mainly targets the fragmentation issue caused by dynamic rescaling in CKKS, compiler instruction scheduling optimization is effective across all schemes. We discuss this observation in more depth in Section VII-F.

3) *Generating Metadata for HEPMU*: The compiler generates metadata that maps high-level ciphertext variables to their constituent RNS components and their current valid `Level`. This metadata is passed to the runtime system. When an FHE operation is offloaded to the Hypnos hardware, the CPU, guided by this compiler-generated information, can instruct the HEPMU to fetch only the necessary RNS components of the input ciphertexts from shared memory to the CU's local memory or register files. This ensures that memory bandwidth and capacity are not wasted on transferring and storing RNS components that are no longer active (e.g., moduli that have been “rescaled away”).

4) *Memory Mapping Analysis*: A critical consideration when generating metadata for the HEPMU is the strategy for mapping RNS components to physical memory addresses. While traditional high-performance computing (HPC) typically aims to maximize DRAM row buffer locality by enforcing physical contiguity, Hypnos adopts a flexible placement strategy driven by RNS components.

In our RNS-based scheme, a single RNS component typically occupies a substantial contiguous memory block (e.g., several megabytes). When the HEPMU performs DMA transfers for such a component, the memory controller operates primarily in burst mode. Although the starting addresses of distinct RNS components may be noncontiguous (scattered) due to dynamic allocation aimed at minimizing memory fragmentation, data access within each individual component remains strictly sequential.

Consequently, the overhead associated with row buffer conflicts occurs only when switching between these large components; this overhead is effectively amortized over the significant duration of the component's transfer. Our analysis indicates that enforcing strict physical contiguity across all RNS components of a ciphertext imposes rigid constraints on the memory allocator and exacerbates fragmentation, while yielding negligible improvements in effective memory bandwidth. Therefore, the Hypnos compiler prioritizes optimizing the temporal scheduling of transfers (i.e., determining when to load data) rather than enforcing spatial locality across distinct RNS components, thereby preserving the flexibility needed to maximize on-chip MU.

5) *Cooperation With HE Paged Management System*: The static analysis performed by the compiler is designed to directly support the HE Paged Management System detailed in Section VI.

By performing this RNS-level analysis at the compilation phase, the Hypnos compiler plays a pivotal role in transforming a standard FHE program into an optimized sequence of operations that fully leverages Hypnos's memory-efficient architecture, thereby reducing data transmission overhead and improving overall system performance. Moreover, this automated approach effectively mitigates potential errors and lessens the substantial workload typically associated with manual compilation efforts.

C. Compilation Flow

The compilation process within the Hypnos Compiler can be broken down into several key stages.

Input Processing and Frontend: The compiler frontend receives the C++ source code. It parses the code to build an

IR. During this phase, calls to FHE library functions related to encryption, decryption, key generation, and homomorphic operations are identified and tagged.

Static Memory Analysis and RNS-Level Optimization: By knowing the RNS `Level` required for subsequent operations, the compiler can identify which RNS components of a ciphertext or key are essential for a given computation and which are currently redundant.

HAL Integration: The compiler utilizes a hardware abstraction layer (HAL) that defines the interface to the Hypnos hardware. This layer provides abstractions for the FHE operators, memory organization, and communication protocols between the ARM CPU and the FHE CUs. FHE library calls identified in the frontend are mapped to these HAL primitives.

Code Partitioning and Generation: Since it is directly integrated with the CPU, FHE applications can be executed entirely within Hypnos.

- 1) *General-Purpose Code*: Standard C++ constructs and non-FHE logic are compiled into machine code for the ARM Cortex-A72 processor.
- 2) *FHE Operator Code*: Identified FHE operations are translated into a sequence of commands for the Hypnos FHE CUs.

Executable File Output: The final output is a single executable file. This file contains the compiled ARM code, the FHE operator command sequences, and metadata generated during the RNS-level optimization phase. This metadata is used at runtime by the CPU-side FHE application and the HEPMU to manage memory efficiently.

The HEPMU's ability to perform fine-grained memory management is substantially enhanced by the compiler's foresight. Without the precise, statically determined information on active RNS components, the hardware would either require overly complex heuristics or rely on coarse-grained information, diminishing its efficiency. Therefore, the Hypnos compiler unlocks the full potential of the HEPMU, significantly improving MU and reducing unnecessary PCIe traffic compared to traditional approaches that manage whole ciphertexts.

VI. HE PAGED MANAGEMENT SYSTEM WORKFLOW

The preceding sections have elaborated in detail on the Hypnos hardware architecture and the design principles of its compiler. This section illustrates the overall workflow of the Hypnos accelerator system through a specific case, with a particular focus on analyzing the operational mechanism of its homomorphic paged memory system. To optimize MU and enhance overall system performance, this homomorphic paged memory system implements fine-grained management of ciphertexts and keys in memory at the granularity of individual RNS components. A core characteristic of this approach lies in its hardware-software codesign, wherein the CPU and the CU collaborate closely. Specifically, the CPU is responsible for executing the high-level FHE application logic, which is compiled by the dedicated Hypnos compiler (detailed in Section V). Conversely, the CU autonomously manages the physical addresses of homomorphic variables and performs fundamental FHE operations. As illustrated in Fig. 7, this paged management system achieves efficient computation

