SoK: Understanding Virtual Machines over Fully Homomorphic Encryption

Song Bian*, Ziwei Ma*, Yunhao Fu*, Haowen Pan*, Ruiyu Shen*, Zhou Zhang*,
Ran Mao†, Zaidong Li*, Yintai Sun*, Ruoyi Li*, Zhenyu Guan*✉
* *School of Cyber Science and Technology, Beihang University,*
Email: {sbian, maziwei, fyhssgss, panhaowen, ruiyushen}@buaa.edu.cn
{zhouzhang, zaidongli, yintaisun, liruoyi, guanzhenyu}@buaa.edu.cn
† *School of Cyber Science and Technology, Beihang University, Email: mr.lavendes@gmail.com*

***Abstract*—We survey the current literature on fully homomorphic encryption (FHE) from a virtual machine (VM) perspective. We observe that programs behave in an intrinsically different way over FHE than on plaintext, forming a new paradigm of computation that is fundamentally constrained by the obliviousness of data. To better understand the computational properties, we compile more than 125 works to formulate a cross-layer systematic view of FHE VMs, with a special focus on understanding the concrete programming models over encrypted data. In particular, we identify three distinct forms of VMs over FHE, namely, the SISD VM, the SIMD VM, and the hybrid VM. To aid the cross-layer systematization, we also discuss how current FHE compilers and hardware accelerators can be applied to enhance the usability and performance of the respective VMs. In the experiment, we test eight VM instructions with varying precision as well as two end-to-end application programs to quantitatively assess the concrete design trade-offs for each of the VM paradigms. We conclude that FHE VMs of different types exhibit distinct utility and performance characteristics, necessitating new design methodologies to be developed in a cross-layer manner.**

1. Introduction

Over the past decade and a half, fully homomorphic encryption (FHE) has progressed from a theoretical proof of concept to an active area of cryptographic research, marked by ongoing standardization efforts [1, 2] and growing practical relevance [3–7]. First proven to exist by [8], FHE provides provable security in allowing general computations to be executed over encrypted data without the need for the decryption keys. Based on this property, the cryptographic primitive is found useful in building a number of advanced cryptographic primitives [9, 10], secure multi-party computation protocols [11–22], and end-to-end privacy-preserving applications [14, 16, 20, 23–32].

While FHE is generally applicable in theory, due to the oblivious nature of data encryption, computational tasks over FHE have drastically different design trade-offs from those of plaintext computation. For example, below a certain data precision threshold, it is known that arithmetic circuits are easier to evaluate than non-algebraic operators over FHE [16, 33–35]. Next, at the cryptographic primitive layer, certain types of computations (e.g., ReLU [36], saturation [37]) are shown to be considerably easier to implement over FHE than others (e.g., floor functions [38–40], comparisons [31, 32, 41]). Combined with the intricate nature of real-world applications, the diverse computational characteristics of different FHE schemes and operators result in an extremely high level of algorithmic design complexity.

To aid the complicated algorithmic design process, a body of literature has emerged to systematize various aspects of FHE. For example, existing works [42–46] study how FHE can be used in applications such as privacy-preserving machine learning, privacy-preserving data analytics, and transciphering. Meanwhile, other surveys include reviews of encryption schemes [47, 48], cryptographic optimizations [49, 50], applications [51], compilers [52], and hardware accelerators [53]. While such works provide close-up inspections of individual research areas, FHE practitioners may still struggle to determine the exact encryption schemes and computation models to implement a particular class of tasks.

Motivation: Despite the above progress, we point out two key properties that need to be carefully examined to fully comprehend how programs behave over FHE:

(a) **Programming Models**: FHE has distinct computational characteristics at both the data and the control levels. On the data level, the discrete nature of FHE ciphertext results in both complex encoding types and inevitable computation errors for continuous data types in real-world applications. On the control level, conditional statements become much more expensive to evaluate, leading to higher design complexity of branching and looping structures. Hence, a combined systematization of the concrete programming models is desired for the development of programs over FHE.

(b) **Performance Trade-offs**: We observe that the efficiency of FHE is tightly coupled with the programming models, the underlying cryptographic primitives, the compilation support, and the low-level hardware platforms. As later demonstrated in Section 7, the end-to-end performance of FHE protocols varies significantly across different FHE schemes, and is also highly dependent on whether designated compilers and hardware accelerators are

available. Therefore, to optimize code execution over FHE, a cross-layer systematization is critical for improving the overall execution efficiency.

To address the above challenges, virtual machines (VMs) over FHE emerge as an effective solution to bridge the gap between high-level program behaviors and low-level cryptographic constructions. Unfortunately, due to the lack of systematic characterizations of the computation models, the utility and performance trade-offs between different FHE VM constructions remain obscure. Thus, we believe that there is a need for a new systematization framework for FHE VMs on different layers of abstraction. Our main contributions are as follows.

- **Defining Computation Models**: To the best of our knowledge, we are the first to formalize the notion of a VM over FHE. We specify the main components of FHE VMs based on the random-access machine model, and exemplify concrete instantiations of VMs based on different types of FHE schemes and operators.
- **Linking Cross-Layer Design Methodologies**: We focus on surveying literature related to FHE from a cross-layer perspective with a systematization strategy centered around the construction of FHE VMs. More specifically, we consolidate design techniques and optimization methods for FHE based on how they can be applied to build better FHE VMs in terms of the utility, latency and security metrics.
- **Benchmarking VMs**: We provide concrete evaluations of the performance of various FHE VMs over a number of representative workloads with varying data precision. The results show that VMs instantiated over different FHE schemes exhibit drastically different performance. Moreover, we observe that distinct ways of implementing the same type of VM can also have a significant impact on the practical efficiency of FHE VMs, especially on end-to-end applications.

This paper is organized as follows. First, in Section 2, we outline the basics and detail an abstraction method for FHE schemes. Second, in Section 3 and Section 4, we formalize the notion of an FHE VM and present a concrete instantiation of the general architecture, and then categorize FHE VMs into SISD, SIMD, and hybrid VMs with their corresponding programming models. Third, the current compilation and hardware acceleration techniques that can be applied to FHE VMs are discussed in Section 5 and Section 6, respectively. Fourth, we benchmark a number of tasks over FHE VMs in Section 7. Lastly, we provide discussions and future directions for FHE VMs in Section 8.

2. Abstracting FHE Schemes

In this section, we first introduce the notations used throughout this work in Section 2.1. We then describe the workflow of a typical round of an FHE protocol, along with the abstraction and categorization of basic FHE schemes in Section 2.2.

2.1. Notations

We use λ to denote the security level, p the plaintext modulus, and q/Q the ciphertext moduli. $\mathbb{Z}_q$ denotes the set of integers modulo q, and $\mathbb{N}$ is the set of natural numbers. For some lattice dimension n/N, we write $R_{N,q} = \mathbb{Z}_q[X]/(X^N + 1)$ for the ring of cyclotomic polynomials modulo $X^N + 1$. We use tilde lower-case letters with square brackets such as $\tilde{a}[X]$ to refer to polynomials in the variable X, and bold lower-case letters such as $\mathbf{a}$ to depict vectors. $\mathbf{a}[i]$ denotes the i-th element in the vector $\mathbf{a}$.

2.2. Scheme Abstraction and Basic Protocol

Let $\mathcal{E}$ be a general FHE scheme where $\mathcal{E} = (\textsc{KeyGen}, \textsc{Encrypt}, \textsc{Decrypt}, \textsc{Ops})$. Existing works have categorized FHE schemes from different perspectives, including generation-based classifications [47, 52, 67–71] and circuit-oriented organizations [53, 72–77]. In contrast, we abstract FHE schemes according to their computation paradigms, namely, single-instruction single-data (SISD) and single-instruction multiple-data (SIMD) schemes. Before introducing the concrete abstractions, we first summarize the basic workflow of an FHE protocol, which consists of the following four steps:

- **Key Generation**: The user runs $(pk, sk) = \textsc{KeyGen}(1^\lambda)$ to obtain a public key pk and a secret key sk with λ-bit security. The public key is shared with the server, while the secret key is kept private.
- **Encryption and Data Upload**: Given some abstract message msg, the user first performs plaintext encoding, and then executes the encryption function to produce a set of ciphertexts $\mathsf{ct}_{\text{in}}^{key} \leftarrow \textsc{Encrypt}(msg, key)$. Note that in an FHE scheme, $key \in \{pk, sk\}$ can be either pk or sk depending on the exact application. Subsequently, the ciphertexts $\mathsf{ct}_{\text{in}}^{key}$ are sent to the server.
- **Homomorphic Evaluation**: Upon receiving ct_{in}, the server applies some function $\mathcal{F}$ directly on ciphertexts, computing $\mathsf{ct}_{\text{out}} \leftarrow \textsc{HomEval}(\mathcal{F}, \mathsf{ct}_{\text{in}})$. The homomorphic property of $\textsc{Encrypt}$ ensures that
$$\mathsf{ct}_{\text{out}}^{key} = \mathcal{F}(\mathsf{ct}_{\text{in}}^{key}) = \textsc{Encrypt}_{key}(f(msg)), \quad (1)$$
while revealing nothing about the plaintexts per se or any intermediate values.
- **Decryption and Result Recovery**: The server returns $\mathsf{ct}_{\text{out}}^{key}$ to the user, who decrypts using sk to recover $f(msg) \leftarrow \textsc{Decrypt}_{sk}(\mathsf{ct}_{\text{out}}^{key})$.

Remark on Multi-Party FHE Schemes: Conventional FHE supports computations only on single-key ciphertexts. To apply FHE in a multi-party computation scenario, we can make use of the threshold FHE (ThFHE) or Multi-Key FHE (MKFHE) schemes. Most standard FHE schemes can be extended to ThFHE [78–81] by using distributed key generation protocols, where each participant holds a private key share to form a joint public key. Some schemes can also be extended to support MKFHE schemes [82–87], enabling homomorphic operations on ciphertexts directly encrypted

TABLE 1: Comparison of Operator Support Across Selected FHE Constructions

Scheme	Scheme Type	$+, -$ and $\cdot$	Key Switching	Func. Boot.	Multiplexer	Poly. Eval.	Ring Product	Automorphism	Boot. Thpt.	Boot. Lat.
AP/DM [54, 55]	SISD	●	●	●	●	○	○	○	Low	Low
TFHE/GINX [56, 57]	SISD	●	●	●	●	○	○	○	Low	Low
BFV [58, 59]	SIMD	●	●	●*	●$^\diamond$	●	●	●	High	High
BGV [60]	SIMD	●	●	●*	●$^\diamond$	●	●	●	High	High
CKKS [61]	SIMD	●	●	●*	●$^\diamond$	●	●	●	High	Vary†
GL [62]	SIMD	●	●	○	●$^\diamond$	●	●	●	High	Vary†

In the table, Func. Boot. denotes functional bootstrapping, either SIMD or SISD. Poly. Eval. refers to homomorphic polynomial evaluation. Boot. Thpt. and Boot. Lat. represent the bootstrapping throughput and latency of the corresponding FHE schemes.
*Functional bootstrapping is not natively supported by the original schemes. It can be realized through extended constructions, including relaxed functional bootstrapping for BFV/BGV [63] and general functional bootstrapping for CKKS [39].
$^\diamond$Multiplexers can generally be realized by composing basic homomorphic operators or Boolean/arithmetic circuits. Beyond such generic constructions, recent studies have further improved the efficiency of encrypted conditional selection for BFV/BGV [64] and CKKS [41, 65].
†Recently, [66] substantially reduced the latency of CKKS bootstrapping to around one second per ciphertext. Consequently, CKKS bootstrapping latency can range from approximately one second to tens of seconds, and is therefore marked as Vary.

under different keys. Additionally, certain schemes combine both multi-key and threshold properties [88, 89]. Most of the abstractions and formulations systematized in this work extend naturally to both ThFHE and MKFHE schemes.

Remark on Other FHE Schemes: We note that other FHE schemes have been proposed based on different hardness assumptions and/or mathematical constructions [90–94]. However, since the VM infrastructures and ecosystems of these schemes are still in the early stages of development, we do not include them in the scope of this paper.

2.2.1. The SISD-FHE Schemes. As shown in Table 1, SISD-FHE schemes encrypt and process a single plaintext per ciphertext, thereby performing homomorphic computation, such as addition, subtraction, constant multiplication, and multiplexer operations in a scalar manner. This design leads to low computational latency, making SISD-FHE schemes well-suited for latency-sensitive applications. For conventional lattice-based SISD-FHE schemes such as AP/DM [54, 55] and TFHE/GINX [56, 57], given a scalar plaintext message $m \in \mathbb{Z}$, the SISD instantiation of $\mathcal{E}$ performs encryption as

$$\mathsf{ct}_{\mathrm{SISD}}(m) = \text{SISD-}\textsc{Encrypt}(m), \tag{2}$$

and we have that $m = \text{SISD-}\textsc{Decrypt}(\mathsf{ct}_{\mathrm{SISD}}(m))$.

SISD Operators: As summarized in Table 1, SISD-OPS consists of the basic additions $(+)$, subtractions $(-)$, and constant multiplications $(\cdot)$ operators, where we have

$$\mathsf{ct}_{\mathrm{SISD}}(m_0 \pm m_1) = \mathsf{ct}_{\mathrm{SISD}}(m_0) \pm \mathsf{ct}_{\mathrm{SISD}}(m_1), \text{ and} \tag{3}$$

$$\mathsf{ct}_{\mathrm{SISD}}(m_0 \cdot m_1) = \mathsf{ct}_{\mathrm{SISD}}(m_0) \cdot m_1 \tag{4}$$

where $\pm$ denotes either an addition or a subtraction operator and two plaintext messages $m_0, m_1 \in \mathbb{Z}$. Note that most existing SISD-FHE schemes do not support direct multiplications between $\mathsf{ct}_{\mathrm{SISD}}$ ciphertexts[1] (i.e., the RINGPRODUCT operator), and thereby lack the ability to evaluate high-degree polynomials over encrypted data. On the other hand, most SISD-FHE schemes support three advanced operators SISD-MUX, KEYSWITCH and SISD-FBS. First, for the homomorphic multiplexer (SISD-MUX) operator, given a selection index $i \in \{0, 1\}$ and two plaintexts m_0 and m_1, we have that

$$\mathsf{ct}_{\mathrm{SISD}}(i \ ?\ m_0 : m_1) = \\ \text{SISD-MUX}(\mathsf{ct}_{\mathrm{KEY}}(i), \mathsf{ct}_{\mathrm{SISD}}(m_0), \mathsf{ct}_{\mathrm{SISD}}(m_1)), \tag{5}$$

where $\mathsf{ct}_{\mathrm{KEY}}$ is a special type of ciphertext that is used to encrypt key material [56]. Second, the homomorphic key switching operator KEYSWITCH changes the encryption key of a ciphertext from one key to another, defined as

$$\mathsf{ct}^{k_1}_{\mathrm{SISD}}(m) = \text{SISD-KS}(\mathsf{ct}^{k_0}_{\mathrm{SISD}}(m), \mathsf{ct}_{\mathrm{KEY}}(k_1)), \tag{6}$$

for two (potentially equal) encryption keys k_0 and k_1. Lastly, the functional bootstrapping (SISD-FBS) is defined as

$$\mathsf{ct}_{\mathrm{SISD}}(f(m)) = \text{SISD-FBS}(\mathsf{ct}_{\mathrm{SISD}}(m), f), \tag{7}$$

for some function f. The most important property for SISD-FHE schemes is that, in most cases, the SISD-FBS operator can be executed in an extremely low-latency manner, enabling fast end-to-end evaluation of small-scale circuits [96–98].

2.2.2. The SIMD-FHE Schemes. The ring variants of the BFV [58, 59], BGV [60], CKKS [61] and GL [62] cryptosystems are the most commonly used SIMD-FHE schemes. As shown in Table 1, in contrast to SISD-FHE schemes, the main feature of SIMD-FHE schemes is their ability to pack a vector of plaintext messages into a single ciphertext. Thus, the key distinction between SIMD-FHE and SISD-FHE schemes lies in that operations on SIMD-FHE ciphertexts are amortized across all embedded plaintext elements within the ciphertext. Given a plaintext vector $\mathbf{m} \in \mathbb{Z}^N$, the SIMD instantiation of $\mathcal{E}$ performs encryption as

$$\mathsf{ct}_{\mathrm{SIMD}}(\mathbf{m}) = \text{SIMD-}\textsc{Encrypt}(\mathbf{m}), \tag{8}$$

1. While DM/CGGI-like cryptosystems can perform the internal product operator [55, 56, 95], the RGSW ciphertext is not the default ciphertext type used in such cryptosystems.

where the original plaintext can be recovered as $\mathbf{m} = \text{SIMD-DECRYPT}(\mathsf{ct}_{\text{SIMD}}(\mathbf{m}))$.

SIMD Operators: Similar to the SISD-FHE schemes, most SIMD-FHE schemes also support the homomorphic addition, subtraction and constant multiplication operators. The only difference is that such operators simultaneously act on a vector of plaintexts. In addition, as shown in Table 1, three distinct operators are associated with SIMD-FHE schemes: the ciphertext-ciphertext multiplication operator RINGPRODUCT [58–61], the homomorphic permutation operator AUTOMORPHISM [33, 34, 61, 99], and the homomorphic polynomial evaluation operator HOMPOLY. Specifically, given two plaintext message vectors $\mathbf{m}_0$ and $\mathbf{m}_1$, we have that

$$\begin{aligned}&\mathsf{ct}_{\text{SIMD}}(\mathbf{m}_0 \cdot \mathbf{m}_1) = \\ &\text{RINGPRODUCT}(\mathsf{ct}_{\text{SIMD}}(\mathbf{m}_0), \mathsf{ct}_{\text{SIMD}}(\mathbf{m}_1)),\end{aligned} \tag{9}$$

where the definition of $\cdot$ in Eq. (9) can vary depending on the exact plaintext encoding [16, 33, 100] (also refer to Appendix C). Similarly, given some permutation π_ι and some plaintext message $\mathbf{m} \in \mathbb{Z}^N$, let $\pi_\iota(\mathbf{m}) = [m_{\iota(0)} \ \cdots \ m_{\iota(N-1)}]$ denote the message permuted under π_ι. Then, it holds that

$$\begin{aligned}\mathsf{ct}_{\text{SIMD}}(\pi_\iota(\mathbf{m})) &= \mathsf{ct}_{\text{SIMD}}([m_{\iota(0)} \ \cdots \ m_{\iota(N-1)}]) \\ &= \text{AUTOMORPHISM}(\mathsf{ct}_{\text{SIMD}}(\mathbf{m}), \iota).\end{aligned} \tag{10}$$

Lastly, for the homomorphic polynomial operator, let $p[X] = \sum_{i=0}^{\ell-1} a_i X^i$ be an ℓ-term univariate polynomial, we write

$$\begin{aligned}\mathsf{ct}_{\text{SIMD}}(p(\mathbf{m})) &= \mathsf{ct}_{\text{SIMD}}\left(\sum_{i=0}^{\ell-1} a_i \mathbf{m}^i\right) \\ &= \text{HOMPOLY}(\mathsf{ct}_{\text{SIMD}}(\mathbf{m}), p[X]),\end{aligned} \tag{11}$$

where $\mathbf{m}^i = (m_0^i, \ldots, m_{N-1}^i)$ denotes the i-th Hadamard power of $\mathbf{m}$.

