LazyArc: Dynamic Out-of-Order Engine for High-Throughput FHE

Omar Ahmed and Nektarios Georgios Tsoutsos

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

Abstract. Fully Homomorphic Encryption (FHE) is a modern cryptographic technique that allows performing computations directly on encrypted data. This makes FHE an indispensable method for privacy-preserving applications, where users' data are encrypted and processed by a potentially untrusted third party. Nevertheless, FHE operations are computationally expensive, often rendering them impractical for realistic scenarios. Notably, a major performance bottleneck for FHE is an operation called bootstrapping that allows refreshing the inherent noise of FHE data to support more computations. In this work, we introduce LazyArc, a versatile lightweight dynamic Out-of-Order (OoO) software engine that enables higher-throughput FHE computations. Our proposed OoO paradigm improves the performance by masking the latency of bootstrapping via a bespoke instruction-bootstrapping scheduling paradigm. Also, we introduce a novel tool, dubbed RegisterMap Tool, to statically analyze FHE arithmetic circuits and track the noise of each ciphertext to allow proactive bootstrapping scheduling. Our approach is evaluated using linear algebra benchmarks and achieves more than 2× speedup compared to in-order baselines, and more than 5× speedup compared to the FHEBooster scheduler.

Keywords: Fully Homomorphic Encryption · Privacy-Preserving Computation · Encrypted Processor · Dynamic Scheduling.

1 Introduction

As modern computing paradigms shift from local execution to outsourced computations on high-performance cloud servers, protecting user data privacy has become a critical challenge. This shift has introduced advances in run time performance that allow regular users to perform complex operations quickly and efficiently. Nevertheless, outsourcing computations to cloud servers introduces significant privacy risks for end users. For example, the cloud may use the data for personalized advertisements or to train their own AI models [25, 29, 30, 32]. Moreover, cloud systems are common targets for various cyber-attacks that threaten our privacy [1, 16, 36, 39].

A promising approach to enable end-to-end secure computations on encrypted data is Fully Homomorphic Encryption (FHE) [20]. FHE supports performing arbitrary computations on encrypted data (ciphertexts) without the need to decrypt them. More formally, if we denote an encryption algorithm E and a decryption algorithm D, then a cloud provider can compute a circuit F on inputs x_1 and x_2 as follows:

$$D(F(E(x_1), E(x_2))) = F(x_1, x_2) \tag{1}$$

In particular, FHE takes encrypted inputs $E(x_1)$ and $E(x_2)$ and returns an encrypted result. By decrypting that output, we receive the same result as if the circuit were evaluated on plaintext values x_1 and x_2. With this powerful capability, we can execute arbitrary computations on encrypted data to maintain their secrecy. Nevertheless, a limitation of FHE is its runtime performance. This is due to the nature of the encoding schemes used to represent ciphertext data as large polynomials. These polynomials encapsulate the plaintext data with a "noise" component, and this noise is what protects the plaintext values and makes an FHE scheme secure.

Performing arithmetic operations on ciphertexts is translated to complex and heavy operations on the underlying polynomials. This leads to having the noise embedded within the polynomials grow, which eventually corrupts the original values. Performing an addition operation between two ciphertexts introduces

minimal noise to the output ciphertext. However, performing a multiplication causes a significant noise growth in the output ciphertext. Without a noise reduction mechanism, we can execute arithmetic circuits with a limited number of addition and multiplication operations.

Some FHE schemes support a technique that can reduce the noise, called bootstrapping. Bootstrapping is a homomorphic decryption operation that processes a ciphertext and reduces its noise. Thus, it allows more computations to be performed on a given ciphertext, hence, we can execute deeper circuits. Nevertheless, bootstrapping causes considerable delays that stall the evaluation of a target circuit.

In addition, noise management during FHE circuits evaluation remains an intricate process that requires domain-specific expertise due to its inherent complexity. When writing FHE circuits, programmers have to track the noise level of each ciphertext and invoke the bootstrapping function when necessary. While naive automatic bootstrapping after every operation prevents noise corruption, it incurs prohibitive computational overhead for FHE programs. Meanwhile, not bootstrapping a noisy ciphertext can corrupt the execution of the circuits. To address this challenge, developers manually set fixed checkpoints within the circuit to bootstrap ciphertexts when reaching this point. However, this process does not scale well, as it is bespoke for each particular circuit. FHE libraries facilitate the estimation of the noise level to determine if bootstrapping is needed. Thus, we can bootstrap a ciphertext as late as possible (ALAP) based on the noise level.

Notably, the concepts of "ciphertext noise" and "ciphertext level" are fundamentally complementary. As ciphertext noise increases, available multiplication levels are consumed. When a ciphertext exhausts its multiplication budget, its noise has reached the maximum allowable threshold. Consequently, bootstrapping restores the multiplication levels of a ciphertext, which is functionally equivalent to reducing its noise. While common FHE libraries, such as OpenFHE, rely on the term "ciphertext level" to denote the remaining operation budget, in this paper, we adopt the term "ciphertext noise" to describe this exact underlying metric.

There are various FHE schemes that are widely used. The most common schemes include BFV [19], BGV [11], CKKS [12], TFHE [15] and FHEW [18]. BFV and BGV support arithmetic circuits that encrypt integers. Meanwhile, the CKKS scheme evaluates arithmetic circuits encrypting real-numbers. The CKKS scheme performs approximate computations on ciphertexts, and these ciphertexts lose precision in deep multiplicative circuits.

The TFHE (often called CGGI) and FHEW schemes are tailored to evaluate Boolean circuits. These schemes offer significantly faster bootstrapping speeds compared to other schemes. However, the amortized latency to execute an arithmetic operation (e.g., addition) is marginally slower than other schemes. The popular OpenFHE library [6] uses a unified codebase for TFHE and FHEW, so in this work we refer to these Boolean FHE schemes interchangeably (FHEW/TFHE) following the OpenFHE nomenclature.

An important distinction between Boolean schemes (TFHE/FHEW) and integer/Real schemes (e.g, CKKS, BGV) lies in whether bootstrapping serves as an integrated computation gate or an infrequent, heavy maintenance procedure. In Boolean schemes, bootstrapping is an integral part of evaluating a homomorphic function. Therefore, executing a Boolean gate (e.g., NAND) involves evaluating the gate operation while the noise refresh happens simultaneously inside a single step called programmable bootstrapping (PBS). Hence, every Boolean gate naturally refreshes the ciphertext noise. Meanwhile, in integer/Real schemes, bootstrapping is completely decoupled from the arithmetic operation. Hence, we can execute multiple arithmetic operations on a ciphertext without bootstrapping it. Once the noise in a ciphertext reaches a particular amount, then we invoke bootstrapping to "reset" the noise budget within the ciphertext.

Modern software library implementations of FHE schemes realize a technique called Scheme Switching (SS). SS enables converting a ciphertext object from one scheme to another, while the data remain encrypted. This feature is highly advantageous as it allows the pipeline to leverage the respective strengths of different schemes. For instance, we can execute a CKKS circuit and switch to FHEW/TFHE to perform bit-wise operations. Unfortunately, SS is an expensive operation and existing techniques are still limited in terms of scalability and performance.

A recent survey [35] categorizes existing methods to accelerate the performance of bootstrapping into three main categories: optimizing homomorphic modular reduction, optimizing encoding and decoding schemes,

and alternative constructions for blind rotations (a complex procedure used to refresh ciphertexts by performing rotations on polynomials).

Furthermore, software-based acceleration techniques often rely on memory-based approaches, as well as improving operations on the underlying polynomials and the Number Theoretic Transform (NTT) used for efficient polynomial multiplication [10, 21, 26]. Also, software approaches often focus on both the usability and performance of FHE, including works focusing on compilers and virtual processor frameworks with specialized scheduling techniques [4, 5, 8, 27, 41, 42]. On the contrary, hardware-based accelerators rely on the design of specialized hardware architectures and parallelization paradigms, exploiting high performance GPUs and FPGAs [7, 9, 24, 33].

In this work, we tackle the execution stalls inherent to the evaluation of FHE circuits, which are represented as a sequence of instructions, evaluated on a virtual machine (VM). Because ciphertexts require periodic bootstrapping to manage noise, these operations introduce runtime delays, called *noise hazards*, that stall the execution pipeline. We propose an Out-of-Order (OoO) execution paradigm to dynamically reorder instruction execution relative to bootstrapping routines. By doing so, our approach hides bootstrapping overheads behind independent computations, effectively boosting circuits performance.

Our OoO approach for FHE is modeled within a lightweight virtual machine (VM), named LazyArc, which can execute arbitrary FHE circuits efficiently using a domain-specific instruction set. In other words, LazyArc is a software-based architecture that interprets and executes FHE instructions, maximizing the computational throughput through out-of-order scheduling of the instructions. Moreover, LazyArc is paired with a novel static circuit analysis tool that introduces hints to the virtual machine to allow optimal bootstrapping scheduling for ciphertexts. LazyArc implements a hybrid architecture that executes both arithmetic and Boolean circuits. Our contributions are summarized as follows:

1. To the best of our knowledge, this is the first work to observe and study *Noise Hazards* in FHE pipelines. Our work provides the first empirical analysis of noise hazards and their associated execution delays.
2. To overcome noise hazards, we introduce LazyArc: a versatile FHE VM that allows Out-of-Order (OoO) execution of FHE instructions, enabling higher execution throughput and better utilization of computational resources. To ensure versatility and compatibility, LazyArc supports both arithmetic and Boolean circuits.
3. To further augment the efficiency of LazyArc, we introduce the *RegisterMap Tool* (RMT), which performs static analysis for FHE circuits before execution and helps LazyArc to bootstrap a ciphertext *proactively*.

2 Related Work

In this section, we review similar work that focuses on related software-based methods to accelerate FHE computations.

The authors of [31] proposed XBOOT, which focuses on the efficiency of performing the XOR operation on CKKS ciphertext. PulpFHE [5] is a virtual processor framework that accelerates the computations of the TFHE scheme on CPUs. Likewise, Juliet [22] focuses on optimizing TFHE computations but on GPUs. HEAP [2] is a parallelized bootstrapper designed for the CKKS scheme by using a hybrid scheme-switching approach. A different approach to accelerate FHE computations is FHEBooster [42], which formulates the bootstrapping scheduling problem as a directed acyclic graph (DAG) and optimizes it using Integer Programming (IP) models.