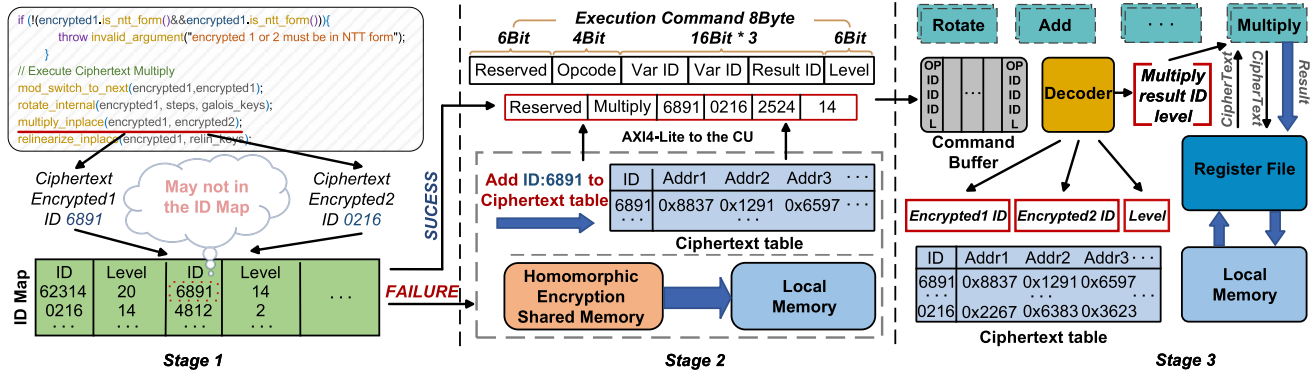


Fig. 7. This diagram illustrates the multistage execution process of a command within Hypnos, providing a detailed account of the entire workflow. Each stage is managed and invoked via specific operation IDs, addresses, and level parameters, ensuring the accurate execution of the encrypted command.

through a multistage operational process, which is crucial for the overall system performance.

A. CPU-Side Orchestration and Command Issuance

The execution of FHE applications on the Hypnos platform commences on the CPU, leveraging outputs from the Hypnos Compiler, which performs RNS-aware static analysis to identify essential RNS components and their respective levels for each operation.

1) Stage 1: Variable Status Inquiry and Command Path:

As depicted in Stage 1 of Fig. 7, when the FHE application encounters an operation designated for hardware acceleration, the CPU first consults a software-maintained data structure, the ID Map. This ID Map tracks the identifiers (IDs) of ciphertext and key variables and their current RNS levels that are presumed to be resident within the CU's managed memory space (i.e., local memory or registered in the CU's hardware tables).

If the ID Map indicates that all operand variables (e.g., Encrypted1 ID 6891 and Encrypted2 ID 0216 in Fig. 7) and their required RNS components are already present and up-to-date within the CU's memory context (the "SUCCESS" path), the CPU proceeds to dispatch an execution command directly to the CU. This command, typically an 8-byte packet as shown, encapsulates the operation code (e.g., Multiply), operand IDs, a result ID, and the pertinent RNS level information necessary for the operation. The command is transmitted to the CU's command buffer via a low-latency AXI4-Lite interface. Subsequently, the CPU pauses execution for that thread or task, awaiting an interrupt from the CU signifying the completion of the FHE operation.

Conversely, if the ID Map reveals that one or more required variables or their specific RNS components are not currently available in the CU's memory (the "FAILURE" path, e.g., Ciphertext Encrypted1 ID 0216 is present but perhaps Ciphertext Encrypted2 ID 6891 is "May not in the ID Map" or its RNS level is incorrect), a data provisioning sequence is initiated. This necessitates actions described in Stage 2 of the workflow.

B. CU-Side: Data Management and Execution

The CU is responsible for the intricate tasks of physical address management for RNS components, dynamic data

replacement (eviction), and the execution of FHE operations, all orchestrated by the HEPMU.

1) Stage 2: RNS Component Provisioning and Table Synchronization:

Following a "FAILURE" condition in Stage 1, where essential RNS components are missing from the CU's local memory, the CPU instructs the HEPMU to fetch these specific components. Guided by the compiler-generated metadata, the HEPMU transfers only the necessary RNS components of the identified variables (e.g., ID : 6891) from the HE Shared Memory into the CU's local memory, as illustrated in Fig. 7. This shared memory is configured as uncacheable to avoid cache coherency overheads, as the CPU primarily handles address information rather than the raw ciphertext data. The sizes of these shared memory spaces for keys and ciphertexts are dynamically allocated by the FHE program running on the CPU to suit application needs.

The page processing unit (PPU) within the HEPMU facilitates this data transfer. Concurrently, the ID Management block updates its hardware-managed KT and CT. These tables, often implemented using register files to support dynamic configurations based on security parameters and application requirements, map the variable IDs to their newly assigned physical addresses within the CU's local memory and store their current RNS Level. For instance, Fig. 7 shows "Add ID:6891 to Ciphertext table" with its corresponding addresses (Addr1, Addr2, Addr3). The CPU's ID Map is then updated to reflect these changes. If the CU's local memory is constrained, the PPU employs a page replacement algorithm, such as LRU, to evict existing RNS components, writing them back to shared memory if modified and necessary. This entire process is governed by the page management finite state machine (PMFSM). This RNS-component-level granularity is crucial for maximizing MU and minimizing fragmentation.