Besides the scheme-specific operators, we also see recent progress in developing functional bootstrapping algorithms for SIMD-FHE schemes [39, 101–103], where a function can be evaluated over a vector of plaintexts, i.e.,

$$\mathsf{ct}_{\text{SIMD}}(f(\mathbf{m})) = \text{SIMD-FUNCBOOT}(\mathsf{ct}_{\text{SIMD}}(\mathbf{m}), f). \tag{12}$$

Note that f acts on $\mathbf{m}$ in a coefficient-wise manner. Additionally, any SIMD-FHE scheme can be made to support the multiplexer operator in a SIMD manner by computing

$$\begin{aligned}&\mathsf{ct}_{\text{SIMD}}(i \ ? \ \mathbf{m}_0 : \mathbf{m}_1) = \\ &\text{RINGPRODUCT}(\mathbf{1} - \mathsf{ct}_{\text{SIMD}}(\mathbf{i}), \mathsf{ct}_{\text{SIMD}}(\mathbf{m}_0)) + \\ &\text{RINGPRODUCT}(\mathsf{ct}_{\text{SIMD}}(\mathbf{i}), \mathsf{ct}_{\text{SIMD}}(\mathbf{m}_1)),\end{aligned} \tag{13}$$

where $\mathbf{i} = \mathbf{0}$ when $i = 0$, and $\mathbf{i} = \mathbf{1}$ when $i = 1$.[2]

2. Here, $\mathbf{0} \in \mathbb{Z}^N$ is a vector of all zeroes, and $\mathbf{1} \in \mathbb{Z}^N$ is a vector of all ones.

3. Systematizing VMs over FHE

In this section, we first provide a formal definition of FHE VMs based on the general computational model. We then analyze the security properties of the model from the perspectives of both FHE security and VM control-flow security. Finally, we provide concrete instantiations for each of the FHE VM types with respect to the security models and data processing paradigms.

3.1. Overview of FHE VM

Figure 1 illustrates the overall architecture of an FHE VM and the execution flow of encrypted programs. The FHE VM provides a virtualized execution layer that enables encrypted applications to be executed through a uniform set of homomorphic instructions and runtime components, effectively abstracting the complexity of the underlying cryptographic schemes. The entire workflow of the FHE VM is composed of three interlinked stages: ① program compilation, ② program initialization, and ③ homomorphic execution. In what follows, we detail the main objective of each stage and briefly sketch the high-level VM components.

① **Program Compilation**: First, the program compilation stage translates a program written in a high-level language into low-level procedures acting on cryptographic objects through the VM compiler. During compilation, the program logic is mapped onto a sequence of standardized instructions selected from the instruction set $\mathcal{I}$ of the target VM. To support diverse computational workloads, most existing FHE compilers adopt a multi-level intermediate representation architecture along with a rich instruction set to optimize the compiled program from an FHE perspective. This stage abstracts hardware-oriented homomorphic circuits into a uniform instruction stream, significantly improving programmability and portability. More details on FHE compilers are provided in Section 5, while those on the formal definition of $\mathcal{I}$ are given in Section 3.2.

② **Execution Initialization**: The VM establishes its execution environment by configuring the FHE scheme $\mathcal{E}$ and initializing the memory system $\mathcal{M}$. First, the compiled instruction stream is loaded into the program memory $\mathcal{M}_{\text{P}}$. The VM then asks the user to produce the evaluation keys and ciphertext inputs that are necessary for running the program. Upon receiving the encrypted data from the user, the input ciphertexts are used to initialize the computation memory $\mathcal{M}_{\text{C}}$, while the evaluation keys (e.g., the relinearization, automorphism and bootstrapping keys) are stored in the key memory $\mathcal{M}_{\text{K}}$. Lastly, the VM instantiates its state space $\mathcal{Q}$ and enters the designated start state $\mathcal{Q}_0 \in \mathcal{Q}$, indicating that the runtime system is ready for execution.

③ **Homomorphic Execution**: The execution engine repeatedly invokes the homomorphic execution cycle δ to process the instruction stream stored in $\mathcal{M}_{\text{P}}$. During each cycle, δ fetches an instruction that operates on the data stored in $\mathcal{M}_{\text{C}}$ and produces a new machine state in $\mathcal{Q}$ that is written to the memories of the VM. Execution continues until a state in $\perp$ is reached. When evaluating deep circuits, δ

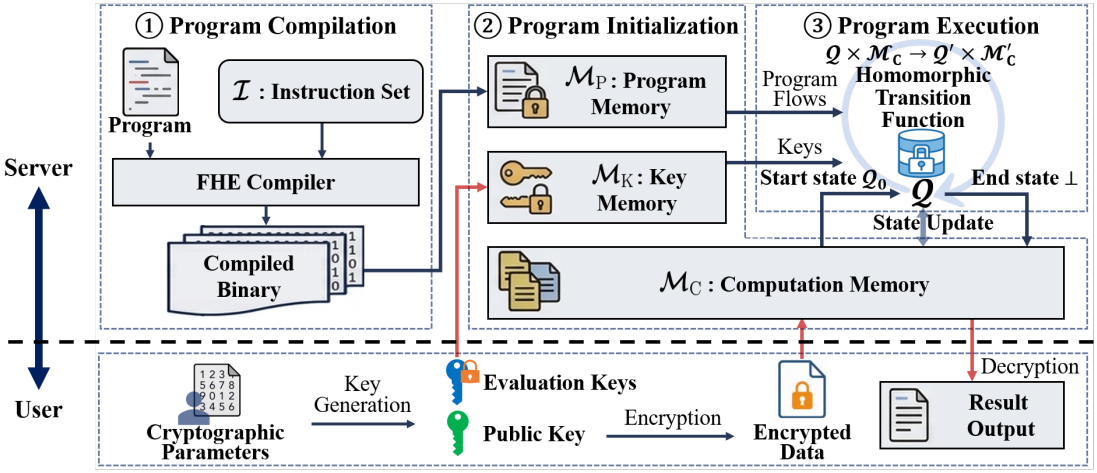


Figure 1: A round of program execution over an FHE VM.

transparently invokes noise-management primitives provided by $\mathcal{E}$, such as bootstrapping, to maintain the correctness of FHE evaluation.

Remark on control-flow security: We note that depending on whether the states in $\mathcal{Q}$ are maintained in plaintext or ciphertext, the FHE VM can be instantiated as either a transparent VM or an oblivious VM. A transparent VM is an FHE VM whose states in $\mathcal{Q}$ are maintained in plaintext, such that its control flow is public and ciphertext-independent. In contrast, an oblivious VM is an FHE VM whose states in $\mathcal{Q}$ are maintained entirely in encrypted form. Due to space limitations, the formal definitions of transparent and oblivious VMs are deferred to Appendix F.

3.2. Formalizing FHE VM

Here, we provide a formal execution model that captures the architecture and semantics of the FHE VM outlined in Section 3.1. An FHE VM is defined by a seven-tuple that describes its architecture and states:

$$\mathsf{FHEVM} = (\mathcal{E}, \mathcal{I}, \mathcal{M}, \mathcal{Q}, \mathcal{Q}_0, \perp, \delta). \tag{14}$$

In what follows, we provide the details for each of the components, while the concrete model instantiations are given in Appendix E.

- $\mathcal{E}$: An FHE scheme, comprising a tuple of algorithms ($\textsc{KeyGen}, \textsc{Encrypt}, \textsc{Decrypt}, \textsc{Ops}$), which defines the underlying cryptographic primitives utilized by the VM instructions.
- $\mathcal{I}$: The set of instructions of the VM, which is a finite set of atomic operations executable by the VM. Such operations are broadly classified into three primary categories, analogous to a traditional ALU: arithmetic operations (e.g., addition, multiplication), logical operations (e.g., AND, OR, and NOT), and bitwise operations (e.g., left and right shifts). Each instruction is implemented using the underlying operators provided by $\mathcal{E}$. A detailed specification of the instructions and their corresponding descriptions is presented in Appendix D.
- $\mathcal{M}$: The memory system, which represents the complete memory architecture of the VM, is defined as a tuple of three independent, addressable memory spaces:

$$\mathcal{M} = (\mathcal{M}_{\mathrm{P}}, \mathcal{M}_{\mathrm{C}}, \mathcal{M}_{\mathrm{K}}). \tag{15}$$

  Specifically, the program memory ($\mathcal{M}_{\mathrm{P}}$) is a read-only memory that holds the instructions of the given program. The computation memory ($\mathcal{M}_{\mathrm{C}}$) is a random-access memory, containing all data read and stored during program execution, including ciphertexts for private inputs and outputs, plaintexts for public values, and all intermediate results. The key memory ($\mathcal{M}_{\mathrm{K}}$) is a read-only memory that stores the evaluation keys required by the homomorphic operators, such as the relinearization keys and the bootstrapping keys.
- $\mathcal{Q}$: A finite set of machine states. Whether the machine states in $\mathcal{Q}$ are represented in plaintext or ciphertext determines a key security property of the specific VM model, which is detailed later in Section 3.3.
- $\mathcal{Q}_0 \in \mathcal{Q}$: The start state.
- $\perp \subseteq \mathcal{Q}$: The set of final states.
- δ: The homomorphic execution cycle (transition function) for executing one instruction, which governs the state transition of the FHE VM. For $\mathtt{q} \in \mathcal{Q}$,

$$\delta(\mathtt{q}, \mathcal{M}_{\mathrm{C}}, \mathtt{i}, \mathcal{M}_{\mathrm{K}}) = (\mathtt{q}', \mathcal{M}'_{\mathrm{C}}), \tag{16}$$

  where $\mathtt{i} \in \mathcal{I}$ is fetched from $\mathcal{M}_{\mathrm{P}}$, $\mathcal{M}_{\mathrm{K}}$ stores the evaluation keys, and $\mathtt{q}' \in \mathcal{Q}$ and $\mathcal{M}'_{\mathrm{C}}$ are the updated state and computation memory, respectively.

Remark on Parameter Instantiation: Any concrete instantiation of an FHE VM needs a specific set of system-level parameters. The parameters include the security level λ and the machine word size ω. In particular, the word size ω defines the fundamental unit for plaintext data within the VM architecture, dictating the width of registers and the operands for instructions. Consequently, the values encrypted within

the underlying FHE scheme $\mathcal{E}$ are also processed at an ω-bit precision. If a computation requires precision exceeding ω, it must be implemented by segmenting the high-precision data across multiple instructions.

3.3. Security Models for FHE VM

We analyze the security of an FHE VM from two distinct perspectives: the FHE ciphertext and the executed program. From the perspective of the FHE ciphertext, security is primarily defined against the traditional adversary models. Most FHE schemes are designed to operate within the semi-honest security model [104], which ensures data confidentiality under the assumption that all parties follow the protocol. However, the semi-honest model does not inherently protect the ciphertext against malicious adversaries who may deviate from the protocol. Ensuring computational integrity against such threats often requires supplementary techniques, such as zero-knowledge proofs [105], to verify that the computation was performed correctly.

From the perspective of the executed program, the security model is defined by the control flow confidentiality of the program. Here, we make a distinction between programs with and without ciphertext-dependent termination conditions. For programs that do not contain ciphertext-dependent termination conditions, the entire control flow of the program can be made public, since all data-dependent conditional branches can be completely unrolled. The unrolling procedure can be easily performed by evaluating all the possible branches and homomorphically extracting the correct control path using a homomorphic multiplexer. Similarly, loops with public, data-independent bounds can be fully unrolled at compile time into a fixed sequence of instructions (corresponding to the transparent loops in [106]). Conversely, if the termination of a program depends on an encrypted value, the computational path cannot be fully unrolled at compile time or at execution time, due to the nature of data encryption. For instance, in a loop that continues until an encrypted condition is met, the total number of iterations inevitably leaks information about the plaintext data of the encrypted condition. Thus, such programs cannot be transformed into a deterministic execution trace without client-server interaction along with additional security measures.

3.4. Instantiating FHE VMs

Here, we provide a concrete instantiation of the abstract FHE VM model defined in Section 3.2 that serves as a general realization of the abstract VM formalization. The instantiation establishes a unified execution framework that serves as the foundation for high-level computations to be systematically translated into the corresponding homomorphic operators over different FHE schemes. The concrete instantiation of each component is given as follows:

- **Encryption Schemes**: $\mathcal{E}$ is a unified encryption scheme that supports both SISD and SIMD FHE functionalities defined as

  $$\mathcal{E} = (\text{SISD-KeyGen}, \text{SISD-Encrypt}, \text{SISD-Decrypt}, \text{SISD-Ops}, \text{SIMD-KeyGen}, \text{SIMD-Encrypt}, \text{SIMD-Decrypt}, \text{SIMD-Ops}).$$

  In other words, under such an instantiation, we can encrypt either a single plaintext message using SISD-Encrypt or a vector of plaintexts using SIMD-Encrypt.
- **Instructions**: $\mathcal{I}$ contains three classes of VM instructions: (i) scalar instructions $\mathcal{I}_{\text{SISD}}$ operating on SISD ciphertexts, (ii) vector instructions $\mathcal{I}_{\text{SIMD}}$ operating on SIMD ciphertexts, and (iii) scheme-switching instructions $\mathcal{I}_{\text{Switch}}$ for converting between scalar and vector ciphertext representations. Due to space limitations, we only give a brief summary of the instruction sets, while the formal details are presented in Appendix D. The scalar instruction set includes arithmetic operations, logical operations and bitwise operations, given by

  $$\mathcal{I}_{\text{SISD}} = \{\texttt{H_SCA_ADD}, \texttt{H_SCA_SUB}, \texttt{H_SCA_MUL}, \texttt{H_SCA_AND}, \texttt{H_SCA_OR}, \texttt{H_SCA_NOT}, \texttt{H_SCA_SHL}, \texttt{H_SCA_SHR}\}.$$

  The vector instruction set includes packed operations, covering arithmetic operations, logical operations, bitwise operations and data-shuffling operations, which are summarized as

  $$\mathcal{I}_{\text{SIMD}} = \{\texttt{H_VEC_ADD}, \texttt{H_VEC_SUB}, \texttt{H_VEC_MUL}, \texttt{H_VEC_AND}, \texttt{H_VEC_OR}, \texttt{H_VEC_NOT}, \texttt{H_VEC_SHL}, \texttt{H_VEC_SHR}, \texttt{H_ROTATE}\}.$$

  The scheme-switching instruction set enables conversions between different data representations and supports vectorization of function evaluation results, i.e.,

  $$\mathcal{I}_{\text{Switch}} = \{\texttt{H_PACK}, \texttt{H_EXTRACT}, \texttt{H_FUN_PACK}\}.$$
- **Memory**: As mentioned above, $\mathcal{M} = (\mathcal{M}_{\text{P}}, \mathcal{M}_{\text{C}}, \mathcal{M}_{\text{K}})$ supports both scalar and vector ciphertext representations. Here, we provide a detailed description of the components and functionalities of each memory space. The program memory $\mathcal{M}_{\text{P}}$ stores the execution sequence of a program, which is defined as

  $$\mathcal{M}_{\text{P}} = (\mathtt{i}_0, \mathtt{i}_1, \ldots, \mathtt{i}_n) \in \mathcal{I}^*,$$

  where each instruction $\mathtt{i}_j \in \mathcal{I}$ represents an executable operation, and the order of instructions determines the execution sequence of the program. The computation memory $\mathcal{M}_{\text{C}}$ stores all operands and intermediate states generated during execution, given by

  $$\mathcal{M}_{\text{C}} = \{\mathsf{ct}_{\text{SISD}}, \mathsf{ct}_{\text{SIMD}}, \mathsf{pt}_{\text{const}}, \mathsf{r}_{\text{tmp}}\}^*.$$

  Here, $\mathsf{pt}_{\text{const}}$ represents plaintext constants and r_{tmp} denotes temporary values produced during execution. Meanwhile, $\mathcal{M}_{\text{K}}$ stores the evaluation keys required by

both the SISD and SIMD subsystems, including public keys, key-switching keys, relinearization keys, Galois keys, and bootstrapping keys.

- **States**: In general, $\mathcal{Q}$ consists of states for ciphertext initialization, homomorphic evaluation, encrypted condition handling, control-flow resolution, noise management, and execution termination, i.e., $\mathcal{Q} = \{\texttt{INIT}, \texttt{EVAL}, \texttt{COND}, \texttt{BRANCH}, \texttt{REFRESH}, \texttt{OUTPUT}, \texttt{ACCEPT}, \texttt{REJECT}\}$, where the start state is defined as $\mathcal{Q}_0 = \texttt{INIT}$, while the set of final states is $\perp = \{\texttt{ACCEPT}, \texttt{REJECT}\}$. Unlike classical virtual machines, the state space of an FHE VM additionally captures encrypted control-flow evaluation and ciphertext lifecycle management. Specifically, `INIT` initializes the encrypted program, ciphertexts, and evaluation keys. `EVAL` executes the current homomorphic instruction, including arithmetic, logical, data-movement, and ciphertext-maintenance operations. `COND` evaluates an encrypted condition required by a conditional instruction, while `BRANCH` determines the subsequent execution path according to the resulting encrypted control information. `REFRESH` performs ciphertext refreshing without executing a regular program instruction, reducing the accumulated noise of encrypted VM states through bootstrapping or related maintenance operations to sustain subsequent homomorphic evaluation. `OUTPUT` finalizes the encrypted execution result. The states `ACCEPT` and `REJECT` represent the terminal outcomes of the VM execution.

4. Programming Models over FHE VMs

In this section, we introduce the concrete programming models constructed over the instantiated FHE VM. To better explain the concrete programming models, we first categorize FHE VMs into three main types in Section 4.1, namely, the SISD VMs (Section 4.2), the SIMD VMs (Section 4.3), and the hybrid VMs (Section 4.4). Each VM type represents a unique approach to balancing the functionality, precision, and efficiency of VMs over FHE.

4.1. VM Categorization

Here, we categorize FHE VMs according to their exposed data parallelism and the mechanisms used to realize complex instructions: SISD VM, SIMD VM, and hybrid VM. As summarized in Table 2, all types of VMs support arithmetic, division, and comparison, but differ in execution granularity, precision strategy, and interoperability.

• **SISD VM** (Section 4.2): The SISD VMs process one piece of plaintext data per instruction, achieving low latency in small-scale computation. As shown in Table 2, most existing SISD VMs adopt radix decomposition and realize functionality through some types of lookup tables (LUTs). Since only a single plaintext is encrypted in a ciphertext, most SISD VMs do not require homomorphic automorphism or scheme conversion operators.

• **SIMD VM** (Section 4.3): The SIMD VMs pack a vector of plaintext messages into one ciphertext and apply each instruction across the vector, trading higher latency for high throughput. To manipulate data in a SIMD manner, automorphism and native vector arithmetic operators are supported. It is also pointed out in Table 2 that the `SIMD-Poly` models do not natively provide bitwise operations or a general precision guarantee, while the `SIMD-Bool` and `SIMD-Carry` models recover broader functionality and arbitrary precision through circuits or radix/RNS decomposition.

• **Hybrid VM** (Section 4.4): The Hybrid VMs combine SIMD arithmetic with SISD-style LUT for flexible program evaluation. Table 2 shows that Hybrid VMs generally target 16-bit precision, while some recent designs [65, 135] support arbitrary precision. Meanwhile, hybrid VMs mostly support scheme conversion, unlike the SISD and SIMD entries. This flexibility exploits the strongest primitive for each program segment, but conversion and scheduling add overhead. Thus, SISD prioritizes latency, SIMD prioritizes throughput, and Hybrid VMs balance both through heterogeneous execution.

4.2. Programming Models of SISD VM

As shown in Table 2, SISD VMs are typically built upon three main types of programming models: the `SISD-Bool` model, the `SISD-Carry` model, and the `SISD-Tree` model. Both the `SISD-Bool` and `SISD-Carry` models employ SISD-FBS to evaluate programs over SISD-FHE ciphertexts, but with distinct circuit topologies. In contrast, the `SISD-Tree` model converts an input program into a (generally binary) branching program, where branching statements are evaluated using the SISD-MUX operator [112, 142]. In what follows, we explain the detailed computational and functional characteristics of each programming models of SISD VMs.

`SISD-Bool` Model: Here, high-level tasks are transformed into Boolean circuits, and the input data are composed into binary digits. Then, SISD-FBS is employed to homomorphically evaluate the given circuit over the binary input data [35, 56]. A standard SISD-FBS operator simultaneously refreshes noise and performs the computation of the prescribed logic gate, allowing the construction of complex Boolean circuits through gate-level composition. As noted in Table 2, the `SISD-Bool` model enables the support of the oblivious VM security model [14, 24, 107], which protects both the program and the data. To optimize the FHE circuit evaluation efficiency, recent works [108, 109, 143–146] extend homomorphic evaluation from single-bit Boolean gates to composite logic gates by combining cryptographic optimizations and circuit-level synthesis techniques [108, 146]. As later shown in Section 7, compared to other programming models for SISD VMs, the `SISD-Bool` model scales well when the word size ω becomes large due to the ability to dynamically optimize the circuit topology.

