HEGIDE: A MIMD Oblivious Processor for Private Function Evaluation over CKKS*

Jules Dumezy[1], Nicolas Ye[1], Pierre-Emmanuel Clet[1], Olive Chakraborty[1] and Aymen Boudguiga[1]

Université Paris-Saclay, CEA-List, France
{jules.dumezy,nicolas.ye,pierre-emmanuel.clet,olive.chakraborty,aymen.boudguiga}@cea.fr

Abstract. While FHE enables computation on encrypted data, protecting the program itself remains a theoretical and practical challenge, often forcing practitioners to choose between exposing proprietary logic or suffering impractical performance penalties. This paper introduces HEGIDE, an oblivious processor based on the (discrete) Cheon-Kim-Kim-Song (CKKS) scheme that bridges the gap between theoretical Private Function Evaluation (PFE) and its practical realization.
Central to our contribution is OSReM (Oblivious Shift Register Memory), a novel memory architecture that circumvents the linear complexity of standard FHE-RAM writes. By treating memory as a shift register, OSReM enables low-latency, constant-time writes without the need for expensive full-memory bootstrapping. HEGIDE leverages a MIMD (Multiple Instruction, Multiple Data) design, utilizing CKKS packing to evaluate distinct program threads in parallel, thus maximizing throughput. While the processor architecture natively supports arbitrary word sizes and instructions, we provide a compiler that manages memory scheduling to abstract the complexity of the shift-register design.
We provide a proof-of-concept full implementation of HEGIDE using the OpenFHE library. Experimental results demonstrate the efficiency of our approach, achieving an amortized cycle time of just 6.4 ms for a 16-bit processor – two orders of magnitude faster in throughput than comparable approaches – offering a viable path for the secure execution of proprietary algorithms on encrypted data.

Keywords: FHE · Private Function Evaluation · Oblivious Processor · CKKS · Oblivious RAM · OSReM · Secure Computation

*This work was partly supported by the France 2030 ANR Projects ANR-22-PECY-003 SecureCompute and ANR-23-PECL-0009 TRUSTINCloudS.

Contents

1 Introduction

The rapid adoption of cloud computing has created a pressing need for secure delegation of computation. Organizations possess proprietary algorithms – ranging from high-frequency trading strategies and fraud detection heuristics to specialized medical diagnostic models – that constitute their core Intellectual Property (IP). While Fully Homomorphic Encryption (FHE) has long been considered the "Holy Grail" for protecting sensitive *data* in such scenarios, it creates a paradox for the code owner: to protect the input, they must expose the program.

In all standard FHE schemes, such as B/FV [Bra12,FV12], BGV [BGV14], CKKS [CKKS17], or DM/CGGI [DM15,CGGI20], the server evaluates a specific, public circuit representation of the client's function. Even if the wire values remain encrypted, the circuit's static properties, including its topology, multiplicative depth, gate count, and wiring, act as a unique fingerprint of the underlying logic. For high-value algorithms, this structural leakage is unacceptable. A competitor could infer critical hyperparameters of a proprietary model – such as the depth of a decision tree or the number of layers in a neural network – simply by analyzing the homomorphic execution trace.

We consider the scenario where an entity holding both a private function f and a private input x wishes to delegate the computation $y = f(x)$ to a semi-honest server. In this setting, the inputs, outputs, and the function's logic must remain hidden from the server's view. While this problem can be framed as Private Function Evaluation (PFE), traditional PFE constructions often rely on Universal Circuits or complex multi-round protocols that suffer from high constant overheads, lack reusability, and scale poorly with program size. Consequently, practitioners are often forced to choose between exposing their proprietary code or accepting impractical performance penalties that hinder real-world deployment.

In this paper, we pursue the practical realization of program privacy through a vectorized, encrypted processor architecture called HEGIDE. Rather than obfuscating a specific circuit, HEGIDE implements a general-purpose processor, where the proprietary program is compiled into encrypted machine code consisting of fixed, data-independent Read-Execute-Write cycles. Because the server observes a uniform sequence of homomorphic operations, the control flow and logic of the underlying algorithm are entirely concealed.

The core design philosophy of HEGIDE is to move away from inefficiently emulating legacy von Neumann architectures. Instead, we introduce a custom architecture natively tailored to the constraints of FHE, specifically optimizing for the deterministic nature of oblivious computation. By treating the program as a stream of encrypted data and aligning the memory and logic units with the deterministic nature of oblivious execution, HEGIDE maximizes the efficiency of the underlying homomorphic primitives.

1.1 Related Work

Our work lies at the intersection of several research areas: the implementation of an FHE virtual processor, the specification of oblivious reads and writes in FHE (which we will refer to as FHE-RAM), and the evaluation of private functions.

Research regarding FHE virtual processors involves designing a general-purpose processor capable of executing an encrypted instruction set on encrypted data. A pioneering study by Brenner et al. [BPS12] utilized the Smart-Vercauteren scheme to implement a minimal instruction set. However, their model required fetching instructions from encrypted memory with linear access complexity, introducing massive overhead. Subsequent works explored Ultimate Reduced Instruction Set Computers (URISC) [TM13, TM14, CA19]. While theoretically Turing complete in the clear, in the encrypted setting these designs require client interaction to detect termination. Moreover, the drastic increase in instruction count rendered them impractical. More recently, Trama et al. [TBC+25] and

Kim [Kim25] leveraged functional bootstrapping in TFHE [CJP21, CBSZ23, KS25] and CKKS [BKSS24, AKP25] to implement 8-bit and 64-bit Arithmetic Logic Units (ALUs), respectively. This line of research provides a viable compilation target for standard Instruction Set Architectures (ISAs), offering an alternative to converting programs into Boolean or arithmetic circuits [CDS15, GSP+21, CLOT21].

In 2024, Azogagh et al. [ADK24] presented the first non-interactive Oblivious Turing Machine (OTM) leveraging TFHE functional bootstrapping. Their construction encrypts the tape as an RLWE ciphertext and manages transitions via blind rotation. However, this TFHE-based OTM suffers from a critical limitation: the tape size is constrained by the polynomial degree N. With typical parameters ($N = 1024$ or 2048) and required redundancy, the effective tape size is reduced to a range of 16 to 32 values, severely limiting expressiveness.

In 2025, Delfour and Killijian [DK25] proposed OUF to circumvent the high round complexity of Azogagh et al.'s OTM. OUF relies on function descriptions (encrypted LUTs) and a dynamic tape updated homomorphically. A typical round involves: (1) extracting inputs via blind rotation, (2) executing all encrypted instructions using Blind Matrix Access (BMA) [ADK24], (3) aggregating results into a single ciphertext via functional keyswitch, and (4) writing the output back to the tape via functional bootstrapping. While OUF improves upon OTM, it retains the same constraints on the tape's size.

A recent library `FHE-RAM` [BM25a] provides oblivious random-access memory with logarithmic read complexity. However, they face a write bottleneck; to hide the access pattern, every write requires a "Read-Before-Write" protocol that traverses a homomorphic tree, making writes significantly more expensive than reads (e.g., 871 ms vs 302 ms for a 1MB heap). The `Phantom` project [BM25b] (pre-alpha release) represents the state-of-the-art in encrypted execution, implementing a fully encrypted RISC-V virtual machine. While `Phantom` offers impressive compatibility with standard binaries, its architecture is based on the traditional von Neumann model, including an encrypted Program Counter (PC). We argue that emulating a dynamic PC under FHE introduces architectural overhead that is redundant for private execution, as oblivious programs must follow a deterministic execution path independent of the input data. We also note the Juliet processor [GMT24], which offers a configurable FHE-based design; however, Juliet evaluates its control flow in plaintext, making it unsuitable for PFE where the program logic must remain hidden.

Distinctions from Existing Designs We distinguish HEGIDE from recent advancements in discrete CKKS integer arithmetic, such as the Homomorphic Integer Computer [Kim25], which demonstrates 64-bit ALU operations over CKKS but does not propose a virtual processor architecture, a memory model, or a compiler framework. HEGIDE builds upon such foundational ALU designs but contributes a complete, general-purpose MIMD oblivious processor with a novel memory architecture and compilation stack.
Earlier FHE processor designs, such as FURISC [CA19], VSP [MBM+21], or `Phantom` [BM25b] emulate legacy von Neumann pipeline stages with an encrypted Program Counter. This imposes performance overhead because the entire pipeline circuit (instruction fetch, decode, and execution) must be fully evaluated every cycle regardless of the actual instruction being executed. Crucially, VSP requires client interaction via its *Resumption* protocol: after each evaluation batch, the client must decrypt a termination flag to determine whether the program has finished, then re-generate clock cycles for the next batch if execution must continue. This fundamentally prevents VSP from achieving fully non-interactive PFE. Beyond interactivity, VSP also suffers from expressiveness limitations: its CAHPv3 ISA provides only a 16-bit datapath with sixteen 16-bit registers, and the implementation is restricted to 512 bytes of ROM and 512 bytes of RAM (at most 256 addressable 16-bit words). This makes the execution of programs of non-trivial size or complexity impractical. FURISC [CA19], as a One Instruction Set Computer (OISC) based on the SBN instruction,

shares the same interactivity limitation noted above for URISCs. When configured with a pre-specified maximum loop count (its only non-interactive mode), it effectively operates as an oblivious decider with a fixed cycle budget, just as HEGIDE does, but with a drastic instruction count inflation: since SBN can only subtract and conditionally branch, simple operations like comparison or data movement require dozens of SBN instructions, rendering real programs impractical (cf. Section 6 for a quantitative comparison).
The relationship between our memory model and standard Oblivious RAM (ORAM) and Private Information Retrieval (PIR) protocols is discussed in Section 3.

1.2 Our Contributions

1. HEGIDE: An Oblivious MIMD CKKS Processor We present HEGIDE (*Homomorphic Encryption-based General-purpose archItecture for Delegated private function Evaluation*), an encrypted processor based on CKKS that enables the non-interactive, fully oblivious evaluation of complex programs. Unlike approximate arithmetic CKKS approaches that can suffer from error accumulation, HEGIDE leverages discrete CKKS functional bootstrapping [BKSS24, AKP25, DAP+26] to perform systematic noise cleaning, ensuring correctness for unbounded computations. Crucially, we exploit the SIMD nature of CKKS to enable Multiple Instruction, Multiple Data (MIMD) execution. By encoding different instructions and addresses in the slots of CKKS ciphertexts, HEGIDE can evaluate entirely different program threads in parallel, maximizing throughput.

2. OSReM: Depth-Optimized Read, Constant-Time Write Memory We introduce OSReM (*Oblivious Shift Register Memory*), a novel memory architecture designed to overcome the linear write bottleneck of standard FHE-RAM. OSReM relies on two key mechanisms:

1. *Vectorized Oblivious Read*: We implement OREAD, a parallelized oblivious read procedure. By leveraging CKKS vectorization, it retrieves distinct values for each slot with a multiplicative depth of just 1, minimizing noise consumption.

2. *Stream Write*: Instead of performing expensive oblivious writes to arbitrary addresses, OSReM operates as a shift register. A new value is appended by shifting the entire memory state. This operation is oblivious by design (constant address) and incurs negligible overhead, avoiding the prohibitive cost of periodic full-memory bootstrapping that would be required by CKKS FHE-RAM designs.

3. Fully Oblivious ALU We design a comprehensive instruction set handling arbitrary word sizes, building on the arithmetic foundation of [Kim25]. We extend this base with novel algorithms for fully oblivious bitwise operations and shifts with encrypted shift amounts, utilizing binary decomposition and barrel shifters. The ALU executes all possible instructions in parallel and uses OREAD to obliviously select the correct result. This approach completely hides the instruction sequence from the server.

4. Implementation and Compiler We provide a proof-of-concept implementation of HEGIDE using the OpenFHE library [BAB+22]. To facilitate programmability, we provide a compiler that translates a subset of Python directly into HEGIDE machine code. The compiler performs liveness analysis and static scheduling to manage the OSReM shift constraints, abstracting the memory complexity for the user. Experimental results demonstrate the efficiency of our approach, which is two orders of magnitude faster in throughput than comparable approaches, ranging from 6.4 ms to 27.8 ms, compared to 1.26 s for `Phantom` [BM25b].

1.3 Technical Overview

To make HEGIDE usable for real-world algorithms, we abstract the cryptographic constraints. First, we support arbitrary word sizes (e.g., 16, 32, or 64-bit integers) via digit decomposition, using functional bootstrapping to clean noise and handle carry propagation. Second, we provide a compilation stack that accepts a subset of Python. The compiler handles the flattening of conditional branches and static memory scheduling for OSReM, allowing developers to focus on logic rather than cryptographic circuit design.

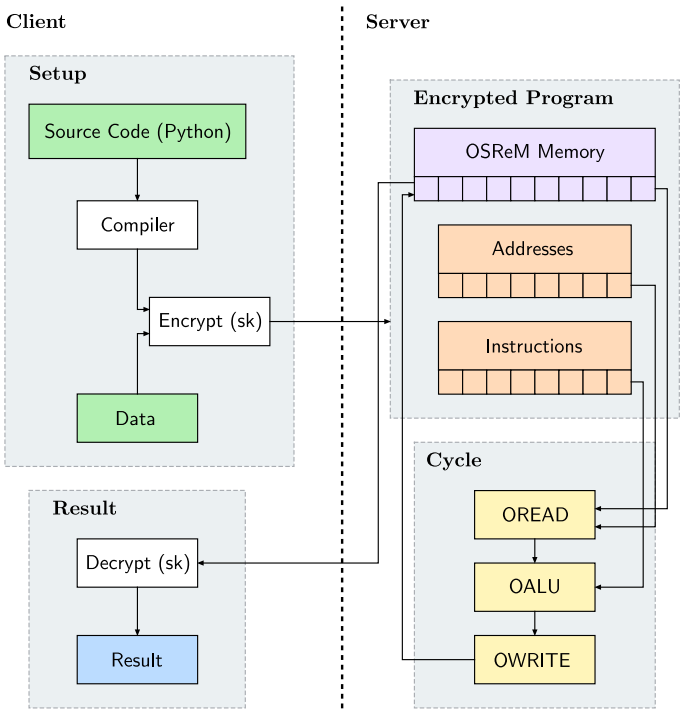

Figure 1: High-level overview of the HEGIDE workflow. The client compiles the Python source code into machine code, which is encrypted along with the data. The server executes the encrypted program on the encrypted data obliviously and returns the result. The client decrypts the final result.