2) Stage 3: FHE Command Execution and Result Propagation:

Once an execution command is received from the CPU (either directly after a "SUCCESS" in Stage 1 or after data provisioning in Stage 2), and the HEPMU confirms the presence of all requisite RNS components in local memory via its CT/KT, the FHE controller within the CU takes charge.

The command is fetched from the Command Buffer and decoded. The ID Management unit, using the operand IDs (e.g., Encrypted1 ID, Encrypted2 ID) and the Level specified in the command, translates these into physical

addresses of the RNS components stored in the CU’s local memory or register file banks. The HEPMU ensures that these components are supplied to the appropriate FHE operational units (e.g., PCMULT, CCADD, RESCALE, ROTATE) within the Execution Unit. The RNS-level parameter is critical, ensuring that only the currently active moduli in the RNS representation are processed, thereby avoiding computation on redundant data.

The designated FHE operation (e.g., “Multiply” in Fig. 7) is then performed. Upon completion, the resulting ciphertext’s RNS components are written back into the CU’s register file or local memory. The HEPMU updates the CT or KT with the new Result ID and its corresponding physical address and new RNS Level.

Finally, the CU issues an interrupt to the CPU, signaling the completion of the command execution. This interrupt may also convey the result ID and its location, enabling the CPU to utilize this result for subsequent computations or to retrieve final data from shared memory if it has been written back by the CU. This completes the command execution cycle.

This meticulously orchestrated workflow, underpinned by the HE paged memory management system, enables Hypnos to significantly reduce data movement, particularly over the PCIe bus, by maintaining a working set of active RNS components within the CU’s proximity. The fine-grained management at the RNS component level drastically improves MU efficiency within the CU’s local memory and the system’s shared memory, mitigating fragmentation. These optimizations collectively contribute to substantial reductions in execution time and enhancements in energy efficiency, especially for data-intensive FHE workloads.

VII. EVALUATION

A. Experimental Setup

1) *Platform*: Hypnos is implemented on the Qiankun FPGA Card from Ant Inc., featuring a dual-core ARM A72 processor, over 1.7 million LUTs, and more than 7000 DSPs. The specific FPGA model is the xcvp1502-vsua2785-2MHP-e-S.

2) *Baseline*: In our experiments, Hypnos was compared with SOTA ASIC-based FHE accelerators, including ARK [14], Craterlake [13], and SHARP [16], and FPGA-based accelerators such as FAB [11] and Poseidon [12]. Some accelerators only implemented portions of the Benchmark. For clarity, we evaluated these architectures based on the complexity of each Benchmark.

3) *Benchmark*: We use the following 4 benchmarks (as shown in Table II) for evaluation.

- 1) *ResNet-20*: Implemented and trained with the CIFAR-10 dataset, featuring 19 convolutional layers and one fully connected layer, with ReLU activation functions [30].
- 2) *Logistic Regression (LR)*: Based on the HELR algorithm using the CKKS scheme [2], incorporating Bootstrapping with a multiplication depth of $L = 38$ evaluated over 10 iterations supported by two Bootstrapping operations.
- 3) *Private Set Intersection*: Evaluated using the SEAL APSI open-source library for computing the intersection of two asymmetric sets, with the sender’s set containing 2^{28} elements and the receiver’s set 2^{10} elements [45].

TABLE I
COMPARISON OF INTEGER AND RNS ARITHMETIC

Operation	Integer Arithmetic	RNS Arithmetic
Addition	$a + b$	$(a_1 + b_1 \bmod q_1, \dots, a_k + b_k \bmod q_k)$
Multiplication	$a \times b$	$(a_1 b_1 \bmod q_1, \dots, a_k b_k \bmod q_k)$

TABLE II
PERFORMANCE COMPARISON OF FOUR BENCHMARK APPLICATIONS WITH THE BASELINE FHE ACCELERATOR, MEASURED IN TWO TIME UNITS: SECONDS (S) AND MILLISECONDS (MS)

Works	ResNet-20 (in s)	LR-Train (in s)	PSI (in ms)	PIR (in ms)
(ASIC) ARK	15.61	0.34	422.73	306.10
(ASIC) Craterlake	15.80	0.25	408.45	303.97
(ASIC) Sharp	15.58	0.18	417.73	305.23
(FPGA) Poseidon	27.07	2.32	689.79	389.77
(FPGA) FAB	22.68	3.22	1168.12	472.30
(FPGA)Hypnos	5.73	1.81	267.57	83.24

- 4) *Private Information Retrieval*: Benchmark evaluates the SEAL private information retrieval (PIR) open-source library for encrypted queries on the server side, involving a query for a single item from a dataset of 2^{20} entries [46].

B. Full-System Performance

The design goal of Hypnos is to provide an architecture capable of running complete FHE applications. We demonstrate the full-system performance of Hypnos in comparison to previous accelerators. As shown in Table II, for data-intensive applications such as ResNet-20, private set intersection (PSI), and PIR, the primary bottleneck is the overhead of memory access. Hypnos can leverage its strengths, achieving up to a $2.75\times$ improvement in performance on ASIC-based accelerators and up to a $4.7\times$ improvement on FPGA-based accelerators. LR-Train, on the other hand, is a HE application that is computationally intensive with minimal memory access; it does not involve large-scale databases or extensive rotating keys. In terms of computational performance, Hypnos achieves a $1.77\times$ improvement over FPGA accelerators, though a performance gap remains compared to ASIC accelerators.