`SISD-Carry` Model: The second subtype constructs fixed-structure LUTs to implement all VM instructions using a fixed set of optimized LUTs [114]. For example, in the `SISD-Carry` model implemented by [114], arithmetic operators, such as additions and multiplications, are implemented using specialized functional bootstrapping that only performs

TABLE 2: Classification of Existing Schemes by VM Computation Model and Functionality

VM Type	Scheme	Security$^{\triangle}$ Model	Programming Model	Decomposition Format	Auto.	Add.	Mult.	Div.	Bit	Comp.	Prec.	Scheme Conv.
SISD	[14, 24, 107]	O	SISD-Bool	Radix*	○	●	●	●	●	●	16-bit$^{\clubsuit}$	N/A
	[56, 75]	T	SISD-Bool	Radix*	○	●	●	●	●	●	Arb.	N/A
	[108, 109]	T	SISD-Bool	Radix*	○	●	●	●	●	●	Arb.†	N/A
	[110–113]	T	SISD-Tree	Radix*	○	●	●	●	●	●	Arb.†	N/A
	[114]	T	SISD-Carry	Radix	○	●	●	●	●	●	Arb.$^{\diamond}$	N/A
SIMD	[115]	O	SIMD-Poly$^{\blacklozenge}$	N/A	●	●	●	●	○	●	Arb.†	N/A
	[116–128]	T	SIMD-Poly	N/A	●	●	●	●	○	●	N/A	N/A
	[66, 94, 129–135]	T	SIMD-Bool	Radix*	●	●	●	●	●	●	Arb.†	○
	[39, 102, 136–139]	T	SIMD-Carry	N/A	●	●	●	●‡	●	●‡	16-bit‡	○
	[140, 141]	T	SIMD-Carry	Radix/RNS	●	●	●	●	●	●	Arb.$^{\diamond}$	○
Hybrid	[73]	T	scheme-hybrid	Radix*	●	●	●	●	●	●	16-bit	○
	[72]	T	scheme-hybrid	N/A	●	●	●	●**	○	●**	16-bit	●
	[106]	T	scheme-hybrid	N/A	●	●	●	●	○	●	16-bit	●
	[99]	T	scheme-hybrid	Radix*	●	●	●	●	●	●	16-bit	●
	[65, 135]	T	encoding-hybrid	Radix$^{\spadesuit}$	●	●	●	●	●	●	Arb.$^{\diamond}$	N/A

$^{\triangle}$T and O denote the support of the transparent and the oblivious program security model, respectively. **These functions are implemented in an SISD style in [72].
$^{\clubsuit}$The precision is limited by the processor instruction set. *Only binary decomposition is supported. †Arbitrary precision is achieved through circuit-based computation.
‡These functions are based on polynomial approximation. $^{\diamond}$Arbitrary precision is achieved through radix/RNS decomposition.
$^{\blacklozenge}$[115] is classified as a SISD VM since the multiple-instruction multiple-data (MIMD) execution is implemented through repeated execution of SIMD-Poly instructions.
$^{\spadesuit}$[65, 135] propose a group-element-based form of plaintext encoding, which can be considered a special type of radix decomposition.

carry-related computations. By using a SISD-Carry model, specific instructions can be realized with fewer bootstrapping operations and reduced evaluation noise. However, such specialized LUTs need to be hand-crafted for each and every type of instruction, which may not be desirable if the target VM has a large number of task-dependent instructions.

SISD-Tree Model: The SISD-Tree model [56, 113] takes a different approach in evaluating circuits over FHE. Roughly speaking, given two ciphertext inputs ct_0, ct_1 and a selection input ct_2, the core idea is to output the homomorphic selection of $\mathsf{ct}_2 \ ? \ \mathsf{ct}_0 \ : \ \mathsf{ct}_1$. Since all logic functions can be implemented by a tree of such selectors, we can evaluate arbitrary Boolean circuits by repeating the selector tree structures. Compared with the SISD-Bool and SISD-Carry models, the SISD-Tree model achieves higher efficiency for deep circuits with few inputs, since the latency of this approach is dominated by the process of converting the input ciphertexts from the $\mathsf{ct}_{\mathrm{SISD}}$ format into the $\mathsf{ct}_{\mathrm{KEY}}$ format, as required by the SISD-MUX operator [56]. Consequently, a number of conversion techniques [112, 139, 147, 148] are proposed to enhance the performance of the SISD-Tree model.

4.3. Programming Models of SIMD VM

SIMD VMs exploit SIMD execution to perform batched homomorphic operations, achieving high throughput for large-scale encrypted computations at the cost of higher latency. As shown in Table 2, we divide the programming models of SIMD VMs into the SIMD-Poly, SIMD-Bool, and SIMD-Carry types, whose main functionalities and computational performance are characterized below.

SIMD-Poly Model: The SIMD-Poly model of SIMD VM leverages the native $+$, $-$, $\cdot$, RingProduct and Automorphism operators of SIMD-FHE schemes to implement arbitrary function evaluation over ciphertext. Therefore, the SIMD-Poly model achieves significantly higher throughput for tasks such as additions and multiplications compared to the SISD VMs. However, when evaluating nonlinear functions, the SIMD-Poly model relies on polynomial approximation techniques, such as Chebyshev interpolation [149, 150], Lagrange interpolation [151], and trigonometric interpolation [152]. For example, [116, 117] propose the method to approximate the comparison (or step) function through polynomial iteration and composition, while [128] leverages the composition of minimax polynomial approximations [153] to approximate the ReLU and max-pooling functions. However, as shown in Table 2, due to the inherent nature of polynomial approximation, the SIMD-Poly model of SIMD VM still suffers from significant noise accumulation and large approximation errors at the discontinuities of nonlinear functions.

SIMD-Bool Model: In the Boolean model of SIMD VM, the core idea is to use the SIMD property to evaluate a given binary circuit on a large number of parallel inputs, enabling high-throughput homomorphic Boolean computation. It is well known that SIMD-FHE schemes such as BGV, BFV and CKKS can emulate binary operations by encrypting a single bit as the plaintext [66, 94, 129–135]. Under such encoding, binary gates can be realized as shallow real-arithmetic circuits, e.g., $i \wedge j = i \cdot j$ and $i \vee j = i + j - i \cdot j$. When building the SIMD-Bool model over approximate FHE schemes such as CKKS, HomPoly is typically used to construct a homomorphic rounding function that periodically rounds real values back to binary digits [116, 129, 153], such that an unbounded-depth circuit can be evaluated. Nevertheless, the SIMD-Bool model suffers from exponential circuit-depth growth as the

target function complexity increases, inducing substantial computational overhead for complex tasks.

`SIMD-Carry` Model: In a `SIMD-Carry` programming model, the arithmetic operations are computed using native SIMD-FHE primitives [41, 136, 154, 155] similarly to those used in `SISD-Carry` model. The important difference here is that since SIMD-FHE schemes natively support both homomorphic additions and multiplications [156], only non-arithmetic operations, such as carry propagation and comparison, are computed using the SIMD-FUNCBOOT operator. More concretely, a high-precision input message is decomposed into a group of plaintexts as further elaborated in Section C. Then, the plaintext segments are separately encrypted to produce the group of ciphertexts. To evaluate a nonlinear function over a group of ciphertexts, the fixed-precision LUTs are first applied to each of the individual ciphertexts in the group, while additional LUTs are used to combine intermediate results across the ciphertext segments. The main advantage for `SIMD-Carry` is that unlike polynomial approximation, SIMD-FUNCBOOT can evaluate arbitrary nonlinear functions with a known precision, independent from the shape of target function. As with the `SIMD-Bool` model, the `SIMD-Carry` model supports the evaluation of general functions over arbitrary-precision data. Nevertheless, as later demonstrated in Section 7, the `SIMD-Bool` and `SIMD-Carry` models exhibit distinct performance characteristics across different tasks.

4.4. Programming Models of Hybrid VM

The goal of a hybrid VM is to take advantage of both SIMD and SISD FHE schemes to further improve the execution efficiency. The key feature of most hybrid VMs is to decompose the input programs into two fundamentally different types of computations, namely, arithmetic and logic computations. Existing approaches to evaluate such computations over hybrid VMs can be broadly classified into two programming models: the `scheme-hybrid` model and the `encoding-hybrid` model. First, the `scheme-hybrid` model [31, 72, 73, 106, 157] utilizes different SIMD- and SISD-FHE schemes to evaluate arithmetic and logic computations, respectively. Under this design, the format-conversion operator realizes a homomorphic conversion between the ciphertext formats of different FHE schemes [148, 157]. More recently, we see the development of the second class of FHE VMs, i.e., `encoding-hybrid` [65, 135], which employs a unified FHE scheme to support both arithmetic and logic computations through separate plaintext encodings. In this case, the format-conversion operator homomorphically converts between the arithmetic and logic plaintext encodings while remaining within the same FHE scheme. In either case, format conversion generally constitutes the heaviest computation in a hybrid VM. Consequently, as later demonstrated in Section 7, although hybrid VMs can attain superior performance for purely arithmetic or purely logical programs, their performance may degrade for programs involving both modes of computations, since frequent changes in computation modes result in the invocation of a large number of conversions.

5. Compilation Support for FHE VMs

Here, we provide a high-level summary of existing compilation techniques that can be applied to FHE VMs. While most current FHE compilers are not designed specifically for program compilation over VMs, we emphasize that a cross-layer systematization of the compilation techniques is critical for making the instantiated FHE VMs usable and deployable.

Table 3 summarizes the main features of existing FHE compilers categorized by their targeted VMs, namely, the SISD VMs, the SIMD VMs, and the hybrid VMs. Here, we compare existing FHE compilers within each VM category across six optimization aspects: program transformation, branch support, loop support, bootstrapping, level management, and multi-level representation.

Program Transformation: The program transformation process consists of optimizations that transform high-level plaintext programs into concrete operators over ciphertexts. Program transformation strategies for SISD VMs, including [75, 108, 114, 158, 159], focus on the design and optimization of the underlying Boolean or LUT circuits. For example, [75, 108] take high-level descriptions (Verilog, C) as input and apply logic synthesis or gate fusion, transforming a sequence of Boolean gates into a single, optimized LUT. For SIMD VMs, a number of compilation techniques are proposed to transform and optimize arithmetic circuits. For instance, [119, 125, 126] can transform high-level operations such as convolutions into an optimized sequence of SIMD vector additions, multiplications and automorphisms. Lastly, for hybrid VMs, in addition to the above transformations, program segmentation needs to be carried out, where the given task is cut into multiple program segments [6, 72, 73, 106]. Then, the compiler needs to determine which concrete FHE scheme to use for each and every program segment.

Branch Statement: Branching is a fundamental control structure for program execution. As shown in Table 3, compilers that support branching statements, such as [106, 132], typically use the SISD-MUX to handle branch statements. Most compilers for SISD VMs [75, 108, 114, 158, 159] can evaluate branch compilation because Boolean circuits have native support for data multiplexing. Since most existing SIMD VM compilers are designed for the `SIMD-Poly` programming model, they provide only limited support for conditional branching.

Loop Statement: Here, we follow the notion in [106], a loop whose termination condition is in plaintext is referred to as a transparent loop, while a loop whose termination condition is in ciphertext is called an oblivious loop. As shown in Table 3, due to the difficulty in determining the termination condition, most existing FHE compilers only support transparent loops. Only a few FHE compilers [75, 106, 108, 115, 132] support the interpretation of (certain types of) oblivious loops, relying on additional client-server interactions.

TABLE 3: Comparison of Compiler Support Across Functionalities

Compiler	VM Type	Surface Language	Program Transformation	Branch	Loop*	Bootstrapping	Level Management	Multi-Level Representation
[75]	SISD VM	C++	●	●	◐	●	○	●
[158] / [114]		C++ / Rust	●	●	◐	●	○	○
[108, 159]		Verilog	●	●	◐	●	○	○
[124]	SIMD VM	C++	●	○	◐	○	●	●
[125]		C++	●	○	◐	●	●	●
[134]		C++	○	○	◐	●	●	○
[121] / [120]		C / Python	●	○	◐	○	○	○
[126, 127]		Python	●	○	◐	○	●	○
[122] / [160]		Python / ONNX	●	○	◐	●	●	●
[118] / [119]		Racket / Julia	●	○	◐	○	○	○
[132]	Hybrid VM	C++	○	●	●	○	○	●
[106]		C++	●	●	●	●	●	●
[72]		C	●	○	◐	●	●	●
[73]		Python	●	○	◐	●	●	●

*◐ refers to the support for transparent loops, and ● refers to both the support for transparent loop and oblivious loops.

Bootstrapping: Supporting the automatic insertion of the bootstrapping operator is critical for improving the usability and performance of FHE VMs, as summarized in Table 3, most SISD VM compilers support the insertion of bootstrapping due to the inherent noise-refreshing nature of the SISD-FBS operator. Conversely, compilers for SIMD VM and hybrid VM vary in their support for automatic bootstrapping management. In particular, some of the SIMD VM compilers optimize programs with shallow multiplicative depth, and thus do not include bootstrapping in the compiled programs [118–121, 124, 126, 127]. On the other hand, general-purpose FHE program compilers [72, 73, 106, 122, 161] and compilers for privacy-preserving neural networks [124, 125, 160] generally support the automatic insertion of bootstrapping.

Level Management: Level management refers to the ability of the compiler to dynamically manage and optimize the multiplicative levels for SIMD-FHE ciphertexts. Thus, level management is generally not necessary for compilers for SISD VM. For the SIMD VM and hybrid VM types, level management is commonly found in compilers for general-purpose programs and programs with deep multiplicative depth [72, 73, 106, 122, 124–127, 160], as reducing the number of unused levels is important for the overall efficiency of the compiled programs.

Multi-Level Representation: The main benefit of building FHE compilers over the multi-level intermediate representation (MLIR) framework is its ability to split the compilation process into different abstraction layers. As shown in Table 3, many recent works, including those in the SISD [75], the SIMD [122, 124, 125, 160], and the hybrid VM types [72, 73, 106, 132], adopt MLIR as a foundation for compiler development. Conversely, certain FHE compilers, such as [108, 114, 118–121, 126, 127, 134, 158, 159], employ a monolithic compiler design (indicated as ○ in the table).

6. Accelerating FHE VMs

Hardware acceleration substantially improves the practical efficiency of FHE virtual machines, with GPUs, FPGAs, and ASICs offering different trade-offs in performance, flexibility, energy efficiency, and cost. Due to space limitations, a more detailed explanation is provided in Section G. GPU accelerators exploit massive parallelism. For SISD VMs, they primarily accelerate programmable bootstrapping by parallelizing multiple SISD-FBS operations [162, 163] or optimizing small-polynomial workloads [164, 165]. For SIMD VMs, existing literature focuses on expensive primitives such as NTT and key switching [166–171], as well as parallel control and data flows [172–174]. Hybrid acceleration remains difficult because different FHE schemes use substantially different parameters [175]. While FPGA accelerators provide reconfigurability and support customized computation patterns and memory layouts for SISD-FBS [176, 177], streamlined pipelines improve NTT, key switching, and data movement for SIMD schemes [178–181]. Reusable functional units can also support multiple schemes [182], although limited on-chip memory, lower frequency, and limited scalability remain constraints. ASICs deliver the highest throughput and energy efficiency through pipelined functional units [183–187] and large-scale polynomial arithmetic [188–195]. Unified architectures support multiple schemes using shared vectorized primitives [76, 77, 196, 197], while processing-near-memory architectures reduce data movement [187, 193, 196]. However, the deployment of ASICs remains limited due to the inflexibility and high manufacturing cost.

7. Evaluation

Here, we test the performance of various types of FHE VMs. We first explain the evaluation setup in Section 7.1, and then provide the benchmark results in Section 7.2 and Section 7.3, respectively.

7.1. Evaluation Setup

Benchmarks: We divide the evaluations of FHE VMs into two parts. First, we benchmark eight representative FHE instructions to assess both the latency and throughput[3] performance. Due to space limitations, we only include performance for addition (ADD), multiplication (MUL), and greater-than (GT) in the main manuscript, and the rest of the evaluation results can be found in Appendix I. Second, for end-to-end program evaluation, we benchmark two representative workloads: P1 and P2. P1 consists of three token transactions in the standard ERC-20 (Ethereum Request for Comments 20) format [198] with 64-bit data precision[4]. Meanwhile, P2 is a TPC-H (Transaction Processing Performance Council Benchmark H) Q6 query [199] evaluated on a database with 32,768 entries, where each relevant input field is represented with 16-bit precision. Further details of the program benchmark definitions, workloads, and implementations are provided in Appendix J.

VM Models and Implementation Details: Here, the instruction measurements are grouped according to the programming models of each VM type, as introduced in Section 4. Specifically, SISD VMs include the `SISD-Bool`, the `SISD-Carry`, and the `SISD-Tree` models, implemented based on [108], [114], and [111, 112], respectively. SIMD VMs include the `SIMD-Poly`, the `SIMD-Bool` and the `SIMD-Carry` models, implemented based on [99], our in-house CKKS library, and [39, 41], respectively. Additionally, the hybrid VM benchmark is implemented based on [32]. Our benchmark evaluations are conducted on a machine equipped with an Intel Xeon Gold 5318Y processor and 512 GB of RAM, using a single thread for computation. The encryption parameters instantiated for each benchmarked VM are chosen to maximize computational efficiency while satisfying the same security level and the required numerical stability and output precision (the complete set of parameters is provided in Appendix H).

7.2. Instruction Benchmarks

Figure 2 compares the latency and throughput of different VM models for ADD, MUL, and GT across bit widths. As shown in Figure 2a, Figure 2b, and Figure 2c, the relative performance of different VM models varies with both instruction type and bit width. In general, SISD VMs provide lower latency at small bit widths, while SIMD VMs achieve higher throughput by exploiting data-level parallelism. Meanwhile, hybrid VMs dynamically integrate the strengths of both SISD and SIMD VMs to balance latency and throughput. To further understand the differences in the programming models, we take a closer look at the comparisons within each VM type as follows.

SISD VMs: The `SISD-Carry` model achieves the lowest latency for simple arithmetic operations such as addition, highlighting the advantage of SISD-style execution for workloads with limited batching opportunities. For heavy arithmetic operations such as multiplication, LUT-based models suffer from rapidly increasing circuit depth with respect to bit width. Although the `SISD-Tree` model is competitive for medium-precision multiplications, its ciphertext conversion overhead from $\mathsf{ct}_{\mathrm{SISD}}$ to $\mathsf{ct}_{\mathrm{KEY}}$ grows linearly with the input bit width, limiting its scalability. For logic-type computations such as greater-than comparison, the `SISD-Carry` model performs better at small bit widths, whereas the `SISD-Bool` model becomes more efficient as the input precision grows.

SIMD VMs: We first observe that the `SIMD-Poly` model delivers the lowest latency for the addition and multiplication instructions. At 16 bits, it is 17.4× faster for ADD and 5.60× faster for MUL than the best SISD model. Nevertheless, under our parameter configuration, the `SIMD-Poly` model cannot directly handle computations at ≥ 64-bit precision. Similarly, while the `SIMD-Poly` model can evaluate nonlinear functions such as GT using polynomial approximation, the evaluation precision is limited as well. In contrast, the `SIMD-Bool` model enables arbitrary-precision computation with high efficiency by transforming the given instructions into logic circuits over the decomposed binary inputs. Meanwhile, `SIMD-Carry`, leveraging decomposition and carry propagation, can perform both polynomial and non-polynomial computations at arbitrary precision. Compared with `SIMD-Bool`, it achieves substantially higher amortized throughput.