Another interesting approach to accelerate FHE is to use Compute-Enabled (CE) RAMs, as proposed in [37]. In [37], the authors leverage the CE-RAM, which allows processing elements directly into data storage, to optimize the underlying operations of the TFHE and the BFV schemes. Wang et al. [40] proposed HE-Booster, which is a polynomial arithmetic acceleration framework targeting GPU architectures. HE-Booster targets both single-GPU and multi-GPU systems to leverage fine-grained data-level parallelism. HE-Booster targets the BGV scheme and achieves faster homomorphic multiplication than similar works.

Agrawal et al. [3] proposed a memory-based design called MAD to accelerate the bootstrapping operation of the CKKS scheme. MAD focuses on reducing the amount of data transfers to/from the main memory

and reducing the number of NTT operations. MAD was evaluated on a GPU architecture, achieving notable speedups over similar frameworks. A recent work in [34] presents CHEHAB RL, a novel framework that utilizes deep reinforcement learning (RL) to automatically optimize the FHE instructions of the BFV scheme. The RL agent learns and applies policies to reduce the execution latency of the code and to bound the noise growth.

Recent advancements in FHE compilers and static scale-management frameworks (e.g., DaCapo [13], HALO [14], ELASM [28], and Fhelipe [27]) optimize circuits at compile-time to minimize the total number of bootstrap operations. These static approaches are fundamentally orthogonal to our work. While such compilers optimize the circuit graph prior to execution, LazyArc operates as a dynamic runtime virtual machine. Compared to these earlier works, LazyArc approaches the acceleration problem from a completely different layer of the stack; it utilizes Out-of-Order evaluation during runtime to mask the latency of any remaining, unavoidable bootstraps.

3 Motivation

This section discusses the concepts of out-of-order execution. Then we define the concept of *noise hazards* and how it negatively impacts FHE pipelines.

3.1 Background: Out-of-Order Execution

Out-of-Order Execution (OoOE) is a dynamic scheduling approach that is used in modern CPUs; it minimizes the number of stall cycles, allowing the processor to reorder the execution of independent instructions, while maintaining the correct logical flow of the computations. Fundamentally, traditional execution paradigms execute the instructions based on their order in a program; however, OoOE executes the instructions based on the availability of the inputs and resources.

With this technique, a CPU can avoid waiting for the preceding instructions to finish and instead process the next instructions immediately if their inputs are available. Tomasulo's algorithm [38] is a fundamental scheduling approach, which detects the dependencies between the instructions and resolves them. In particular, Tomasulo's algorithm detects and resolves Write After Read (WAR), and Write After Write (WAW) data hazards [23]. It relies on distributed control and renaming hardware registers to resolve the dependencies between the instructions.

3.2 Noise Hazards in FHE Pipelines

Recall that computations on ciphertexts increase the noise that protects the underlying plaintext values. At some point, performing more operations on a noisy ciphertext will render it invalid and corrupt the underlying plaintext. Hence, bootstrapping must be invoked at a particular point to refresh this ciphertext when reaching a specific noise threshold. This threshold is determined based on the parameters used to set up an FHE scheme. Thus, when expressing an FHE circuit, we have to estimate the noise amount whenever a ciphertext is processed and bootstrap it accordingly (this is called As Late As Possible bootstrapping).

In FHE pipelines, there are two types of hazards: **data hazards** when some instructions are strictly dependent on earlier instructions, and **noise hazards**. A *noise hazard occurs when the FHE pipeline stalls because a ciphertext cannot be used without bootstrapping – a dependency between* ***computations*** *and* ***bootstrapping***. Notably, the overhead introduced by bootstrapping is significantly larger than the delays introduced by FHE addition and multiplication. We formally define noise hazards as follows.

Definition 1. Noise Hazard is a type of pipeline stalls that occur in FHE circuits due to the excess accumulation of noise in a ciphertext. Hence, a noisy ciphertext entails bootstrapping, which in turn causes significant delays. A noisy ciphertext is available in its register, but using it for further computations will corrupt it.

Table 1: The average latency of addition and multiplication versus bootstrapping under different parameters with 128-bit security level. These values were obtained by averaging the latency over 10 invocations of each function.

Ring Dimension	Mult. Depth	Add (ms)	Mult. (ms)	Bootstrap (ms)	Boots./ Add	Boots./ Mult.
131072	30	34	671	29218	859	44
	29	24	609	27068	1128	44
	27	22	498	26070	1185	52
65536	20	10	277	9831	983	35
	19	9	258	9521	1058	37
	17	7	178	8798	1257	49

Since a register is used to store a ciphertext, we will use the term "register" and "ciphertext" interchangeably to refer an encrypted value. To prove the aforementioned observation of noise hazards, we provide the empirical analysis shown in Table 1. The table shows the timings of bootstrapping, addition, and multiplication averaged over 10 invocations for each. This analysis was performed based on the OpenFHE [6] library v1.5.1. The ring dimension is the degree of the polynomial ring used to construct ciphertexts. It is automatically selected by the FHE library to match a standard 128-bit security. The last two columns show the relative latency of bootstrapping over addition and multiplication, respectively. Therefore, based on Table 1, it becomes evident that bootstrapping is the main bottleneck and a single bootstrap invocation can be masked by multiple additions and multiplications.

Consequently, masking bootstrapping latency is our primary optimization target, because its latency is significantly higher than arithmetic operations. Therefore, LazyArc's OoO engine specifically masks these stalls by executing independent instructions of the workload while a bootstrap is pending. As shown in Table 1, a single bootstrapping invocation accommodates 859–1257 additions or 35–49 multiplications. However, this ratio cannot be fully exploited in real-world applications for two reasons:

1. Circuits often do not have a large number of **independent** instructions (e.g., 465 instructions are independent of any other instructions).
2. Table 1 shows the latency of each operation when run individually, so each operation utilizes all the available system resources. Hence, the resources are reserved for only one operation. However, our OoO paradigm overlaps the bootstrapping with other arithmetic operations. Therefore, the system resources are shared between instructions and bootstrapping.

To illustrate this concept, consider the example circuit shown in Figure 1. The left part of the figure shows a sequence of FHE instructions with their corresponding noise budget. Instructions `I1` to `I6` initialize new ciphertext registers. A fresh ciphertext has the minimum noise required to hide the plaintext. Instruction `I7` performs the multiplication $F_0 \times F_1$ and writes the result to F_2. Hence, F_2 accumulates more noise. As the ciphertext is involved in more multiplication, it enters the depletion region (between cycles 20 and 76). The depletion region is where a ciphertext is involved in a series of computations, accumulating more noise on it. Following the depletion region, instruction `I16` performs another multiplication (for a total of 10 consecutive multiplications involving F_2). The output of `I16` becomes too noisy, so execution cannot proceed without bootstrapping. Once bootstrapping is done (cycle 98), the execution proceeds to `I17` until `I19`.

Observe how bootstrapping stalls the execution of the circuit: `I17` and `I19` wait until the 15 cycles of bootstrapping are consumed, although they do not require the output of `I16` as input. This is a direct example of a noise hazard. In fact, instruction `I16` consumes its 7 cycles, after which F_2 is available in its memory. However, using F_2 directly will corrupt the underlying plaintext. This motivates our primary research question: is it possible to hide the latency of bootstrapping and proceed with executing `I17` and `I19`? This is what we aim to address with the design of LazyArc.

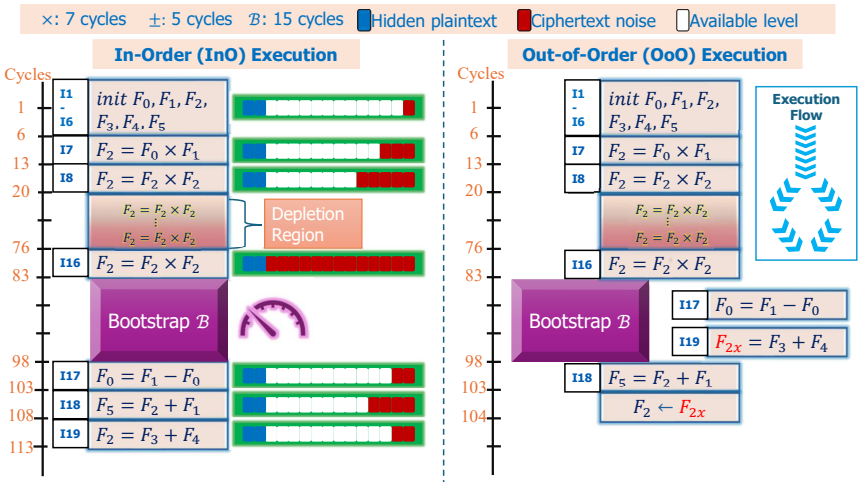


Fig. 1: A circuit initializes 6 ciphertexts (assuming 1 cycle for each) and performs arithmetic operations. Instructions I1-I6 cannot be executed OoO. Instruction I18 depends on I16. I17 and I19 are independent of I16 and I18. The right-side figure depicts the typical scenario to achieve OoOE.

To resolve this, we assign tags (temporary identifiers for original registers) to outputs that are still pending in the pipeline. The tags allow eliminating conflicts between registers, which in turn resolves the bootstrapping-computation dependencies. As shown in the right part of Figure 1, the pipeline can proceed in-order until `I16`. After `I16`, the pipeline can evaluate the bootstrapping, while executing `I17` and `I19`. Instead of letting `I19` write the result to F_2, it writes to a tagged register, F_{2x}. Then `I18` reads the value from the original F_2 register (output of `I16`) to compute F_5. Finally, F_2 is updated from the tagged register F_{2x}.

3.3 High-Level Architecture Overview

LazyArc is a virtual machine that dynamically schedules FHE instructions to avoid noise hazards. This is achieved by adapting an OoO execution paradigm to decouple bootstrapping from the instructions' operations. There are fundamental differences between LazyArc and CPU dynamic scheduling algorithms. First, LazyArc is a **software-based architecture** that enables efficient dynamic scheduling for FHE circuits, meanwhile CPU scheduling algorithms are general methods applied on hardware processors.