C. Energy Efficiency

We utilize the energy-delay product (EDP) as the metric for energy efficiency. Table III presents the energy efficiency performance of Hypnos compared with ASIC-based accelerators ARK and Craterlake, and the FPGA-based accelerator Poseidon across four benchmarks. In ResNet-20, PSI and PIR, Hypnos outperforms both ASIC and FPGA accelerators, achieving enhancements of $30.1\times$ and $20.7\times$, respectively. This is primarily due to the substantial data transfers, which contribute to significant PCIe overhead. Hypnos’s architecture excels in managing these demands. However, in the LR-Train

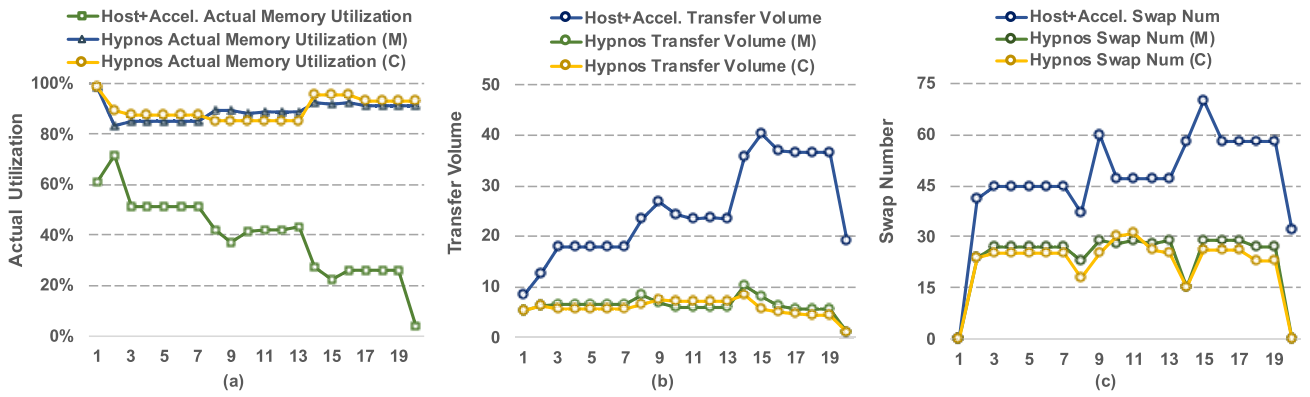


Fig. 8. How (a) MU, (b) transfer volume, and (c) swap number change at each layer of ResNet-20 under encrypted execution when comparing Hypnos to traditional architectures, with both systems featuring a uniform memory size of 16 GB and differing in granularity: the traditional architecture managing memory at the granularity of entire ciphertexts, while Hypnos operates at the granularity of a single RNS component.

TABLE III
EFFICIENCY ANALYSIS. WE USE EDP AS THE METRIC.
LOWER IS BETTER

Work	ResNet-20	LR-Train	PSI	PIR
ARK	68545.1	32.5	50267.4	26356.8
Craterlake	51675.5	12.9	34533.3	19125.8
Poseidon	87619.1	643.6	56892.1	18165.3
Hypnos	4159.9	415.1	9070.9	877.9

applications, Hypnos does not match the performance of ASIC accelerators in terms of EDP, due to the FPGA's limited resources, yet it still surpasses the FPGA accelerator Poseidon.

D. Memory Performance Analysis

In conventional ASIC and FPGA-based FHE accelerator designs, a Host + Accelerator topology is typically used. In this architecture, the Host system needs to coordinate data transfer and manage accelerator resources, usually communicating via bandwidth-limited and high-latency PCIe buses. To quantitatively evaluate this, we use ResNet-20 as a representative benchmark for data-intensive applications. Fig. 8 provides a comparative analysis of Hypnos, driven by both manual compilation [Hypnos (M)] and the Hypnos Compiler [Hypnos (C)], against an abstracted Host + Accelerator model. This three-way comparison across critical memory-related metrics highlights the progressive benefits of the Hypnos architecture and the further enhancements brought by its compiler. The layers of the CNN are represented on the horizontal axis, and all systems are configured with a uniform 16 GB off-chip memory on the accelerator side.

1) *Memory Utilization*: Defined as the percentage of RNS components actively used for computation relative to the total RNS components loaded into memory for the current layer's ciphertexts and keys. As illustrated in Fig. 8(a), traditional Host + Accelerator models suffer from low MU. This is because entire ciphertexts, including RNS components that become redundant after operations, continue to occupy memory space. The MU in such systems can plummet below 20% toward the final layers of the network, where ciphertext levels

are lowest. The Hypnos architecture, even with Hypnos (M), significantly improves MU by leveraging the HEPMU for RNS-component level management. However, the Hypnos (C) further enhances MU. This superior performance is attributed to the compiler's precise RNS-level static analysis and its ability to generate metadata that allows the HEPMU to load and maintain only the strictly necessary RNS components in local memory. Across several intermediate layers, Hypnos (C) shows a slight drop in MU compared to Hypnos (M), due to the compiler making wiser global decisions regarding such retention. This is a deliberate design choice; some variables that are frequently reused across computation stages are strategically kept in memory even if they are not immediately needed by the current instruction, to avoid the latency of repeatedly fetching them from shared memory and thereby optimize overall performance.