Hybrid VMs: For a hybrid VM, the addition and multiplication instructions are typically executed using the SIMD-FHE schemes, while GT is delegated to the SISD-FHE scheme. When an instruction is executed independently, we manually select and benchmark the most efficient programming model, without considering the conversion overheads. However, when the instruction is part of a longer computation pipeline, both the single-instruction latency and the conversion overheads need to be considered.

7.3. Program Benchmarks

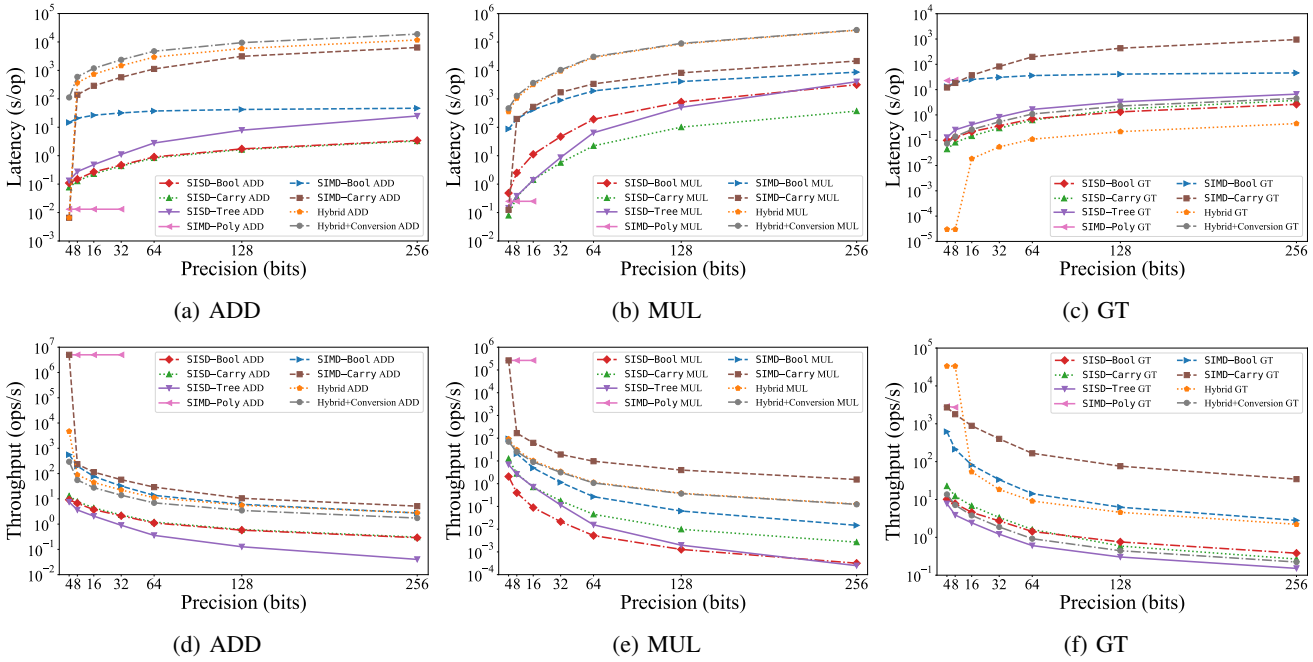
To better examine the overall effectiveness of FHE VMs, we benchmark the latency and throughput of different VM models using two end-to-end programs, namely, P1 and P2, as specified in Section 7.1[5].

As shown in Figure 3, the `SISD-Carry` model and the Hybrid VM achieve the lowest latency under program P1, indicating that SISD architectures are highly efficient for small-scale programs. In contrast, `SIMD-Bool` and `SIMD-Carry` require approximately 7.41× and 13.33× as much execution time compared with the `SISD-Carry`

3. For SIMD FHE VMs, throughput is measured assuming fully packed ciphertexts, with each instruction processing all available plaintext slots in parallel.

4. It is noted that transactions between accounts are typically hard to batch, since each transaction is directed from a specific user A to a specific user B. Thus, we only run three transactions in parallel to represent realistic application scenarios.

5. The `SIMD-Poly` model of the SIMD VM is excluded from the program benchmark due to its lack of support for high-precision instructions.



(a) ADD (b) MUL (c) GT

(d) ADD (e) MUL (f) GT

Figure 2: The performance of (a, d) addition, (b, e) multiplication and (c, f) greater-than instructions in terms of latency and throughput, respectively.

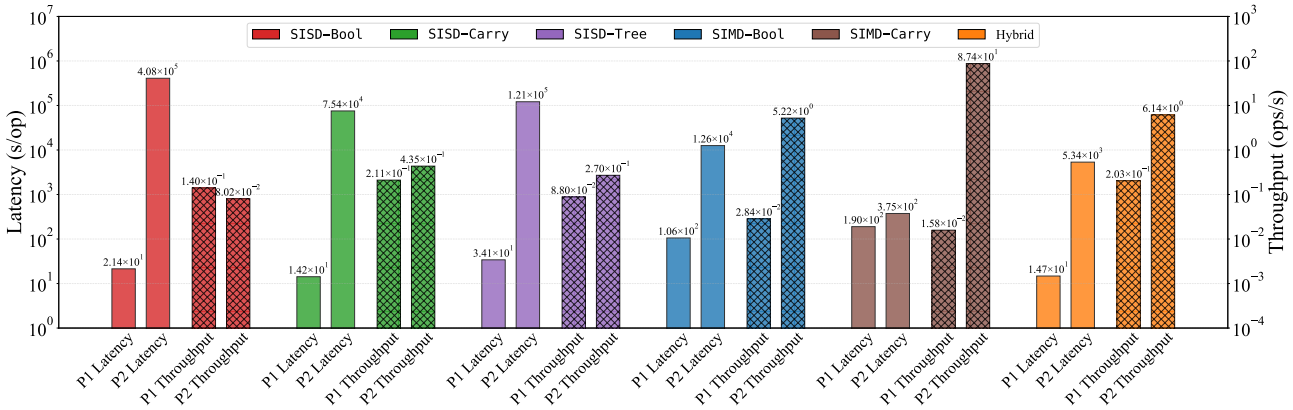


Figure 3: Performance comparison of FHE transactions and data queries across different VM models.

model, respectively, since most SIMD slots remain unused. Conversely, P2 evaluates a TPC-H query over a large number of database entries, where SISD VMs exhibit limited scalability while SIMD VMs achieve lower latency and higher throughput. In particular, the `SIMD-Carry` model achieves over a 200× speedup in latency and throughput over SISD-based models, demonstrating the advantage of efficient batch processing. Moreover, although the `SIMD-Bool` model exhibits lower latency for certain individual instructions, a significant number of AUTOMORPHISM operators need to be inserted for end-to-end programs with inter-slot summations, leading to degraded efficiency. Lastly, the hybrid VM model achieves a balanced trade-off between performance and flexibility.

8. Discussions and Future Directions

8.1. Guidelines for VM Selection

Based on the performance characteristics of different FHE VM models, we provide an operational guideline for selecting an appropriate VM for a target workload.

For workloads dominated by transactional operations with small batch sizes, strong control-flow dependencies, and frequent comparisons (e.g., balance checks, conditional updates, or access control), SISD VM architectures are generally preferred. In particular, `SISD-Carry`-based or `SISD-Tree`-based VMs are more suitable when the input precision is moderate, as they minimize per-instruction

overhead and avoid the overhead of underutilized SIMD ciphertext slots.

For workloads that are data-parallel and throughput-oriented, where operations are mostly arithmetic (e.g., multiplication, addition, and aggregation) and can be executed independently across large datasets, SIMD VM architectures are more appropriate. In this case, the programming models of SIMD VMs are to be selected to fully exploit SIMD-style ciphertext packing, especially when the number of data items is large, even at the cost of higher per-instruction latency.

For workloads where both transactional logic and large-scale data processing coexist, hybrid VM architectures can be more performant. In such cases, the execution strategy needs to minimize ciphertext format conversion overheads, which can be non-trivial in practice. Therefore, hybrid VMs become advantageous only when each computation segment is sufficiently large to amortize the cost of format conversion, while cross-model switching remains infrequent.

8.2. Future Directions

We point out three potential directions for future research. First, the efficient instantiation of oblivious VMs with provable security remains an open research direction. In particular, hiding the program termination condition is known to be challenging for both FHE and general secure multi-party computation protocols [200]. Second, the development of advanced FHE primitives is fundamental to addressing the current dilemma between the data precision and the execution performance. In its current state, increasing data precision tends to cause super-linear increases in latency and substantial degradation in throughput, which can be prohibitive for real-world applications. Last but not least, one of the most vital research directions is the establishment of VM-specific, open-source compilation ecosystems to further abstract away cryptographic complexity. While existing compilers have already provided a certain degree of utility, the programming models are yet too complex to allow for massive program development by non-FHE-experts.

Appendix A. Open Science Policy

We are committed to supporting transparency, reproducibility, and open access for this work. To facilitate independent validation of our experimental results, we provide the source code and benchmark implementations to reproduce the main evaluations presented in this paper. All publicly releasable research artifacts will be made available at https://anonymous.4open.science/r/FHEVM_benchmarks-E1BF/. These materials are intended to enable other researchers to reproduce our results, compare alternative FHE VM designs under consistent settings, and extend the evaluation framework for future research.

Appendix B. Notations

We summarize the notations used in this work in Table A1.

Appendix Table A1: Notation Summary

Notation	Description
msg	Abstract message (scalar or vector of real numbers)
M	Real-valued message
$\mathbf{M}$	Vector of real-valued message
λ	Security level
n, N	Lattice dimension
p	Plaintext modulus
q, Q	Ciphertext moduli
$\mathbb{Z}_p$	Set of integers modulo p
$\mathbb{Z}_q$	Set of integers modulo q
$\mathbb{Z}_p^N$	Set of N-vector over $\mathbb{Z}_p$
$R_{N,p}$	Plaintext ring $\mathbb{Z}_p[X]/(X^N+1)$
$R_{N,Q}$	Ciphertext ring $\mathbb{Z}_Q[X]/(X^N+1)$
$\tilde{a}[X]$	Polynomial in variable X
m	Plaintext in $\mathbb{Z}_p$
$\mathbf{m}$	Plaintext vector with N components in $\mathbb{Z}_p$
$\hat{m}_{int}$	Decomposition of M
$\hat{m}_{vec}$	Decomposition of $\mathbf{M}$
ct	FHE ciphertext
$\mathsf{ct}_{\mathrm{SISD}}$	Ciphertexts for SISD schemes
$\mathsf{ct}_{\mathrm{SIMD}}$	Ciphertexts for SIMD schemes
σ	Plaintext embedding
ρ	Plaintext precision
β	Ciphertext segment precision
ω	Machine word size
$\mathcal{E}$	General FHE scheme
$\mathcal{I}$	ISA of the FHE VM
$\mathcal{M}$	Memory system of the FHE VM
$\mathcal{Q}$	Finite set of machine states
δ	Transition function of the FHE VM defining one instruction execution

Appendix C. Plaintext Encodings

Plaintext encoding transforms scalar or vector messages into plaintext representations supported by FHE. Both scalar and vector plaintexts may be partitioned into multiple groups, while vector plaintexts can additionally be embedded into a polynomial plaintext space. The two main encoding procedures are plaintext decomposition and plaintext embedding.

Plaintext Decomposition: Plaintext decomposition partitions a high-precision plaintext into multiple lower-precision components. The decomposition is applied directly to a scalar plaintext and component-wise to a vector plaintext. We consider two common methods: radix decomposition and residue number system (RNS) decomposition.

In radix decomposition, a plaintext is represented by a sequence of digits with respect to a radix B. For a scalar plaintext m, the decomposition satisfies $m = \sum_i m_i B^i$, where each m_i is a lower-precision radix digit. For a vector plaintext $\mathbf{m}$, the same decomposition is applied component-wise as $\mathbf{m} = \sum_i \mathbf{m}_i B^i$. The resulting groups preserve the positional significance of the original plaintext. Radix decomposition therefore supports selective access to different precision ranges and is well suited to non-linear operations such as comparison, bit extraction, shifting, and division.

In RNS decomposition, a plaintext is represented by its residues with respect to a set of pairwise coprime moduli. For a scalar plaintext m, each decomposed component is given by $m_i = m \bmod q_i$, where q_i is an RNS modulus. For a vector plaintext $\mathbf{m}$, the residues are computed component-wise as $\mathbf{m}_i = \mathbf{m} \bmod q_i$. The original plaintext can be reconstructed from the residue components using the Chinese remainder theorem, provided that it lies within the supported reconstruction range. Since the residue components are independent, RNS decomposition enables parallel modular computation and is particularly suitable for arithmetic-intensive and linear homomorphic operations.

Radix and RNS decomposition provide complementary plaintext organizations. Radix decomposition exposes positional components of a plaintext, whereas RNS decomposition exposes independent modular components. Homomorphic conversion between the two representations allows different operations to use the decomposition form most suitable for their evaluation.

Plaintext Embedding: Plaintext embedding maps an encoded plaintext into the algebraic plaintext space required by the underlying FHE scheme. For scalar plaintext spaces, this mapping may be direct. For polynomial plaintext spaces, a vector plaintext $\mathbf{m}$ is mapped to a polynomial $\tilde{m} = \sigma(\mathbf{m})$ in the corresponding plaintext ring. The embedding is applied independently to every decomposed plaintext group. Two common methods are coefficient embedding and slot embedding. In coefficient embedding, the entries of a vector are mapped directly to the coefficients of a polynomial, yielding $\tilde{m}_{\mathrm{coeff}}(x) = \sum_{j=0}^{N-1} \mathbf{m}[j] x^j$. This representation supports computations based on coefficient-wise data organization [16, 100]. In slot embedding, the vector entries are encoded into independent plaintext slots through an algebraic encoding map, such that $\tilde{m}_{\mathrm{slot}}(\zeta_j) = \mathbf{m}[j]$ at designated evaluation points ζ_j [33]. This representation enables SIMD-style evaluation, in which a homomorphic operation is applied simultaneously to multiple plaintext elements.

Appendix Table A2: Example ISA for a SISD VM with an ω-bit Word Size.

Instruction	Input(s)	Output	Description
H_SCA_ADD	$\mathsf{ct}_{\mathrm{SISD}}(m_1), \mathsf{ct}_{\mathrm{SISD}}(m_2)$	$\mathsf{ct}_{\mathrm{SISD}}(m_1 + m_2)$	Homomorphic scalar addition
H_SCA_ADD	$\mathsf{ct}_{\mathrm{SISD}}(m_1), m_2$	$\mathsf{ct}_{\mathrm{SISD}}(m_1 + m_2)$	Homomorphic plaintext-ciphertext scalar addition
H_SCA_SUB	$\mathsf{ct}_{\mathrm{SISD}}(m_1), \mathsf{ct}_{\mathrm{SISD}}(m_2)$	$\mathsf{ct}_{\mathrm{SISD}}(m_1 - m_2)$	Homomorphic scalar subtraction
H_SCA_SUB	$\mathsf{ct}_{\mathrm{SISD}}(m_1), m_2$	$\mathsf{ct}_{\mathrm{SISD}}(m_1 - m_2)$	Homomorphic plaintext-ciphertext scalar subtraction
H_SCA_MUL	$\mathsf{ct}_{\mathrm{SISD}}(m_1), m_2$	$\mathsf{ct}_{\mathrm{SISD}}(m_1 \cdot m_2)$	Homomorphic plaintext-ciphertext scalar multiplication
H_SCA_AND	$\mathsf{ct}_{\mathrm{SISD}}(m_1), \mathsf{ct}_{\mathrm{SISD}}(m_2)$	$\mathsf{ct}_{\mathrm{SISD}}(m_1 \wedge m_2)$	Homomorphic scalar bitwise AND
H_SCA_OR	$\mathsf{ct}_{\mathrm{SISD}}(m_1), \mathsf{ct}_{\mathrm{SISD}}(m_2)$	$\mathsf{ct}_{\mathrm{SISD}}(m_1 \vee m_2)$	Homomorphic scalar bitwise OR
H_SCA_NOT	$\mathsf{ct}_{\mathrm{SISD}}(m)$	$\mathsf{ct}_{\mathrm{SISD}}(\neg m)$	Homomorphic scalar bitwise NOT
H_SCA_SHL	$\mathsf{ct}_{\mathrm{SISD}}(m), c$	$\mathsf{ct}_{\mathrm{SISD}}(m \ll c)$	Homomorphic scalar left shift by c bits, where c is a positive integer
H_SCA_SHR	$\mathsf{ct}_{\mathrm{SISD}}(m), c$	$\mathsf{ct}_{\mathrm{SISD}}(m \gg c)$	Homomorphic scalar right shift by c bits
H_SCA_RADIXTORNS	$\mathsf{ct}_{\mathrm{SISD}}(m_{\mathrm{radix}})$	$\mathsf{ct}_{\mathrm{SISD}}(m_{\mathrm{RNS}})$	Homomorphically convert a radix decomposed $\mathsf{ct}_{\mathrm{SISD}}$ ciphertext into an RNS decomposed $\mathsf{ct}_{\mathrm{SISD}}$ ciphertext.
H_SCA_RNSTORADIX	$\mathsf{ct}_{\mathrm{SISD}}(m_{\mathrm{RNS}})$	$\mathsf{ct}_{\mathrm{SISD}}(m_{\mathrm{radix}})$	Homomorphically convert an RNS decomposed $\mathsf{ct}_{\mathrm{SISD}}$ ciphertext into a radix decomposed $\mathsf{ct}_{\mathrm{SISD}}$ ciphertext.

Appendix Table A3: Example ISA for a SIMD VM with an ω-bit Word Size.