The "All-Data" Paradigm By definition, FHE protects data, not functions. To protect the program logic, HEGIDE treats the function *as* data. Instead of evaluating a circuit specific to a client's algorithm, the server executes a public, fixed processor circuit. The proprietary algorithm is compiled into a stream of encrypted machine code (opcodes and addresses). The server executes this stream blindly; because of the semantic security of CKKS, it cannot distinguish between an Add and a Mul, or a read from Address A versus a read from Address B.

The Oblivious Decider Model A fundamental constraint of FHE is that the server cannot branch based on encrypted values. Consequently, the execution trace must be entirely independent of the input data. This reality forces all secure programs to be transformed into *Oblivious Deciders*: programs that execute a deterministic, linear sequence of instructions where loops are unrolled to worst-case bounds and conditional branches are "flattened" (evaluating all paths and selecting the result obliviously).

In this model, the program counter used by architectures like `Phantom` becomes an architectural mismatch. Since the instruction sequence is deterministic and known at compile time, maintaining an encrypted PC to find the next instruction adds unnecessary complexity. HEGIDE embraces this by discarding the PC entirely, performing static analysis at compile time, and using the SReM memory architecture to fit the oblivious decider model.

Efficient Oblivious Write with SReM Vectorized oblivious FHE-RAM for leveled FHE schemes (B/FV, BGV, and CKKS) incurs a linear overhead for every write, as writing consumes depth that must be restored via bootstrapping.[1] However, because HEGIDE operates on oblivious deciders, the memory access pattern is statically known. We exploit this predictability through (O)SReM (*(Oblivious) Shift Register Memory*). Instead of performing expensive random-access writes, the processor treats memory as a shift register where every write is a deterministic append to the tail. Since the write index is fixed and public, the operation requires no homomorphic selection logic, making writes virtually "free" in terms of multiplicative depth, noise, and computation time. The compiler uses static liveness analysis to ensure data is recirculated or ejected at the correct cycle, trading memory provision for the elimination of the write bottleneck.

MIMD Vectorization A single FHE thread is inherently slow due to cryptographic overhead. However, the underlying CKKS scheme supports vectorization, packing N slots (typically 2^{15} or 2^{16}) into a single ciphertext. Standard approaches utilize this for Single Instruction, Multiple Data (SIMD) operations. HEGIDE transforms this capability into a Multiple Instruction, Multiple Data (MIMD) architecture. Since the instruction stream is encrypted data, we can pack distinct opcodes into different slots of the instruction ciphertext. For example, slot 0 can execute an Add on its data, while slot 1 executes a Mul, and slot 2 performs a left shift Shl. This allows the server to execute thousands of independent program threads, or parallelize a massive data-parallel workload simultaneously, yielding a high amortized throughput despite the latency of individual operations. Furthermore, thanks to the semantic security of CKKS, the encrypted program bytecode can be safely cached by the server and reused; the client only needs to upload a new encrypted memory state to execute the same algorithm on new inputs.

2 Preliminaries

2.1 Notation

For $N > 1$ a power of two, we denote by $\mathcal{R}$ the polynomial ring $\mathbb{Z}[X]/(X^N + 1)$. For an integer $q > 1$, $\mathcal{R}_q$ denotes the quotient ring $\mathcal{R}/q\mathcal{R} = \mathbb{Z}_q[X](X^N + 1)$. All logarithms are expressed in base 2 unless explicitly mentioned. We denote vectors using bold letters (e.g., $\mathbf{v}$). For $x \in \mathbb{R}$, $\lfloor x \rceil$ denotes the nearest integer to x. For an integer x, we denote by $[x]_q$ the centred reduction of x modulo q, uniquely represented in the interval $(-q/2, q/2]$. For polynomials and vectors, the round and modular reduction operations are applied coefficient-wise. We use $x \leftarrow S$ to denote uniform sampling from a finite set S, and $x \leftarrow \chi$ to denote sampling x according to a probability distribution χ. Finally, we denote the

[1] If we remove the vectorization constraint, memory could be mapped to the slots of a single ciphertext, enabling access via conditional homomorphic rotations, with a logarithmic complexity. However, this design sacrifices MIMD parallelism, as slots are consumed for storage rather than independent execution threads. Since the ALU cost is dominated by full-ciphertext functional bootstrapping, this approach would yield the same single-thread latency while reducing the amortized throughput by a factor of N.

Kronecker delta for $i, j \in \mathbb{N}$ by:

$$\delta_{i,j} := \begin{cases} 1 & \text{if } i = j, \\ 0 & \text{if } i \neq j. \end{cases}$$

2.2 Hybrid RLWE-CKKS scheme

We present the hybrid RLWE-CKKS FHE scheme from [AKP25] for discrete CKKS functional bootstrapping. Since discrete CKKS operates with integer plaintext, we utilize RLWE ciphertexts for exact encryption and decryption.[2]

Standard CKKS operations are used for carrying out most computations, with the exception of the digit decomposition procedure presented in [AKP25]. To avoid confusion between parameters, we denote RLWE moduli without an apostrophe (e.g., q) and CKKS moduli with an apostrophe (e.g., q'_0).

Symmetric RLWE We denote the RLWE scheme by RLWE. It is parametrized by a plaintext modulus p, a ciphertext modulus q and error/key distributions $\chi_{\mathsf{err}}, \chi_{\mathsf{key}}$ over $\mathcal{R}$. The scheme RLWE is defined by the following operations:

$$\begin{aligned} \mathsf{RLWE.Encrypt}(m, \mathsf{sk}) &= (-a \cdot \mathsf{sk} + (q/p)m + e, a) && \text{for } a \leftarrow \mathcal{R}, \\ & && \quad e \leftarrow \chi_{\mathsf{err}} \\ \mathsf{RLWE.Decrypt}(\mathsf{ct}, \mathsf{sk}) &= \lfloor (p/q) \cdot (b + a \cdot \mathsf{sk}) \rceil && \text{for } \mathsf{ct} = (b, a) \end{aligned}$$

RNS-CKKS We use the RNS variant of the CKKS scheme [CHK+18b], which we denote by CKKS. For a power of two scaling factor Δ and a multiplicative depth $L \in \mathbb{N}$, we define a moduli chain $Q'_\ell = \prod_{i=0}^{\ell} q'_i$ for levels $\ell \in [\![0, L]\!]$. The q'_i are prime, with $q'_0 > \Delta$, and for all $\ell \in [\![1, L]\!]$, $Q'_\ell / Q'_{\ell-1} \approx \Delta$.

CKKS encrypts complex vectors $\mathbf{z} \in \mathbb{C}^{N/2}$ by encoding them into a polynomial plaintext $\mathsf{pt} \in \mathcal{R}$ via the canonical embedding τ. The canonical embedding $\tau : \mathbb{R}[X]/(X^N + 1) \to \mathbb{C}^{N/2}$ is the isomorphism mapping a polynomial $P(X)$ to the vector of its evaluation at the primitive $2N$-th roots of unity modulo the cyclotomic polynomial. We refer to elements in the polynomial ring as being in coefficient representation, and their image under τ as being in slot representation. Thanks to the isomorphism, addition and multiplication of polynomials correspond to component-wise addition and multiplication of the complex vectors in the slots.

Let χ_{pkenc} be the public key distribution on $\mathcal{R}$. We define the CKKS scheme as:

$\mathsf{CKKS.Setup}(1^\lambda) : \mathsf{params} = \{N, \{Q'_\ell\}_\ell, \Delta, \chi_{\mathsf{key}}, \chi_{\mathsf{err}}, \chi_{\mathsf{pkenc}}\}$

$\mathsf{CKKS.KeyGen}(\mathsf{params}) : \mathsf{sk} \leftarrow \chi_{\mathsf{key}},\ a \leftarrow \mathcal{R}_{Q'_L},\ e \leftarrow \chi_{\mathsf{err}}, \mathsf{pk} \leftarrow (-a \cdot \mathsf{sk} + e, a)$

$\mathsf{CKKS.Encode}(\mathbf{z}) = \lfloor \tau^{-1}(\Delta \mathbf{z}) \rceil \in \mathcal{R}$

$\mathsf{CKKS.Encrypt}(\mathsf{pt}, \mathsf{pk}) = [u \cdot \mathsf{pk} + (\mathsf{pt} + e_0, e_1)]_{Q'_L} \in \mathcal{R}^2_{Q'_L}$ for $u \leftarrow \chi_{\mathsf{pkenc}}, e_0, e_1 \leftarrow \chi_{\mathsf{err}}$

$\mathsf{CKKS.Decrypt}(\mathsf{ct}, \mathsf{sk}) = b + a \cdot \mathsf{sk} \in \mathcal{R}$

$\mathsf{CKKS.Decode}(\mathsf{pt}) = \tau(\mathsf{pt}/\Delta) \in \mathbb{C}^{N/2}$

Note that we will treat CKKS ciphertexts as encrypting vectors of N real elements to match the dimension of the RLWE ciphertexts. Implementation-wise, this is achieved by utilizing both the real and imaginary parts of the $N/2$ complex slots, which may require separating these components into distinct ciphertexts during computation. For clarity, we omit these implementation details and describe our algorithms as operations directly on vectors of N real elements.

[2] Note that those RLWE ciphertexts can be seen as coefficient-encoded BFV ciphertexts.

2.3 Functional Bootstrapping

While the original CKKS bootstrapping [CHK+18a] serves primarily to refresh multiplicative depth, recent advancements [BKSS24, AKP25] introduce functional bootstrapping. This paradigm extends the standard procedure to evaluate an arbitrary function $f : \mathbb{Z}_p \to \mathbb{Z}_p$ on encrypted data. Crucially, the input CKKS ciphertext does not encrypt a message $\mathbf{m} \in (\mathbb{Z}_p + i\mathbb{Z}_p)^{N/2} \subset \mathbb{C}^{N/2}$ exactly, rather, it encrypts an approximation $\mu = \mathbf{m} + \varepsilon \in \mathbb{C}^{N/2}$. Provided the error is small (see the analysis in [AKP25]), this error is reduced during functional bootstrapping and allows for the unique determination of the underlying message.

We adopt the framework of [AKP25], aiming to transform an input RLWE ciphertext encrypting (exact) messages in $\mathbb{Z}_p^N$ into a bootstrapped RLWE ciphertext encrypting their evaluation under f. The core mechanism relies on a polynomial $T_f \in \mathbb{C}[X]$ constructed via first-order[3] Hermite interpolation, which satisfies the following conditions for all $k \in [\![0, p-1]\!]$:

$$\mathrm{Re}(T_f(\zeta_p^k)) = f(k) \qquad \mathrm{Re}(T_f'(\zeta_p^k)) = 0$$

where $\zeta_p = \exp(2i\pi/p)$. The derivative condition ensures that small errors in the input do not propagate to the output, effectively "snapping" the noise value closer to the grid point. The functional bootstrapping procedure is as follows:

RLWE-to-CKKS conversion. Given an input RLWE ciphertext ct with modulus q encrypting $m(X) \in \mathcal{R}_p$, we start by switching the modulus to CKKS base modulus q_0', and scale the result by Δ/q_0' to obtain a CKKS ciphertext ct' encoding the message $\Delta m(X)/p$. We then raise the modulus to Q_L', changing the message to $\Delta m(X)/p + q_0' I(X)$, where I captures the inherent modulo-q_0' overflows. Finally, we perform the homomorphic encoding procedure $\mathsf{CoeffsToSlots}$, putting each coefficient t_i of $t(X)$ in a different slot.

Homomorphic Evaluation. We evaluate a Chebyshev interpolation of the (scaled) complex exponential map to obtain $\mathsf{ct}'_{\exp}$ with $\hat{m}_i \approx \exp(2i\pi t_i) = \exp(2i\pi m_i/p) = \zeta_p^{m_i}$ in its slots. We then evaluate the Hermite interpolation T_f on $\mathsf{ct}'_{\exp}$ and extract the real part, yielding $\tilde{m}_i \approx f(m_i)$ in the slots.

CKKS-to-RLWE conversion. We switch back to coefficient encoding with $\mathsf{SlotsToCoeffs}$ and scale the result by $Q'/(\Delta p)$ for Q' the CKKS modulus after bootstrapping. We can then perform a final modulus switch from Q' to q to obtain an RLWE ciphertext containing a valid encryption of $f(m(X))$ (for f applied coefficient-wise).

Note that the final CKKS-to-RLWE conversion can be delayed to perform additional computations in CKKS. When we need to evaluate different LUTs on the same ciphertext, we will use the MVB algorithm [AKP25, DAP+26].

Failure Probability. The modular reduction step assumes the message (plus overflow) falls within a specific interval. Exceeding this bound causes a wrapping failure. We mitigate this issue using the sparse-secret encapsulation technique [BTH22], which reduces the failure probability to negligible levels ($< 2^{-128}$) for all used parameter sets in this paper, without incurring a performance penalty.