The second and most important difference stems from the problems solved by LazyArc and other dynamic scheduling algorithms. On the one hand, hardware-based algorithms resolve **resource and data conflicts** that arise among the instructions. On the other hand, LazyArc resolves **noise hazards** that arise between instructions and bootstrapping. In FHE circuits, an instruction executes and writes its output to a destination register; thus, the output is available. However, if the output is used in further computations, it will corrupt the computation.

The third difference is that CPU scheduling aims to maximize instruction-level parallelism (ILP) by concurrently executing independent instructions. However, LazyArc aims to reduce execution time by *hiding the latency of the complex bootstrapping procedure*. Ultimately, CPU dynamic scheduling algorithms are **instruction-to-instruction hardware schedulers**, but LazyArc is an **instruction-to-bootstrapping software scheduler**.

3.4 Virtual Machine Formalization

LazyArc operates as a virtual machine applying OoO scheduling to FHE circuits. It maps the instructions from their queue to special data structures (Streams and Tags Table in Figure 2a, explained later in Section 4) to detect bootstrapping dependencies and assign tags to registers. Bootstrapping is considered as a high-latency functional unit. Crucially, this formalization enables LazyArc to exclusively track noise growth in ciphertexts to bypass *noise hazards*. This fine-grained dispatch extracts *Bootstrapping-Level Scheduling* opportunities, advancing beyond simple asynchronous scheduling to parallelize computations.

Furthermore, LazyArc is designed as a hardware-agnostic control-plane architecture that decouples instruction scheduling and dependency resolution from the underlying execution backend. LazyArc logically separates the execution engine into a Control Plane (the Instruction Queue, RegisterMap Tool, and dynamic OoO scheduler) and a Data Plane (the Processor and Bootstrapper modules). While our prototype is evaluated using a software implementation (built on OpenFHE) to facilitate rapid benchmarking across complex FHE workloads, the architectural principles of LazyArc (i.e., Out-of-Order execution, RegisterMap tracking, and proactive bootstrapping) apply directly to hardware execution engines. In our software prototype, the functional units are CPU worker threads invoking C++ primitives. In a hardware-accelerated system (e.g., FPGA or ASIC), these functional units could be instantiated as specialized processing elements, such as Number Theoretic Transform (NTT) units or modular reduction pipelines. Crucially, the LazyArc control plane operates identically regardless of whether the functional unit is a software thread or a hardware execution pipeline.

4 LazyArc: A Dynamic Out-of-Order Engine

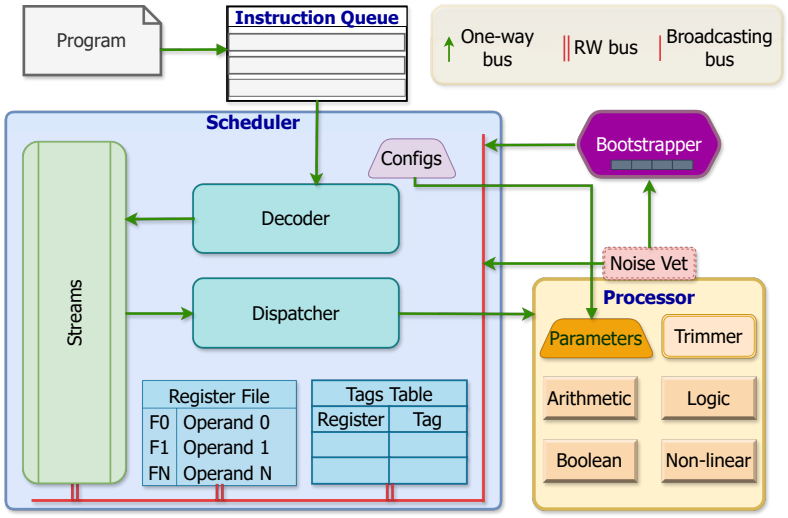
Our main objective is to hide the latency introduced by bootstrapping, which in turn results in increasing the throughput of FHE execution pipelines, achieving better utilization of computational resources, and better run time performance. Towards that end, we need to realize out-of-order execution (OoOE) in the FHE domain. Figure 2 shows a detailed view of the LazyArc virtual machine. It consists of three fundamental modules: the Scheduler, the Processor, and the Bootstrapper. The Scheduler consists of the units: Decoder, Dispatcher, Streams, Register File, Tags Table, and Noise Vet (attached to the Processor).

The Decoder is responsible for fetching and decoding the instructions from the queue and posting them to the Streams unit. Streams, shown in Figure 2b, is a special data structure for storing the instruction's operands. LazyArc has six Streams: `AddSubStream` for addition and subtraction instructions, `MultStream` for multiplication instructions, `ComplexStream` for compound instructions such as computing mean and variance, `NonLinearStream` for non-linear instructions such as division and square root, `BooleanStream` for Boolean operations, and `LogicStream` for comparison instructions such as greater than and equality checks.

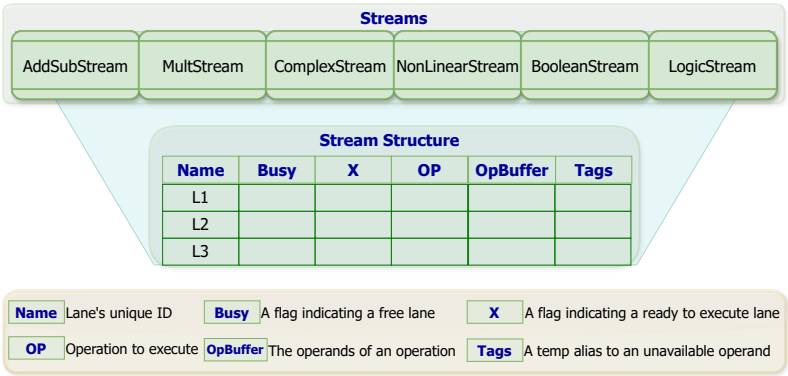
Each Stream consists of lanes, where each lane is a special structure that includes six buffers to store the state of an instruction. The `Name` buffer holds the name of the lane, while the `Busy` buffer is a flag to show if the lane is available to store an instruction. The `X` (eXecute) buffer indicates whether the instruction is ready to execute or not. The `OP` buffer holds which operation to be performed. The `OpBuffer` buffer stores pointers to the operands needed for the operation. The `Tags` buffer holds the tag (temporary identifier) of the unavailable operand.

The `OpBuffer` and `Tags` buffers cannot have the same operands at the same time because storing an operand in one of the buffers means that the other buffer does not store it. For example, assume that we have the instruction $F_3 = F_0 + F_1$. If the registers F_0 and F_1 have their noiseless values ready, they are stored in `OpBuffer`. Conversely, if either F_0 or F_1 is noisy or unavailable, we assign a tag to the register and store it in `Tags`. When `Tags` is free, it means that `OpBuffer` is full, and thus the `X` buffer is set to `True`, indicating that an instruction can be dispatched to the Processor (a software-based engine that invokes an underlying FHE library).

After decoding instructions and posting them to their streams, the Dispatcher fetches an instruction from a Stream and passes it to the Processor. The Processor executes the instruction and passes the result to the Noise Vet unit. The Noise Vet is a lightweight unit that is responsible for deciding whether to forward



(a) The fundamental modules of LazyArc.



(b) The Streams module. Each stream consists of Lanes.

Fig. 2: The architectural design of LazyArc.

the outputs to the Bootstrapper or post them to the Broadcasting bus. The Noise Vet checks the noise of a ciphertext; if it exceeds a threshold or has a flag for early bootstrapping, it forwards the outputs to the Bootstrapper. Otherwise, the outputs are posted to the Broadcasting bus to write the results into their buffers. The Bootstrapper is responsible for removing the noise from a ciphertext and posting it to the Broadcasting bus.

The Scheduler has two additional units: the Register File (RF) and the Tags Table (TT). The RF stores the available registers and their operands. *A register is considered available if it stores a ciphertext* ***and*** *its noise level is below the critical threshold.* Thus, the available register should exist in the RF, while the unavailable register should exist in the TT. The TT stores the unavailable register name and the name of the lane that holds its output. Here, the name of the lane that will hold a pending output serves as a tag for later instructions.

As shown in Figure 2a, LazyArc uses the Configs unit to store the configurations needed to evaluate FHE circuits. A subset of these configurations (e.g., evaluation keys) are passed to the Processor, which stores them in the Parameters unit. The Processor has four computational units and a special unit. The computation units (based on the instruction set of [5]) are: The Arithmetic unit for operations such as add

and multiply, the Logic unit for comparisons, the Boolean unit for performing bit-wise operations, and the Non-linear unit for operations such as division.

The Trimmer is a specialized functional unit that truncates the fractional component of a ciphertext, evaluating it strictly as an integer to prevent precision noise from corrupting Boolean logic selectors (for example, if a ciphertext is 1.01, Trimmer transforms it to 1.00). This unit evaluates the smooth step function $S(x) = 3x^2 - 2x^3$ [17]. It is useful for performing comparisons with the multiplexer gate. The multiplexer uses a selector bit, which has to be 0 or 1. If the bit is 0.001 or 1.001, this could corrupt the result. Thus, we restore the bit's precision before using it.

4.1 Dynamic Noise Hazard Detection and Mitigation

Recall that noise hazards occur because some ciphertexts are not ready for further computations due to noise accumulation. Normally, when the Processor finishes executing an instruction, it outputs the result, so that any waiting instructions can use it. However, while the Processor's outputs are structurally available, they may be saturated with noise that can corrupt the subsequent results. Furthermore, subsequent independent instructions are forced to stall.