Instruction	Input(s)	Output	Description
H_VEC_ADD	$\mathsf{ct}_{\mathrm{SIMD}}(\mathbf{m}_1), \mathsf{ct}_{\mathrm{SIMD}}(\mathbf{m}_2)$	$\mathsf{ct}_{\mathrm{SIMD}}(\mathbf{m}_1 + \mathbf{m}_2)$	Homomorphic vector addition
H_VEC_ADD	$\mathsf{ct}_{\mathrm{SIMD}}(\mathbf{m}_1), \mathbf{m}_2$	$\mathsf{ct}_{\mathrm{SIMD}}(\mathbf{m}_1 + \mathbf{m}_2)$	Homomorphic plaintext-ciphertext vector addition
H_VEC_SUB	$\mathsf{ct}_{\mathrm{SIMD}}(\mathbf{m}_1), \mathsf{ct}_{\mathrm{SIMD}}(\mathbf{m}_2)$	$\mathsf{ct}_{\mathrm{SIMD}}(\mathbf{m}_1 - \mathbf{m}_2)$	Homomorphic vector subtraction
H_VEC_SUB	$\mathsf{ct}_{\mathrm{SIMD}}(\mathbf{m}_1), \mathbf{m}_2$	$\mathsf{ct}_{\mathrm{SIMD}}(\mathbf{m}_1 - \mathbf{m}_2)$	Homomorphic plaintext-ciphertext vector subtraction
H_VEC_MUL	$\mathsf{ct}_{\mathrm{SIMD}}(\mathbf{m}_1), \mathbf{m}_2$	$\mathsf{ct}_{\mathrm{SIMD}}(\mathbf{m}_1 \cdot \mathbf{m}_2)$	Homomorphic plaintext–ciphertext element-wise Hadamard product, corresponding to polynomial convolution $\mathbf{m}_1 \cdot \mathbf{m}_2$ over $R_{N,q}$
H_VEC_MUL	$\mathsf{ct}_{\mathrm{SIMD}}(\mathbf{m}_1), \mathsf{ct}_{\mathrm{SIMD}}(\mathbf{m}_2)$	$\mathsf{ct}_{\mathrm{SIMD}}(\mathbf{m}_1 \cdot \mathbf{m}_2)$	Homomorphic ciphertext–ciphertext element-wise Hadamard product, corresponding to polynomial convolution $\mathbf{m}_1 \cdot \mathbf{m}_2$ over $R_{N,q}$
H_VEC_AND	$\mathsf{ct}_{\mathrm{SIMD}}(\mathbf{m}_1), \mathsf{ct}_{\mathrm{SIMD}}(\mathbf{m}_2)$	$\mathsf{ct}_{\mathrm{SIMD}}(\mathbf{m}_1 \wedge \mathbf{m}_2)$	Homomorphic vector element-wise bitwise AND
H_VEC_OR	$\mathsf{ct}_{\mathrm{SIMD}}(\mathbf{m}_1), \mathsf{ct}_{\mathrm{SIMD}}(\mathbf{m}_2)$	$\mathsf{ct}_{\mathrm{SIMD}}(\mathbf{m}_1 \vee \mathbf{m}_2)$	Homomorphic vector element-wise bitwise OR
H_VEC_NOT	$\mathsf{ct}_{\mathrm{SIMD}}(\mathbf{m})$	$\mathsf{ct}_{\mathrm{SIMD}}(\neg \mathbf{m})$	Homomorphic vector element-wise bitwise NOT
H_VEC_SHL	$\mathsf{ct}_{\mathrm{SIMD}}(\mathbf{m}), c$	$\mathsf{ct}_{\mathrm{SIMD}}(\mathbf{m} \ll c)$	Homomorphic vector element-wise left shift by c bits
H_VEC_SHR	$\mathsf{ct}_{\mathrm{SIMD}}(\mathbf{m}), c$	$\mathsf{ct}_{\mathrm{SIMD}}(\mathbf{m} \gg c)$	Homomorphic vector element-wise right shift by c bits
H_ROTATE	$\mathsf{ct}_{\mathrm{SIMD}}(\mathbf{m}), c$	$\mathsf{ct}_{\mathrm{SIMD}}(\mathbf{m}[X^{5^c}])$	Homomorphic vector rotation
H_ECD	$\mathsf{ct}_{\mathrm{SIMD}}(\sigma_s(\mathbf{m}))$	$\mathsf{ct}_{\mathrm{SIMD}}(\sigma_c(\mathbf{m}))$	Homomorphically convert a slot embedding $\mathsf{ct}_{\mathrm{SIMD}}$ ciphertext into a coefficient embedding $\mathsf{ct}_{\mathrm{SIMD}}$ ciphertext, i.e., $\mathsf{ct}_{\mathrm{SIMD}}(\sigma_c(\mathbf{m})) = \textsc{StoC}(\mathsf{ct}_{\mathrm{SIMD}}(\sigma_s(\mathbf{m})))$
H_DCD	$\mathsf{ct}_{\mathrm{SIMD}}(\sigma_c(\mathbf{m}))$	$\mathsf{ct}_{\mathrm{SIMD}}(\sigma_s(\mathbf{m}))$	Homomorphically convert a coefficient embedding $\mathsf{ct}_{\mathrm{SIMD}}$ ciphertext into a slot embedding $\mathsf{ct}_{\mathrm{SIMD}}$ ciphertext, i.e., $\mathsf{ct}_{\mathrm{SIMD}}(\sigma_s(\mathbf{m})) = \textsc{CtoS}(\mathsf{ct}_{\mathrm{SIMD}}(\sigma_c(\mathbf{m})))$
H_RADIXTORNS	$\mathsf{ct}_{\mathrm{SIMD}}(\mathbf{m}_{\mathrm{radix}})$	$\mathsf{ct}_{\mathrm{SIMD}}(\mathbf{m}_{\mathrm{RNS}})$	Homomorphically convert a radix decomposed $\mathsf{ct}_{\mathrm{SIMD}}$ ciphertext into an RNS decomposed $\mathsf{ct}_{\mathrm{SIMD}}$ ciphertext.
H_RNSTORADIX	$\mathsf{ct}_{\mathrm{SIMD}}(\mathbf{m}_{\mathrm{RNS}})$	$\mathsf{ct}_{\mathrm{SIMD}}(\mathbf{m}_{\mathrm{radix}})$	Homomorphically convert an RNS decomposed $\mathsf{ct}_{\mathrm{SIMD}}$ ciphertext into a radix decomposed $\mathsf{ct}_{\mathrm{SIMD}}$ ciphertext.

Appendix D. ISA of the VMs

We summarize the ISAs of the proposed FHE VMs. As summarized in Table A2, SISD VMs support scalar operations including arithmetic, bitwise, shift, and decomposition base conversion instructions. As illustrated in Table A3,

Appendix Table A4: Example ISA for a Hybrid VM with an ω-bit Word Size.

Instruction	Input(s)	Output	Description
H_PACK	$\{\mathsf{ct}_{\mathrm{SISD}}(m_0), ..., \mathsf{ct}_{\mathrm{SISD}}(m_{n-1})\}$	$\mathsf{ct}_{\mathrm{SIMD}}(\mathbf{m})$	Homomorphically pack n $\mathsf{ct}_{\mathrm{SISD}}$ ciphertexts into one $\mathsf{ct}_{\mathrm{SIMD}}$ ciphertext where the slot form of $\mathbf{m}$ is $\{m_0, ..., m_{n-1}\}$,
H_FUN_PACK	$\{\mathsf{ct}_{\mathrm{SISD}}(m_0), ..., \mathsf{ct}_{\mathrm{SISD}}(m_{n-1})\}$	$\mathsf{ct}_{\mathrm{SIMD}}(f(\mathbf{m}))$	Homomorphically pack n $\mathsf{ct}_{\mathrm{SISD}}$ ciphertexts into one $\mathsf{ct}_{\mathrm{SIMD}}$ ciphertext and evaluate an arbitrary function f on the packed plaintext vector.
H_EXTRACT	$\mathsf{ct}_{\mathrm{SIMD}}(\mathbf{m}), i$	$\mathsf{ct}_{\mathrm{SISD}}(m_i)$	Homomorphically extract the i-th coefficient m_i from $\mathsf{ct}_{\mathrm{SIMD}}(\mathbf{m})$

the SIMD VM enables vectorized arithmetic, bitwise, shift, rotation, along with instructions for embedding conversion between slot and coefficient embedding and for decomposition base conversion. The hybrid FHE VM unifies the ISAs of both SISD and SIMD VMs, and therefore encompasses the ISAs defined in Table A2 and Table A3. In addition, to support ciphertexts from both SISD and SIMD VMs, the hybrid VM introduces a set of scheme conversion instructions, as proposed in Table A4.

Appendix E. Instantiations of FHE VMs

According to the data-processing model supported by the underlying FHE schemes, we instantiate the FHE VM as the SISD VM, the SIMD VM, and the hybrid VM. We assume that all instantiated VMs have a word size of ω, i.e., all VM instructions operate on data elements with a word size of ω bits.

E.1. SISD VM

The SISD VM is an instantiation of the FHE VM designed to process one unit of data at a time, reflecting the SISD property of the underlying FHE scheme. A SISD VM instantiates the data-processing components $(\mathcal{E}, \mathcal{I}, \mathcal{M})$ of the FHE VM tuple in Equation (14) as follows:

- $\mathcal{E}$ is an SISD FHE scheme, such as [55, 56].
- $\mathcal{I}$ consists of scalar operations including arithmetic operations (H_SCA_ADD, H_SCA_SUB, H_SCA_MUL), logical operations (H_SCA_AND, H_SCA_OR, H_SCA_NOT, H_SCA_XOR), and bitwise operations (H_SCA_SHL, H_SCA_SHR).
- Within the memory $\mathcal{M}$, $\mathcal{M}_{\mathrm{C}}$ stores the plaintexts (e.g., public constants) and both the $\mathsf{ct}_{\mathrm{SISD}}$ and the $\mathsf{ct}_{\mathrm{SIMD}}$ ciphertexts. However, as mentioned above, VM instructions only take $\mathsf{ct}_{\mathrm{SISD}}$ ciphertext groups as inputs. Meanwhile, $\mathcal{M}_{\mathrm{K}}$ stores the keys of the SISD-FHE scheme, including the key-switching keys and bootstrapping keys.

E.2. SIMD VM

The SIMD VM is an instantiation of the FHE VM designed to process multiple data units using a single instruction. A SIMD VM instantiates the data-processing components $(\mathcal{E}, \mathcal{I}, \mathcal{M})$ of the FHE VM tuple in Equation (14) as follows:

- $\mathcal{E}$ is a SIMD FHE scheme, such as [58, 61].
- $\mathcal{I}$ consists of vector operations including arithmetic operations (H_VEC_ADD, H_VEC_SUB, H_VEC_MUL), logical operations (H_VEC_AND, H_VEC_OR, H_VEC_NOT, H_VEC_XOR), bitwise operations (H_VEC_SHL, H_VEC_SHR), and data-shuffling operations (H_ROTATE).
- Within the memory $\mathcal{M}$, $\mathcal{M}_{\mathrm{C}}$ stores the plaintext data that include both the public constants and the pre-computed polynomial coefficients. $\mathcal{M}_{\mathrm{C}}$ also contains $\mathsf{ct}_{\mathrm{SIMD}}$ ciphertexts, and all instructions primarily operate on $\mathsf{ct}_{\mathrm{SIMD}}$. On the other hand, $\mathcal{M}_{\mathrm{K}}$ stores the keys of the SIMD-FHE scheme, including relinearization keys, Galois keys, and bootstrapping keys.

SIMD VMs offer high efficiency for data-parallel tasks, which can be useful in applications such as privacy-preserving machine learning and statistical data analysis.

E.3. Hybrid VM

The hybrid VM aims to combine the strengths of both the SISD and SIMD VMs, and instantiates the data-processing components $(\mathcal{E}, \mathcal{I}, \mathcal{M})$ of the FHE VM tuple in Equation (14) as follows:

- $\mathcal{E}$ is a composite FHE scheme that integrates both SISD FHE schemes and SIMD FHE schemes.
- $\mathcal{I}$ is a combined set of the instruction sets from both the SISD and SIMD VMs. A key difference is that $\mathcal{I}$ also includes scheme-switching instructions (H_PACK, H_EXTRACT) to convert between scalar and vector data representations.
- $\mathcal{M}$ represents the union of the memory components from both the SISD and SIMD VMs. Consequently, $\mathcal{M}_{\mathrm{C}}$ holds both single-unit and packed vector ciphertexts, while $\mathcal{M}_{\mathrm{K}}$ stores the combined set of keys required by both the SISD and SIMD FHE schemes.

A hybrid VM allows complex applications to leverage both the flexible control flow of SISD VMs for swift decision-making and the parallel computation capability of SIMD VMs for heavy arithmetic tasks.

Appendix F. Instantiating Control-Flow Security

In addition to the data-flow instantiation discussed in Appendix E, we also need to address the control-flow security of FHE programs, as introduced in Section 3.3. The distinction is based on whether the control flow is public

(ciphertext-independent) or encrypted (ciphertext-dependent), leading to the classification of transparent and oblivious VMs.

Transparent VM: A transparent VM is an FHE VM whose computational program conforms to a public, ciphertext-independent control-flow model. In such a program, although the sensitive data are encrypted, the control flow, i.e., which instructions are executed and in what order, is completely public. In other words, for a transparent VM, the state $\mathtt{q}_p \in \mathcal{Q}$ and the program π are public. Accordingly, the current instruction $\mathtt{i}_p \in \mathcal{I}$ can be determined from the public program and state. The execution cycle δ_p is instantiated as

$$\delta_p(\mathtt{q}_p, \mathcal{M}_{\mathrm{C}}, \mathtt{i}_p, \mathcal{M}_{\mathrm{K}}) = (\mathtt{q}'_p, \mathcal{M}'_{\mathrm{C}}), \tag{A1}$$

where $\mathtt{i}_p$ is fetched from $\mathcal{M}_{\mathrm{P}}$ according to the public program π and the current state $\mathtt{q}_p$.

Oblivious VM: An FHE VM is considered an oblivious VM if the control flow depends on one or more ciphertext values. Such a dependency prevents the execution path or total execution length from being determined in advance from public information, as the computational path cannot be fully unrolled at either compile time or run time without client-server interactions. To protect the sensitive control flow, the entire operational state, including the program, is typically maintained in encrypted form. Therefore, unlike transparent VMs, the program π is encrypted as $\mathsf{Enc}(\pi)$, while $q_e \in \mathcal{Q}$ denotes an encrypted machine state. The execution cycle, δ_e, executes an encrypted instruction $\mathcal{I}_e \in \mathcal{I}$, taking all components as ciphertext inputs to produce the next encrypted state. To formally characterize this process within a unified private mapping framework, we define an encrypted execution context $\mathcal{X}$ that encapsulates the encrypted program $\mathsf{Enc}(\pi)$, the encrypted instruction $\mathcal{I}_e$, and the current encrypted state $q_e \in \mathcal{Q}$:

$$\mathcal{X} = (\mathsf{Enc}(\pi), q_e, \mathcal{I}_e). \tag{A2}$$

The execution cycle δ_e is then instantiated over the private execution context $\mathcal{X}$ and the current memory $\mathcal{M}$, producing the updated execution context $\mathcal{X}'$ and memory $\mathcal{M}'$:

$$\delta_e(\mathcal{X}, \mathcal{M}) = (\mathcal{X}', \mathcal{M}'). \tag{A3}$$

By encapsulating the sensitive execution context in $\mathcal{X}$, the transition becomes ciphertext-dependent, while the underlying program logic remains hidden from the VM operator.

We note that most existing oblivious FHE VMs rely on client-server interactions to determine the termination states of the programs [14, 107]. How to build oblivious VMs entirely over FHE in a non-interactive manner remains an open question.

Appendix G. Comparing Hardware FHE Accelerators

Table A5 summarizes recent hardware accelerators for FHE VMs, categorized by their target VM types, architectures, and supported FHE schemes. In what follows, we outline the distinct trade-offs in functionality, performance, and cost for each of the acceleration platforms.

G.1. Application-Specific Accelerators for FHE VMs

In SISD VMs, ASIC implementations such as MATCHA [183] and Morphling [184] address the bootstrapping bottleneck in TFHE through FFT/IFFT optimizations. Both designs support SISD-FBS and logical operations, where the optimization of bootstrapping remains the decisive factor for system performance. Strix [185] utilizes a streaming and fully pipelined architecture to accelerate ciphertext processing in SISD-FHE. Matrix [186] balances off-chip bandwidth and on-chip storage according to the execution flow of SISD-FBS. For SIMD VMs, ASIC architectures such as Cheetah [191], F1 [188], CraterLake [189], ARK [190], SHARP [192], ABC-FHE [201] and BASALISC [202] target the main bottlenecks of large-scale polynomial arithmetic, namely NTT, key switching, and automorphism, through specialized functional units and extensive on-chip storage to enhance parallelism and data reuse. ISHEPC [193] extends ASIC parallelism to logical computation by encoding binary values into ciphertexts and using arithmetic operations to emulate bitwise logic, thus bridging logical and arithmetic domains within hardware. Lu et al. [194] utilize a reconfigurable processing element to flexibly support all operations for multi-scheme, reducing encryption latency and cutting storage requirements. FAST [195] integrates recently proposed cryptographic optimizations to propose a method that supports multiple key-switching. ASIC designs for Hybrid VMs further aim to merge SISD and SIMD models under a unified architecture. UFC [77] abstracts FHE functionalities into universal primitives, allowing shared hardware units to execute both SIMD-FHE and SISD-FHE workloads. Trinity [76] uses cluster-based configurable architecture and dynamic workload scheduling to flexibly allocate resources, achieving balanced performance across mixed workloads. Presto [197] uses the Module-Learning-with-Errors representation to unify diverse FHE schemes, decomposing their distinct operations into a complete, common set of vector operations to accelerate. APACHE [196] proposes a layered near-memory computing architecture to significantly enhance the utilization rates of both computational resources and memory bandwidth.

G.2. FPGA Design for FHE VMs

For SISD VMs, FPT [176] exploits the inherent noise tolerance of TFHE to replace resource-intensive floating-point or integer arithmetic with optimized fixed-point arithmetic, substantially improving SISD-FBS efficiency. By leveraging FPGA reconfigurability, FPT [176] enables fine-grained task-level parallelism in gate-level operations without excessive resource overhead. Zhang et al. [177] organize memory operations in a special way to facilitate the implementation of the external product. In SIMD VMs, FPGA accelerators such as HEAX [178] and Poseidon [179] address the high computational costs of large-scale NTTs, key switching, and data movement. FxHENN [180] further demonstrates FPGA efficiency for HE-based CNNs through intra-layer

Appendix Table A5: Summary of Recent FHE Hardware Designs

VM Type	Design	Arch.	Fabri.*/OSS**	FHE Scheme	Arith.	Div.	Auto.	Bit	Poly.	Hybrid Key
SISD VM	FPT [176]	FPGA	●	CGGI	○	○	N/A	●	N/A	N/A
	Zhang et al. [177]	FPGA	○	CGGI	○	○	N/A	●	N/A	N/A
	MATCHA [183]	ASIC	○	CGGI	○	○	N/A	●	N/A	N/A
	Morphling [184]	ASIC	○	CGGI	○	○	N/A	●	N/A	N/A
	Strix [185]	ASIC	○	CGGI	○	○	N/A	●	N/A	N/A
	Matrix [186]	ASIC	○	CGGI	○	○	N/A	●	N/A	N/A
	Memfhe [187]	ASIC	○	CGGI	●†	●†	N/A	●	N/A	N/A
	Concrete-cuda [162]	GPU	●	CGGI	●†	●†	N/A	●	N/A	N/A
	Narisada et al. [163]	GPU	●	CGGI	●†	○	N/A	○	N/A	N/A
	Xiao et al. [164]	GPU	○	CGGI	●†	●†	N/A	●	N/A	N/A
	Velofhe [165]	GPU	○	CGGI	●†	●†	N/A	●	N/A	N/A
SIMD VM	HEAX [178]	FPGA	○	CKKS	●	●	●	●♠	●	○
	Poseidon [179]	FPGA	○	CKKS	●	●	●	●♠	●	○
	FxHENN [180]	FPGA	○	CKKS	●	●	●	●♠	●	○
	FAB [181]	FPGA	○	CKKS	●	●	●	●♠	●	○
	Cheetah [191]	ASIC	○	BFV	●	●	●	●♠	●	○
	F1 [188]	ASIC	○	BGV/BFV/GSW/CKKS	●	●	●	●♠	●	○
	CraterLake [189]	ASIC	○	BGV/GSW/CKKS	●	●	●	●♠	●	○
	ARK [190]	ASIC	○	CKKS	●	●	●	●♠	●	○
	SHARP [192]	ASIC	○	CKKS	●	●	●	●♠	●	○
	ISHEPC [193]	ASIC	●	CKKS	●	○	●	●	○	○
	Lu et al. [194]	ASIC	●	BFV/BGV/CKKS	●	●	●	●♠	●	○
	FAST [195]	ASIC	○	CKKS	●	●	●	●♠	●	○
	ABC-FHE [201]	ASIC	○	CKKS	●	●	●	●♠	●	○
	BASALISC [202]	ASIC	●	BGV	●	●	●	●♠	●	○
	Lee et al. [203]	ASIC	●	CKKS	●	●	●	●♠	●	○
	Over100x [166]	GPU	◐	BFV/BGV/CKKS	●	●	●	●♠	●	○
	TensorFHE [167]	GPU	◐	BFV/BGV/CKKS	●	●	●	●♠	●	○
	GME [204]	GPU	○	CKKS	●	●	●	●♠	●	○
	HE-Booster [168]	GPU	○	BFV/BGV/CKKS	●	●	●	●♠	●	○
	Phantom [169]	GPU	●	BFV/BGV/CKKS	●	●	●	●♠	●	○
	Badawi et al. [172]	GPU	○	BFV/BGV	●	●	●	●♠	●	○
	HEonGPU [173]	GPU	●	BFV/BGV/CKKS	●	●	●	●♠	●	○
	WarpDrive [170]	GPU	○	BFV/BGV/CKKS	●	●	●	●♠	●	○
	Neo [171]	GPU	○	BFV/BGV/CKKS	●	●	●	●♠	●	○
	CAT [174]	GPU	●	BFV/BGV/CKKS	●	●	●	●♠	●	○
Hybrid VM	Trinity [76]	ASIC	○	CKKS/CGGI	●	●	●	●	●	●
	UFC [77]	ASIC	○	CKKS/CGGI	●	●	●	●	●	●
	Presto [197]	ASIC	●	BFV/BGV/CKKS/CGGI	●	●	●	●	●	○
	APACHE [196]	ASIC	●	BFV/BGV/CKKS/CGGI	●	●	●	●	●	○
	HEAP [205]	FPGA	○	CKKS/CGGI	●	●	●	●	●	●
	Chameleon [175]	GPU	○	CKKS/CGGI	●	●	●	●	●	●

*Physically taped out. **Open source or has released its core code.
†Implemented based on Boolean circuit. ♠Implemented based on arithmetic circuit.

pipelining and inter-layer resource reuse, sharing BRAM buffers and computation modules across layers to minimize memory footprint. FAB [181] leverages recent algorithms for bootstrapping while being cognizant of the computational and memory constraints. For Hybrid VMs, HEAP [205] uses smaller bootstrapping keys to reduce the amount of data to be read and a variety of hardware optimizations to improve computational performance.