2.4 Threat Model

Our goal is to protect both the input and output data, and the program logic. We assume an *honest-but-curious* server that performs the computation correctly, but attempts to infer information from the encrypted data structures and memory access patterns. We

[3]Higher order interpolation can also be used for better precision.

explicitly note that this assumption is required only for the *correctness* of the computation. Regarding *privacy*, even a malicious server who deviates from the protocol (e.g., by skipping cycles or altering instructions) cannot learn any information about the input data or the program logic, as all operations are performed on IND-CPA secure ciphertexts.

We assume a single-key setting where the program and data are encrypted under the same key. A typical use case is a financial institution that wishes to outsource the execution of a proprietary trading strategy (the program) on sensitive market data (the input) to a powerful cloud provider. The institution owns both the intellectual property (the algorithm) and the sensitive data. By encrypting both under its private key, it can utilize the cloud's computational resources while ensuring the provider learns absolutely nothing about the trading logic or the financial positions. Note that scenarios involving distinct program and data owners (e.g., a hospital running a third-party diagnostic model) can also be reduced to this model via a trusted setup or distributed key generation.

We aim to achieve *full obliviousness*. Formally, this implies that for any two programs $\mathcal{P}_1, \mathcal{P}_2$ and any two inputs x_1, x_2, provided that both programs compile to the same number of processor cycles (runtime) T, utilize the same memory size M, word size W and instruction set $\mathbb{I}$, the server's view during execution of $\mathsf{Encrypt}(\mathcal{P}_1)$ on $\mathsf{Encrypt}(x_1)$ must be computationally indistinguishable from the execution of $\mathsf{Encrypt}(\mathcal{P}_2)$ on $\mathsf{Encrypt}(x_2)$.[4]

Our security model allows for the leakage of four parameters:

1. *Runtime* T: The server knows the total number of cycles executed.
2. *Memory* M: The server learns the total size of the allocated memory.
3. *Word size* W: The server learns the word size of the virtual processor.
4. *Instruction Set*: The server learns the instruction set of the virtual processor.

We explicitly note that this model protects the *computation process*, not the *computation result*. We do not claim protection against leakage derived from the decrypted output, i.e., differential privacy or circuit privacy guarantees.

Chosen-Ciphertext Attacks (CCA) and Strict No-Feedback Policy. While the computation process relies on IND-CPA security, in standard deployments, a malicious server observing whether a final decryption succeeds or fails could potentially act as a Chosen-Ciphertext Attack (CCA) oracle. To formally guarantee privacy against such active adversaries and maintain the security claims established above, our threat model enforces a strict no-feedback policy. The client must operate strictly offline and provide no feedback to the server on decryption success or failure, or on the validity of the returned result. By ensuring the user's behavior is predefined and independent of the decrypted outcome, we sever the feedback loop required to mount a CCA attack.

3 OSReM: A Shift-Register Memory Model

We established in Subsection 1.3 that we place ourselves in the oblivious deciders model. While this is a strong constraint on the class of executable programs, it provides a notable advantage: we can statically determine the entire sequence of memory accesses at compile time. Standard oblivious architectures fail to exploit this predictability: they usually rely on a Random Access Memory model, where the program can read and write at any address at any cycle.

In the context of vectorized oblivious FHE-RAM for leveled FHE schemes, this RAM write is a known bottleneck: writing to an encrypted index is inherently expensive ($O(M)$)

[4] We can pad the programs so that the number of processor cycle, memory and word size are the same for both. Also, we could set $\mathbb{I} = \mathbb{I}_1 \cup \mathbb{I}_2$.

and consumes depth for conditional selection. This requires frequent and prohibitively expensive full-memory bootstrapping.

Relationship with ORAM and PIR It is important to distinguish our setting from both Oblivious RAM (ORAM) and Private Information Retrieval (PIR), as the three address fundamentally different problems despite superficial similarities. ORAM protocols (e.g., Path ORAM [SvDS+13], Circuit ORAM [WCS15]) operate at the *physical address* level: the server can observe which memory cells are accessed, and ORAM hides the access pattern by shuffling and re-encrypting data after each access, incurring $O(\log N)$ amortized overhead per operation. In our FHE setting, the problem is different: the memory *indices themselves* are encrypted, so the server already cannot observe which address is being accessed. The $O(M)$ cost of our oblivious read is not a pattern-hiding overhead but a consequence of computing on encrypted indices: the server must evaluate a weighted sum over all memory positions because it cannot determine which position is selected. ORAM techniques do not help here, since they assume the server can route requests to specific physical cells, which is precisely what FHE encryption prevents.
A single oblivious read is structurally similar to a Private Information Retrieval (PIR) [CGKS95, KO97] query: in both cases, a value is retrieved from a database without revealing the query index. However, PIR is inherently a *read-only* primitive, designed for a client-server model where the client performs computation between queries. A processor requires interleaved reads and writes every cycle, executed entirely by the server with no client interaction, making the PIR abstraction insufficient.
Recent FHE-RAM constructions [BM25a] address both reads and writes on encrypted data, achieving logarithmic-depth reads via homomorphic tree structures. However, they still face a write bottleneck ($O(M)$ depth) because they must support arbitrary, dynamic write patterns: each write requires a "Read-Before-Write" traversal of the tree to preserve the access-pattern hiding guarantees of the structure.

We argue that for oblivious deciders, where the full access schedule is known at compile time, this cost is unnecessary. Since the lifespan of every variable is known, we can replace the general-purpose RAM model with a shift register that aligns with the temporal locality of data. We introduce SReM (Shift-Register Memory), a memory model designed to decouple the cost of reading from the cost of writing.

SReM Model Unlike a standard RAM where data resides at fixed indices, SReM treats memory as a dynamic, shifting stream of size M.

Random Access Read. The program can read a value from any index $i \in [\![0, M-1]\!]$ in the current state. This preserves the flexibility required for complex algorithms.

Stream Write. The write operation is fixed. The whole memory is shifted by one position, discarding the head (element at index 0), and the new value is written at the tail (element at index $M-1$).

Definition 1 (SReM Primitives)**.** Let $\mathbf{s}_t = (s_{t,0}, \ldots, s_{t,M-1})$ denote the memory state at cycle t. The three SReM primitives are:

Init(v): Given an initial data vector $\mathbf{v} = (v_0, \ldots, v_{M-1})$, set $\mathbf{s}_0 \leftarrow \mathbf{v}$.

Read(i): For $i \in [\![0, M-1]\!]$, return $s_{t,i}$.

Write(v): Shift the state and append v to the tail: $\mathbf{s}_{t+1} \leftarrow (s_{t,1}, s_{t,2}, \ldots, s_{t,M-1}, v)$. The value $s_{t,0}$ at the head is discarded.

In SReM, the write becomes a simple pointer update. It requires zero homomorphic operations and, therefore, does not consume multiplicative depth or add noise. However,

we lose some flexibility compared to RAM. We quantify this loss in the next section and demonstrate how to overcome it.

3.1 Comparison of SReM and RAM

Let M be the capacity of the SReM memory state. At any processor cycle t, the memory state is a vector $\mathbf{s}_t = (s_0, \ldots, s_{M-1})$, where s_0 is the head (the element scheduled for eviction) and the new value is written to the tail at position $M-1$.

Theoretical analysis Let $L_t \subseteq [\![0, M-1]\!]$ be the set of indices corresponding to live variables, that is, variables required by the program at some future cycle $t' > t$. We define the *load factor* $\alpha_t = |L_t|/M$.

A fundamental constraint of SReM is that we can only write one value per cycle. This creates a scheduling conflict at the head s_0, depicted in Figure 2:

1. *Maintenance cycle*: If $0 \in L_t$ (head s_0 is live), it cannot be discarded. The processor must execute a Cpy instruction, reading s_0 and re-appending it back to the tail. The memory state rotates, but no useful computation occurs.
2. *Computation cycle*: If $0 \notin L_t$ (head s_0 is dead), the processor can execute a useful instruction and write the new result to the tail, discarding the dead value s_0.

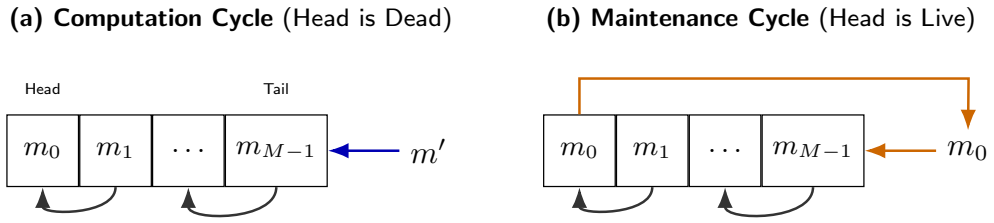

Figure 2: SReM Scheduling States. (a) Computation Cycle: The variable at the head is dead and discarded; a new result m' enters the tail. (b) Maintenance Cycle: The head variable is live; it is recirculated to the tail to prevent data loss.

We now prove that we can deterministically bound the number of maintenance cycles by memory provisioning.

Definition 2 (Memory Overhead Factor)**.** For a program requiring a maximum of K simultaneously live variables, and an SReM capacity M, we define the memory overhead factor as $\gamma = M/K$.

Definition 3 (End-Packed Program)**.** A program is *end-packed* if, at initialization, all live variables are stored at the end of the memory state vector (indices $M-K$ to $M-1$), leaving the first elements of the vector dead.

Lemma 1. *Let $\mathcal{P}$ be an end-packed program of size T with a memory overhead factor of $\gamma \geq 1$. Between any processor cycle t and $t+M$, the proportion of maintenance cycles (live head) is at most $1/\gamma$.*

Proof. Since the shift register only puts new variables at its tail, between any processor cycle t and $t+M$, the head of the register exactly runs through the variables stored at cycle t. Thus, exactly $|L_t|$ cycles are performed on live variables. Since the memory overhead factor is γ, $\gamma|L_t| \leq M$. Therefore, the proportion of maintenance cycles is at most $|L_t|/M \leq 1/\gamma$. □

Theorem 1 (Deterministic overhead)**.** *Let $\mathcal{P}$ be an end-packed program of size T with a memory overhead factor $\gamma \geq 1$. Then, the proportion of maintenance cycles is at most $1/\gamma$.*

Proof. We consider two cases for T:

- *Case 1*: $T = k \cdot M + t$ with $t \in [\![0, (1-1/\gamma)M]\!]$ and $k \in \mathbb{N}$. Since $\mathcal{P}$ is end-packed, the first t cycles are computation cycles, performed on dead variables. Then, applying Lemma 1 k times shows that between cycle t and $t + k \cdot M$, the proportion of maintenance cycles is at most $1/\gamma$.
- *Case 2*: $T = (1 - 1/\gamma)M + k \cdot M + t$ with $t \in [\![0, M/\gamma]\!]$ and $k \in \mathbb{N}$. Since $\mathcal{P}$ is end-packed, the first $(1 - 1/\gamma)M$ cycles are computation cycles, performed on dead variables. Since $t \leq M/\gamma$, it follows that at most $1/\gamma$ of the operations are performed on live variables during the first $(1 - 1/\gamma)M + t$ cycles. Then applying Lemma 1 k times concludes the proof.

□

By Theorem 1 and by delegating some static analysis to the compiler, we effectively trade storage space and proportional read latency for "free" writes, eliminating the primary bottleneck of oblivious write in FHE.

3.2 Oblivious SReM in CKKS

We now instantiate an oblivious SReM (OSReM) in CKKS for HEGIDE. Let M be the size of the memory state, that is the number of addresses, and ℓ_m the CKKS level of the memory state.

3.2.1 Oblivious Read

For a given index $x \in [\![0, M-1]\!]$, we perform an oblivious read as in Algorithm 1.

Algorithm 1: CKKS Oblivious Read: OREAD

Input: Address indicators $\{\mathsf{ct}\delta_{x,i}\}_{i=0}^{M-1} \in (\mathcal{R}^2_{Q'_{\ell_m}})^M$, Memory State $\mathbf{Mem} \in (\mathcal{R}^2_{Q'_{\ell_m}})^M$

Output: Target Data $\mathsf{REG} \in \mathcal{R}^2_{Q'_{\ell_m - 1}}$

$\mathsf{REG} \leftarrow \sum_{i=0}^{M-1} \mathsf{Mul}(\mathsf{ct}\delta_{x,i}, \mathbf{Mem}[i])$;

return REG;

Formal Encoding of Indicator and Memory Ciphertexts To support parallel execution threads, CKKS natively utilizes SIMD operations via slot packing. Let N be the ring dimension, providing N available plaintext slots. Let $\mathbf{x} = (x_0, \ldots, x_{N-1}) \in [\![0, M-1]\!]^N$ be the vector of target addresses for each execution thread. We encode these addresses into M distinct indicator ciphertexts $\mathsf{ct}\delta_{\mathbf{x},i}$ for $i \in [\![0, M-1]\!]$. For a given memory index i, the plaintext vector $\mathbf{v}_i \in \mathbb{R}^N$ is constructed such that its j-th slot is $\mathbf{v}_i[j] = \delta_{x_j,i}$, and encrypted as $\mathsf{ct}\delta_{\mathbf{x},i} \leftarrow \mathsf{CKKS.Encrypt}(\mathsf{CKKS.Encode}(\mathbf{v}_i))$. Similarly, the encrypted memory $\mathbf{Mem} = \{\mathsf{MEM}_0, \ldots, \mathsf{MEM}_{M-1}\}$ is constructed "column-wise". The ciphertext MEM_i encrypts the values at address i, encoding the vector $\mathbf{m}_i = (m_{i,0}, \ldots, m_{i,N-1})$. During the OREAD operation, the indicator ciphertexts act as one-hot selectors in a homomorphic weighted sum across this encrypted memory.