LazyArc addresses this issue through two techniques: Online Ciphertext Analysis (OCA) and Register-Tag Aliasing (RTA). With OCA, instead of writing the results immediately to the buffers, we dedicate a lightweight Noise Vet unit to offload noise analysis from other units and to decide where to route the outputs. Consequently, execution units are decoupled from noise management, allowing them to dedicate computational cycles exclusively to circuit evaluation. The RTA technique (as shown in Figure 1) aims to replace the names of the later instructions' registers with tags that will refer to the original registers. For an unavailable ciphertext, its tag corresponds to the name of the lane that will hold it. Thus, the pipeline executes later instructions while bootstrapping the output of the preceding instructions. Then the original registers are updated with the new outputs. Notably, the use of the Streams and the TT in the Scheduler is what enables RTA, and in turn out-of-order execution to eliminate noise hazards. TT stores a register and the name of the lane that will store its value.

4.2 Proactive Bootstrapping via RegisterMap

The standard approach for noise management invokes bootstrapping As Late As Possible (ALAP). However, our analysis demonstrates that ALAP scheduling is frequently suboptimal. To highlight this observation, consider the example shown in Figure 3: The left side of the figure shows an example of the ALAP bootstrapping strategy. Register F_2 is depleted from instruction `I8` to the end of the depletion region (`I15`). The noise in F_2 at `I15` does not trigger bootstrapping, since it is below the critical threshold, but it reaches this threshold at `I19`. In this ALAP bootstrapping scenario, we bootstrap F_2 at `I19` and then the pipeline continues to execute `I20` and the rest of the circuit. Alternatively, the proactive bootstrapping approach (with OoOE) bootstraps F_2 at the last instruction of the depletion region (`I15`). During the 15 cycles of bootstrapping, the pipeline can execute `I16`–`I18` concurrently. Therefore, the 15 cycles of bootstrapping are totally masked.

Enabling proactive bootstrapping is achieved using our novel `RegisterMap` Tool (RMT). The RMT is based on our bespoke data structure named `RegisterMap` (RM). The RMT is a static analysis tool that takes an FHE circuit and the configuration of the FHE scheme and outputs the same circuit but with bootstrapping hints. The hints inform the pipeline to bootstrap the output of a particular instruction earlier before the noise accumulation exceeds the critical threshold. Early (proactive) bootstrapping allows us to further mask the impacts of noise hazards and therefore accelerate the overall execution.

The RM is a comprehensive data structure that captures each register in a circuit and statically tracks the noise growth in each instruction. Figure 4 shows an example circuit analyzed and captured in the RM data structure. The RM captures individual registers in each line and constructs a map for it. The map includes seven properties, as indicated in the figure. The `Noise` property is an estimate of the noise accumulated in the destination register. A fresh ciphertext has the minimum noise, which is set to 1. Notably, the `Noise` and the `Dist` (Distance) properties help the RMT to output the proactive bootstrapping hints.

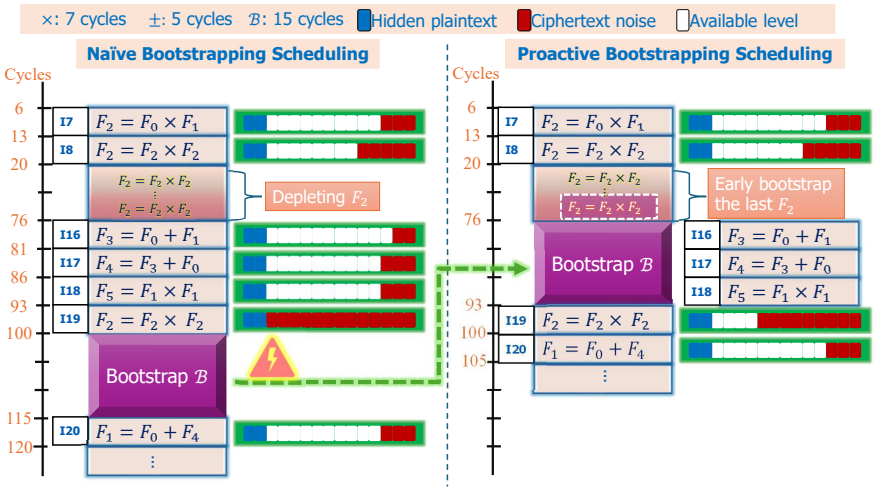
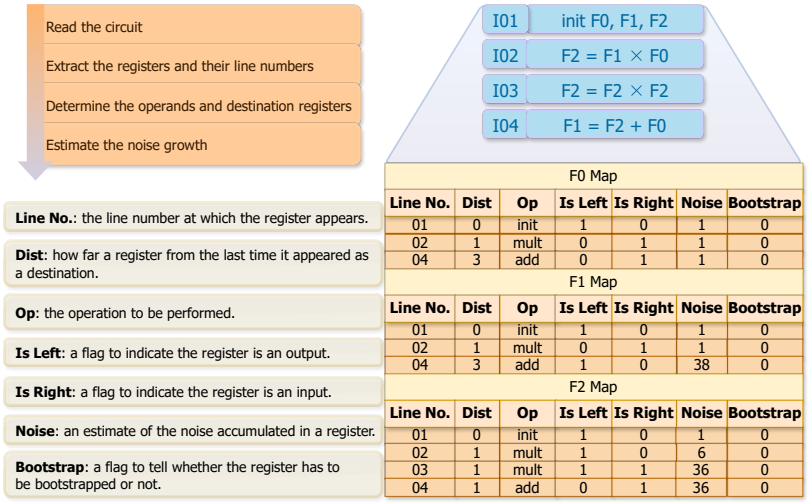


Fig. 3: ALAP vs. proactive bootstrapping.



F0 Map

Line No.	Dist	Op	Is Left	Is Right	Noise	Bootstrap
01	0	init	1	0	1	0
02	1	mult	0	1	1	0
04	3	add	0	1	1	0

F1 Map

Line No.	Dist	Op	Is Left	Is Right	Noise	Bootstrap
01	0	init	1	0	1	0
02	1	mult	0	1	1	0
04	3	add	1	0	38	0

F2 Map

Line No.	Dist	Op	Is Left	Is Right	Noise	Bootstrap
01	0	init	1	0	1	0
02	1	mult	1	0	6	0
03	1	mult	1	1	36	0
04	1	add	0	1	36	0

Fig. 4: The RM structure captures registers in the circuit.

As the RMT scans the circuit, it adds a new row of properties for each register. Each row represents the status of a register at a particular line. The `Noise` property is updated according to the type of instruction and whether the register is a destination or an input to an instruction. If the register is a destination, then its noise increases. If it is an input register, its noise remains the same.

Instructions introduce varying degrees of noise depending on their underlying homomorphic operations. In effect, we categorize the instructions into buckets, where instructions within the same bucket introduce a fixed amount of noise. For example, `bucket0` includes instructions such as `mov` and `load` that do not add additional noise to the destination register. Meanwhile, `bucket1` includes instructions such as `add` and `sub` that add the sum of the noise of the input registers. Similarly, `bucket3` includes more expensive instructions such as `mult` and `div` that double the sum of the noise of the input registers to the destination register. In general, the noise is computed according to this equation:

Algorithm 1: RegisterMap (RM) Construction

Input : Circuit $Cir = (ins_0, \ldots, ins_{n-1})$; Boots-hint set $H \subseteq \{0, \ldots, n-1\} \times \text{Reg}$
Output: RegisterMap M: reg $\mapsto$ ordered list of rows

1 **Function** `BuildRegisterMap(`Cir, H`):`
2 $M \leftarrow \emptyset$; $N_{fresh} = 1$
3 $live \leftarrow \emptyset$ ▷ `register → current noise`
4 $lastWrite \leftarrow \emptyset$ ▷ `register → index of last write`
5 **foreach** $ins \in Cir$ in order **do**
6 $idx \leftarrow ins.index$; $dst \leftarrow ins.dst$; $op \leftarrow ins.operation$; $S \leftarrow ins.srcs$
7 **foreach** $s \in S$ **do** ▷ `Snapshot every source operand` *`before`* `this op runs`
8 $n_s \leftarrow live[s]$ **if** $s \in live$ **else** N_{fresh}
▷ `Append to M[s] <index, distance, op, isLeft, isRight, noise, bootstrap>`
9 AppendRow$(M[s],\ \langle idx,\ idx - lastWrite[s],\ op,\ \mathsf{F},\ \mathsf{T},\ n_s,\ \mathsf{F}\rangle)$
10 **if** $dst \neq \perp$ **then** ▷ `Destination noise, Equation 2`
11 **if** $S = \emptyset$ **then**
12 $raw \leftarrow N_{\text{fresh}}$ ▷ `No source operands`
13 **else**
14 $raw \leftarrow \max_{s \in S}$ **if** $\text{bucket}(ins) = 0$ **else** $raw \leftarrow W_{\text{bucket}(op)} \cdot \sum_{s \in S} live[s]$
▷ `Bootstrap decision: forced hint` *`or`* `ALAP crossing of` τ
15 $forced \leftarrow (idx, dst) \in H$; $boot \leftarrow forced \vee \lfloor \log_2 raw \rfloor \geq \tau$; $noise \leftarrow 2^{\tau - \omega}$ **if** $boot$ **else** raw
▷ `Append to M[s] <index, distance, op, isLeft, isRight, noise, bootstrap>`
16 AppendRow$\big(M[dst],\ \langle idx,\ idx - lastWrite[dst],\ op,\ \mathsf{T},\ (dst{\in}S),\ noise,\ boot\rangle\big)$
17 $live[dst] \leftarrow noise$; $lastWrite[dst] \leftarrow idx$
18 **return** M

$$dst.noise = \begin{cases} src.noise & \text{instr.} \in bucket0 \\ bucketWeight\times & \\ (src_1.noise + src_2.noise) & \text{otherwise.} \end{cases} \tag{2}$$

When the noise reaches a threshold τ, it sets the `Bootstrap` property to `True`. Then, it reduces the register's noise by some fixed amount ω. The values of τ and ω are determined heuristically from the multiplicative depth set for the FHE scheme. Since the concept of "noise" is complementary to "levels", Equation 2 is fundamentally tracking the multiplication level of a ciphertext based on the noise estimation.