2) *Transfer Volume*: This metric quantifies the volume of data transferred over the PCIe bus (or equivalent host-accelerator interface in the abstracted model) at each layer of the ResNet-20 application. Fig. 8(b) demonstrates that while initial transfer volumes for loading the first layer's data are comparable, both Hypnos (M) and Hypnos (C) achieve substantial reductions in data transfer for intermediate layers compared to the Host + Accelerator model. Notably, up to 73.55% and 79.36% reduction are achieved by Hypnos (M) and Hypnos (C) compared with the traditional models. This superior reduction by the codesign approach is a direct consequence of its more effective MU and precise RNS component management, which minimizes the need for premature eviction and subsequent re-fetching of data.

3) *Swap Number*: This counts the instances of memory variable swaps occurring due to insufficient capacity in the accelerator's local memory to hold all concurrently active data. The frequency of swaps is linked to MU efficiency. As shown in Fig. 8(c), both Hypnos (M) and Hypnos (C) diminish the total number of swap operations by more than 50% compared to traditional architectures. The Hypnos (C) again demonstrates an advantage over manual compilation by further reducing swap numbers in several layers. This reduction is primarily due to the increased effective memory capacity afforded by the compiler's more precise management of data at the RNS component level. Fewer swaps directly

TABLE IV
COMPILER ABLATION

Method	ResNet-20	LR-Train	PSI	PIR	Compile Time	Expertise
Hypnos (M)	6.11s	1.89s	278.27ms	86.72ms	Few Days - Few Weeks	High
Hypnos (C)	5.73s (93.7%)	1.81s (95.7%)	267.57ms (96.1%)	83.24ms (95.9%)	Few Secs - Few Mins	Low

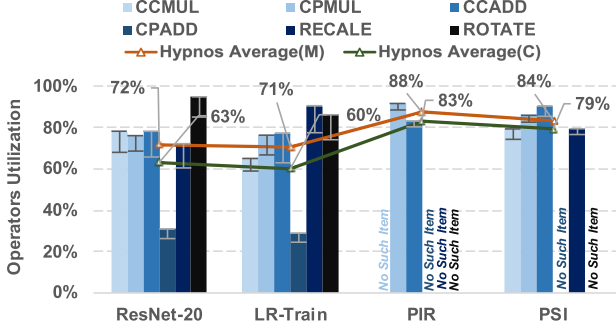


Fig. 9. This is the utilization of each Operator during the run in four benchmarks and the average utilization of each Operator in each benchmark by Hypnos.

translate into reduced PCIe traffic and fewer pipeline stalls within the computational units, leading to faster execution.

E. Operator Utilization

In addition to the raw execution time, one of the key aspects of Hypnos' compiler global optimization strategy is its ability to enhance the utilization of FHE operators within CUs. Fig. 9 breaks down operator utilization across various benchmarks, comparing average utilization achieved through manual compilation [Hypnos Average (M)] versus Hypnos compiler [Hypnos Average (C)]. Across all benchmarks, and especially in CKKS-based applications such as ResNet-20 and LR-Train, RNS-level optimizations and global command scheduling have the largest impact, with the Hypnos compiler consistently achieving higher average operator utilization. For ResNet-20, the compiler boosts average operator utilization from 60% manually to 72% via the compiler. Similarly, for LR-Train, utilization increases from 63% manually to 71% via the compiler. This improvement is attributed to the compiler's holistic view of the application DAG, enabling it to reschedule operations and manage data dependencies more effectively. While manual optimization may excel at fine-tuning specific critical paths, the compiler's automated global analysis can lead to more balanced workload distribution and reduced idle times for various FHE operators. For PSI and PIR, due to their relatively regular dataflows and simple computations, the Hypnos compiler does not outperform manual compilation by much.

F. Compiler Performance

Table IV presents an ablation study comparing the performance of applications compiled with the Hypnos Compiler against meticulously optimized manual compilation. For complex CKKS-based applications such as ResNet-20 and LR-Train, the Hypnos compiler delivers significant performance improvements. In ResNet-20, Hypnos (C) improves upon Hypnos (M)'s 6.11 s by 6.2%. Similarly, in LR-Train, the

TABLE V
HARDWARE RESOURCE UTILIZATION

Resources	LUT(K)	FF(K)	BRAM	DSP
Used	860	1,115	2,511	4,990
Percentage	(50.02%)	(32.41%)	(65.36%)	(67.07%)

compiler's output improves upon the hand-compiled version (1.89 s) by 4.2%. These accelerations are primarily attributed to the compiler's sophisticated RNS-aware static analysis capabilities, which enable it to perform fine-grained memory management on RNS components and have global instruction scheduling abilities, thus optimizing execution workflows beyond manual capability.

Even for BFV-based applications such as PSI and PIR, which have static RNS component sizes and therefore cannot benefit extensively from dynamic RNS memory management, the Hypnos compiler still provides performance improvements. The gains for these BFV-based schemes come from the compiler's global optimization of instruction ordering. These improvements are not as pronounced as those seen in CKKS applications but still highlight the compiler's ability to optimize beyond RNS-specific functionality.

Crucially, these performance benefits are coupled with a significant reduction in development effort. As indicated in Table IV, manual compilation can take from days to weeks and requires high professional expertise. In contrast, the Hypnos Compiler achieves these results with compilation times in the order of seconds to minutes, demanding significantly lower developer expertise ("Expertise: Low"). This highlights the Hypnos Compiler's superior comprehensive cost-benefit, transforming a potentially hard-to-program hardware accelerator into an accessible and highly efficient system, effectively bridging the gap from a high-performance hardware unit to a truly deployable FHE framework.