Correctness and complexity The correctness follows directly from the definition of the indicator values: for a valid address x, exactly one indicator $\mathsf{ct}\delta_{x,i}$ encrypts the non-zero selector, where $i = x$, while others encrypt 0. The weighted sum therefore collapses to just $\mathbf{Mem}[x]$. The algorithm requires M ciphertext-ciphertext multiplications and $M-1$ additions, and the multiplicative depth is 1. Note that we use lazy relinearization, that is, perform the relinearization after the weighted sum, saving $M-1$ costly relinearizations.

Noise analysis The algorithm performs homomorphic multiplications between the binary indicators and the memory state, followed by a summation of M terms. Assuming the input noise is bounded by $\varepsilon_{\mathsf{in}}$, the worst-case (resp. average-case) noise growth after multiplication and summation is

$$|\varepsilon_{\mathsf{out}}| = O(M) \cdot |\varepsilon_{\mathsf{in}}| \qquad (\text{resp. } |\varepsilon_{\mathsf{out}}| = O(\sqrt{M}) \cdot |\varepsilon_{\mathsf{in}}|).$$

Since we target moderate memory sizes and will perform functional bootstrapping at each cycle on the read value, this noise growth is manageable.

MIMD Parallelism In the context of oblivious read, the SIMD capability detailed above directly translates to MIMD memory access. This allows for vectorized addressing, where every slot in the ciphertext vector accesses a distinct memory state at an independent address in a single operation (see Figure 3). Because the provided indicators select the target memory address for each slot independently, the weighted sum operation is performed independently in each slot. Specifically, slot k computes:

$$\sum_{i=0}^{M-1} \delta_{x_k,i} \cdot m_{i,k} = m_{x_k,k}$$

which is exactly a read at address x_k in the k-th memory state.

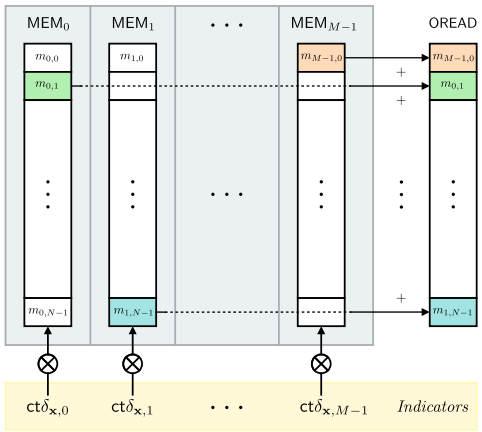

Figure 3: Visualization of the MIMD memory access. The memory consists of M ciphertexts, where slot j represents an independent execution thread. The OREAD operation computes a weighted sum of the memory columns using packed indicator ciphertexts $\mathsf{ct}\delta_{\mathbf{x},i}$. In this example, the indicators select $m_{0,1}$ from MEM_0, $m_{1,N-1}$ from MEM_1, and $m_{M-1,0}$ from MEM_{M-1}, aggregating them into the final result ciphertext.

Obliviousness We formally state the security guarantee of OREAD.

Theorem 2 (Obliviousness of OREAD)**.** *For any encrypted memory state* $\mathbf{Mem} \in (\mathcal{R}^2_{Q'_\ell})^M$ *and two distinct vector of addresses* $\mathbf{x}_1, \mathbf{x}_2 \in [\![0, M-1]\!]^{N/2}$*, the view of the server generated by* $\mathsf{OREAD}(\{\mathsf{Encrypt}(ct\delta_{\mathbf{x}_1,j})\}_{j=0}^{M-1}, \mathbf{Mem})$ *is computationally indistinguishable from the view generated by* $\mathsf{OREAD}(\{\mathsf{Encrypt}(ct\delta_{\mathbf{x}_2,j})\}_{j=0}^{M-1}, \mathbf{Mem})$*, assuming the underlying CKKS scheme is IND-CPA secure.*

Proof. The security of OREAD follows strictly from the semantic security of CKKS. Indeed, OREAD computes a weighted sum of all memory ciphertexts where the weights are encrypted indicators (0 or 1). Since the server observes only the execution of the fixed multiplexer circuit and IND-CPA secure ciphertexts, any ability to distinguish access patterns would imply an ability to break the underlying encryption scheme. □

Note that this oblivious read algorithm requires the client to provide M encryption indicators, resulting in a communication and space complexity of $O(M)$ per read instruction. While this overhead is acceptable for compiled programs that are uploaded once and executed frequently (amortizing the bandwidth cost over many executions), it may be limiting in bandwidth-constrained settings. We address this in Appendix B by introducing a compressed read mechanism that trades computational complexity for up to $O(1)$ communication of a single RLWE ciphertext with a small ciphertext modulus.

3.2.2 Oblivious Write

The oblivious write in SReM is designed as a deterministic shift of the memory state, and a write at a determined position, the tail. This operation is inherently oblivious as the access pattern is fixed and independent of the data. No homomorphic operations are used, and its runtime is negligible compared to even a single homomorphic addition.

Algorithm 2: Oblivious Write: OWRITE

Input: Register $\mathsf{REG} \in \mathcal{R}^2_{Q'_{\ell_m}}$, Memory State $\mathbf{Mem} \in (\mathcal{R}^2_{Q'_{\ell_m}})^M$

Output: New Memory State $\mathbf{Mem} \in (\mathcal{R}^2_{Q'_{\ell_m}})^M$

```
for i ← 0 to M − 2 do
    Mem[i] ← Mem[i + 1];
Mem[M − 1] ← REG;
return Mem;
```

4 HEGIDE: An Oblivious MIMD CKKS Processor

Let W be the word size in bits for the HEGIDE processor, and M the size of its OSReM memory state. We denote by p and d the plaintext modulus and radix basis for the ALU, respectively, both of which are chosen to be powers of two for simplicity.

4.1 Oblivious Memory

Our architecture requires a memory capable of storing M logical words of bit-width W, that is M elements of $\mathbb{Z}_2w$. However, the underlying hybrid scheme operates on elements with a plaintext modulus p. To support arbitrary word sizes W, we employ a multi-bank strategy. We decompose each W-bit integer into $k = \lceil W/\log p \rceil$ digits in base p. Consequently, the

single logical memory is physically implemented as k parallel OSReM instances, where the i-th instance stores the i-th digit of every word.

The choice of p involves a trade-off: a larger p reduces the number of banks k (saving RAM and bandwidth) but increases the complexity of the ALU's digit decomposition step. To optimize runtime performance, we set $p = d$ in our default configuration, eliminating the need to decompose from base p to base d. Alternative configurations for bandwidth- or memory-constrained scenarios are discussed in Appendix B.

Since the memory is physically striped across k independent ciphertext vectors of size M, the OREAD and OWRITE operations are simply vectorized: the processor performs k reads (resp. writes) in parallel to retrieve (resp. update) a full W-bit word.

4.2 Oblivious ALU

The HEGIDE ALU executes all instructions in the ISA in parallel and then obliviously selects the correct result – the instruction to be executed – using OREAD. This design enables HEGIDE to leverage state-of-the-art instructions based on functional bootstrapping, as introduced by Kim [Kim25]. However, while this foundation efficiently handles standard arithmetic, it is insufficient for a general-purpose processor. Most notably, existing FHE implementations of bitwise shifts depend on cleartext shift amounts, violating the requirement to hide the operands.

For HEGIDE, we implement a set $\mathbb{I}$ of 11 instructions. To maximize efficiency, the ALU operates on two different representations of the operands simultaneously: a binary decomposition and a radix decomposition $d > 2$.

Bitwise Operations: Not, And, Or, Xor, Shl, Shr, Eq. Binary decomposition is the natural representation for logical operations. We evaluate Not, And, Or, and Xor using the bivariate polynomials $P_{\mathsf{Not}} = 1 - X$, $P_{\mathsf{And}} = XY$, $P_{\mathsf{Or}} = 1 - (1 - X)(1 - Y)$ and $P_{\mathsf{Xor}} = (X - Y)^2$. For the shift left Shl and shift right Shr instructions, we implement a fully oblivious barrel shifter. Unlike previous works that require cleartext shift amounts, HEGIDE utilizes the binary decomposition of the shift amount to perform a sequence of $\log W$ homomorphic multiplexers (MUX). This avoids the precision loss associated with multiplying or dividing by powers of two. The equality Eq instruction is performed by xoring the two operands, and computing the Or of the result.

Radix Operations: Add, Sub, Mul, Cpy. For arithmetic operations such as Add and Mul, which require carry-propagation logic and modular reduction via functional bootstrapping, choosing $d > 2$ offers higher *bit-throughput*. Specifically, the ratio of the execution time for a base-d functional bootstrapping over $\log d$ bits is smaller than the execution time of a binary functional bootstrapping, making the higher radix more efficient for multi-bit arithmetic. We adopt the design from Kim [Kim25] for Add, Sub, and Mul. Finally, the Cpy instruction acts as a unary identity, which the compiler uses for OSReM maintenance cycles, i.e., recirculating a live head.

A complete algorithmic description of the new opcodes is given in Appendix C.

ALU Pipeline The core of the ALU logic is defined in Algorithm 3.

Decomposition and Cleaning. The input registers are decomposed into base 2. This is done using multi-value functional bootstrapping, which evaluates the identity and bit extraction LUTs on $\mathbb{Z}_d$. This effectively cleans both the base d and base 2 digits.

Parallel Evaluation. The set $\mathbb{I}$ of all instructions is evaluated in parallel, using the radix or binary decomposition. This results in a vector of results, **ISAMem**, which can be

viewed as a small temporary memory containing every possible instruction outcome for the given operand.

Oblivious Selection. The final result is retrieved from **ISAMem** using OREAD with the encrypted instruction.

Algorithm 3: Oblivious ALU: OALU

Input: Instruction indicators $\mathbf{Instr} \in (\mathcal{R}^2_{Q'_{\ell_m+1}})^{|\mathbb{I}|}$, Registers $\mathsf{REG}_1, \mathsf{REG}_2 \in (\mathcal{R}^2_{Q'_{\ell_m-1}})^k$ where $k = \lceil W/\log d \rceil$

Output: Target Data $\mathsf{RES} \in (\mathcal{R}^2_{Q'_{\ell_m}})^k$

```
/* 1.Digit decomposition with functional bootstrapping for noise
   cleaning                                                      */
```

1 $\mathbf{digits}_1, \mathbf{bits}_1 \leftarrow \mathsf{MVB}(\mathsf{REG}_1, d, \{\mathrm{Id}, \mathrm{bit}_1, \ldots, \mathrm{bit}_{\log d}\})$;

2 $\mathbf{digits}_2, \mathbf{bits}_2 \leftarrow \mathsf{MVB}(\mathsf{REG}_2, d, \{\mathrm{Id}, \mathrm{bit}_1, \ldots, \mathrm{bit}_{\log d}\})$;

```
/* 2.  Parallel instruction execution                           */
```

3 $\mathbf{ISAMem} \leftarrow \emptyset$;

4 **foreach** $I \in \mathbb{I}$ **do**

5 $\quad \mathbf{ISAMem}.\mathsf{append}(I(\mathbf{bits}_1, \mathbf{bits}_2, \mathbf{digits}_1, \mathbf{digits}_2))$;

```
/* 3.  Result selection via OREAD                               */
```

6 $\mathsf{RES} \leftarrow \mathsf{OREAD}(\mathbf{Instr}, \mathbf{ISAMem})$;

7 **return** RES;

Correctness and Complexity The correctness of the ALU follows from the correctness of the functional bootstrapping [AKP25], the instruction set, and the OREAD algorithm. The complexity is dominated by the parallel evaluation of the ISA. In an ideal multi-threaded environment, the runtime is bounded by the latency of the most expensive instruction, which is Mul in our case, as we need to perform $\lceil \log_d W \rceil$ functional bootstrapping. Since in most configurations, $|\mathbb{I}| \ll M$, the selection via OREAD is significantly faster than for memory access.

MIMD Parallelism Because the instruction indicators are packed CKKS ciphertexts, they can encrypt a vector of distinct instructions. This allows each slot in the ciphertext to select a different result from **ISAMem**, enabling different slots to execute distinct instructions. This allows the parallel execution of distinct program threads across the slots.

4.3 Processor Cycle

With the previously introduced building blocks, we can define a cycle for HEGIDE. Unlike traditional processor architectures (e.g., a RISC-V five-stage pipeline), HEGIDE intentionally has no instruction fetch or decode stages: since the entire instruction stream is statically scheduled at compile time, these stages would only add redundant overhead. Instead, each cycle receives its pre-computed encrypted operand addresses and instruction indicators directly, reducing the processor to a fixed Read-Execute-Write sequence. A program is compiled into a list of cycles, where each cycle executes the sequence described in Algorithm 4.