The RMT uses a bespoke optimization algorithm to generate the bootstrapping hints. The algorithm iterates over individual registers and inspects their maps. If some register F_x at row i needs to be bootstrapped, the algorithm evaluates its source registers to identify which exhibits the highest noise. Assume that the source register with the highest noise is `srcX`. The optimization algorithm marks `srcX` for proactive bootstrapping at the last line j it appears as a destination register (s.t. $j < i$) under two conditions: (1) The distance of `srcX` is greater than or equal to a threshold π, which defines the minimum number of instructions between the line where the register must be bootstrapped and the last time it appeared as a destination. (2) `srcX` is not marked for bootstrapping at line j.

Algorithm 1 lists the pseudocode to construct the RM for a given input circuit, while Algorithm 2 lists the pseudocode to analyze the circuit and applies the proactive bootstrapping hints. The RMT starts with executing Algorithm 2, which begins with building the RM (line 4) on an empty hint set (H). Then, the loop at line 6 iterates over the entries of the map (M) and checks if the destination register is marked for bootstrapping or not (line 7). If a register is marked for bootstrapping, then the block of lines 9–14 check which source register has the highest noise and applies the proactive bootstrapping hint accordingly.

Algorithm 1 is only responsible for constructing the RM and estimating the noise based on the operation of the instruction. The algorithm starts with iterating over the lines of the circuit (line 5). Then, line 6 fetches the line number within the circuit (idx), the destination (dst) register, the operation of the instruction (op),

Algorithm 2: Proactive Bootstrapping via RegisterMap

Input : Circuit Cir; distance threshold π; max rounds R
Output: Boots-hint set $H \subseteq \{0, \ldots, n-1\} \times \text{Reg}$

```
1  Function PlaceProactiveHints(Cir, π, R):
2    H ← ∅
3    for t ← 1 to R do
4      M ← BuildRegisterMap(Cir, H)
5      H' ← H
6      foreach row ρ ∈ M do
7        if ρ.boot ∧ ρ.left then
8          C ← { s ∈ ρ.srcs : M[s] is defined }
9          if C ≠ ∅ then
             ▷ srcX: source with the highest noise entering ρ
10           s* ← arg max_{s∈C} InNoise(s, ρ.index)
             ▷ last write of s* before ρ
11           j ← LastWriteBefore(s*, ρ.index)
12           if j ≠ ⊥ then
               ▷ Cond. 1: work ≥ π; Cond. 2: s* not boosted at j
13             if ρ.index − j.index ≥ π ∧ ¬j.boot then
14               H' ← H' ∪ {(j.index, s*)}
15     if H' = H then
16       break
17     H ← H'
     ▷ final pass, consistent with H
18   M ← BuildRegisterMap(C, H)
19   return H
```

and the source operands (S). Then, the loop at line 7 creates a map for each source operand and appends a row of properties to it. Then, the `if` block at lines 10–17 applies the estimation of noise according to Equation 2 and updates the bootstrapping and noise properties of the destination register in its map.

The RMT is a **static analyzer** that runs on the user side without executing the circuit. Thus, if the circuit defines a loop, the code inside the loop is evaluated once. Therefore, we remark that the RMT may not capture all optimization opportunities. In order for the RMT to be 100% accurate, then it must unroll loops, which in turn means evaluating the circuit (at least partially). However, this is against the philosophy of outsourced computation, where users execute a circuit in its entirety on remote servers. Therefore, the RMT is optimized to analyze circuits with unrolled loops. Otherwise, we rely on ALAP bootstrapping to detect noise hazards. Because the RMT structures every instruction into discrete maps, its memory overhead scales linearly with the instruction count. This typically bounds the memory footprint to roughly 3× the size of the compiled circuit. The static analysis completes on the order of seconds, even for very deep circuits.

The performance gains of the RMT are significantly impacted by the τ, π and ω parameters; suboptimal parameter configurations can severely degrade performance. For instance, the distance threshold π determines the minimum number of instructions to be executed concurrently if we invoke bootstrapping early. If π is too large (e.g., 100), then no ciphertext will be marked for proactive bootstrapping unless there are at least 100 instructions that can overlap with bootstrapping. Hence, proactive bootstrapping might never occur, and the pipeline ends up following the ALAP strategy. If π is too small (e.g., 1), then proactive bootstrapping can be invoked at any point as long as there is at least 1 instruction to overlap with bootstrapping. This causes the risk of premature bootstrapping. Our empirical analysis identified that setting π to 5 offers the best option to proactively invoke bootstrapping while ensuring that its latency can be sufficiently masked.

The choice of RMT parameters depends on the multiplicative depth configured for CKKS and how deep the circuit is. For example, assume that CKKS is configured with a depth of 30, which means that we have 29 multiplication levels (0 bits of extra noise) and 1 level for bootstrapping. After bootstrapping a ciphertext,

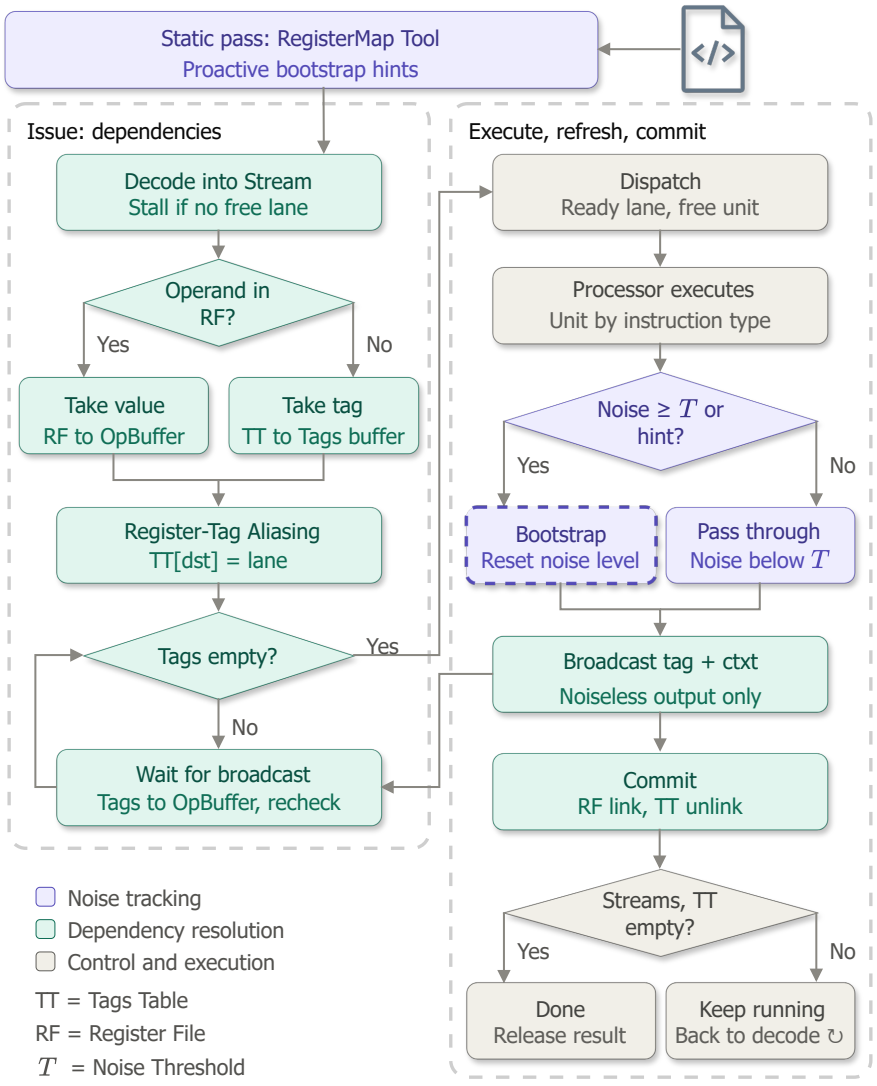


Fig. 5: This flowchart depicts the workflow of LazyArc virtual machine.

it will have 13 multiplication levels (16 noise bits in the ciphertext). Therefore, we set τ to 16, meaning we mark a ciphertext to be bootstrapped when the noise is at least 16 bits. Consequently, ω is defined to be 0–13, meaning that bootstrapping removes ω noise bits that make the ciphertext have 16 noise bits.

Figure 5 visualizes the flow of our proposed virtual machine. Bootstrapping is considered the module with the highest latency and it is running in parallel with other modules. Hence, as long as the Bootstrapper queue has waiting ciphertexts, it is always processing them regardless of other modules.

Algorithm 3: Mapping an approximate integer value into $\mathbb{Z}_Q[x]/(x^d)$

Input : Approximate integer value $v \geq 0$, degree d, modulus Q
Output: A polynomial $D \in \mathbb{Z}_Q[x]/(x^d)$

$v = \lfloor v \rceil$;
$mod \leftarrow 1$; $maxCoef \leftarrow Q$; $exp \leftarrow d$;
while $exp > 0$ **do**
  if $exp \pmod 2 == 1$ **then**
    $mod \leftarrow mod \times maxCoef$;
  $maxCoef \leftarrow maxCoef \times maxCoef$;
  $exp \leftarrow \lfloor exp/2 \rfloor$;
$mask \leftarrow mod - 1$; $n \leftarrow v \ \& \ mask$; $D \leftarrow \emptyset$;
for $i \leftarrow 0$ **to** $d - 1$ **do**
  Append $(n \pmod Q)$ to D;
  $n \leftarrow \lfloor n/Q \rfloor$;
return D;

4.3 Hybrid Arithmetic-Boolean Execution Engine

To allow LazyArc to process arithmetic and Boolean circuits, we propose an encoding technique to seamlessly switch a CKKS ciphertext (real-numbers domain) to FHEW ciphertext (binary domain), using existing scheme switching methods. Scheme switching (SS) is an expensive operation that is under active research.

There are two strategies to perform SS in OpenFHE: the first is a CKKS/FHEW switching, and the second is a BFV/CKKS switching. The first approach allows fine control on the bits of a ciphertext, and thus more flexibility to perform arbitrary computations. However, this approach is limited to small values (< 8). The latter approach is more efficient for larger values, but is dedicated to BFV and CKKS ciphertexts.