G. FPGA Resource Utilization

Hypnos operates on an ARM + FPGA heterogeneous computing platform. Table V delineates the resource utilization within the PL region. This includes the CU, controllers, register files, and the HEPMU. A significant portion of the DSP blocks is allocated to operator computations, whereas the DSP blocks within the HEPMU primarily support the hardware implementation of the LRU replacement algorithm and manage physical addresses. In the context of the HEPMU, BRAM is utilized for caching homomorphic encrypted variables. It is worth noting that the ARM processor, NoC, and DDR controller on the FPGA card are implemented as hard IP cores and thus do not consume any of the FPGA's programmable resources.

H. Sensitivity

Fig. 10 explores (a) and (b) impact of memory management granularity based on RNS components and (c) size of the

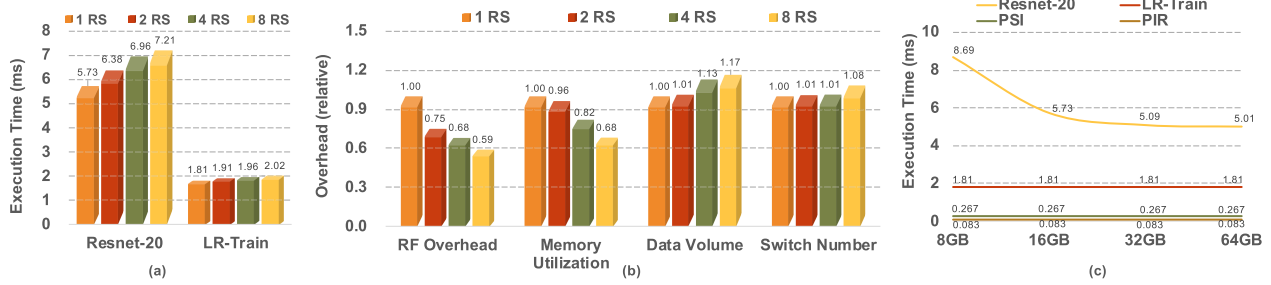


Fig. 10. Analysis of sensitivity. (a) and (b) Explore the granularity of memory management within ResNet-20 and LR-Train. (c) Examines the impact of memory size on the Hypnos for selected benchmarks.

memory. Larger RNS granularity leads to increased memory fragmentation, reducing FHE storage system efficiency and overall performance. Using ResNet-20 as an example, a trend can be observed: increasing the granularity further to $L_{\max}$ would reduce MU, increase data transfer volume and exchange frequency, bringing them back to pre-optimization levels. For applications with large data volumes, increasing off-chip memory from 8 to 16 GB significantly reduces execution time, but the benefit becomes small beyond 16 GB, which is mainly related to the size of the application's working set. Smaller data volume applications are less sensitive to memory size, indicating that memory size allocation should be tailored to meet specific application requirements.

VIII. CONCLUSION

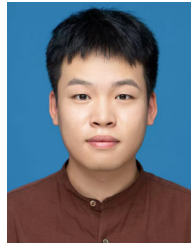
In this article, we prototype the Hypnos. By integrating a CPU, the CU can directly access the homomorphically encrypted data. While specialized hardware like Hypnos offers significant performance potential, efficiently programming these architectures for complex FHE schemes remains a substantial hurdle. Manually optimizing FHE code for such hardware, including managing RNS levels and orchestrating data movement for the HEPMU, is error-prone, time-consuming, and demands deep hardware-specific expertise. Hypnos demonstrates a new paradigm for FHE acceleration, which tightly integrates computation, memory management, and control flow on a single device. This is crucial for overcoming the system-level bottlenecks that have hindered previous solutions.

Although our current work is based on the heterogeneous FPGA platform for prototyping to evaluate the feasibility of actual deployment, the proposed HEPMU and Hypnos compiler principles are not limited to specific devices, such as integrating an open-source soft-core CPU instead of a hard-core ARM CPU in non-SoC platforms. Combining HEPMU with a more powerful ASIC platform is our ultimate goal. We hope our work can inspire new design concepts for domain-specific accelerators targeting FHE applications.