Algorithm 4: HEGIDE Cycle: HGCYCLE

Input: Address indicators $\mathbf{Adr}_1, \mathbf{Adr}_2 \in (\mathcal{R}^2_{Q'_{\ell_m}})^M$, Instruction indicators $\mathbf{Instr} \in (\mathcal{R}^2_{Q'_{\ell_{m+1}}})^{|\mathbb{I}|}$, Memory State $\mathbf{Mem} \in (\mathcal{R}^2_{Q'_{\ell_m}})^M$

Output: New Memory State $\mathbf{Mem}' \in (\mathcal{R}^2_{Q'_{\ell_m}})^M$

/* 1. Oblivious retrieval of operands */

1 $\mathsf{REG}_1 \leftarrow \mathsf{OREAD}(\mathbf{Adr}_1, \mathbf{Mem})$;

2 $\mathsf{REG}_2 \leftarrow \mathsf{OREAD}(\mathbf{Adr}_2, \mathbf{Mem})$;

/* 2. Oblivious instruction */

3 $\mathsf{RES} \leftarrow \mathsf{OALU}(\mathbf{Instr}, \mathsf{REG}_1, \mathsf{REG}_2)$;

/* 3. State update */

4 **return** $\mathsf{OWRITE}(\mathsf{RES}, \mathbf{Mem})$;

Correctness and complexity The correctness of the cycle follows directly from the correctness of its components. For moderate memory sizes, the computational complexity is dominated by OALU, as it involves multiple functional bootstrapping operations to clean and evaluate the operands, whereas the memory operations are primarily linear weighted sums. OALU involves digit decomposition and barrel-shifting logic which requires up to $O(W)$ functional bootstrappings. This dominates the cycle cost. Consequently, the total bootstrapping complexity for a program of T cycles scales with $O(TW)$.

Noise Analysis and Stability A fundamental requirement for any FHE-based processor is the ability to run for an arbitrary number of cycles without noise-induced failure. In HEGIDE, noise stability is achieved through the systematic application of functional bootstrapping at the start of the OALU pipeline.

During the multi-value bootstrapping of Algorithm 3, the read operands are effectively cleaned by the Hermite interpolation used in the functional bootstrapping. As characterized in [AKP25], for a given (low enough) input noise, the output noise is notably reduced. Consequently, any noise accumulated during the OREAD operation in the current cycle, or inherited from the results of the previous cycle, is reduced before the actual instruction logic is evaluated.

This "clean-on-read" architecture ensures that the noise magnitude at any point in the processor cycle remains bounded. This systematic noise-cleaning procedure allows HEGIDE to support unbounded execution depth, making it suitable for complex, long-running algorithms. Note that for an ISA with high noise-inducing instructions, a second cleaning step with functional bootstrapping can be performed immediately before the write operation, adding minimal overhead (k functional bootstrapping in parallel) while providing tighter noise control.

MIMD Parallelism Because every operation in HGCYCLE (Algorithm 4) is vectorized across the N slots of the ciphertext ($N/2$ real and $N/2$ imaginary for CKKS), the processor naturally operates in MIMD fashion, where each slot represents an independent execution thread. This allows the execution of up to N different programs on N different data streams, or, as presented in Appendix D, the multi-threaded execution of cooperating programs, utilizing encrypted CKKS rotations to enable communication between threads.

4.4 Input and Output

The execution of HEGIDE requires a setup phase where the client compiles the program and encrypts the necessary data structures. The input consists of:

- *Encrypted Memory State*: The initial memory $\mathbf{Mem}_0$, containing the encrypted inputs and global variables.

- *Encrypted Instruction Stream*: The sequence of encrypted information for each cycle $t \in [\![0, T-1]\!]$, comprising the address indicators $\mathbf{Adr}_1^{(t)}, \mathbf{Adr}_2^{(t)}$ and the instruction indicators $\mathbf{Instr}^{(t)}$.

The server executes the T cycles blindly. For the output, we propose three modes, depending on the algorithm.

1. *Last Element*: The server returns the result of the last computation, which is the tail of the OSReM. This minimizes communication.

2. *Full State Dump*: The server returns the entire memory state.

3. *Sanitized Dump*: To prevent leakage from intermediate dead variables, the server masks the memory state by multiplying each element by an encrypted binary mask (0 for dead, 1 for live), and then returns the full state.

Note that since the memory is maintained in the CKKS slot domain, the server must first apply SlotsToCoeffs to the result ciphertexts to convert them to RLWE form. This will minimize bandwidth, as $q \approx q'_0$ is the smallest ciphertext modulus.

4.5 Security

We conclude the architectural description by formally establishing the security of HEGIDE. We rely on the security of the OREAD primitive established in Theorem 2.

Theorem 3 (Full Obliviousness of HEGIDE)**.** *Let $\mathcal{P}_1, \mathcal{P}_2$ be two programs compiled for HEGIDE having the same number of cycles T, memory size M, word size W, utilizing the same instruction set $\mathbb{I}$. Assuming the underlying CKKS scheme is IND-CPA secure, the view of the server during the execution of $\mathcal{P}_1$ is computationally indistinguishable from the view during the execution of $\mathcal{P}_2$.*

Proof. The server's view consists of the encrypted inputs (memory state and instruction stream) and the trace of operations performed during the T cycles. First, by the IND-CPA security of CKKS, the encrypted initial memory states and instruction streams are indistinguishable for $\mathcal{P}_1$ and $\mathcal{P}_2$, as their sizes are determined by the public parameters T, M, W and $\mathbb{I}$.

Second, consider the execution trace. In every cycle t, the server performs a fixed sequence of operations: two OREAD calls, one OALU call, and one OWRITE. The memory access patterns are protected by the obliviousness of OREAD (cf. Theorem 2). Since the server executes OREAD on encrypted address indicators, it cannot distinguish which memory cells are accessed.

Then, the server evaluates the multi-value bootstrapping and all instructions in $\mathbb{I}$ identically for both programs. The selection of the valid result is performed via OREAD on the internal **ISAMem** using encrypted instruction indicators. By Theorem 2, this selection reveals nothing about the chosen instruction.

Finally, the OWRITE operation is a deterministic shift independent of the data.

Since every sub-component of the cycle generates a computationally indistinguishable view or a deterministic trace, the composition of T cycles maintains this property. Thus, HEGIDE achieves full obliviousness. □

5 Compilation

To bridge the gap between high-level logic and the OSReM architecture, we develop a dedicated compiler that transforms a subset of Python into a structured machine code representation for HEGIDE. The compiler's primary objective is to linearize execution and manage the physical mapping of variables within the shift-register memory. Currently, the compiler supports scalar variables and static control flow; complex data structures like lists, which require dynamic oblivious addressing, are left for future work. The compilation pipeline consists of the following stages:

Front-end Transformation. The compiler parses source code into an Abstract Syntax Tree (AST) to enforce the constraints of an oblivious decider. To eliminate data-dependent control flow, the compiler performs if-arithmetization: standard `if` c`:` $x = a$ `else:` $x = b$ blocks are converted into the functional equivalent $x = c \cdot a + (1 - c) \cdot b$. This ensures both branches are evaluated, maintaining a uniform sequence of operations. Finally, complex nested expressions are decomposed into three-address code to match the single-instruction-per-cycle design of HEGIDE.

Memory Scheduling. Since the program is an oblivious decider, the compiler statically determines the access trace. For each variable v, the compiler identifies the cycle of its last use, t_{end}. We define the Time-To-Live (TTL) at cycle t as $\mathrm{TTL}_v(t) = t_{\mathsf{end}} - t$. A variable is considered live as long as $\mathrm{TTL}_v(t) \geq 0$.

Greedy Initialization. As discussed in Section 3, the compiler produces an end-packed initial state to postpone maintenance cycles. Global variables and inputs are allocated to the initial memory state using a greedy heuristic: variables with the largest TTL are placed near the tail of the memory state, while short-lived variables are placed closer to the head.

Static Memory Simulation. The compiler performs a cycle-by-cycle simulation of the OSReM state. At each cycle t, it inspects the variable at the head (index 0). If the variable is still live, a head collision is detected. To resolve the collision, the compiler injects a Cpy instruction into the instruction stream, which re-circulates the variable to the tail. The simulation state is updated to reflect this additional shift, and the process continues until all instructions are scheduled. Note that determining the optimal schedule to minimize Cpy operations is a non-trivial combinatorial optimization problem. Independent instruction reordering could also be used to prevent the injection of a Cpy.

Code Generation. Upon successful simulation, the compiler generates a linear assembly-like intermediate representation (`.ahg`). This is then assembled into the final machine code: a JSON-formatted map containing the initial memory state, memory and word sizes, as well as the sequence of opcodes and operand addresses. This file serves as the cleartext "program binary" which the client subsequently encrypts and uploads to the server for execution.

Application Scope and MIMD Parallelization While the compilation pipeline detailed above currently generates a single instruction stream for a given program, the massive slot parallelism of HEGIDE ($N/2$ complex slots taken as up to N real slots) is designed to be fully utilized by parallel workloads. We categorize the application scope into four paradigms:

1. *Same program, same data (Sequential)*: Least favorable case, resulting in low slot utilization as only one program is evaluated on one slot.

2. *Same program, same data (Parallelizable)*: Efficient, utilizing slots to vectorize internal algorithm loops.

3. *Same program, different data (SIMD)*: Highly efficient, ideal for batch processing such as batched AI inference.

4. *Different program, different data (MIMD)*: Highly efficient, providing true independent multi-threading.

As demonstrated in Section 6, HEGIDE is at least two orders of magnitude faster in amortized time compared to previous approaches. Consequently, even a modest occupancy of roughly a thousand slots (out of N) suffices to match or exceed the single-thread throughput of the fastest existing encrypted processor. It is also worth noting that some of the above regimes could be combined, for example, to compute a single program on 32 threads on 256 different data, utilizing 8192 slots.
Extending the compiler to natively orchestrate and schedule multi-threaded MIMD programs is a natural avenue for future work. In our current proof-of-concept, to leverage these highly efficient SIMD and MIMD execution modes, the parallelization is managed via the host interface. The user compiles the distinct programs into their respective JSON representations independently. These compiled assemblies are then loaded into a C++ `std::vector` and passed to a program-loading routine, which manually maps the independent instruction sequences and datasets to the corresponding slots in the CKKS plaintexts prior to encryption.

6 Implementation and Experimental Results

6.1 System Details

All benchmarks were conducted on a dual-socket server equipped with two Intel Xeon Platinum 8268 processors (24 cores each, 2.90 GHz) and 192 GB of RAM, running Ubuntu 24.04.3 LTS. Our proof-of-concept implementation of HEGIDE is built upon OpenFHE v1.4.2, compiled with clang++ using native optimizations.[5] We utilize the sparse key encapsulation mode with fixed manual scaling and hybrid key switching. To leverage hardware acceleration, we utilize the Intel HEXL [BKS+21] backend, which provides AVX-512-optimized kernels for polynomial arithmetic. However, as our server does not possess AVX512 IFMA52 instructions, we default to the AVX2 kernels. Benchmarks without HEXL acceleration are provided in Appendix E for comparison.

We fix the ring dimension to $N = 2^{16}$ and employ full slot packing for both RLWE and CKKS ciphertexts to maximize throughput. Targeting a security level of $\lambda \geq 128$ bits, we set the modulus capacity threshold $\log(Q'_L P')$ to 1533 bits (following Table 3 of [BMTPH21]) and the scaling factor to $\Delta = 2^{48}$. The plaintext modulus is set to $p = 16$. Consequently, representing a virtual word of size W requires decomposing the data across $k = \lceil W / \log p \rceil$ independent ciphertexts.

6.2 Results

Latency and Amortized Cycle Time Since HEGIDE is an oblivious architecture, its runtime is deterministic and depends solely on the number of processor cycles, independent of the input data. We therefore report the cycle latency and the amortized cycle time. The amortized time represents the effective cost per program thread when the processor is fully saturated (MIMD).

[5] A public release of the source code is planned, subject to internal approval.

Table 1: Latency and amortized cycle time (latency/N) for various word sizes W and memory capacities M. Measured with HEXL backend, averaged over 5 cycles.

W	$\log M$	$\log(Q'_L P')$	Latency (s)	Amtz. time (ms)
16	4	1476	418	6.38
16	8	1524	473	7.22
16†	12	1452	822	12.55
32	4	1476	819	12.49
32	8	1524	917	14.00
32†	12	1452	1070	16.33
48	4	1524	1340	20.45
48	8	1452	1349	20.59
48†	12	1500	1590	24.26
64†	4	1524	1820	27.77
64†	8	1452	1823	27.81

Several compression techniques can be used to trade off between memory consumption and computation speed (see Appendix B for details). In our experiments, we default to the *Low* compression mode, which offers the best latency. However, entries marked with a † utilize *Medium* compression to remain within the server's RAM limits; this mode introduces additional computational overhead, which accounts for the observed performance degradation. Benchmarks without HEXL acceleration are provided in Appendix E.

Micro-benchmarks and Bottlenecks Breaking down the cycle reveals that the oblivious ALU dominates the runtime, accounting for approximately $90 - 95\%$ of the cycle latency depending on the configuration. The oblivious read consumes the majority of the remaining time, while the oblivious write is negligible (around 200 µs on average). We notice that the cycle time scales as $O(\log M + W)$ for this range of parameters.

Profiling reveals that our implementation is currently memory-bound rather than compute-bound. Specifically, the underlying library (OpenFHE) performs frequent deep copies of large ciphertexts during bootstrapping. This results in severe cache contention, with profiling tools indicating Last Level Cache (LLC) miss rates exceeding 90%. This architectural limitation prevents us from fully saturating the 48 cores of our test platform, and on average, only 10 threads are used. Eliminating these redundant copies and improving memory locality, potentially via a custom backend or optimized allocator, could theoretically yield a notable speedup.

Virtual Frequency We can interpret the amortized performance as a virtual frequency $f_{\text{virt}} = 1/T_{\text{amtz}}$, representing the effective clock speed of the virtual processor per data slot. For our 16-bit configuration, we achieve a virtual frequency of approximately 157 Hz. While this is orders of magnitude slower than silicon, it is sufficient for many real-time control loops and privacy-preserving analytics where throughput is more critical than single-thread latency.

RAM Consumption Memory usage in HEGIDE is static after initialization, determined solely by the memory size M, word size W, and the compression level. Table 2 summarizes the peak RAM usage for the described configuration.