To allow the first SS approach to support larger values, we adapt an encoding scheme that encodes a plaintext into a structure that can be encrypted in CKKS and uses existing SS techniques to switch a ciphertext from CKKS to FHEW. This approach maps a plaintext value from the real-number domain to a polynomial of degree d with coefficients modulo Q (s.t. $d = 2k$ and $Q = 2^n$, for some $k, n \in \mathbb{Z}$), i.e.,

$$\xi : v \in \mathbb{R} \rightarrow \mathbb{Z}_Q[x]/(x^d) \tag{3}$$

Algorithm 3 shows the encoding function ξ. This algorithm is similar to the gadget decomposition technique. Gadget decomposition is a method that takes a ciphertext and splits it into smaller elements to control noise growth during homomorphic operations. Both gadget decomposition and Algorithm 3 rely on decomposing larger values into smaller chunks. However, Algorithm 3 takes a single scalar value and packs its base-Q digits into the coefficients of a single polynomial $D(x)$. $D(x)$ is then encrypted as a single (packed) plaintext.

The degree d is set according to the desired range of plaintext values. Thus, larger plaintext values require a higher degree. For instance, assuming the coefficient modulus $Q = 8$, to encode the value 13, a degree d of 2 is sufficient, while the value 123321 requires a degree of 6. Generally, the degree required to encode a value x can be estimated as $d = \lceil log_Q(x) \rceil$.

Performing arithmetic operations in FHEW requires implementing special Boolean circuits to perform polynomial addition and multiplication. Multiplication on the level of Boolean circuits is more complex and expensive than other operations. This overhead arises because multiplying two ciphertexts translates into multiplying the plaintext polynomials used to encode them.

Notably, Boolean FHE schemes inherently avoid noise hazards because ciphertexts are bootstrapped immediately after evaluating a Boolean gate. Gate operations and bootstrapping are tightly coupled operations. Thus, if we were to decouple them (i.e., perform the gate operation and then later bootstrap the ciphertext), this will not introduce any benefits beyond delaying the bootstrap operation. The ciphertext must be bootstrapped before being used in subsequent operations. Unlike CKKS ciphertext that can be used for

Table 2: Benchmarks sizes and descriptions.

Benchmark	Size	Description	Max Depth
Newton-Schulz	3x3 4x4 5x5	A family of iterative algorithms often used for computing the matrix inverse.	19
Polynomial Multiplication	deg-8 deg-16	Multiplying two polynomials of degree-8 and degree-16. We overloaded the circuit with multiplication instructions to trigger bootstrapping.	17
Matrix Multiplication	8x8 16x16	Multiplying two matrices of size 8x8 and 16x16. We overloaded the circuit with multiplication instructions to trigger bootstrapping.	17
Statistics	Size 4 Size 8	Computing the mean, variance, STD, maximum, and minimum values of a set of values.	29
Private Set Intersection (PSI)	12x12 16x16 24x24 30x30	Identifying common elements between two encrypted sets.	27
Stress Test Addition/Multiplication	1 block 2 blocks 3 blocks 4 blocks	A suite to test the proactive bootstrapping by executing circuits of large multiplicative depth. Each block involves 25 consecutive multiplication followed by a set of additions or multiplications.	30
Unbalanced Decision Tree (UDT)	Depth 5	A binary decision tree of tree depth of 5. The tree is unbalanced; the root's left sub-tree is a leaf, and the right side goes up to depth 5.	27

Table 3: The CKKS parameters setup for a 128-bit security (the precision is fixed to 59 bits).

Multiplicative Depth	Ring Dimension	Cyclotomic Order	Ciphertext Modulus Size (bits)
30	131072	262144	1830
29			1772
27			1654
20	65536	131072	1241
19			1182
17			1064

several computations without bootstrapping, Boolean ciphertext cannot be used until being bootstrapped *immediately* after evaluating a homomorphic gate.

5 Experimental Evaluation

Table 2 summarizes our benchmarks and the multiplicative depths used to initialize CKKS. We implemented LazyArc using the OpenFHE library v1.5.1, and all our benchmarks use 128-bit security parameters. The parameters are shown in Table 3. Our experiments were performed on a Linux machine with an AMD EPYC 7532 32-Core Processor with 128-GB of RAM. We set 120-min as a time limit for each benchmark.

Table 4 lists the execution time (in seconds) for six benchmarks compared to their baselines. All our benchmarks were implemented from scratch to match our ISA syntax that is parsed by LazyArc. Our virtual machine utilizes a hybrid multi-threading paradigm: coarse-grained Task-Level Parallelism (TLP) to decouple bootstrapping from instruction execution, combined with fine-grained Data-Level Parallelism (DLP) for the underlying computations on ciphertext polynomials. The columns OoO+ and InO+ represent Out-of-Order and In-Order execution with the hybrid multi-threading paradigm, while the columns OoO and InO represent Out-of-Order and In-Order execution with TLP only. For each pair of experiments, the column S shows the speedup gained by overcoming noise hazards. S is computed by the equation $Time_{InO}/Time_{OoO}$.

Table 4: Execution times in seconds (rounded) for Out-of-Order vs. In-Order configurations. The '+' suffix denotes execution with fine-grained Data-Level Parallelism (DLP) enabled within OpenFHE.

		With DLP (+)			Without DLP		
Circuit	**Size**	**OoO+**	**InO+**	S	**OoO**	**InO**	S
Newton–Schulz	3×3	211	519	2.46×	318	692	2.18×
	4×4	536	616	1.15×	1479	1559	1.05×
	5×5	1272	2816	2.21×	1898	3511	1.85×
Poly. Mult.	deg-8	368	743	2.02×	585	1093	1.87×
	deg-16	1664	2692	1.62×	2662	3973	1.49×
Matrix Mult.	8×8	199	411	2.07×	340	561	1.65×
	16×16	922	1578	1.71×	1776	2522	1.42×
Stats.	4	482	509	1.06×	654	676	1.03×
	8	1045	1112	1.06×	1540	1571	1.02×
PSI	12×12	89	208	2.34×	140	302	2.16×
	16×16	672	1577	2.35×	1086	1991	1.83×
	24×24	1301	2101	1.61×	2153	3627	1.68×
	30×30	2002	3291	1.64×	3253	5400	1.66×
UDT	Depth 5	835	952	1.14×	2482	3384	1.36×

We highlight that although some benchmarks, such as Matrix and Polynomial Multiplications, are embarrassingly parallel workloads, the speedup achieved by LazyArc is mainly attributed to the overlapping of computations with bootstrapping. In fact, we do not perform ad hoc parallelization for each workload. Instead, all the speedup is attributed to the instruction-bootstrapping scheduling and to the ciphertext-level parallelization implemented by OpenFHE.

To demonstrate the efficiency of the `RegisterMap` Tool, we perform a set of stress tests. These tests are designed to include a deep multiplicative depth to intentionally deplete a ciphertext and to include independent instructions in between. We implemented two suites of stress tests: The first is `ST.Add`, which is a circuit that involves a depletion region followed by a set of additions. The second suite is `ST.Mult` which is a circuit that involves a depletion region followed by a set of multiplications. Each depletion region followed by additions or multiplications is referred to as a block. Thus, to test deeper circuits, we repeat the block up to 4 times. The depletion region consists of 25 consecutive multiplications. For an `ST.Add` block, the depletion region is followed by 10 additions, 2 multiplications, and another 5 additions. For an `ST.Mult` block, the depletion region is followed by 17 multiplications. To complement these synthetic suites, we also evaluate an Unbalanced Decision Tree (UDT) inference circuit. The UDT serves as a realistic machine learning workload that naturally exhibits the heterogeneous, deep dependency graphs required to trigger proactive bootstrapping.

Table 5 shows a comparative analysis of the run time of LazyArc executing the circuits with and without proactive bootstrapping versus the in-order execution. The circuits were tested with the hybrid multithreading paradigm. The multiplicative depth of the stress test circuits was set to 30 (optimized with RMT parameters $\tau = 18, \omega = 13, \pi = 5$). In contrast, the UDT utilized a multiplicative depth of 27. Because the UDT exhibits a significantly denser and more heterogeneous dependency graph than the linear stress tests, the RMT allows architectural parameter tuning to prevent the static noise heuristic from triggering premature hints on highly branched paths. By calibrating the threshold for this structural complexity, the UDT was efficiently evaluated using $\tau = 52, \omega = 17, \pi = 5$, demonstrating the architecture's adaptability to heterogeneous workloads.

Table 6 shows the performance of the fundamental operations on FHEW ciphertext under scheme switching. We used degree-4 polynomials and the modulus Q (the coefficient modulus) is set to 8. We constructed the adder, subtractor, and multiplier circuits utilizing the homomorphic Boolean gates supported by the

Table 5: Comparison of Proactive OoO+, OoO+, and InO+ execution. All times are in seconds. S is the speedup of Proactive OoO+ over InO+.

Circuit	Size	Proactive OoO+	OoO+	InO+	S
ST.Add	1	37.63	46.22	69.85	1.86×
	2	58.83	73.41	144.27	2.45×
	3	85.19	105.90	213.21	2.5×
	4	104.83	132.13	277.93	2.65×
ST.Mult	1	40.20	54.01	84.16	2.09×
	2	68.40	92.38	166.12	2.43×
	3	98.20	125.48	241.17	2.46×
	4	130.60	159.51	321.12	2.46×
UDT	Depth 5	765.37	834.82	952.22	1.24×

Table 6: Run time of the core FHEW operations.

Operation	Run time (s)	Operation	Run time (s)
BinAdd	133.26	XOR Gate	62.99
BinSub	128.07	AND Gate	70.01
BinMult	3137.03	OR Gate	67.14

FHEW scheme. Multiplication is the most expensive operation due to its circuit complexity. The Boolean gate operations are significantly faster compared to multiplication. Thus, we recommend using SS for evaluating Boolean gates only, while delegating the arithmetic operations to CKKS. Since Boolean operations do not suffer from noise hazards, the circuits were executed in-order with DLP.

We highlight that the results in Table 6 are based on switching from the CKKS scheme to the FHEW scheme. The initialization of the FHEW parameters occurs simultaneously when initializing the CKKS parameters. Both are initialized with parameters to guarantee 128-bit of security. Also, the performance of the Boolean operations may seem slower than the performance of the actual scheme. This overhead arises because the pipeline operates on ciphertexts that have undergone scheme switching from CKKS to FHEW.