G.3. GPU Design for FHE VMs

For SISD VMs, Concrete-cuda [162] provides a GPU backend that accelerates SISD-FBS and extends its capabilities to construct arithmetic operations, enabling efficient evaluation of complex Boolean circuits. Narisada et al. [163] enhance arithmetic performance in multi-threaded environments by exploiting redundant representation, reducing inter-thread dependencies and improving throughput. Xiao et al. [164] design a novel parallelization strategy allowing an automated optimization that partitions SISD-FBS into multiple GPU thread blocks. Velofhe [165] develops a hybrid four-step

NTT approach for fast polynomial multiplication. For SIMD VMs, Over100x [166], TensorFHE [167] and Phantom [169] focus on optimizing core bottlenecks such as NTT, key switching, and memory bandwidth limitations. GME [204] designs a custom GPU architecture to reduce data transfer and improve modular exponentiation efficiency. HE-Booter [168] introduces a new inter-thread synchronization mechanism to efficiently use thread-level parallelism for NTT. Badawi et al. [172] follow the data parallelism principle to evenly distribute the workload of SIMD-FHE across available GPUs, improving computational performance. HEonGPU [173] and CAT [174] both use memory management mechanisms to reduce the GPU memory requirements of SIMD-FHE applications. WarpDrive [170] and Neo [171] perform detailed optimizations of NTT on GPUs to improve the computation efficiency of large parameter polynomials. For Hybrid VMs, Chameleon [175] proposes a scalable NTT acceleration design that adapts to larger SIMD-FHE polynomials and smaller SISD-FHE polynomials.

Appendix H. Detailed Benchmark Parameter Settings

The cryptographic parameters used in our benchmarks are organized according to the three FHE VM types: SISD, SIMD, and hybrid VMs. We report the major cryptographic parameters that determine the computational scale and ciphertext representation of each model, while omitting low-level implementation parameters.

H.1. SISD VMs

The parameter settings for the evaluated SISD VMs are as follows.

`SISD-Bool` Model: The `SISD-Bool` model uses a polynomial degree of $N = 2048$ and a 64-bit torus representation, corresponding to a native ciphertext modulus of 2^{64}. The programmable bootstrapping decomposition uses 2 levels with a base log of 11. The plaintext modulus is 8, corresponding to a 3-bit plaintext space, and the model does not use a separate carry modulus.

`SISD-Carry` Model: The `SISD-Carry` model uses a polynomial degree of $N = 2048$ and a 64-bit torus representation, corresponding to a native ciphertext modulus of 2^{64}. The programmable bootstrapping decomposition uses 1 level with a base log of 23. The message modulus and carry modulus are both 4, providing 4 bits of combined message-and-carry capacity per radix block.

`SISD-Tree` Model: The `SISD-Tree` model uses a polynomial degree of $N = 2048$ and a 64-bit torus representation, corresponding to a native ciphertext modulus of 2^{64}. Depending on the evaluation route, programmable bootstrapping uses 2 levels with a base log of 15 or 3 levels with a base log of 12. The plaintext modulus is 2, corresponding to a 1-bit Boolean plaintext space, and the model does not use a separate carry modulus.

H.2. SIMD VMs

The parameter settings for the evaluated SIMD VMs are as follows.

`SIMD-Poly` Model: The `SIMD-Poly` model uses a polynomial modulus degree of $N = 65536$ and supports 32,768 SIMD slots. The ciphertext modulus is approximately $\log_2 Q = 1240$ bits, with 20 evaluation levels. The modulus size is configured with 59-bit scaling moduli and a 60-bit first modulus.

`SIMD-Bool` Model: The `SIMD-Bool` model uses a polynomial modulus degree $N = 131072$ and supports 65,536 SIMD slots. The ciphertext modulus is approximately $\log_2 Q = 1535$ bits, with 25 evaluation levels. The modulus size is configured with 59-bit scaling moduli and a 60-bit first modulus.

`SIMD-Carry` Model: The `SIMD-Carry` model uses a polynomial modulus degree $N = 65536$ and supports 32,768 SIMD slots. The ciphertext modulus is approximately $\log_2 Q = 2243$ bits, with 37 evaluation levels. The modulus size is configured with 59-bit scaling moduli and a 60-bit first modulus.

H.3. Hybrid VMs

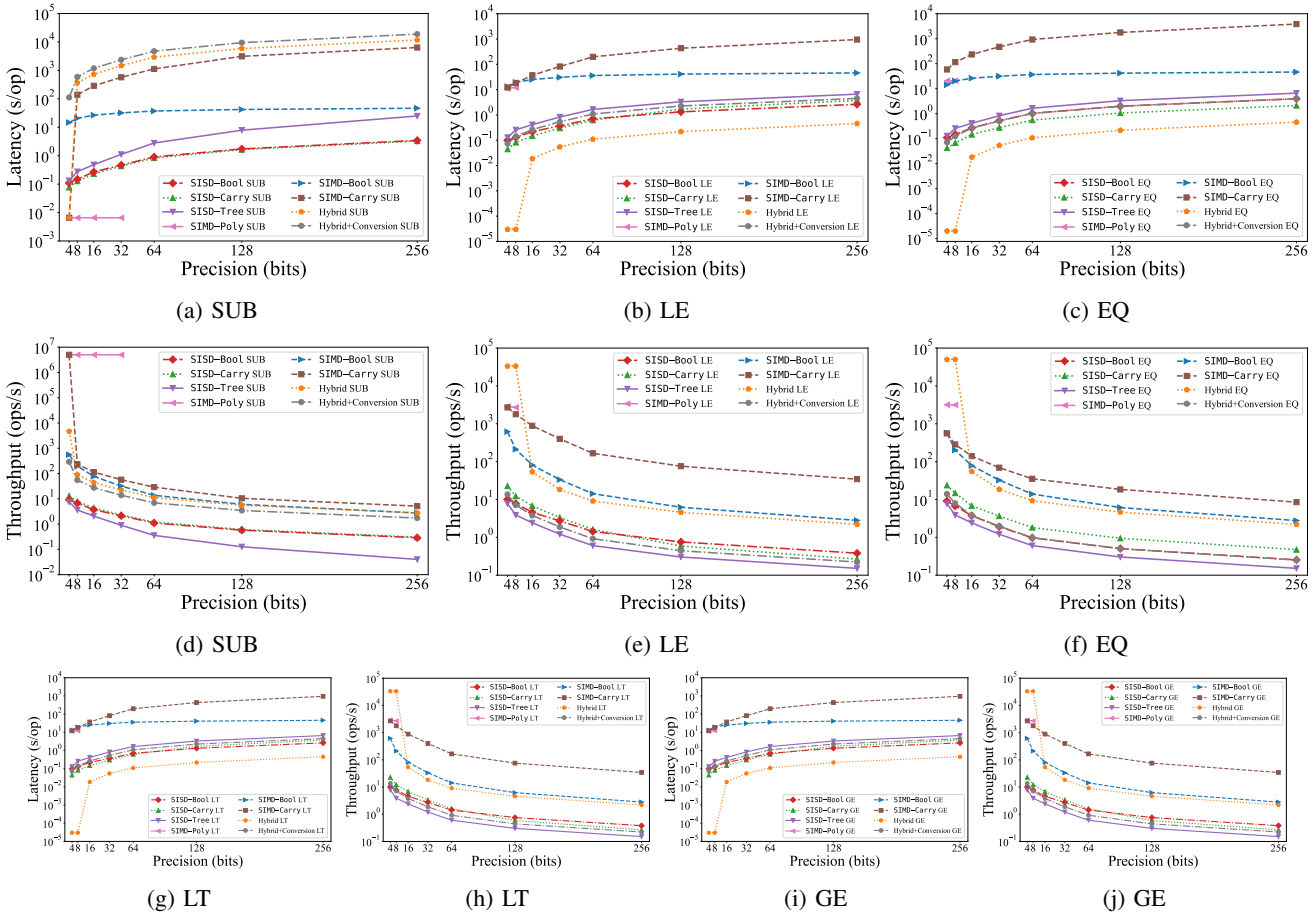
The hybrid model combines SISD and SIMD computation. The SISD component uses a polynomial degree of $N = 1024$, a 32-bit torus representation, and a programmable bootstrapping decomposition with 3 levels and a base log of 6. Its plaintext modulus is 8, corresponding to a 3-bit plaintext space, while comparison outputs are Boolean ciphertexts. The SIMD component uses a polynomial degree of $N = 65536$ with 32,768 SIMD slots. The coefficient modulus is approximately $\log_2 Q = 949$ bits.

Appendix I. Supplementary Instruction Experiments

In the main text, we benchmark only addition (ADD), multiplication (MUL), and greater-than comparison (GT). Here, we further extend the experiments to include subtraction (SUB) and four additional comparison operations: less-than (LT), greater-than-or-equal-to (GE), less-than-or-equal-to (LE), and equality (EQ). The latency and throughput curves of these five instructions under different programming models are shown in Figure A1.

Across all VM models, SUB exhibits nearly the same latency as ADD. This consistency arises because both operations share similar arithmetic structures. Consequently, SUB and ADD display equivalent computational costs and scaling trends across precision levels.

For comparison operations, LT, GE, and LE follow the same latency pattern as GT, since these comparisons share common underlying logic or can be converted among one another, for instance, $A < B$ can be transformed into $B > A$, $A \geq B$ can be implemented as $\neg(A < B)$, and $A \leq B$ can be implemented as $\neg(A > B)$. Their execution paths



(a) SUB (b) LE (c) EQ

(d) SUB (e) LE (f) EQ

(g) LT (h) LT (i) GE (j) GE

Appendix Figure A1: The performance of (a, d) subtraction, (b, e) less-than-or-equal-to, (c, f) equality (g, h) less-than, and (i, j) greater-than-or-equal-to, instructions in terms of latency and throughput respectively.

therefore share nearly the same computational structure, leading to overlapping latency curves.

In contrast, EQ and GT differ in their underlying computational logic: EQ evaluates equality across all relevant bits or values, while GT is based on comparing relative magnitude. As a result, the latency and throughput of EQ differ from those of GT.

To further analyze the trade-offs among different programming models beyond individual latency and throughput measurements, we visualize the frequency-throughput trade-offs of ADD and GT instructions in Figure A2, where execution frequency is defined as $1/\text{latency}$. The results show that different VM models occupy distinct regions in the performance space. SISD, SIMD, and hybrid models exhibit different balances between execution frequency and throughput, reflecting their underlying execution granularity and parallelism strategies.

Appendix J. Program Benchmark Details

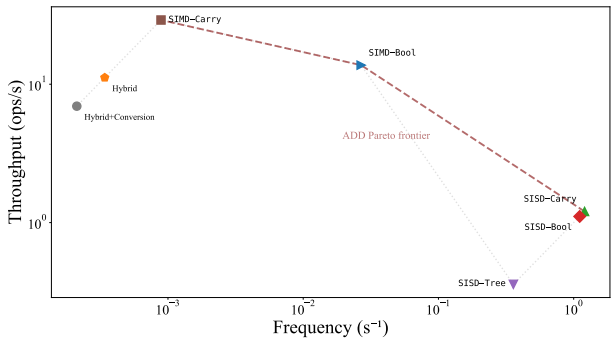
We further characterize P1 and P2 from multiple aspects to better understand the representativeness of our benchmarks.

Background on Benchmarks: The ERC-20 workload (P1) models a simplified token transfer operation in a blockchain system, where each transaction updates account balances subject to a balance sufficiency constraint. Specifically, given an account A transferring amount m to account B, the computation is defined as:

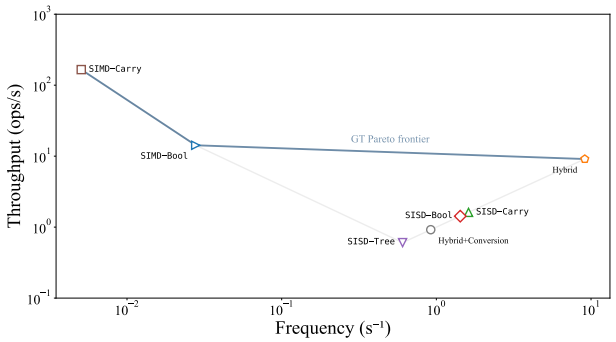
$$\begin{aligned} \text{sign} &= \mathbb{I}(A_{\text{balance}} \geq m), \\ m' &= m \cdot \text{sign}, \\ A'_{\text{balance}} &= A_{\text{balance}} - m', \\ B'_{\text{balance}} &= B_{\text{balance}} + m', \end{aligned}$$

where $\mathbb{I}(\cdot)$ is an indicator function that outputs 1 if the condition holds and 0 otherwise. The variable sign acts as a gating factor that conditionally enables the transfer: when $A_{\text{balance}} \geq m$, the transaction proceeds with $m' = m$; otherwise, $m' = 0$, and the balances remain unchanged. This workload involves both comparison operations and arithmetic updates, and introduces conditional dependencies that reduce the available parallelism.

TPC-H Query 6 (P2) is a query from the TPC-H benchmark that computes total revenue over a filtered subset



(a) ADD



(b) GT

Appendix Figure A2: Frequency-throughput trade-offs of FHE VM models for addition and greater-than instructions.

of records:

$$\text{Revenue} = \sum \left(l_{\text{extendedprice}} \cdot l_{\text{discount}} \right),$$

subject to the constraints:

$$\begin{aligned} l_{\text{shipdate}} &\in [d, d + 1 \text{ year}), \\ l_{\text{discount}} &\in [p - 0.01, p + 0.01], \\ l_{\text{quantity}} &< t. \end{aligned}$$

Here, d, p, and t denote external query parameters. The computation is dominated by element-wise multiplication, filtering, and global aggregation over a large relation, making it highly suitable for SIMD-style batch processing.

SIMD Utilization: Due to potential data dependencies between accounts in real transaction scenarios, P1 generally offers limited batching opportunities, reflecting practical constraints in transactional workloads. Accordingly, we use three parallel transactions as a representative small-batch setting. In contrast, P2 extensively exploits SIMD parallelism by packing 32,768 database entries into ciphertext slots.

Implementation Details: Both workloads are implemented directly on top of the underlying FHE libraries rather than high-level domain-specific languages, allowing explicit control over the generated homomorphic operations and VM instructions.

Program Size: The implementations of P1 and P2 contain approximately 700 and 1,100 lines of benchmark-specific code, respectively, across the evaluated FHE libraries. The implementations are written in C++ or Rust depending on the underlying library.

References

[1] M. Albrecht, M. Chase, H. Chen, J. Ding, S. Goldwasser, S. Gorbunov, S. Halevi, J. Hoffstein, K. Laine, K. Lauter, S. Lokam, D. Micciancio, D. Moody, T. Morrison, A. Sahai, and V. Vaikuntanathan, *Homomorphic Encryption Standard*, pp. 31–62. Springer International Publishing, 2021.

[2] J.-P. Bossuat, R. Cammarota, I. Chillotti, B. Curtis, W. Dai, H. Gong, E. Hales, D. Kim, B. Kumara, C. Lee, X. Lu, C. Maple, A. Pedrouzo-Ulloa, R. Player, Y. Polyakov, L. Ruiz, Y. Song, and D. Yhee, "Security guidelines for implementing homomorphic encryption," *IACR CiC*, 2025.

[3] Apple Inc., "Private cloud compute security guide." https://security.apple.com/documentation/private-cloud-compute, 2024.

[4] Zama, "FHEVM: A cross-chain protocol for confidential smart contracts." https://github.com/zama-ai/fhevm/blob/main/fhevm-whitepaper.pdf, 2025.

[5] Fhenix, "CoFHE: Confidential computation for EVM-compatible blockchains." https://www.fhenix.io/, 2026. Accessed: 2026-08-15.

[6] J. Kun, "How google is making private ai practical with homomorphic encryption," 2026. https://blog.google/security/how-google-is-making-private-ai-practical-with-homomorphic-encryption/.

[7] W. Kim, C. Lee, J. Lee, A. Passelègue, and D. Stehlé, "Asynchronous lagrange-based threshold FHE with smaller modulus overhead," in *CRYPTO* (N. Heninger and M. Rosulek, eds.), vol. 16801 of *Lecture Notes in Computer Science*, (Cham), pp. 3–35, Springer, 2026.

[8] C. Gentry, "Fully homomorphic encryption using ideal lattices," in *STOC*, (New York, NY, USA), p. 169–178, Association for Computing Machinery, 2009.

[9] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. Vadhan, and K. Yang, "On the (im)possibility of obfuscating programs," *J. ACM*, vol. 59, May 2012.

[10] D. Boneh, A. Sahai, and B. Waters, "Functional encryption: Definitions and challenges," in *Theory of Cryptography*, pp. 253–273, 2011.

[11] Y. Huang, D. Evans, and J. Katz, "Private set intersection: Are garbled circuits better than custom protocols?," in *NDSS*, 2012.

[12] B. Chor, O. Goldreich, E. Kushilevitz, and M. Sudan, "Private information retrieval," in *FOCS*, (USA), p. 41, IEEE Computer Society, 1995.

[13] O. Goldreich and R. Ostrovsky, "Software protection and simulation on oblivious rams," *J. ACM*, vol. 43, p. 431–473, May 1996.

[14] K. Matsuoka, R. Banno, N. Matsumoto, T. Sato, and S. Bian, "Virtual secure platform: A five-stage pipeline

processor over TFHE," in *USENIX Security*, pp. 4007–4024, 2021.

[15] P. Mishra, R. Lehmkuhl, A. Srinivasan, W. Zheng, and R. A. Popa, "Delphi: A cryptographic inference service for neural networks," in *USENIX Security*, pp. 2505–2522, 2020.

[16] Z. Huang, W. Lu, C. Hong, and J. Ding, "Cheetah: Lean and fast secure two-party deep neural network inference," in *USENIX Security*, pp. 809–826, 2022.

[17] S. Bian, W. Jiang, Q. Lu, Y. Shi, and T. Sato, "NASS: Optimizing secure inference via neural architecture search," in *ECAI*, pp. 1746–1753, 2020.

[18] S. Bian, T. Wang, M. Hiromoto, Y. Shi, and T. Sato, "ENSEI: Efficient secure inference via frequency-domain homomorphic convolution for privacy-preserving visual recognition," in *CVPR*, pp. 9400–9409, 2020.

[19] W. jie Lu, Z. Huang, Z. Gu, J. Li, J. Liu, K. Ren, C. Hong, T. Wei, and W. Chen, "BumbleBee: Secure two-party inference framework for large transformers," *IACR ePrint*, 2023.

[20] R. A. Popa, C. M. S. Redfield, N. Zeldovich, and H. Balakrishnan, "CryptDB: Protecting confidentiality with encrypted query processing," in *SOSP*, p. 85–100, 2011.

[21] S. Eskandarian and M. Zaharia, "ObliDB: oblivious query processing for secure databases," in *VLDB*, p. 169–183, 2019.

[22] T. Hackenjos, F. Hahn, and F. Kerschbaum, "SAGMA: Secure aggregation grouped by multiple attributes," in *SIGMOD*, pp. 587–601, 2020.