A significant portion of this memory footprint is due to alignment overhead. Our RNS implementation uses 48-bit prime moduli stored in 64-bit machine words, leaving the upper 16 bits (25%) unused. Advanced techniques, such as grafting [CCK+25], could enable us to pack moduli more tightly, reducing the memory footprint by approximately 25% and

Table 2: Peak RAM consumption of HEGIDE for varying memory and word sizes.

W	$\log M$	Total RAM (GB)	Amtz. RAM. (MB)
16	4	93.3	1.42
16	8	90.5	1.38
16†	12	98.9	1.51
32	4	137	2.09
32	8	150.2	2.29
32†	12	132.3	2.02
48	4	107.3	1.64
48	8	112.4	1.71
48†	12	178.9	2.73
64†	4	136.4	2.08
64†	8	142.8	2.18

improving cache locality. Additionally, around 50 GB of RAM are used by the various keys needed for rescaling and bootstrapping.

6.3 Comparison with State-of-the-Art

To contextualize our performance, we compare HEGIDE against existing FHE-based execution environments. These systems differ not only in performance but also in design philosophy. `Phantom` [BM25b] and VSP [MBM+21] prioritize compatibility with standard toolchains: `Phantom` implements a fully encrypted RISC-V virtual machine, accepting standard binaries, while VSP provides a C-compatible compilation flow via its CAHPv3 ISA and LLVM backend. This "plug-and-play" approach lowers the barrier to entry but forces the processor to emulate traditional pipeline stages (fetch, decode, execute) under FHE, incurring significant per-cycle overhead. OUF [DK25] takes a different route, encoding functions as encrypted LUTs evaluated via TFHE functional bootstrapping, but its expressiveness is constrained by the tape size. FURISC [CA19] uses a single SBN instruction, at the cost of massive instruction count inflation. HEGIDE instead designs an architecture natively tailored to FHE constraints: by discarding the program counter and traditional pipeline in favor of statically scheduled Read-Execute-Write cycles and the SReM memory model, it avoids the overhead of emulating constructs that are redundant in the oblivious decider model.

To enable a fair hardware comparison, we ran both `Phantom` and VSP on the same server used for our benchmarks (Section 6). On our reference decision-tree program (see the next paragraph), `Phantom` achieves a cycle time of approximately 1.26 s using its AVX2+FMA backend, and VSP approximately 3.28 s per cycle on the ruby 5-stage pipeline with the Iyokan AVX2 evaluator. Both systems process a single program at a time, so their per-cycle time is also their single-thread throughput. The VSP authors estimate FURISC's per-cycle time at approximately 1278 s. OUF exhibits round latencies ranging from a few seconds to tens of seconds depending on the function set. In contrast, HEGIDE achieves an amortized cycle time between 6.4 ms and 27.8 ms across $N = 2^{16}$ MIMD slots, a throughput speedup of over two orders of magnitude compared to `Phantom` and VSP, and nearly three orders of magnitude compared to OUF.

End-to-End Benchmark: Decision Tree Evaluation The target application scope of HEGIDE is PFE, where the encrypted program embodies proprietary logic such as a decision tree or scoring model. We therefore evaluate an oblivious depth-4 binary decision tree (15 internal nodes, 16 leaves, 4 feature comparisons against constant thresholds)

on HEGIDE, `Phantom`, and VSP, on the same hardware described in Section 6. In all three systems the traversal path is a secret function of encrypted data, so every one of the $2^D - 1 = 15$ internal nodes must be evaluated regardless of the input. The three implementations share the tree structure and produce the same expected leaf on our sample input.
VSP's LLVM-CAHP toolchain lowers the C source to 41 CAHPv3 cycles; `Phantom`'s Rust frontend emits 30 RISC-V cycles; HEGIDE compiles to 60 cycles, because each if/else arithmetizes into four operations, i.e. four cycles, since the oblivious decider model has no native branch primitive under FHE. The $60 > 41 > 30$ ordering reflects the cost of expressing conditionals arithmetically rather than via a hardware branch, not an increase in underlying work; the compensation is that HEGIDE's per-cycle cost is two orders of magnitude lower.
We measure the wall-clock time with AVX2+FMA enabled.

- VSP ($K = 41$, `ruby` 5-stage pipeline, Iyokan AVX2, 96 CPU workers): 3.28 s per cycle, end-to-end **121.9** s, peak memory 1.56 GB.
- `Phantom` ($K = 30$, Poulpy `FFT64Avx` backend with the AVX2+FMA, 64 threads): 1.26 s per cycle, end-to-end **41.4** s, peak memory 1.93 GB.
- HEGIDE ($K = 60$, $W = 16$, $M = 256$): 473 s per cycle and 28400 s end-to-end, amortized to **7.22** ms per cycle and **433** ms end-to-end across $N = 2^{16}$ MIMD slots; peak memory 90.5 GB total, or ~ 1.4 MB per thread.

The VSP authors estimate FURISC's per-cycle time at approximately 1278 s. For the same tree, FURISC would require hundreds of SBN cycles and project to days of wall-clock time.

Timing Analysis Two regimes emerge, corresponding to distinct use cases. For *interactive, single-query* evaluation–e.g. a one-shot oblivious credit check or a real-time triage decision where a user waits for a single answer–HEGIDE's per-program latency of almost 8 hours is prohibitive: `Phantom`'s 41 s and VSP's 122 s remain the only practical options. For *throughput-oriented* evaluation–batch scoring of many records against the same encrypted tree, or any workload with embarrassingly-parallel internal threads–HEGIDE becomes the better choice as soon as the batch exceeds roughly **230** parallel programs (the break-even point against VSP's per-program latency) or **690** parallel programs (the break-even against `Phantom`'s), a threshold trivially reached in any realistic analytics or ML-inference pipeline. At full slot utilization ($N = 2^{16}$ programs per batch), HEGIDE's aggregate throughput is **96**× `Phantom`'s and **281**× VSP's, and its **433** ms amortized latency per thread sits two orders of magnitude below either single-program baseline.

Memory Analysis In the same batch regime, HEGIDE's per-thread memory footprint (~ 1.4 MB) is three orders of magnitude smaller than VSP's or `Phantom`'s, matching the end-to-end latency ratio (Table 2). The same advantage applies to *data-parallel* single-program workloads (the same encrypted program run on distinct encrypted inputs) and to any program whose internal loop can be vectorized across slots, which together cover most of the PFE design space.
VSP's reference implementation is hard-capped at 512 bytes of ROM and 512 bytes of RAM. Trees with richer feature vectors–categorical encodings, aggregated statistics, or ensembles of scorecards–would exceed these bounds and require a larger and significantly slower CAHPv3 memory configuration. `Phantom` scales in memory but has no MIMD path: its per-program latency grows linearly with batch size. HEGIDE's word and memory dimensions scale continuously through the FHE parameter space, and the MIMD structure

exploits every available slot for free throughput, making it the most practical option among encrypted processors for throughput-oriented PFE workloads.

HEGIDE on GPU The CPU-only measurements above are conservative: HEGIDE's workload is particularly well-matched to GPU execution. Each cycle reduces to a handful of large, regular homomorphic operations over a ring dimension $N \geq 2^{14}$, with no control-flow divergence that aligns one-to-one with the SIMT execution model of modern GPUs. This is precisely the regime in which mature CKKS GPU backends deliver their largest speedups, as shown in [CYK+26], which offers 13ms bootstrapping with similar parameters on an RTX 5090. By contrast, TFHE-based processors such as VSP or OUF operate on much smaller rings ($N \sim 2^{10}$) with frequent fine-grained bootstrapping, a regime where kernel-launch overhead and limited per-operation parallelism cap the achievable GPU gain, with bootstrapping records just under 1ms on a similar GPU. We therefore expect GPU acceleration to *widen*, rather than narrow, HEGIDE's throughput advantage in the batch regime. The underlying architectural lessons–the OSReM shift-register memory and the all-paths arithmetic selection–are orthogonal to the concrete FHE scheme and could, in principle, be applied to other encrypted processor designs to reduce their per-cycle cost in the same way.

7 Concluding Remarks

In this work, we introduced HEGIDE, an architecture that enables the efficient execution of encrypted programs on encrypted data. By replacing standard RAM with the novel OSReM shift-register memory, we eliminated the prohibitive cost of oblivious writes, enabling constant-time updates for oblivious algorithms. OSReM exploits a key insight specific to the oblivious decider model: since memory access patterns are entirely determined at compile time, the general-purpose access-pattern hiding of ORAM and PIR is unnecessary, and a shift-register design can achieve zero-cost writes while keeping reads at depth 1. Coupled with a MIMD design that leverages CKKS packing, HEGIDE achieves an amortized cycle time of 6.4 ms, demonstrating that program privacy is achievable without sacrificing practical throughput. End-to-end benchmarks on representative PFE workloads confirm that HEGIDE's amortized throughput exceeds that of existing encrypted processors by two orders of magnitude. Furthermore, the architecture is inherently flexible, enabling configurable trade-offs between computation speed and memory footprint, and supporting powerful extensions described in the Appendix.

Future work will focus on enhancing the compiler with automatic MIMD vectorization, exploring floating-point instruction sets to support a broader range of scientific computations, and leveraging software and hardware acceleration to make HEGIDE viable for real-world applications.

References

[ADK24] Sofiane Azogagh, Victor Delfour, and Marc-Olivier Killijian. Oblivious turing machine. In *2024 19th European Dependable Computing Conference (EDCC)*, pages 17–24, Piscataway, NJ, USA, 2024. IEEE.

[AKP25] Andreea Alexandru, Andrey Kim, and Yuriy Polyakov. General functional bootstrapping using CKKS. In Yael Tauman Kalai and Seny F. Kamara, editors, *Advances in Cryptology – CRYPTO 2025, Part III*, volume 16002 of *Lecture Notes in Computer Science*, pages 304–337, Santa Barbara, CA, USA, August 17–21, 2025. Springer, Cham, Switzerland.

[BAB+22] Ahmad Al Badawi, Andreea Alexandru, Jack Bates, Flavio Bergamaschi, David Bruce Cousins, Saroja Erabelli, Nicholas Genise, Shai Halevi, Hamish Hunt, Andrey Kim, Yongwoo Lee, Zeyu Liu, Daniele Micciancio, Carlo Pascoe, Yuriy Polyakov, Ian Quah, Saraswathy R.V., Kurt Rohloff, Jonathan Saylor, Dmitriy Suponitsky, Matthew Triplett, Vinod Vaikuntanathan, and Vincent Zucca. OpenFHE: Open-source fully homomorphic encryption library. Cryptology ePrint Archive, Report 2022/915, 2022. https://eprint.iacr.org/2022/915.

[BGV14] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (leveled) fully homomorphic encryption without bootstrapping. *ACM Transactions on Computation Theory (TOCT)*, 6(3):1–36, 2014.

[BK25] Dan Boneh and Jaehyung Kim. Homomorphic encryption for large integers from nested residue number systems. In Yael Tauman Kalai and Seny F. Kamara, editors, *Advances in Cryptology – CRYPTO 2025, Part III*, volume 16002 of *Lecture Notes in Computer Science*, pages 338–370, Santa Barbara, CA, USA, August 17–21, 2025. Springer, Cham, Switzerland.

[BKS+21] Fabian Boemer, Sejun Kim, Gelila Seifu, Fillipe DM de Souza, Vinodh Gopal, et al. Intel HEXL (release 1.2). https://github.com/intel/hexl, September 2021.

[BKSS24] Youngjin Bae, Jaehyung Kim, Damien Stehlé, and Elias Suvanto. Bootstrapping small integers with CKKS. In Kai-Min Chung and Yu Sasaki, editors, *Advances in Cryptology – ASIACRYPT 2024, Part I*, volume 15484 of *Lecture Notes in Computer Science*, pages 330–360, Kolkata, India, December 9–13, 2024. Springer, Singapore, Singapore.

[BM25a] Jean-Philippe Bossuat and Janmajaya Mall. Fhe-ram. Online: https://github.com/phantomzone-org/fhe-ram, May 2025.

[BM25b] Jean-Philippe Bossuat and Janmajaya Mall. Phantom. Online: https://github.com/phantomzone-org/phantom, Dec 2025.

[BMTPH21] Jean-Philippe Bossuat, Christian Mouchet, Juan Troncoso-Pastoriza, and Jean-Pierre Hubaux. Efficient bootstrapping for approximate homomorphic encryption with non-sparse keys. In *Advances in Cryptology – EUROCRYPT 2021: 40th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, October 17–21, 2021, Proceedings, Part I*, page 587–617, Berlin, Heidelberg, 2021. Springer-Verlag.

[BPS12] Michael Brenner, Henning Perl, and Matthew Smith. How practical is homomorphically encrypted program execution? an implementation and performance evaluation. In Geyong Min, Yulei Wu, Lei (Chris) Liu, Xiaolong Jin, Stephen A. Jarvis, and Ahmed Yassin Al-Dubai, editors, *11th IEEE International Conference on Trust, Security and Privacy in Computing and Communications, TrustCom 2012, Liverpool, United Kingdom, June 25-27, 2012*, pages 375–382, Piscataway, NJ, USA, 2012. IEEE Computer Society.

[Bra12] Zvika Brakerski. Fully homomorphic encryption without modulus switching from classical GapSVP. In Reihaneh Safavi-Naini and Ran Canetti, editors, *Advances in Cryptology – CRYPTO 2012*, volume 7417 of *Lecture Notes in Computer Science*, pages 868–886, Berlin, Heidelberg, 2012. Springer.