To evaluate circuits that include a combination of arithmetic and Boolean operations, we simulate the core operations of the Simon and Speck ciphers. These algorithms perform arithmetic operations, which are delegated to CKKS, and Boolean operations (mostly XORs), which are delegated to FHEW. Because FHEW's per-gate functional bootstrapping prevents noise accumulation, these workloads do not experience noise hazards and were executed in-order with the DLP paradigm. The run times of these benchmarks are shown in Table 7. To isolate the performance of the hybrid Arithmetic-Boolean pipeline, we evaluate the core timing execution of individual cipher rounds, omitting the full key-schedule initialization required for end-to-end cryptographic deployment.

5.1 Evaluation against Prior Schedulers and Runtimes

The work most similar to ours is FHEBooster [42]. While both FHEBooster and LazyArc optimize multi-core FHE scheduling, they utilize fundamentally different paradigms. FHEBooster is a static compile-time scheduler that uses computationally heavy Integer Programming (IP) solvers to minimize bootstrapping counts, strictly executing predetermined task queues in-order per thread. **It fundamentally still leaves the execution pipeline fully stalled whenever bootstrapping occurs.**

In contrast, LazyArc is a dynamic Out-of-Order (OoO) runtime architecture. Instead of treating noise accumulation as a static path constraint, LazyArc formalizes it as a dynamic "noise hazard," resolving it at run time. Crucially, LazyArc dynamically evaluates independent instructions while a bootstrap is

Table 7: Benchmarking the Arithmetic-Boolean pipeline on Simon and Speck ciphers for different number of rounds. All times are in minutes (rounded).

Benchmark	**1 Round**	**2 Rounds**	**3 Rounds**
Simon Cipher	21	43	52
Speck Cipher	5	8	11

Table 8: A qualitative comparison of LazyArc vs. FHEBooster.

Work	**FHEBooster [42]**	**LazyArc**
Goal	Reduce the number of bootstrappings	Mask the latency of bootstrapping
Schedule	Integer programming (IP)	Out-of-Order (OoO)
Runtime	Static, in-order Heavy static analysis	Dynamic, out-of-order Lightweight static analysis
Optimize Paradigm	Applies parallelism to execute independent instructions. It stalls when encountering dependencies	Schedules the instructions with bootstrapping. It minimizes the stalls caused by bootstrapping
Bootstrap Strategy	**Static–reactive**: a circuit is **statically** processed before execution to determine exact nodes to bootstrap. It introduces significant pre-execution overhead	**Dynamic–proactive**: automatic bootstrapping while evaluating instructions concurrently. Proactive bootstrapping is possible with no overhead

actively pending (i.e., a latency-masking capability is impossible within FHEBooster's rigid static queues). This dynamic adaptability of LazyArc enables higher throughput on deeper FHE circuits while slashing preprocessing overhead from minutes down to mere seconds. We show a qualitative comparison with FHEBooster in Table 8. All of the FHEBooster benchmarking was conducted on the same machine that we used to evaluate LazyArc.

To establish a direct quantitative comparison, we leverage the open-source Logistic Regression (LR) inference workload provided by FHEBooster. Table 9 presents the evaluation of both FHEBooster and LazyArc across datasets with 16, 32, and 64 attributes. All the circuits were evaluated on a multi-threaded environment on LazyArc and FHEBooster with a fixed multiplicative depth of 17. Notably, FHEBooster exhibits limited expressiveness for arbitrary circuit design because its workloads are constrained by a rigid graph representation. Furthermore, while FHEBooster was originally evaluated on additional synthetic workloads, these circuits lacked sufficient specification to enable faithful replication. Consequently, to ensure a fair and reproducible evaluation, our comparative analysis strictly focuses on the realistic LR benchmark.

PulpFHE [5] shares similarities with LazyArc, as both are software-based architectures that utilize the same instruction set. However, they differ significantly in their operational focus: PulpFHE optimizes its functional units for evaluating TFHE Boolean circuits, whereas LazyArc serves as a dynamic scheduler tailored for the CKKS scheme. A performance comparison between the two frameworks is presented in Table 10. It is important to note that, due to the different backend FHE schemes employed, this comparison should be viewed as an assessment of overall architectural efficiency and scalability, rather than a direct evaluation of their respective parallelization or scheduling mechanisms.

EasyFHE [43] is a SOTA work that is relatively similar to LazyArc. EasyFHE features GPU-backend optimizations, such as removing redundancy and instruction scheduling. The instruction scheduling approach aims to overlap memory transfers with computations to hide **memory latency**. In contrast, LazyArc is a CPU-based virtual machine, where its dynamic scheduling aims to hide the **bootstrapping latency**.

Table 9: A quantitative comparison of LazyArc vs. FHEBooster. The circuits use a fixed multiplicative depth of 17. Speedup = FHEBooster/LazyArc.

Circuit	**FHEBooster (secs)**	**LazyArc (secs)**	**Speedup**
LR_16	28	5	5.60×
LR_32	32	8	4.00×
LR_64	51	13	3.92×

Table 10: Execution times in seconds (rounded) for LazyArc (OoO+) vs PulpFHE. All PulpFHE circuits utilized 16-bit data. "NaN" indicates the circuit did not finish within 120 minutes. S = PulpFHE/LazyArc.

Circuit	**Size**	**OoO+**	**PulpFHE**	S
Newton-Schulz	3×3	211	498	2.36×
	4×4	494	2303	4.66×
	5×5	1272	NaN	–
Poly. Mult.	deg-8	368	610	1.66×
	deg-16	1664	2328	1.4×
Matrix Mult.	8×8	199	4701	23.62×
	16×16	922	NaN	–
Statistics	4	482	1076	2.23×
	8	1045	1173	1.12×
PSI	12×12	89	2585	29.04×
	16×16	672	4556	6.78×
	24×24	1301	NaN	–
	30×30	2002	NaN	–
UDT	Depth 5	835	NaN	–

6 Discussion

Sources of Throughput Gains. There are two complementary reasons for the performance improvements of LazyArc. First, LazyArc allows executing instructions Out-of-Order while bootstrapping another ciphertext concurrently. This is demonstrated in the columns 6-8 of Table 4, where a standard OoO paradigm achieves 1.02–2.18× speedup. To further complement this speedup, we enable additional data-level parallelism, where both bootstrapping and instructions run on multiple cores on the polynomial level. This is demonstrated in columns 3-5 in Table 4. Our reported performance gain is up to 2.46×.

Loop Unrolling and Fallback Penalty. For the RMT, the number of iterations must be fixed (so loops can be statically unrolled). Because the RMT is a static-analysis tool, it is designed to operate without evaluating the circuit (either fully or partially) to compute iterations. Evaluating the circuit at the user-side violates the core philosophy of computation outsourcing, where the full computation must be performed on the server-side. If loops cannot be statically unrolled, the RMT treats them as single-instruction blocks. This results in an ALAP bootstrapping fallback where LazyArc still performs OoO execution, but proactive bootstrapping optimizations are disabled. The penalty in this case is simply a reversion to standard ALAP performance speeds.

RMT Model Accuracy and Sensitivity. Our noise estimation model (Equation 2) provides a high-fidelity approximation of noise growth by tracking multiplications and additions. In fact, Equation 2 focuses on reflecting how noise grows with intensive instructions. This estimation becomes inaccurate if ciphertexts are initialized with low multiplication levels, because Equation 2 assumes that all ciphertext are fresh with 0

extra noise. By understanding how many levels of multiplication a fresh ciphertext allows and how many levels we get after bootstrapping, we can tune τ and ω precisely.

Distinction from OpenFHE Runtime. OpenFHE is an FHE library; it provides the necessary functionalities to run encrypted computations on a particular FHE scheme. In contrast, LazyArc is a virtual machine that leverages OpenFHE as a backend execution engine. OpenFHE only relies on polynomial-level parallelization to accelerate the computations. **Notably, noise analysis and bootstrapping is completely the user's responsibility.** Furthermore, OpenFHE does not perform dynamic scheduling to reorder the instructions. Conversely, LazyArc performs Out-of-Order scheduling to mask the latency of bootstrapping. Moreover, the RMT aims to maximize the number of instructions to be executed within the bootstrapping period to further enhance the performance. Also, LazyArc automatically detects noise hazards and resolves them without manual user intervention.

Numerical accuracy and correctness. LazyArc does not alter the logical flow or algorithmic dependencies of a computation to achieve out-of-order (OoO) execution. Consequently, the architecture guarantees that OoO evaluation of a circuit yields results identical to standard sequential execution. The numerical accuracy of the output is determined solely by the underlying cryptographic scheme and its parameterization. As an approximate homomorphic encryption scheme, CKKS natively evaluates arithmetic with floating-point approximations. In deep circuits with complex operations, minor precision loss is expected due to inherent noise growth and scale management. For instance, an exact plaintext value of 3 may decrypt to 2.973. Crucially, this precision behavior is an intrinsic property of the CKKS encoding and scheme parameterization, and is entirely independent of LazyArc's scheduling architecture.

7 Conclusion

This work introduced LazyArc, a lightweight FHE virtual machine built for Out-of-Order execution of FHE instructions. Our analysis formalizes the concept of noise hazards in FHE circuits and analyzes how they cause unnecessary stalls in FHE pipelines. LazyArc is the first work to realize dynamic instruction-bootstrapping scheduling for FHE circuits. Our proposed dynamic scheduling approach efficiently avoids noise hazards and accelerates the execution of CKKS circuits. We complement LazyArc with a new static analysis tool, named the `RegisterMap` Tool (RMT), which statically analyzes FHE circuits and outputs bootstrapping hints to allow proactive bootstrapping to further reduce FHE execution stalls. LazyArc was evaluated on a computationally intensive set of benchmarks, such as the Private Set Intersection and Newton-Schulz algorithms, and demonstrated up to 2.46× speedup compared to the naive sequential execution paradigm, and about 5.60× speedup compared to FHEBooster that utilizes Integer Programming models. In addition, the proactive bootstrapping strategy demonstrated a speedup of up to 2.65× compared to the In-Order baselines.