REFERENCES

- [1] C. Gentry, "Fully homomorphic encryption using ideal lattices," in *Proc. forty-first Annu. ACM Symp. Theory Comput.*, May 2009, pp. 169–178.
- [2] K. Han, S. Hong, J. H. Cheon, and D. Park, "Logistic regression on homomorphically encrypted data at scale," *Proc. AAAI Conf. Artif. Intell.*, vol. 33, no. 1, pp. 9466–9471, Jul. 2019.
- [3] H. Chabanne, R. Lescuyer, J. Milgram, C. Morel, and E. Prouff, "Recognition over encrypted faces," in *Proc. 4th Int. Conf. MSPN*, Paris, France, Jun. 2019, pp. 174–191.
- [4] L. Folkerts, C. Gouert, and N. G. Tsoutsos, "REDsec: Running encrypted discretized neural networks in seconds," *Cryptol. ePrint Arch.*, 2021.
- [5] Z. Huang, W.-j. Lu, C. Hong, and J. Ding, "Cheetah: Lean and fast secure two-party deep neural network inference," in *Proc. 31st USENIX Secur. Symp. (USENIX Secur.)*, 2022, pp. 809–826.
- [6] K. Cong, D. Das, J. Park, and H. V. L. Pereira, "SortingHat: Efficient private decision tree evaluation via homomorphic encryption and transpiling," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Nov. 2022, pp. 563–577.
- [7] C. Juvekar, V. Vaikuntanathan, and A. Chandrakasan, "GAZELLE: A low latency framework for secure neural network inference," in *Proc. 27th USENIX Secur. Symp.*, 2018, pp. 1651–1669.
- [8] X. Ren et al., "HEDA: Multi-attribute unbounded aggregation over homomorphically encrypted database," *Proc. VLDB Endowment*, vol. 16, no. 4, pp. 601–614, Dec. 2022.
- [9] K. Matsuoka, R. Banno, N. Matsumoto, T. Sato, and S. Bian, "Virtual secure platform: A five-stage pipeline processor over TFHE," in *Proc. 30th USENIX Secur. Symp. (USENIX Secur.)*, 2021, pp. 4007–4024.
- [10] E. Lee et al., "Low-complexity deep convolutional neural networks on fully homomorphic encryption using multiplexed parallel convolutions," in *Proc. Int. Conf. Mach. Learn.*, 2022, pp. 12403–12422.
- [11] R. Agrawal et al., "FAB: An FPGA-based accelerator for bootstrappable fully homomorphic encryption," in *Proc. IEEE Int. Symp. High-Perform. Comput. Archit. (HPCA)*, Feb. 2023, pp. 882–895.
- [12] Y. Yang, H. Zhang, S. Fan, H. Lu, M. Zhang, and X. Li, "Poseidon: Practical homomorphic encryption accelerator," in *Proc. IEEE Int. Symp. High-Perform. Comput. Archit. (HPCA)*, Feb. 2023, pp. 870–881.
- [13] N. Samardzic et al., "CraterLake: A hardware accelerator for efficient unbounded computation on encrypted data," in *Proc. 49th Annu. Int. Symp. Comput. Archit.*, New York, NY, USA, Jun. 2022, pp. 173–187.
- [14] J. Kim et al., "ARK: Fully homomorphic encryption accelerator with runtime data generation and inter-operation key reuse," in *Proc. 55th IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, Oct. 2022, pp. 1237–1254.
- [15] S. Kim et al., "BTS: An accelerator for bootstrappable fully homomorphic encryption," in *Proc. 49th Annu. Int. Symp. Comput. Archit.*, 2022, pp. 711–725.
- [16] J. Kim, S. Kim, J. Choi, J. Park, D. Kim, and J. H. Ahn, "SHARP: A short-word hierarchical accelerator for robust and practical fully homomorphic encryption," in *Proc. 50th Annu. Int. Symp. Comput. Archit.*, Jun. 2023, pp. 1–15.
- [17] N. Samardzic et al., "F1: A fast and programmable accelerator for fully homomorphic encryption," in *Proc. MICRO-54: 54th Annu. IEEE/ACM Int. Symp. Microarchitecture*, Oct. 2021, pp. 238–252.
- [18] R. Agrawal, L. De Castro, C. Juvekar, A. Chandrakasan, V. Vaikuntanathan, and A. Joshi, "MAD: Memory-aware design techniques for accelerating fully homomorphic encryption," in *Proc. 56th Annu. IEEE/ACM Int. Symp. Microarchitecture*, Oct. 2023, pp. 685–697.
- [19] H. Wang, Y. Yang, H. Lu, and X. Li, "Hypnos: Memory efficient homomorphic processing unit," in *Proc. 62nd ACM/IEEE Design Autom. Conf. (DAC)*, Jun. 2025, pp. 1–7.
- [20] J. H. Cheon, A. Kim, M. Kim, and Y. Song, "Homomorphic encryption for arithmetic of approximate numbers," in *Proc. 23rd Int. Conf. theory Appl. Cryptol. Inf. Secur. Adv. Cryptol. ASIACRYPT*, Hong Kong, 2017, pp. 409–437.
- [21] V. Lyubashevsky, C. Peikert, and O. Regev, "On ideal lattices and learning with errors over rings," in *Proc. 29th Annu. Int. Conf. Theory Appl. Cryptograph. Techn. Adv. Cryptol. EUROCRYPT*, 2010, pp. 1–23.