[BTH22] Jean-Philippe Bossuat, Juan Ramón Troncoso-Pastoriza, and Jean-Pierre Hubaux. Bootstrapping for approximate homomorphic encryption with negligible failure-probability by using sparse-secret encapsulation. In Giuseppe Ateniese and Daniele Venturi, editors, *ACNS 2022: 20th International Conference on Applied Cryptography and Network Security*, volume 13269 of *Lecture Notes in Computer Science*, pages 521–541, Rome, Italy, June 20–23, 2022. Springer, Cham, Switzerland.

[CA19] Ayantika Chatterjee and Khin Mi Mi Aung. *FURISC: FHE Encrypted URISC Design*, pages 87–115. Springer Singapore, Singapore, 2019.

[CBSZ23] Pierre-Emmanuel Clet, Aymen Boudguiga, Renaud Sirdey, and Martin Zuber. Combo: A novel functional bootstrapping method for efficient evaluation of nonlinear functions in the encrypted domain. In Nadia El Mrabet, Luca De Feo, and Sylvain Duquesne, editors, *Progress in Cryptology - AFRICACRYPT 2023*, pages 317–343, Cham, 2023. Springer Nature Switzerland.

[CCK+25] Jung Hee Cheon, Hyeongmin Choe, Minsik Kang, Jaehyung Kim, Seonghak Kim, Johannes Mono, and Taeyeong Noh. Grafting: Decoupled scale factors and modulus in rns-ckks. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security*, CCS '25, page 1098–1112, New York, NY, USA, 2025. Association for Computing Machinery.

[CDS15] Sergiu Carpov, Paul Dubrulle, and Renaud Sirdey. Armadillo: A compilation chain for privacy preserving applications. In Feng Bao, Steven Miller, Sherman S. M. Chow, and Danfeng Yao, editors, *Proceedings of the 3rd International Workshop on Security in Cloud Computing, SCC@ASIACCS '15, Singapore, Republic of Singapore, April 14, 2015*, pages 13–19, New York, NY, USA, 2015. ACM.

[CGGI20] 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.

[CGKS95] Benny Chor, Oded Goldreich, Eyal Kushilevitz, and Madhu Sudan. Private information retrieval. In *36th Annual Symposium on Foundations of Computer Science*, pages 41–50, Milwaukee, Wisconsin, October 23–25, 1995. IEEE Computer Society Press.

[CHK+18a] Jung Hee Cheon, Kyoohyung Han, Andrey Kim, Miran Kim, and Yongsoo Song. Bootstrapping for approximate homomorphic encryption. In Jesper Buus Nielsen and Vincent Rijmen, editors, *Advances in Cryptology - EUROCRYPT 2018 - 37th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tel Aviv, Israel, April 29 - May 3, 2018 Proceedings, Part I*, volume 10820 of *Lecture Notes in Computer Science*, pages 360–384, Cham, 2018. Springer.

[CHK+18b] Jung Hee Cheon, Kyoohyung Han, Andrey Kim, Miran Kim, and Yongsoo Song. A full RNS variant of approximate homomorphic encryption. In Carlos Cid and Michael J. Jacobson Jr., editors, *Selected Areas in Cryptography - SAC 2018 - 25th International Conference, Calgary, AB, Canada, August 15-17, 2018, Revised Selected Papers*, volume 11349 of *Lecture Notes in Computer Science*, pages 347–368, Cham, 2018. Springer.

[CJP21] Ilaria Chillotti, Marc Joye, and Pascal Paillier. Programmable bootstrapping enables efficient homomorphic inference of deep neural networks. In *Cyber Security Cryptography and Machine Learning: 5th International Symposium, CSCML 2021, Be'er Sheva, Israel, July 8–9, 2021, Proceedings 5*, pages 1–19, Cham, 2021. Springer.

[CKKS17] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yong Soo Song. Homomorphic encryption for arithmetic of approximate numbers. In Tsuyoshi Takagi and Thomas Peyrin, editors, *Advances in Cryptology – ASIACRYPT 2017, Part I*, volume 10624 of *Lecture Notes in Computer Science*, pages 409–437, Hong Kong, China, December 3–7, 2017. Springer, Cham, Switzerland.

[CLOT21] Ilaria Chillotti, Damien Ligier, Jean-Baptiste Orfila, and Samuel Tap. Improved programmable bootstrapping with larger precision and efficient arithmetic circuits for tfhe. In *Advances in Cryptology – ASIACRYPT 2021: 27th International Conference on the Theory and Application of Cryptology and Information Security, Singapore, December 6–10, 2021, Proceedings, Part III*, page 670–699, Berlin, Heidelberg, 2021. Springer-Verlag.

[CYK+26] Wonseok Choi, Hyunah Yu, Jongmin Kim, Hyesung Ji, Jaiyoung Park, and Jung Ho Ahn. Theodosian: A deep dive into memory-hierarchy-centric fhe acceleration, 2026. Accepted at ISPASS 2026.

[DAP+26] Jules Dumezy, Andreea Alexandru, Yuriy Polyakov, Pierre-Emmanuel Clet, Olive Chakraborty, and Aymen Boudguiga. Evaluating larger lookup tables using ckks. In *IACR Transactions on Cryptographic Hardware and Embedded Systems*, volume 1, Bochum, Germany, 2026. Ruhr-Universität Bochum. (to appear).

[DK25] Victor Delfour and Marc-Olivier Killijian. Ouf: Oblivious universal function with domain specific optimizations. In *Proceedings of the 24th IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, Piscataway, NJ, USA, 2025. IEEE. (to appear).

[DM15] Léo Ducas and Daniele Micciancio. Fhew: Bootstrapping homomorphic encryption in less than a second. In Elisabeth Oswald and Marc Fischlin, editors, *Advances in Cryptology – EUROCRYPT 2015*, pages 617–640, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.

[FV12] Junfeng Fan and Frederik Vercauteren. Somewhat practical fully homomorphic encryption. Cryptology ePrint Archive, Report 2012/144, 2012.

[GMT24] Charles Gouert, Dimitris Mouris, and Nektarios Georgios Tsoutsos. Juliet: A configurable processor for computing on encrypted data. Cryptology ePrint Archive, Report 2024/1089, 2024.

[GSP+21] Shruthi Gorantala, Rob Springer, Sean Purser-Haskell, William Lam, Royce J. Wilson, Asra Ali, Eric P. Astor, Itai Zukerman, Sam Ruth, Christoph Dibak, Phillipp Schoppmann, Sasha Kulankhina, Alain Forget, David Marn, Cameron Tew, Rafael Misoczki, Bernat Guillen, Xinyu Ye, Dennis Kraft, Damien Desfontaines, Aishe Krishnamurthy, Miguel Guevara, Irippuge Milinda Perera, Yurii Sushko, and Bryant Gipson. A general purpose transpiler for fully homomorphic encryption. *CoRR*, abs/2106.07893, 2021.

[Kim25] Jaehyung Kim. Efficient homomorphic integer computer from ckks. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2025(4):873–898, Sep. 2025.

[KO97] Eyal Kushilevitz and Rafail Ostrovsky. Replication is NOT needed: SINGLE database, computationally-private information retrieval. In *38th Annual Symposium on Foundations of Computer Science*, pages 364–373, Miami Beach, Florida, October 19–22, 1997. IEEE Computer Society Press.

[KS25] Kamil Kluczniak and Leonard Schild. Fdfb2: Full-domain functional bootstrapping with function amortization. In *Proceedings of the 13th Workshop on Encrypted Computing & Applied Homomorphic Computing*, WAHC '25, page 13–25, New York, NY, USA, 2025. Association for Computing Machinery.

[MBM+21] Kotaro Matsuoka, Ryotaro Banno, Naoki Matsumoto, Takashi Sato, and Song Bian. Virtual secure platform: A five-stage pipeline processor over TFHE. In Michael D. Bailey and Rachel Greenstadt, editors, *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, pages 4007–4024. USENIX Association, 2021.

[SvDS+13] Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher W. Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path ORAM: an extremely simple oblivious RAM protocol. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *ACM CCS 2013: 20th Conference on Computer and Communications Security*, pages 299–310, Berlin, Germany, November 4–8, 2013. ACM Press.

[TBC+25] Daphné Trama, Aymen Boudguiga, Pierre-Emmanuel Clet, Renaud Sirdey, and Nicolas Ye. Designing a general-purpose 8-bit (t)fhe processor abstraction. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2025(2):535–578, Mar. 2025.

[TM13] Nektarios Georgios Tsoutsos and Michail Maniatakos. Investigating the application of one instruction set computing for encrypted data computation. In Benedikt Gierlichs, Sylvain Guilley, and Debdeep Mukhopadhyay, editors, *Security, Privacy, and Applied Cryptography Engineering*, pages 21–37, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[TM14] Nektarios Georgios Tsoutsos and Michail Maniatakos. Heroic: homomorphically encrypted one instruction computer. In *Proceedings of the Conference on Design, Automation & Test in Europe*, DATE '14, Leuven, BEL, 2014. European Design and Automation Association.

[WCS15] Xiao Wang, T.-H. Hubert Chan, and Elaine Shi. Circuit ORAM: On tightness of the Goldreich-Ostrovsky lower bound. In Indrajit Ray, Ninghui Li, and Christopher Kruegel, editors, *ACM CCS 2015: 22nd Conference on Computer and Communications Security*, pages 850–861, Denver, CO, USA, October 12–16, 2015. ACM Press.

A CKKS Operations

We detail the homomorphic operations for the RNS variant of CKKS used in our implementation. All operations assume input ciphertexts share the same ciphertext modulus (i.e., are at the same level). If the moduli levels Q'_ℓ differ, we align them by applying the DropLevels procedure to the ciphertext at the higher level, reducing it to match the lower level.

KeySwitchGen($\mathsf{sk_{in}}, \mathsf{sk_{out}}$) : Generates a switching key to transform a ciphertext encrypting a message under secret key $\mathsf{sk_{in}}$ to one under $\mathsf{sk_{out}}$. Let P' be the gadget modulus

and w the decomposition base (both powers of two). We define the decomposition depth as $d_{\mathsf{swk}} = \lceil \log_w(Q'_L) \rceil$. We sample a uniform vector $\mathbf{a}' \leftarrow (\mathcal{R}_{Q'_L P'})^{d_{\mathsf{swk}}}$ and an error vector $\mathbf{e}' \leftarrow (\chi_{\mathsf{err}})^{d_{\mathsf{swk}}}$. The switching key $\mathsf{swk} = \{(k_{j,0,k_{j,1}})\}_{j=0}^{d_{\mathsf{swk}}-1}$ is constructed as:

$$k_{j,0} = -a'_j \cdot \mathsf{sk}_{\mathsf{out}} + e'_j + P' w^j \cdot \mathsf{sk}_{\mathsf{in}}, \quad k_{j,1} = a'_j.$$

We specifically generate the relinearization key as $\mathsf{rlk} \leftarrow \mathsf{KeySwitchGen}(\mathsf{sk}^2, \mathsf{sk})$ and the conjugation key as $\mathsf{cnk} \leftarrow \mathsf{KeySwitchGen}(\sigma(\mathsf{sk}), \mathsf{sk})$, where $\sigma : X \mapsto X^{-1}$ is the conjugation automorphism.

KeySwitch(ct, swk): Applies the key switching operation to a ciphertext $\mathsf{ct} = (c_0, c_1) \in \mathcal{R}^2_{Q'_\ell}$. The component c_1 is decomposed into d_{swk} digits $\{c_{1,j}\}$ in base w, such that $c_1 = \sum c_{1,j} w^j$. The output is computed by evaluating the dot product with the switching key and rescaling by the gadget modulus:

$$\mathsf{ct}' = \left(c_0 + \left\lfloor \frac{1}{P'} \sum_j c_{1,j} k_{j,0} \right\rceil, \quad \left\lfloor \frac{1}{P'} \sum_j c_{1,j} k_{j,1} \right\rceil \right) \mod Q'_\ell$$

DropLevels(ct, k): Reduces the modulus of a ciphertext $\mathsf{ct} \in \mathcal{R}^2_{Q'_\ell}$ by k levels. The function returns the same polynomials modulo $Q'_{\ell-k}$.

Add($\mathsf{ct}_1, \mathsf{ct}_2$): Computes the component-wise addition of two ciphertexts $\mathsf{ct}_1, \mathsf{ct}_2 \in \mathcal{R}^2_{Q'_\ell}$. Returns $\mathsf{ct}_{\mathsf{add}} = [\mathsf{ct}_1 + \mathsf{ct}_2]_{Q'_\ell}$.

Mul($\mathsf{ct}_1, \mathsf{ct}_2$): Performs homomorphic multiplication of two ciphertexts $\mathsf{ct}_1 = (b_1, a_1)$ and $\mathsf{ct}_2 = (b_2, a_2)$. We first compute the tensor product components $(d_0, d_1, d_2) = (b_1 b_2, a_1 b_2 + a_2 b_1, a_1 a_2)$ modulo Q'_ℓ. The quadratic term d_2 (corresponding to sk^2) is relinearized using rlk:

$$\mathsf{ct}_{\mathsf{mul}} = [(d_0, d_1) + \mathsf{KeySwitch}((0, d_2), \mathsf{rlk})]_{Q'_\ell}$$

The result has a scaling factor equal to the product of the input scaling factors.

Rescale(ct): Reduces the scaling factor and modulus of a ciphertext $\mathsf{ct} \in \mathcal{R}^2_{Q'_\ell}$ to manage noise growth. It computes $\mathsf{ct}' = \lfloor (q'_\ell)^{-1} \cdot \mathsf{ct} \rceil$ and switches the modulus to $Q'_{\ell-1}$. The new scaling factor becomes $\Delta_{new} = \Delta_{old}/q'_\ell$.