By resolving the inherent noise hazards of FHE pipelines through dynamic out-of-order scheduling and static proactive hints, LazyArc bridges the gap between rigid compile-time schedules and dynamic execution unpredictability. Furthermore, by seamlessly supporting hybrid Arithmetic-Boolean workloads, LazyArc establishes a comprehensive, high-throughput architectural foundation for the next generation of privacy-preserving outsourced computations.

Acknowledgment

The authors would like to acknowledge the support of the National Science Foundation (Award 2239334).

References

1. Fargana Abdullayeva. Cyber resilience and cyber security issues of intelligent cloud computing systems. *Results in Control and Optimization*, 12:100268, 2023.

2. Rashmi Agrawal, Anantha Chandrakasan, and Ajay Joshi. HEAP: A fully homomorphic encryption accelerator with parallelized bootstrapping. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*, pages 756–769, 2024.
3. Rashmi Agrawal, Leo De Castro, Chiraag Juvekar, Anantha Chandrakasan, Vinod Vaikuntanathan, and Ajay Joshi. Mad: Memory-aware design techniques for accelerating fully homomorphic encryption. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 685–697, 2023.
4. Omar Ahmed, Charles Gouert, and Nektarios Georgios Tsoutsos. PEEV: Parse encrypt execute verify–a verifiable FHE framework. *IEEE Access*, 2024.
5. Omar Ahmed and Nektarios Georgios Tsoutsos. PulpFHE: Complex instruction set extensions for FHE processors. Cryptology ePrint Archive, Paper 2024/1315, 2024.
6. Ahmad Al Badawi et al. OpenFHE: Open-source fully homomorphic encryption library. In *proceedings of the 10th workshop on encrypted computing & applied homomorphic cryptography*, pages 53–63, 2022.
7. Ahmad Al Badawi, Bharadwaj Veeravalli, Jie Lin, Nan Xiao, Matsumura Kazuaki, and Aung Khin Mi Mi. Multi-GPU design and performance evaluation of homomorphic encryption on GPU clusters. *IEEE Transactions on Parallel and Distributed Systems*, 32(2):379–391, 2020.
8. Asra Ali, Jaeho Choi, Bryant Gipson, Shruthi Gorantala, Jeremy Kun, Wouter Legiest, Lawrence Lim, Alexander Viand, Meron Zerihun Demissie, and Hongren Zheng. HEIR: A universal compiler for homomorphic encryption, 2025.
9. Weihang an, Benjamin M Case, Gengran Hu, Shuhong Gao, and Yingjie Lao. An ultra-highly parallel polynomial multiplier for the bootstrapping algorithm in a fully homomorphic encryption scheme. *Journal of Signal Processing Systems*, 93(6):643–656, 2021.
10. Jean-Philippe Bossuat, Christian Mouchet, Juan Troncoso-Pastoriza, and Jean-Pierre Hubaux. Efficient bootstrapping for approximate homomorphic encryption with non-sparse keys. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 587–617. Springer, 2021.
11. Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (leveled) fully homomorphic encryption without bootstrapping. *ACM Transactions on Computation Theory (TOCT)*, 6(3):1–36, 2014.
12. Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. Homomorphic encryption for arithmetic of approximate numbers. In *International conference on the theory and application of cryptology and information security*, pages 409–437. Springer, 2017.
13. Seonyoung Cheon, Yongwoo Lee, Dongkwan Kim, Ju Min Lee, Sunchul Jung, Taekyung Kim, Dongyoon Lee, and Hanjun Kim. {DaCapo}: Automatic bootstrapping management for efficient fully homomorphic encryption. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 6993–7010, 2024.
14. Seonyoung Cheon, Yongwoo Lee, Hoyun Youm, Dongkwan Kim, Sungwoo Yun, Kunmo Jeong, Dongyoon Lee, and Hanjun Kim. Halo: Loop-aware bootstrapping management for fully homomorphic encryption. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, pages 572–585, 2025.
15. Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. TFHE: fast fully homomorphic encryption over the torus. *Journal of Cryptology*, 33(1):34–91, 2020.
16. Savita Devi and Taran Singh Bharti. A review on detection and mitigation analysis of distributed denial of service attacks and their effects on the cloud. *International Journal of Cloud Applications and Computing (IJCAC)*, 12(1):1–21, 2022.
17. Nir Drucker, Guy Moshkowich, Tomer Pelleg, and Hayim Shaul. BLEACH: Cleaning errors in discrete computations over CKKS. *Journal of Cryptology*, 37(1):3, 2024.
18. Léo Ducas and Daniele Micciancio. FHEW: bootstrapping homomorphic encryption in less than a second. In *Annual international conference on the theory and applications of cryptographic techniques*, pages 617–640. Springer, 2015.
19. Junfeng Fan and Frederik Vercauteren. Somewhat practical fully homomorphic encryption. Cryptology ePrint Archive, Paper 2012/144, 2012.
20. Craig Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the forty-first annual ACM symposium on Theory of computing*, pages 169–178, 2009.
21. Jia-Zheng Goey, Wai-Kong Lee, Bok-Min Goi, and Wun-She Yap. Accelerating number theoretic transform in GPU platform for fully homomorphic encryption. *Journal of Supercomputing*, 77(2), 2021.
22. Charles Gouert, Dimitris Mouris, and Nektarios Georgios Tsoutsos. Juliet: A configurable processor for computing on encrypted data. *IEEE Transactions on Computers*, 73(9):2335–2349, 2024.
23. John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.
24. Lei Jiang, Qian Lou, and Nrushad Joshi. Matcha: A fast and energy-efficient accelerator for fully homomorphic encryption over the torus. In *Proceedings of the 59th ACM/IEEE Design Automation Conference*, pages 235–240, 2022.

25. Amin Karami, Milad Shemshaki, and Mustansar Ali Ghazanfar. Exploring the ethical implications of ai-powered personalization in digital marketing. *Data Intelligence*, 3, 2024.
26. Jongmin Kim et al. Ark: Fully homomorphic encryption accelerator with runtime data generation and inter-operation key reuse. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1237–1254. IEEE, 2022.
27. Aleksandar Krastev, Nikola Samardzic, Simon Langowski, Srinivas Devadas, and Daniel Sanchez. A tensor compiler with automatic data packing for simple and efficient fully homomorphic encryption. *Proceedings of the ACM on Programming Languages*, 8(PLDI):126–150, 2024.
28. Yongwoo Lee, Seonyoung Cheon, Dongkwan Kim, Dongyoon Lee, and Hanjun Kim. {ELASM}:{Error-Latency-Aware} scale management for fully homomorphic encryption. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 4697–4714, 2023.
29. Christian Meurisch and Max Mühlhäuser. Data protection in AI services: A survey. *ACM Computing Surveys (CSUR)*, 54(2):1–38, 2021.
30. Fatemehsadat Mireshghallah, Mohammadkazem Taram, Praneeth Vepakomma, Abhishek Singh, Ramesh Raskar, and Hadi Esmaeilzadeh. Privacy in deep learning: A survey, 2020.
31. Chao Niu, Zhicong Huang, Zhaomin Yang, Yi Chen, Liang Kong, Cheng Hong, and Tao Wei. Xboot: Free-XOR gates for CKKS with applications to transciphering. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2025(4):118–144, 2025.
32. Dijana Peras and Renata Mekovec. A conceptualization of the privacy concerns of cloud users. *Information & computer security*, 30(5):653–671, 2022.
33. Nikola Samardzic et al. CraterLake: a hardware accelerator for efficient unbounded computation on encrypted data. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, pages 173–187, 2022.
34. Bilel Sefsaf, Abderraouf Dandani, Abdessamed Seddiki, Arab Mohammed, Eduardo Chielle, Michail Maniatakos, and Riyadh Baghdadi. Chehab rl: Learning to optimize fully homomorphic encryption computations. In *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 342–360, 2026.
35. Huajie Shen, Qian Xu, Bo Yu, Yuhan Yang, and Wei He. Bootstrapping in approximate fully homomorphic encryption: a research survey. *Cybersecurity*, 8(1):87, 2025.
36. Hamed Tabrizchi and Marjan Kuchaki Rafsanjani. A survey on security challenges in cloud computing: issues, threats, and solutions. *The journal of supercomputing*, 76(12):9493–9532, 2020.
37. Jonathan Takeshita, Dayane Reis, Ting Gong, Michael T Niemier, Xiaobo Sharon Hu, and Taeho Jung. Accelerating finite-field and torus fully homomorphic encryption via compute-enabled (s) ram. *IEEE Transactions on Computers*, 73(10):2449–2462, 2023.
38. R. M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of Research and Development*, 11(1):25–33, 1967.
39. Gurpreet Singh Walia and P Deepalakshmi. Protecting cloud computing environments from cyber threats with AI-powered machine learning systems. In *2025 Global Conference in Emerging Technology (GINOTECH)*, pages 1–6. IEEE, 2025.
40. Zhiwei Wang, Peinan Li, Rui Hou, Zhihao Li, Jiangfeng Cao, XiaoFeng Wang, and Dan Meng. He-booster: An efficient polynomial arithmetic acceleration on gpus for fully homomorphic encryption. *IEEE Transactions on Parallel and Distributed Systems*, 34(4):1067–1081, 2023.
41. Rick Weber, Ryan Orendorff, Ghada Almashaqbeh, and Ravital Solomon. Parasol compiler: Pushing the boundaries of FHE program efficiency. In *2026 IEEE Symposium on Security and Privacy (SP)*, pages 3471–3489, 2026.
42. Tommy White, Charles Gouert, Chengmo Yang, and Nektarios Georgios Tsoutsos. FHE-BOOSTER: Accelerating fully homomorphic execution with fine-tuned bootstrapping scheduling. In *2023 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 293–303. IEEE, 2023.
43. Jianyu Zhao, Xueyu Wu, Guang Fan, Mingzhe Zhang, Shoumeng Yan, Lei Ju, and Zhuoran Ji. A framework for developing and optimizing fully homomorphic encryption programs on GPUs. In *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 25–40, 2026.