[23] Q. Lou and L. Jiang, "She: A fast and accurate deep neural network for encrypted data," in *NeuralIPS*, 2019.

[24] R. Banno, K. Matsuoka, N. Matsumoto, S. Bian, M. Waga, and K. Suenaga, "Oblivious online monitoring for safety LTL specification via fully homomorphic encryption," in *CAV*, pp. 447–468, 2022.

[25] L. Folkerts, C. Gouert, and N. G. Tsoutsos, "REDsec: Running encrypted discretized neural networks in seconds," in *NDSS*, 2023.

[26] W. Lu, Z. Huang, C. Hong, Y. Ma, and H. Qu, "PEGASUS: Bridging polynomial and non-polynomial evaluations in homomorphic encryption," in *IEEE S&P*, pp. 1057–1073, 2021.

[27] K. Cong, D. Das, J. Park, and H. V. L. Pereira, "SortingHat: Efficient private decision tree evaluation via homomorphic encryption and transciphering," in *ACM CCS*, pp. 563–577, 2022.

[28] R. Akhavan Mahdavi, H. Ni, D. Linkov, and F. Kerschbaum, "Level up: Private non-interactive decision tree evaluation using levelled homomorphic encryption," in *ACM CCS*, pp. 2945–2958, 2023.

[29] Q. Lou and L. Jiang, "HEMET: A homomorphic-encryption-friendly privacy-preserving mobile neural network architecture," in *ICML*, pp. 7102–7110, 2021.

[30] X. Ren, L. Su, Z. Gu, S. Wang, F. Li, Y. Xie, S. Bian, C. Li, and F. Zhang, "HEDA: Multi-attribute unbounded aggregation over homomorphically encrypted database," *VLDB*, pp. 601–614, 2022.

[31] S. Bian, Z. Zhang, H. Pan, R. Mao, Z. Zhao, Y. Jin, and Z. Guan, "HE^{3}DB: An efficient and elastic encrypted database via arithmetic-and-logic fully homomorphic encryption," in *ACM CCS*, pp. 2930–2944, 2023.

[32] Z. Zhang, S. Bian, Z. Zhao, R. Mao, H. Zhou, J. Hua, Y. Jin, and Z. Guan, "ArcEDB: An arbitrary-precision encrypted database via (amortized) modular homomorphic encryption," in *ACM CCS*, 2024.

[33] N. P. Smart and F. Vercauteren, "Fully homomorphic SIMD operations," *Des. Codes Cryptogr.*, vol. 71, no. 1, pp. 57–81, 2014.

[34] Microsoft Research, "Microsoft SEAL (release 3.7)." https://github.com/Microsoft/SEAL, Sept. 2021.

[35] I. Chillotti, M. Joye, and P. Paillier, "Programmable bootstrapping enables efficient homomorphic inference of deep neural networks," in *CSCML*, pp. 1–19, 2021.

[36] Z. Yang, C. Niu, B. Wei, Z. Huang, C. Hong, and T. Wei, "RBOOT: Accelerating homomorphic neural network inference by fusing ReLU within bootstrapping." Cryptology ePrint Archive, Paper 2025/1534, 2025.

[37] S. Bian, Y. Fu, D. Zhao, H. Pan, Y. Jin, J. Sun, H. Qiao, and Z. Guan, "FHECAP: An encrypted control system with piecewise continuous actuation," *TIFS*, vol. 20, pp. 4551–4566, 2025.

[38] Z. Liu, D. Micciancio, and Y. Polyakov, "Large-precision homomorphic sign evaluation using FHEW/TFHE bootstrapping," in *ASIACRYPT*, (Cham), pp. 130–160, Springer Nature Switzerland, 2022.

[39] A. Alexandru, A. Kim, and Y. Polyakov, "General functional bootstrapping using CKKS," in *CRYPTO*, p. 304–337, 2025.

[40] S. Bian, Y. Fu, R. Shen, H. Pan, A. Wang, and Z. Guan, "High-precision functional bootstrapping for CKKS from fourier extension," in *EUROCRYPT*, pp. 274–303, Springer, 2026.

[41] S. Bian, H. Pan, J. Hu, Z. Zhang, Y. Fu, J. Hua, Y. Chen, B. Zhang, Y. Jin, J. Dong, and Z. Guan, "Engorgio: An Arbitrary-Precision Unbounded-Size hybrid encrypted database via quantized fully homomorphic encryption," in *USENIX Security*, (Seattle, WA), pp. 8441–8460, USENIX Association, Aug. 2025.

[42] C. Niu, B. Wei, Z. Huang, Z. Yang, C. Hong, M. Wang, and T. Wei, "SoK: FHE-friendly symmetric ciphers and transciphering," *IACR TCHES*, vol. 2025, no. 3, pp. 583–613, 2025.

[43] L. K. L. Ng and S. S. M. Chow, "SoK: Cryptographic neural-network computation," in *IEEE S&P*, pp. 497–514, 2023.

[44] M. Zbudila, A. Suresh, H. Yalame, O. Mirzamohammadi, A. Abidin, and B. Preneel, "SoK: Connecting the dots in privacy-preserving ML - systematization of MPC protocols and conversions between secret sharing schemes." Cryptology ePrint Archive, Paper

2025/1679, 2025.

[45] J. Xue, X. Xin, W. Zhang, M. Zheng, Q. Song, M. Zhou, Y. Dong, D. Wang, X. Chen, J. Xie, L. Wang, D. Mohaisen, H. Wu, and Q. Lou, "SoK: Can fully homomorphic encryption support general AI computation? A functional and cost analysis," 2025.

[46] W. Liu, L. You, Y. Shao, X. Shen, G. Hu, J. Shi, and S. Gao, "From accuracy to approximation: A survey on approximate homomorphic encryption and its applications," *Computer Science Review*, vol. 55, p. 100689, 2025.

[47] C. Marcolla, V. Sucasas, M. Manzano, R. Bassoli, F. H. Fitzek, and N. Aaraj, "Survey on fully homomorphic encryption, theory, and applications," *Proceedings of the IEEE*, vol. 110, no. 10, pp. 1572–1609, 2022.

[48] A. Adir, E. Aharoni, N. Drucker, R. Levy, H. Shaul, and O. Soceanu, *Homomorphic Encryption for Data Science (HE4DS)*. Professional and Applied Computing, Cham: Springer Cham, 2024.

[49] C. Gouert, D. Mouris, and N. G. Tsoutsos, "SoK: New insights into fully homomorphic encryption libraries via standardized benchmarks." Cryptology ePrint Archive, Paper 2022/425, 2022.

[50] H. Shen, Q. Xu, B. Yu, Y. Yang, and W. He, "Bootstrapping in approximate fully homomorphic encryption: a research survey," *Cybersecurity*, vol. 8, no. 1, p. 87, 2025.

[51] D. Aronoff, A. Bhat, P. Chatzigiannis, M. Minaei, S. Raghuraman, R. M. Townsend, and N. X.-Y. Zhang, "SoK: Fully-homomorphic encryption in smart contracts." Cryptology ePrint Archive, Paper 2025/527, 2025.

[52] A. Viand, P. Jattke, and A. Hithnawi, "SoK: Fully homomorphic encryption compilers," in *IEEE S&P*, pp. 1092–1108, IEEE, 2021.

[53] J. Zhang, X. Cheng, L. Yang, J. Hu, X. Liu, and K. Chen, "SoK: Fully homomorphic encryption accelerators," *ACM Comput. Surv.*, vol. 56, Oct. 2024.

[54] J. Alperin-Sheriff and C. Peikert, "Faster bootstrapping with polynomial error," in *CRYPTO*, pp. 297–314, Springer, 2014.

[55] L. Ducas and D. Micciancio, "FHEW: Bootstrapping homomorphic encryption in less than a second," in *EUROCRYPT*, pp. 617–640, 2015.

[56] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène, "TFHE: Fast fully homomorphic encryption over the torus," *J. Cryptol.*, vol. 33, no. 1, pp. 34–91, 2020.

[57] N. Gama, M. Izabachène, P. Q. Nguyen, and X. Xie, "Structural lattice reduction: Generalized worst-case to average-case reductions and homomorphic cryptosystems," in *EUROCRYPT*, pp. 528–558, 2016.

[58] J. Fan and F. Vercauteren, "Somewhat practical fully homomorphic encryption," *IACR ePrint*, 2012.

[59] Z. Brakerski, "Fully homomorphic encryption without modulus switching from classical gapsvp," in *CRYPTO*, pp. 868–886, 2012.

[60] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, "(leveled) fully homomorphic encryption without bootstrapping," in *ITCS*, pp. 309–325, 2012.

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

[62] C. Gentry and Y. Lee, "Fully homomorphic encryption for matrix arithmetic," in *CRYPTO*, pp. 363–395, Springer, 2026.

[63] Z. Liu and Y. Wang, "Relaxed functional bootstrapping: A new perspective on BGV/BFV bootstrapping," in *International conference on the theory and application of cryptology and information security*, pp. 208–240, Springer, 2024.

[64] I. Iliashenko and V. Zucca, "Faster homomorphic comparison operations for bgv and bfv," *Proceedings on Privacy Enhancing Technologies*, vol. 2021, no. 3, pp. 246–264, 2021.

[65] M. Gao and H. Zheng, "FHE for SIMD arithmetic logic units with amortized $o(1)$ bootstrapping per ciphertext." CRYPTO, 2026.

[66] Y. Bae, J. H. Cheon, J. Kim, and D. Stehlé, "Bootstrapping bits with CKKS," in *EUROCRYPT*, p. 94–123, 2024.

[67] Y. Lindell, *Tutorials on the Foundations of Cryptography*. Springer, 2017.

[68] M. Joye, "SoK: Fully homomorphic encryption over the [discretized] torus," *TCHES*, pp. 661–692, 2022.

[69] M. Creeger, "The rise of fully homomorphic encryption: Often called the holy grail of cryptography, commercial fhe is near.," *Queue*, vol. 20, no. 4, pp. 39–60, 2022.

[70] E. Bezuglova and N. Kucherov, "An overview of modern fully homomorphic encryption schemes," in *APAMCS*, pp. 300–311, Springer, 2022.

[71] T. V. T. Doan, M.-L. Messai, G. Gavin, and J. Darmont, "A survey on implementations of homomorphic encryption schemes," *J Supercomput*, vol. 79, pp. 15098–15139, 2023.

[72] S. Bian, Z. Zhao, Z. Zhang, R. Mao, K. Suenaga, Y. Jin, Z. Guan, and J. Liu, "HEIR: A unified representation for cross-scheme compilation of fully homomorphic computation," in *NDSS*, 2024.

[73] Google, "HEIR: Homomorphic encryption intermediate representation." https://github.com/google/heir, 2024.

[74] E. Chielle, O. Mazonka, N. G. Tsoutsos, and M. Maniatakos, "E^3: A framework for compiling C++ programs with encrypted operands," *IACR ePrint*, 2018.

[75] S. Gorantala, R. Springer, and S. P. , et. al., "A general purpose transpiler for fully homomorphic encryption," *IACR ePrint*, 2021.

[76] X. Deng, S. Fan, Z. Hu, Z. Tian, Z. Yang, J. Yu, D. Cao, D. Meng, R. Hou, M. Li, *et al.*, "Trinity: A general purpose fhe accelerator," in *MICRO*, pp. 338–351, IEEE, 2024.

[77] M. Zhou, Y. Nam, X. Wang, Y. Lee, C. Wilkerson, R. Kumar, S. Taneja, S. Mathew, R. Cammarota,

and T. Rosing, "UFC: A unified accelerator for fully homomorphic encryption," in *MICRO*, pp. 352–365, IEEE, 2024.
[78] G. Asharov, A. Jain, A. López-Alt, E. Tromer, V. Vaikuntanathan, and D. Wichs, "Multiparty computation with low communication, computation and interaction via threshold fhe," in *EUROCRYPT*, pp. 483–501, Springer, 2012.
[79] D. Boneh, R. Gennaro, S. Goldfeder, A. Jain, S. Kim, P. M. Rasmussen, and A. Sahai, "Threshold cryptosystems from threshold fully homomorphic encryption," in *CRYPTO*, pp. 565–596, Springer, 2018.
[80] M. Dahl, D. Demmler, S. El Kazdadi, A. Meyre, J.-B. Orfila, D. Rotaru, N. P. Smart, S. Tap, and M. Walter, "Noah's ark: Efficient threshold-FHE using noise flooding," in *WAHC*, pp. 35–46, 2023.
[81] C. Mouchet, S. Chatel, A. Pyrgelis, and C. Troncoso, "Helium: Scalable MPC among lightweight participants and under churn," in *ACM CCS*, pp. 3038–3052, 2024.
[82] A. López-Alt, E. Tromer, and V. Vaikuntanathan, "On-the-fly multiparty computation on the cloud via multikey fully homomorphic encryption," in *STOC*, pp. 1219–1234, 2012.
[83] P. Mukherjee and D. Wichs, "Two round multiparty computation via multi-key fhe," in *EUROCRYPT*, pp. 735–763, Springer, 2016.
[84] T. Kim, H. Kwak, D. Lee, J. Seo, and Y. Song, "Asymptotically faster multi-key homomorphic encryption from homomorphic gadget decomposition," in *ACM CCS*, pp. 726–740, 2023.
[85] B. Xiang, J. Zhang, K. Wang, Y. Deng, and D. Feng, "NTRU-based bootstrapping for mk-fhes without using overstretched parameters," in *ASIACRYPT*, pp. 241–270, Springer, 2024.
[86] M. Clear and C. McGoldrick, "Multi-identity and multi-key leveled fhe from learning with errors," in *CRYPTO*, pp. 630–656, Springer, 2015.
[87] B. Xiang, S. Min, I. Hwang, Z. Wang, H. He, Y. Wei, K. Yang, J. Zhang, Y. Deng, and Y. Yu, "HERDS: Multi-key fully homomorphic encryption with sublinear bootstrapping," in *EUROCRYPT*, (Cham), pp. 214–242, Springer Nature Switzerland, 2026.
[88] H. Chen, I. Chillotti, and Y. Song, "Multi-key homomorphic encryption from TFHE," in *ASIACRYPT*, pp. 446–472, Springer, 2019.
[89] S. Badrinarayanan, A. Jain, N. Manohar, and A. Sahai, "Secure MPC: laziness leads to god," in *ASIACRYPT*, pp. 120–150, Springer, 2020.
[90] J. Hoffstein, J. Pipher, and J. H. Silverman, "NTRU: A ring-based public key cryptosystem," in *ANTS*, pp. 267–288, Springer, 1998.
[91] D. Stehlé and R. Steinfeld, "Making NTRU as secure as worst-case problems over ideal lattices," in *EUROCRYPT*, pp. 27–47, Springer, 2011.
[92] F.-H. Liu and H. Wang, "Batch bootstrapping I: a new framework for simd bootstrapping in polynomial modulus," in *EUROCRYPT*, pp. 321–352, Springer, 2023.
[93] F.-H. Liu and H. Wang, "Batch bootstrapping II: bootstrapping in polynomial modulus only requires o˜(1) fhe multiplications in amortization," in *EUROCRYPT*, pp. 353–384, Springer, 2023.
[94] D. Micciancio and J. Sorrell, "Ring packing and amortized FHEW bootstrapping," in *ICALP*, pp. 100:1–100:14, 2018.
[95] J. Park and S. Rovira, "Efficient TFHE bootstrapping in the multiparty setting," *IEEE Access*, vol. 11, pp. 118625–118638, 2023.
[96] A. Guimarães, E. Borin, and D. F. Aranha, "Revisiting the functional bootstrap in TFHE," *IACR CHES*, vol. 2021, no. 2, pp. 229–253, 2021.
[97] H. Okada, S. Kiyomoto, and C. Cid, "Integer-wise functional bootstrapping on TFHE: applications in secure integer arithmetics," *Information*, vol. 12, no. 8, p. 297, 2021.
[98] K. Kluczniak and L. Schild, "FDFB: Full domain functional bootstrapping towards practical fully homomorphic encryption," *IACR CHES*, vol. 2023, no. 1, pp. 501–537, 2023.
[99] A. Badawi, J. Bates, F. Bergamaschi, *et al.*, "OpenFHE: Open-source fully homomorphic encryption library," in *WAHC*, pp. 53–63, 2022.
[100] S. Bian, D. Kundi, K. Hirozawa, W. Liu, and T. Sato, "APAS: Application-specific accelerators for RLWE-based homomorphic linear transformations," *IEEE TIFS*, vol. 16, pp. 4663–4678, 2021.
[101] D. Lee, S. Min, and Y. Song, "Functional bootstrapping for packed ciphertexts via homomorphic LUT evaluation," *Cryptology ePrint Archive*, 2024.
[102] J. Dumezy, A. Alexandru, Y. Polyakov, P. Clet, O. Chakraborty, and A. Boudguiga, "Evaluating larger lookup tables using CKKS," *IACR Cryptol. ePrint Arch.*, p. 1301, 2025.
[103] S. Ma, T. Huang, A. Wang, and X. Wang, "Practical dense-key bootstrapping with subring secret encapsulation," in *ASIACRYPT*, pp. 101–130, 2025.
[104] R. Canetti, "Security and composition of multiparty cryptographic protocols," *J. Cryptology*, vol. 13, pp. 143–202, 2000.
[105] Z. Zhou, Y. Li, Y. Wang, Z. Yang, B. Zhang, C. Hong, T. Wei, and W. Chen, "ZHE: Efficient zero-knowledge proofs for HE evaluations," in *IEEE S&P*, pp. 3328–3346, IEEE, 2025.
[106] S. Bian, Z. Zhao, R. Shen, Z. Zhang, R. Mao, D. Li, Y. Liu, M. Waga, K. Suenaga, Z. Guan, J. Hua, Y. Jin, and J. Liu, "Chloe: Loop transformation over fully homomorphic encryption via multi-level vectorization and control-path reduction," in *IEEE S&P*, 2025.
[107] A. Chatterjee and I. Sengupta, "FURISC: FHE encrypted URISC design." Cryptology ePrint Archive, Paper 2015/699, 2015.
[108] Z. Guan, R. Mao, Q. Zhang, Z. Zhang, Z. Zhao, and S. Bian, "Autohog: Automating homomorphic gate design for large-scale logic circuit evaluation," *IEEE TCAD*, 2024.
[109] A. Singh, S. Das, A. Chakraborty, R. Sadhukhan,

A. Chatterjee, and D. Mukhopadhyay, "FHEDA: Efficient circuit synthesis with reduced bootstrapping for torus FHE," in *EuroSP*, pp. 841–859, 2024.

[110] M. Izabachène and J.-P. Bossuat, "TETRIS: Composing fhe techniques for private functional exploration over large datasets," 2024.

[111] R. Wang, J. Ha, X. Shen, X. Lu, C. Chen, K. Wang, and J. Lee, "Refined TFHE leveled homomorphic evaluation and its application," in *ACM CCS*, pp. 2309–2323, 2025.

[112] R. Wang, Y. Wen, Z. Li, X. Lu, B. Wei, K. Liu, and K. Wang, "Circuit bootstrapping: Faster and smaller," in *EUROCRYPT*, p. 342–372, 2024.

[113] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène, "Improving TFHE: faster packed homomorphic operations and efficient circuit bootstrapping." Cryptology ePrint Archive, Paper 2017/430, 2017.

[114] Zama, "FHEVM: A full-stack framework for integrating fully homomorphic encryption (FHE) with blockchain applications," 2025. https://github.com/zama-ai/fhevm.

[115] J. Dumezy, N. Ye, P.-E. Clet, O. Chakraborty, and A. Boudguiga, "HEGIDE: A MIMD oblivious processor for private function evaluation over CKKS." Cryptology ePrint Archive, Paper 2026/1187, 2026.

[116] J. H. Cheon, D. Kim, and D. Kim, "Efficient homomorphic comparison methods with optimal complexity," in *ASIACRYPT 2020*, p. 221–256, 2020.