B Compression

HEGIDE supports four compression modes to balance the trade-off between communication bandwidth, RAM consumption, and server-side computation. These modes primarily affect the encoding of memory addresses and instructions, as detailed in Table 3.[6]

The plaintext modulus p can be increased to decrease the number of OSReM banks needed to store words of size W (which are decomposed into digits of base p). As the internal radix $d = 16$ remains fixed, increasing p introduces an additional overhead during the oblivious ALU phase, as inputs must be decomposed from base p to both base d and base 2. However, the oblivious read phase becomes faster due to the reduction in the number of memory banks. Note that we could leverage the results of [BK25] to support even larger precision ($p = 2^W$), further reducing the memory footprint.

[6] For our current implementation, compression is still experimental for Medium compression, and High compression is not yet supported. Stable compression modes will be available in the first public release of the source code.

Table 3: Compression modes and their impact on parameters. p denotes the plaintext modulus. We report the number of ciphertexts in the memory, the number of ciphertexts per address read, and the number of ciphertexts per instruction execution. Entries marked with $*$ indicate a single RLWE ciphertext with modulus q.

Compression	p	Memory	Address	Instruction
Off	16	$M \cdot W/4$	M	$\lvert\mathbb{I}\rvert$
Low	16	$M \cdot W/4$	$d \log_d M$	$\lvert\mathbb{I}\rvert$
Medium	65536	$M \cdot W/16$	$d \log_d M$	$\lvert\mathbb{I}\rvert$
High	65536	$M \cdot W/16$	1^*	1^*

Regarding instructions, we always provide fully expanded indicators, except in the *High* compression mode, as $|\mathbb{I}|$ is small enough to be negligible. In the *High* compression mode, both addresses and instructions rely on CMT-FBT-OREAD, described below. For *Low* and *Medium* compression, we rely on CMT-OREAD for address read, also presented below.

B.1 CMT-FBT for Oblivious Read

We observe that a memory read operation $y = \mathbf{Mem}[x]$ is equivalent to evaluating a Lookup Table (LUT) where the input x serves as the address and the LUT coefficients represent the memory contents. Based on this, we construct CMT-FBT-OREAD, a CKKS oblivious read primitive derived from the *Collapsed Multiplexer Tree Functional Bootstrapping* (CMT-FBT) [DAP+26].

CMT-FBT utilizes a multiplexer structure to create a one-hot vector for value selection from a raw address in $\mathbb{Z}_M$. We adapt this structure by replacing the plaintext LUT of the original bootstrapping with encrypted CKKS ciphertexts representing the memory state. The algorithm leverages *Functional Digit Decomposition* (FDD) to decompose the encrypted address into digits of radix d (where $d \mid M$). The procedure then computes one-hot indicator vectors from these digits, which drive a multiplexer tree to obliviously select the target ciphertext. This is equivalent to a weighted sum, which guarantees obliviousness.

Algorithm 5: Alternative Oblivious Read: CMT-FBT-OREAD

Input: Address $\mathsf{ADR} \in \mathcal{R}_q^2$, Memory State $\mathbf{Mem} \in (\mathcal{R}_{Q'_\ell}^2)^M$

Output: Target Data $\mathsf{REG} \in \mathcal{R}_{Q'_{\ell-2}}^2$

/* 1. Decompose address into digits and compute indicators */

1 $\{\mathsf{ct}_{\delta,k}\}_{1\le k\le \log_d M} \leftarrow \mathsf{FDD}(\mathsf{ADR}, \{\mathsf{LUT}(\delta_{j,d})\}_{0\le j<d}, d, M)$;

/* 2. Oblivious Selection via Collapsed Multiplexer Tree */

2 $\mathsf{REG} \leftarrow \mathsf{CMT}(\{\mathsf{ct}_{\delta,k}\}_k, \mathbf{Mem})$;

3 **return** REG;

Algorithm Adaptation and Complexity The core of the selection logic is CMT, adapted from Algorithm 6 in [DAP+26]. The standard CMT algorithm decomposes the evaluation into Kronecker products of the one-hot vectors of the address digits (LSD and MSD components) followed by a vector-matrix-vector multiplication with the inverse vectorisation of the LUT. Crucially, in CMT-FBT-OREAD, the LUT is the encrypted memory state **Mem**, which has the following consequences:

- *Ciphertext-Ciphertext Multiplication*: We replace the scalar multiplications used in CMT-FBT with ciphertext-ciphertext multiplications Mul. We employ lazy relin-

earization and rescaling to minimize overhead: following the Kronecker products evaluation, we perform $M + \sqrt{M}$ tensor products but defer the relinearization and rescaling until after the aggregation (sum) step, going from $M + \sqrt{M}$ to $\sqrt{M} + 1$ relinearization and rescaling. While transitioning to Mul consumes one additional multiplicative level, it enables the processing of encrypted data.

- *Reduced Noise Growth*: The noise analysis is similar to CMT-FBT, but replacing MulInt with Mul introduces an additional rescaling. This rescaling reduces the noise induced by the multiplication, which could otherwise amplify noise by up to $p/2$ in the worst case. Consequently, we do not require explicit "cleaning" steps for the memory sizes and plaintext moduli considered in this paper.

The complexity of CMT-FBT-OREAD is $O(M)$, and the multiplicative depth is $\log(\log_d M)+1$. Although this asymptotic complexity matches OREAD, the use of functional bootstrapping in FDD significantly increases the constant factor and computation time.

Multi-Value Optimization In our context, where we must read from multiple OSReM banks to reconstruct a decomposed word, we leverage a "multi-value" approach. We compute the expensive FDD step and Kronecker products only once, and then apply the CMT operation to the different memory states in parallel. This significantly amortizes the computation cost.

B.2 CMT-OREAD

For *Low* and *Medium* compression, the client pre-computes the indicators normally returned by the functional digit decomposition. This involves sending $d \log_d M$ one-hot indicators directly. The server then skips the FDD step and only performs the CMT algorithm. We refer to this version as CMT-OREAD.

For our memory bandwidth-bound application, the *Low* compression mode actually performs better than the *Off*, given that there are fewer ciphertexts per address.

C Complete ISA Description

We describe the algorithms used to implement the 11 instructions of the proposed instruction set.

Radix The Add, Sub, and Mul instructions are implemented using the same carry-propagation algorithm as Kim [Kim25]. The only difference is that, where possible, we skip the identity bootstrapping of the carry to save computation time.

The Cpy instruction acts as an identity function: it takes the input operands (which are decomposed in the OALU) and simply recomposes the first operand into a valid register ciphertext.

Binary The Not, And, Or and Xor instructions are evaluated using the polynomials $P_{\mathsf{Not}} = 1 - X$, $P_{\mathsf{And}} = XY$, $P_{\mathsf{Or}} = 1 - (1 - X)(1 - Y)$ and $P_{\mathsf{Xor}} = (X - Y)^2$, which consume at most one multiplicative level.

For the shift instructions, we only present the left shift (Algorithm 6), as the right shift follows the exact same blueprint (with reversed index logic). The BinRecompose algorithm recomposes the bitwise decomposition back to the radix decomposition (base d), consuming one multiplicative level.

Algorithm 6 consumes $\log(W) + 1$ multiplicative levels (due to the W sequential MUX layers). If we run out of multiplicative levels, we can bootstrap all W ciphertexts in **value** mid-execution using functional bootstrapping.

Algorithm 6: Shift Left: Shl

Input: Binary decomposition of value **value** $\in (\mathcal{R}^2_{Q'_\ell})^W$ and shift amount **shift** $\in (\mathcal{R}^2_{Q'_\ell})^W$

Output: Result Register RES $\in (\mathcal{R}^2_{Q'_{\ell-\log W-1}})^k$

zero $\leftarrow$ Encrypt(0);
`/* Iterate over the bits of the shift amount (Barrel Shifter) */`
for $k \leftarrow 0$ **to** $\log W - 1$ **do**
  temp $\leftarrow \{\text{zero}, \ldots, \text{zero}\}$;
  `/* Iterate over the bits of the value */`
  for $i \leftarrow 0$ **to** $W - 1$ **do**
    $j \leftarrow i - 2^k$;
    $v \leftarrow (j < 0)$? zero : **value**$[j]$;
    `/* MUX: if shift[k] is 1, take v; else take value[i] */`
    $m \leftarrow \text{Mul}(\textbf{shift}[k], \text{Sub}(v, \textbf{value}[i]))$;
    temp$[i] \leftarrow \text{Add}(\textbf{value}[i], m)$;
  value $\leftarrow$ **temp**;
RES $\leftarrow$ BinRecompose(**value**);
return RES;

For the Eq instruction (Algorithm 7), we rely on P_{Xor} and P_{Or} to test equality. We consume the same multiplicative depth as the shift left and right algorithms, as we perform one Xor, and $\log W$ Or.

D Extensions

While the core design of HEGIDE focuses on a standard 2-operand integer architecture, the flexibility of the underlying OREAD and OALU primitives allows for significant extensions.

D.1 N-ary Instructions

The current architecture fetches two operands per cycle. However, certain algorithms (e.g., fused multiply-add, vectorized accumulations) benefit from ternary or n-ary operations. Extending HEGIDE to support n operands is straightforward: it requires performing n parallel OREAD operations at the start of the cycle. The OALU is then expanded to evaluate functions $f(\mathbf{op}_1, \ldots, \mathbf{op}_n)$. While this increases the ciphertext inputs per cycle linearly, the amortized cost remains low as the OREAD operations are performed in parallel and the depth of the OALU typically grows only logarithmically with n.

D.2 Global Memory (OROM)

OSReM is optimized for temporal locality, but some programs require frequent access to global constants or lookup tables (e.g., S-boxes, decision trees) that persist throughout execution. We can integrate a standard Oblivious Read-Only Memory (OROM) alongside OSReM. In this model, the processor performs an auxiliary OREAD on a static memory state **Rom**. To utilize this data, the OALU inputs are augmented with a multiplexer that selects between the dynamic OSReM operand and the static OROM value based on an encrypted control bit in the instruction stream. This enables the processor to seamlessly integrate register-based variables with global constants, thereby reducing the number of maintenance cycles.

Algorithm 7: Equality: Eq

Input: Binary decompositions $\mathbf{bits}_1, \mathbf{bits}_2 \in (\mathcal{R}^2_{Q'_\ell})^W$

Output: Result Register $\mathsf{RES} \in (\mathcal{R}^2_{Q'_{\ell-\log W-1}})^k$

```
/* Bitwise XOR: result is 1 if bits differ, 0 otherwise          */
for i ← 0 to W − 1 do
    x[i] ← Square(Sub(b1[i], b2[i]));
/* Tree-based OR reduction to detect any differences              */
L ← W;
while L > 1 do
    H ← L/2;
    for i ← 0 to H − 1 do
        v1 ← Mul(Sub(1, x[i]), Sub(1, x[i + H]));
        x[i] ← Sub(1, v1);
    L ← H;
RES[0] ← Sub(1, x[0]);
/* Fill remaining register digits with encrypted zeros            */
for i ← 1 to k − 1 do
    RES[i] ← Encrypt(0);
return RES;
```

D.3 Floating Point ISA

Our implementation utilizes discrete CKKS to emulate exact integer arithmetic. However, the architecture is agnostic to the data type. By replacing the digit-decomposition-based ALU with circuits for floating-point arithmetic (e.g., using integer circuits to emulate IEEE 754 mantissa/exponent logic), HEGIDE can support scientific computing workloads. Note that emulating IEEE floats obliviously is significantly more expensive than integer arithmetic due to the complexity of renormalization and rounding steps, likely requiring deeper circuits and more frequent functional bootstrapping.

D.4 Inter-Thread Communication

The MIMD nature of HEGIDE allows processing N independent threads in the N slots of a CKKS ciphertext. By default, these threads are isolated from each other. To enable cooperative multi-threading (where thread i needs data from thread j), we can implement an oblivious routing network. This can be realized using a Benes Network or a similar butterfly permutation network implemented via CKKS rotations. This network operates as a multi-stage barrel shifter, consisting of $2 \log N - 1$ stages. In each stage, data is conditionally swapped or rotated based on control bits derived from the target destination indices. While more expensive, this extension transforms HEGIDE from a vector of isolated processors into a fully connected parallel computer, capable of executing algorithms like parallel sorting or FFTs obliviously.

E Additional Experimental Results

E.1 Non HEXL Benchmarks

For completeness, we evaluate the performance of HEGIDE using the standard backend of OpenFHE, with AVX-512 hardware acceleration disabled.

Table 4: Latency and amortized cycle time (latency/N) for various word sizes W and memory capacities M. Measured with the standard (non HEXL) backend, averaged over 5 cycles.

W	$\log M$	$\log(Q'_L P')$	Latency (s)	Amtz. time (ms)
16	4	1476	567	8.64
16	8	1524	615	9.39
$16^\dagger$	12	1452	1120	17.09
32	4	1476	1107	16.88
32	8	1524	1160	17.69
$32^\dagger$	12	1452	1425	21.74
48	4	1524	1806	27.56
48	8	1452	1829	27.92
$48^\dagger$	12	1500	2108	32.17
$64^\dagger$	4	1524	2336	35.65
$64^\dagger$	8	1452	2317	35.35

Table 4 presents the same benchmarks as Table 1, without the HEXL backend. Consistent with the profiling analysis in Section 6, the removal of arithmetic acceleration results in a relatively modest average latency increase of 1.32×, while keeping the cycle scaling in $O(\log M + W)$. This confirms that the current bottleneck lies primarily in memory bandwidth and cache efficiency (due to large ciphertext sizes) rather than polynomial arithmetic throughput. RAM consumption remains consistent with the HEXL configuration.