- [22] J. H. Cheon, K. Han, A. Kim, M. Kim, and Y. Song, “A full RNS variant of approximate homomorphic encryption,” in *Proc. Int. Conf. Sel. Areas Cryptogr.*, vol. 11349, 2019, pp. 347–368.
- [23] M. Kim, D. Lee, J. Seo, and Y. Song, “Accelerating the operations from key decomposition technique,” in *Proc. Annu. Int. Cryptol. Conf.* Cham, Switzerland: Springer, 2023, pp. 70–92.
- [24] H. Hasse, *Number Theory*. Berlin, Germany: Walter de Gruyter GmbH & Co KG, 2022.
- [25] K. Ireland and M. I. Rosen, *A Classical Introduction to Modern Number Theory*, vol. 84. Cham, Switzerland: Springer, 1990.
- [26] J.-C. Bajard, J. Eynard, M. A. Hasan, and V. Zucca, “A full RNS variant of FV like somewhat homomorphic encryption schemes,” in *Proc. Int. Conf. Sel. Areas Cryptogr.*, 2016, pp. 423–442.
- [27] S. Halevi, Y. Polyakov, and V. Shoup, “An improved RNS variant of the BFV homomorphic encryption scheme,” in *Proc. Cryptographers Track RSA Conf.* Cham, Switzerland: Springer, 2019, pp. 83–105.
- [28] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, “(Leveled) fully homomorphic encryption without bootstrapping,” *ACM Trans. Comput. Theory*, vol. 6, no. 3, pp. 1–36, Jul. 2014.
- [29] *Microsoft SEAL*, Microsoft Research, Redmond, WA, USA, Mar. 2022. [Online]. Available: <https://github.com/Microsoft/SEAL>
- [30] J.-W. Lee et al., “Privacy-preserving machine learning with fully homomorphic encryption for deep neural network,” *IEEE Access*, vol. 10, pp. 30039–30054, 2022.
- [31] Y. Lee et al., “HECATE: Performance-aware scale optimization for homomorphic encryption compiler,” in *Proc. IEEE/ACM Int. Symp. Code Gener. Optim. (CGO)*, Apr. 2022, pp. 193–204.
- [32] A. S. Tanenbaum and H. Bos, *Modern Operating Systems*. London, U.K.: Pearson, 2015.
- [33] D. Hankerson, A. J. Menezes, and S. Vanstone, *Guide to Elliptic Curve Cryptography*. Cham, Switzerland: Springer, 2006.
- [34] P. Barrett, “Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor,” in *Proc. Conf. Theory Appl. Cryptograph. Techn.* Cham, Switzerland: Springer, 1986, pp. 311–323.
- [35] R. C. Agarwal and C. S. Burrus, “Number theoretic transforms to implement fast digital convolution,” *Proc. IEEE*, vol. 63, no. 4, pp. 550–560, 1975.
- [36] L. A. Belady, “A study of replacement algorithms for a virtual-storage computer,” *IBM Syst. J.*, vol. 5, no. 2, pp. 78–101, 1966.
- [37] 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 *Proc. 41st ACM SIGPLAN Conf. Program. Lang. Design Implement.*, Jun. 2020, pp. 546–561.
- [38] A. Viand, P. Jattke, and A. Hithnawi, “SoK: Fully homomorphic encryption compilers,” in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2021, pp. 1092–1108.
- [39] A. Viand, P. Jattke, M. Haller, and A. Hithnawi, “HECO: Fully homomorphic encryption compiler,” in *Proc. 32nd USENIX Secur. Symp. (USENIX Secur.)*, 2023, pp. 4715–4732.
- [40] S. Bian et al., “HEIR: A unified representation for cross-scheme compilation of fully homomorphic computation,” *Cryptol. ePrint Arch.*, 2023.
- [41] E. Crockett, C. Peikert, and C. Sharp, “ALCHEMY: A language and compiler for homomorphic encryption made easy,” in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2018, pp. 1020–1037.
- [42] D. W. Archer et al., “RAMPARTS: A programmer-friendly system for building homomorphic encryption applications,” in *Proc. 7th ACM Workshop Encrypted Comput. Appl. Homomorphic Cryptogr.*, Nov. 2019, pp. 57–68.
- [43] L. Li et al., “ANT-ACE: An FHE compiler framework for automating neural network inference,” in *Proc. 23rd ACM/IEEE Int. Symp. Code Gener. Optim.*, Mar. 2025, pp. 193–208.
- [44] V. A. Alfred, S. L. Monica, and D. U. Jeffrey, *Compilers Principles, Techniques & Tools*. London, U.K.: Pearson Education, 2007.
- [45] *Microsoft APSI*, Microsoft Research, Redmond, WA, USA, Dec. 2023. [Online]. Available: <https://github.com/microsoft/apsi>
- [46] *Microsoft SealPIR*, Microsoft Research, Redmond, WA, USA, Jun. 2023. [Online]. Available: <https://github.com/microsoft/SealPIR>



Haoxuan Wang is currently pursuing the Ph.D. degree with the Institute of Computing Technology (ICT), Chinese Academy of Sciences (CAS), University of Chinese Academy of Sciences (UCAS), Beijing, China.

His main research focuses on homomorphic encryption accelerators, confidential computing, and computer architecture.



Yinghao Yang received the Ph.D. degree in computer science from the Institute of Computing Technology (ICT), Chinese Academy of Sciences (CAS), Beijing, China, in 2025.

He is currently an Assistant Researcher with the State Key Laboratory of Processors, ICT, CAS. His research interests include privacy-preserving computing acceleration, fully homomorphic encryption (FHE) hardware acceleration, and trusted execution environment (TEE).



Shangjie Pan is currently pursuing the Ph.D. degree with the Institute of Computing Technology (ICT), Chinese Academy of Sciences (CAS), University of Chinese Academy of Sciences (UCAS), Beijing, China.

His research interests include hardware security, system security, and trusted execution environments.



Hang Lu received the Ph.D. degree in computer science from the Institute of Computing Technology (ICT), Chinese Academy of Sciences (CAS), Beijing, China, in 2015.

He is currently a Professor with ICT, CAS, University of Chinese Academy of Sciences (UCAS), Beijing. His research interests include FHE acceleration, FPGA accelerator design, and a trusted execution environment.



Xiaowei Li (Senior Member, IEEE) received the Ph.D. degree in computer science from the Institute of Computing Technology (ICT), CAS, Beijing, China, in 1991.

In 2000, he joined ICT, CAS, as a Professor. His current research interests include VLSI testing, design for testability, design verification, dependable computing, and wireless sensor networks.