[117] J. H. Cheon, D. Kim, D. Kim, H. Lee, and K. Lee, "Numerical method for comparison on homomorphically encrypted numbers," in *ASIACRYPT*, pp. 415–445, 2019.

[118] M. Cowan, D. Dangwal, A. Alaghi, C. Trippel, V. T. Lee, and B. Reagen, "Porcupine: a synthesizing compiler for vectorized homomorphic encryption," in *PLDI*, pp. 375–389, 2021.

[119] D. Archer, J. Trilla, J. Dagit, A. Malozemoff, Y. Polyakov, K. Rohloff, and G. Ryan, "RAMPARTS: A programmer-friendly system for building homomorphic encryption applications," in *WAHC*, pp. 57–68, 2019.

[120] R. Malik, K. Sheth, and M. Kulkarni, "Coyote: A compiler for vectorizing encrypted arithmetic circuits," in *ASPLOS*, pp. 118–133, 2023.

[121] R. Recto and A. C. Myers, "A compiler from array programs to vectorized homomorphic encryption," *CoRR*, 2023.

[122] A. Viand, P. Jattke, M. Haller, and A. Hithnawi, "HECO: Fully homomorphic encryption compiler," in *USENIX Security*, pp. 4715–4732, 2023.

[123] S. Park, W. Song, S. Nam, H. Kim, J. Shin, and J. Lee, "HEaaN. MLIR: An optimizing compiler for fast ring-based homomorphic encryption," *PLDI*, vol. 7, no. PLDI, pp. 196–220, 2023.

[124] Y. Lee, D. Kim, D. Lee, and H. Kim, "ELASM: Error-latency-aware scale management for fully homomorphic encryption," in *USENIX Security*, pp. 1–15, 2023.

[125] S. Cheon, Y. Lee, D. Kim, J. M. Lee, S. Jung, T. Kim, D. Lee, and H. Kim, "DaCapo: Automatic bootstrapping management for efficient fully homomorphic encryption," in *USENIX Security*, 2024.

[126] 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 *PLDI*, pp. 546–561, 2020.

[127] S. Chowdhary, W. Dai, K. Laine, and O. Saarikivi, "EVA improved: Compiler and extension library for CKKS," in *WAHC*, pp. 43–55, 2021.

[128] J. Lee, E. Lee, J. Lee, Y. Kim, Y. Kim, and J. No, "Precise approximation of convolutional neural networks for homomorphically encrypted data," *IEEE Access*, vol. 11, pp. 62062–62076, 2023.

[129] N. Drucker, G. Moshkowich, T. Pelleg, and H. Shaul, "BLEACH: Cleaning errors in discrete computations over CKKS," *J. Cryptol.*, vol. 37, no. 1, 2023.

[130] H. Chen and K. Han, "Homomorphic lower digits removal and improved FHE bootstrapping," in *EUROCRYPT 2018*, vol. 10820, pp. 315–337, 2018.

[131] S. Halevi and V. Shoup, "Bootstrapping for helib," in *EUROCRYPT*, vol. 9056, pp. 641–670, 2015.

[132] E. Chielle, O. Mazonka, H. Gamil, and M. Maniatakos, "Accelerating fully homomorphic encryption by bridging modular and bit-level arithmetic," in *ICCAD*, 2022.

[133] J. Kim and T. Noh, "Modular reduction in CKKS." Cryptology ePrint Archive, Paper 2024/1638, 2024.

[134] A. Viand and H. Shafagh, "Marble: Making fully homomorphic encryption accessible to all," in *WAHC*, pp. 49–60, 2018.

[135] Z. Brakerski, O. Friedman, D. Golan, A. Gurny, D. Mutzari, and O. Sheinfeld, "REFHE: Fully homomorphic ALU," in *EUROCRYPT*, (Cham), pp. 183–213, Springer Nature Switzerland, 2026.

[136] Y. Bae, J. Kim, D. Stehlé, and E. Suvanto, "Bootstrapping small integers with ckks," in *ASIACRYPT*, p. 330–360, 2024.

[137] A. Guimarães, H. V. L. Pereira, and B. V. Leeuwen, "Amortized bootstrapping revisited: Simpler, asymptotically-faster, implemented," *IACR ePrint*, p. 14, 2023.

[138] Z. Liu and Y. Wang, "Amortized functional bootstrapping in less than 7 ms, with o˜(1) polynomial multiplications," in *ASIACRYPT*, pp. 101–132, 2023.

[139] G. D. Micheli, D. Kim, D. Micciancio, and A. Suhl, "Faster amortized FHEW bootstrapping using ring automorphisms," in *PKC*, vol. 14604, pp. 322–353, 2024.

[140] D. Boneh and J. Kim, "Homomorphic encryption for large integers from nested residue number systems," in *CRYPTO*, vol. 16002, pp. 338–370, 2025.

[141] J. Kim, "Efficient homomorphic integer computer from CKKS," *IACR TCHES*, vol. 2025, no. 4, pp. 873–898, 2025.

[142] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène, "Faster fully homomorphic encryption: Bootstrapping in less than 0.1 seconds," in *Advances*

in Cryptology - ASIACRYPT, vol. 10031, pp. 3–33, 2016.

[143] K. Matsuoka, Y. Hoshizuki, T. Sato, and S. Bian, "Towards better standard cell library: Optimizing compound logic gates for TFHE," in *WAHC*, pp. 63–68, 2021.

[144] S. Carpov, "A fast heuristic for mapping Boolean circuits to functional bootstrapping." Cryptology ePrint Archive, Paper 2024/1204, 2024.

[145] S. Carpov, M. Izabachène, and V. Mollimard, "New techniques for multi-value input homomorphic evaluation and applications," in *CT-RSA*, p. 106–126, 2019.

[146] J. Mono, K. Kluczniak, and T. Güneysu, "Improved circuit synthesis with multi-value bootstrapping for fhew-like schemes," *IACR TCHES*, vol. 2024, no. 4, pp. 633–656, 2024.

[147] I. Chillotti, D. Ligier, J. Orfila, and S. Tap, "Improved programmable bootstrapping with larger precision and efficient arithmetic circuits for TFHE," in *ASIACRYPT*, pp. 670–699, 2021.

[148] H. Chen, W. Dai, M. Kim, and Y. Song, "Efficient homomorphic conversion between (Ring) LWE ciphertexts," in *ACNS*, pp. 460–479, 2021.

[149] R. Mayans, "The chebyshev equioscillation theorem," *JOMA*, vol. 6, 2006.

[150] T. Rivlin, "The chebyshev polynomials, pure and applied mathematics," 1974.

[151] F. B. Hildebrand, *Introduction to numerical analysis*. Courier Corporation, 1987.

[152] D. S. Greenstein, "Interpolation and approximation (philip j. davis)," *SIAM Review*, vol. 7, no. 1, pp. 151–152, 1965.

[153] E. Lee, J.-W. Lee, J.-S. No, and Y.-S. Kim, "Minimax approximation of sign function by composite polynomial for homomorphic comparison," *IEEE TDSC*, vol. 19, no. 6, pp. 3711–3727, 2022.

[154] J. H. Cheon, G. Hanrot, J. Kim, and D. Stehlé, "SHIP: A shallow and highly parallelizable CKKS bootstrapping algorithm," in *EUROCRYPT*, pp. 398–428, 2025.

[155] H. Pan, R. Gan, Y. Fu, Y. Sun, Z. Zhang, Y. Wang, Y. Chen, B. Zhang, H. Zhou, Y. Tong, Z. Guan, J. Dong, and S. Bian, "HANNS: Low-storage non-interactive approximate private nearest neighbor search with sublinear comparison complexity." Cryptology ePrint Archive, Paper 2026/1397, 2026.

[156] G. Cha, D. Park, and J.-W. Lee, "Improved radix-based approximate homomorphic encryption for large integers via lightweight bootstrapped digit carry," in *EUROCRYPT*, pp. 243–273, 2026.

[157] Alibaba, "OpenPEGASUS." https://github.com/Alibaba-Gemini-Lab/OpenPEGASUS, 2021.

[158] C. Gouert and N. G. Tsoutsos, "Romeo: Conversion and evaluation of HDL designs in the encrypted domain," in *DAC*, pp. 1–6, 2020.

[159] C. Gouert, D. Mouris, and N. G. Tsoutsos, "HELM: Navigating homomorphic encryption through gates and lookup tables," *IEEE TIFS*, 2025.

[160] L. Li, J. Lai, P. Yuan, T. Sui, Y. Liu, Q. Zhu, X. Zhang, L. Xiao, W. Chen, and J. Xue, "ANT-ACE: An fhe compiler framework for automating neural network inference," in *CGO*, pp. 193–208, 2025.

[161] A. Ali, J. Choi, B. Gipson, S. Gorantala, J. Kun, W. Legiest, L. Lim, A. Viand, M. Z. Demissie, and H. Zheng, "HEIR: A universal compiler for homomorphic encryption," 2025.

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

[163] S. Narisada, H. Okada, K. Fukushima, S. Kiyomoto, and T. Nishide, "Gpu acceleration of high-precision homomorphic computation utilizing redundant representation," in *WAHC*, pp. 1–9, 2023.

[164] Y. Xiao, F.-H. Liu, Y.-T. Ku, M.-C. Ho, C.-F. Hsu, M.-C. Chang, S.-H. Hung, and W.-C. Chen, "GPU acceleration for FHEW/TFHE bootstrapping," *IACR TCHES*, vol. 2025, no. 1, pp. 314–339, 2025.

[165] S. Shen, H. Yang, Z. Liu, Y. Liu, X. Lu, W. Dai, L. Zhou, Y. Zhao, and R. C. Cheung, "VeloFHE: GPU acceleration for FHEW and TFHE bootstrapping," *IACR TCHES*, vol. 2025, no. 3, pp. 81–114, 2025.

[166] W. Jung, S. Kim, J. H. Ahn, J. H. Cheon, and Y. Lee, "Over 100x faster bootstrapping in fully homomorphic encryption through memory-centric optimization with gpus," *IACR TCHES*, pp. 114–148, 2021.

[167] S. Fan, Z. Wang, W. Xu, R. Hou, D. Meng, and M. Zhang, "TensorFHE: Achieving practical computation on encrypted data using GPGPU," in *IEEE HPCA*, pp. 922–934, IEEE, 2023.

[168] Z. Wang, P. Li, R. Hou, Z. Li, J. Cao, X. Wang, and D. Meng, "HE-Booster: an efficient polynomial arithmetic acceleration on gpus for fully homomorphic encryption," *IEEE TPDS*, vol. 34, no. 4, pp. 1067–1081, 2023.

[169] H. Yang, S. Shen, W. Dai, L. Zhou, Z. Liu, and Y. Zhao, "Phantom: A cuda-accelerated word-wise homomorphic encryption library," *IEEE TDSC*, vol. 21, no. 5, pp. 4895–4906, 2024.

[170] G. Fan, M. Zhang, F. Zheng, S. Fan, T. Zhou, X. Deng, W. Tang, L. Kong, Y. Song, and S. Yan, "WarpDrive: GPU-based fully homomorphic encryption acceleration leveraging tensor and CUDA cores," in *IEEE HPCA*, pp. 1187–1200, IEEE, 2025.

[171] D. Jiao, X. Deng, Z. Wang, S. Fan, Y. Chen, D. Meng, R. Hou, and M. Zhang, "Neo: Towards efficient fully homomorphic encryption acceleration using tensor core," in *ISCA*, (New York, NY, USA), p. 107–121, Association for Computing Machinery, 2025.

[172] A. Al Badawi, B. Veeravalli, J. Lin, N. Xiao, M. Kazuaki, and A. Khin Mi Mi, "Multi-GPU design and performance evaluation of homomorphic encryption on GPU clusters," *IEEE TPDS*, vol. 32, no. 2, pp. 379–391, 2021.

[173] A. Ş. Özcan and E. Savaş, "HEonGPU: a GPU-based fully homomorphic encryption library 1.0," *Cryptology ePrint Archive*, 2024.

[174] Q. Li and R. Zong, "CAT: A GPU-accelerated FHE framework with its application to high-precision private dataset query," *arXiv preprint arXiv:2503.22227*, 2025.

[175] Z. Wang, H. He, L. Zhao, P. Li, Z. Li, D. Meng, and R. Hou, "Chameleon: An efficient FHE scheme switching acceleration on GPUs," *IEEE TPDS*, vol. 36, no. 11, pp. 2264–2280, 2025.

[176] M. Van Beirendonck, J.-P. D'Anvers, F. Turan, and I. Verbauwhede, "FPT: A fixed-point accelerator for torus fully homomorphic encryption," in *ACM CCS*, pp. 741–755, 2023.

[177] J. Zhang, A. Cui, and Y. Jin, "Acceleration of the bootstrapping in TFHE by FPGA," *IEEE TETC*, vol. 13, no. 2, pp. 496–511, 2025.

[178] M. S. Riazi, K. Laine, B. Pelton, and W. Dai, "HEAX: An architecture for computing on encrypted data," in *ASPLOS*, pp. 1295–1309, 2020.

[179] Y. Yang, H. Zhang, S. Fan, H. Lu, M. Zhang, and X. Li, "Poseidon: Practical homomorphic encryption accelerator," in *IEEE HPCA*, pp. 870–881, IEEE, 2023.

[180] Y. Zhu, X. Wang, L. Ju, and S. Guo, "FxHENN: FPGA-based acceleration framework for homomorphic encrypted CNN inference," in *IEEE HPCA*, pp. 896–907, IEEE, 2023.

[181] R. Agrawal, L. de Castro, G. Yang, C. Juvekar, R. Yazicigil, A. Chandrakasan, V. Vaikuntanathan, and A. Joshi, "FAB: An FPGA-based accelerator for bootstrappable fully homomorphic encryption," in *IEEE HPCA*, pp. 882–895, 2023.

[182] E. Pattuk, M. Kantarcioglu, H. Ulusoy, and B. Malin, "CheapSMC: A framework to minimize secure multiparty computation cost in the cloud," in *DBSec*, pp. 285–294, 2016.

[183] L. Jiang, Q. Lou, and N. Joshi, "MATCHA: A fast and energy-efficient accelerator for fully homomorphic encryption over the torus," in *DAC*, pp. 235–240, 2022.

[184] A. Putra, J.-Y. Kim, *et al.*, "Morphling: A throughput-maximized TFHE-based accelerator using transform-domain reuse," in *IEEE HPCA*, pp. 249–262, IEEE, 2024.

[185] A. Putra, Prasetiyo, Y. Chen, J. Kim, and J.-Y. Kim, "Strix: An end-to-end streaming architecture with two-level ciphertext batching for fully homomorphic encryption with programmable bootstrapping," in *MICRO*, (New York, NY, USA), p. 1319–1331, Association for Computing Machinery, 2023.

[186] L. Liang, Z. Gu, F. Zhang, Z. Chen, Z. Li, X. Fan, D. Niu, M. Li, Z. Li, Z. Wang, H. Zheng, Y. Cai, and Y. Xie, "Matrix: Multi-cipher structures dataflow for parallel and pipelined TFHE accelerator," *ACM TACO*, vol. 22, Sept. 2025.

[187] S. Gupta, R. Cammarota, and T. Šimunić, "MemFHE: End-to-end computing with fully homomorphic encryption in memory," *ACM TECS*, vol. 23, no. 2, pp. 1–23, 2024.

[188] N. Samardzic, A. Feldmann, A. Krastev, S. Devadas, R. Dreslinski, C. Peikert, and D. Sanchez, "F1: A fast and programmable accelerator for fully homomorphic encryption," in *MICRO*, pp. 238–252, 2021.

[189] N. Samardzic, A. Feldmann, A. Krastev, N. Manohar, N. Genise, S. Devadas, K. Eldefrawy, C. Peikert, and D. Sanchez, "Craterlake: a hardware accelerator for efficient unbounded computation on encrypted data," in *ISCA*, pp. 173–187, 2022.

[190] J. Kim, G. Lee, S. Kim, G. Sohn, M. Rhu, J. Kim, and J. H. Ahn, "Ark: Fully homomorphic encryption accelerator with runtime data generation and inter-operation key reuse," in *MICRO*, pp. 1237–1254, IEEE, 2022.

[191] B. Reagen, W.-S. Choi, Y. Ko, V. T. Lee, H.-H. S. Lee, G.-Y. Wei, and D. Brooks, "Cheetah: Optimizing and accelerating homomorphic encryption for private inference," in *IEEE HPCA*, pp. 26–39, IEEE, 2021.

[192] 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 *ISCA*, (New York, NY, USA), Association for Computing Machinery, 2023.

[193] L. Lei, Y. Zhu, X. Zou, Y. He, Z. Zhang, Z. Guan, H. Yang, Y. Liu, S. Bian, and H. Jia, "An eDRAM-based in-situ-computing processor for homomorphic encryption evaluation on the edge," in *IEEE ESSERC*, pp. 464–467, IEEE, 2024.

[194] S. Lu, W. Zhu, B. Yang, J. Yang, T. Dai, C. Chen, X. Han, J. Yang, H. Wang, M. Zhu, *et al.*, "A 28nm 4.05 μJ/encryption 8.72 kHMul/s reconfigurable multi-scheme fully homomorphic encryption processor for encrypted client-server computing," in *IEEE ISSCC*, vol. 68, pp. 01–03, IEEE, 2025.

[195] S. Fan, X. Deng, L. Kong, G. Shi, G. Fan, D. Meng, R. Hou, and M. Zhang, "FAST: An fhe accelerator for scalable-parallelism with tunable-bit," in *ISCA*, (New York, NY, USA), p. 92–106, 2025.

[196] L. Ding, S. Bian, P. He, and J. Zhang, "APACHE: A processing-near-memory architecture for multi-scheme fully homomorphic encryption," *IEEE Transactions on Computers*, vol. 75, no. 6, pp. 2128–2140, 2026.

[197] L. Lei, Y. Duan, C. Peng, Y. Zhu, G. Du, Z. Guan, H. Yang, Y. Liu, Z. Gu, S. Bian, *et al.*, "Presto: A unified risc-v-compatible soc for multi-scheme fhe acceleration over module lattice," in *IEEE HCS*, pp. 1–40, IEEE Computer Society, 2025.

[198] F. Vogelsteller and V. Buterin, "ERC-20: Token standard," *Stiftung Ethereum*, vol. 169, 2015.

[199] T. P. P. Council, "TPC BENCHMARK^{TM} H Standard Specification," tech. rep., 2022.

[200] M. Hastings, B. Hemenway, D. Noble, and S. Zdancewic, "SoK: General purpose compilers for secure multi-party computation," in *IEEE S&P*, pp. 1220–1237, 2019.

[201] S. Yune, H. Lee, A. Putra, H. Cho, C. D. Manh, J. Jeon, and J.-Y. Kim, "ABC-FHE: A resource-efficient accelerator enabling bootstrappable parameters for client-side fully homomorphic encryption," *arXiv preprint*

arXiv:2506.08461, 2025.

[202] R. Geelen, M. Van Beirendonck, H. V. Pereira, B. Huffman, T. McAuley, B. Selfridge, D. Wagner, G. Dimou, I. Verbauwhede, F. Vercauteren, *et al.*, "BASALISC: programmable hardware accelerator for bgv fully homomorphic encryption," *arXiv preprint arXiv:2205.14017*, 2022.

[203] H. Lee, H. Kwon, and Y. Lee, "A 2.7-to-13.3 μJ/boot/slot flexible RNS-CKKS processor in 28nm CMOS technology for FHE-based privacy-preserving computing," in *IEEE ISSCC*, vol. 67, pp. 296–298, IEEE, 2024.

[204] K. Shivdikar, Y. Bao, R. Agrawal, M. Shen, G. Jonatan, E. Mora, A. Ingare, N. Livesay, J. L. Abellán, J. Kim, *et al.*, "Gme: Gpu-based microarchitectural extensions to accelerate homomorphic encryption," in *MICRO*, pp. 670–684, 2023.

[205] R. Agrawal, A. Chandrakasan, and A. Joshi, "HEAP: A fully homomorphic encryption accelerator with parallelized bootstrapping," in *ISCA*, pp. 756–769, 2024.