Janus-FHE: Reducing Microarchitectural Leakage in GPU-Based Homomorphic Encryption

Kashfia Farheen and Nektarios Georgios Tsoutsos

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

Abstract. Homomorphic Encryption (HE) enables secure cloud computing through computations on encrypted data, but the physical execution of HE workloads on shared GPUs can still expose relevant metadata through microarchitectural behavior. We present a case study of BFV relinearization in HEonGPU, a state-of-the-art GPU HE library, and show that repeated executions exhibit structured, reproducible variation in shared-L2 activity across relinearization-key contexts. Our analysis further shows that the regular control-flow structure observed in the optimized HEonGPU binary is compiler-dependent rather than enforced by the source implementation. Together, these observations motivate an HE design in which execution regularity is established explicitly rather than obtained as a consequence of compilation.

We present JANUS-FHE, a GPU framework for BFV multiplication and relinearization that restructures the computation around fixed-width large-integer arithmetic. JANUS reconstructs ciphertext coefficients from the RNS representation, uses Kronecker substitution to express polynomial multiplication as large-integer multiplication, and evaluates the resulting convolution through a Stockham-organized Discrete Galois Transform (DGT). JANUS carries this fixed execution structure through product recovery, BFV scale-and-round, and method-I relinearization, using fixed-width computation and branchless masked arithmetic throughout. This formulation makes the control flow, memory-access schedule, and execution geometry depend on public parameters rather than the ciphertext or evaluation-key values. Under the same shared-L2 methodology used for HEonGPU, JANUS exhibits fewer recurring key-conditioned differences, while SASS-level analysis confirms that its intended regular execution structure is preserved after compilation. At $N = 8192$, JANUS completes the full multiplication-and-relinearization path in 0.350 ms, compared with 0.153 ms for HEonGPU, representing a 2.29× latency overhead for JANUS's parameter-determined execution structure.

Keywords: Homomorphic Encryption · Side Channel Leakage · Shared Cache Fingerprinting.

1 Introduction

Cloud computing has made large-scale remote computation economically attractive [4] and is organized around on-demand access to shared configurable resources [34]. However, it also forces users to trust third-party infrastructure with sensitive workloads and confidential data. Even with regulatory frameworks, the risk of malicious exploitation remains high. Standard encryption techniques do not provide a solution to this dilemma as they are able to protect the data only in storage and transit. In order to perform meaningful computations on the data, it has to be decrypted. Homomorphic Encryption (HE) [22] represents a cryptographic breakthrough in this scenario. HE allows computations to be done on encrypted data; the output of a ciphertext computation would be the encrypted form of the output of the same computation using those corresponding plaintexts. HE can therefore ensure the privacy of the data that is to be processed on a cloud server, as the data remain encrypted throughout computation.

However, practical HE is computationally expensive. Modern lattice-based HE schemes such as BFV [18], BGV [11], and CKKS [13] rely on high-degree polynomial arithmetic which imposes substantial computational overhead. While there are FPGA and hardware acceleration options [46], GPUs are an appealing execution platform. They offer massive parallelism, high memory bandwidth, and strong throughput for transform-heavy arithmetic. Prior GPU-accelerated HE frameworks have demonstrated substantial speedups by exploiting Residue Number System (RNS) decomposition [9, 45], modular arithmetic parallelism, and

transform-based convolution. Representative examples include BFV implementations on CUDA GPUs [8], multi-GPU HE designs [7], optimized GPU NTT pipelines [38,41], GPU HE libraries such as HEonGPU and Cheddar [14, 39], broader GPU-focused analyses of HE execution [37], and finite-field acceleration support for HE arithmetic [32].

However, the same GPU features that make HE fast also complicate its physical security. GPUs are shared, highly parallel resources with complex memory hierarchy, using a Single Instruction, Multiple Thread (SIMT) architecture where groups of threads execute in lockstep. This execution model, together with shared caches, memory systems, and other hardware resources, creates microarchitectural observation surfaces that have been exploited in prior GPU side-channel work. In a shared-GPU environment, one relevant concern is fingerprinting: prior work has shown that repeated measurements of timing, contention, performance counters, and other microarchitectural activity can distinguish workloads, execution contexts, or user activity [5, 19, 28, 36, 50]. For GPU-accelerated HE, this question is particularly relevant because performance-critical operations repeatedly process ciphertext- and evaluation-key-derived intermediate values through highly optimized GPU kernels. The cryptographic security of schemes such as BFV does not, by itself, require these implementations to exhibit input-independent control flow or uniform microarchitectural behavior. Consequently, implementation choices that improve throughput may also introduce execution variability that is invisible at the cryptographic abstraction layer but observable in the underlying hardware. This motivates examining GPU HE implementations not only for correctness and performance, but also for the regularity of their execution behavior.

To examine how this concern appears in an existing GPU HE implementation, we consider HEonGPU [39], a state-of-the-art GPU framework for HE. We first examine BFV relinearization in HEonGPU at the source and compiled-instruction levels. In HEonGPU's CUDA code, the modular-reduction routine is written as a loop whose number of iterations can depend on the value being reduced. In the optimized release build, however, the compiler converts this loop into branchless, predicated instructions, so the compiled kernel follows the same control-flow structure for the tested inputs. The regularity we observe in the binary therefore comes from compiler optimization rather than being enforced by the source code itself. Separately, we show that, even under a fixed replay schedule, the kernel `divide_round_lastq_kernel` exhibits sparse but structured variation in shared-L2 cache behavior. Across repeated measurements, this variation differs across relinearization-key contexts and shows reproducible structure. We use hardware performance counters as a high-fidelity diagnostic instrument to characterize this behavior; access to these privileged counters is not assumed for the unprivileged co-resident attacker. The relinearization key is public and may be reused across many operations in the same evaluation context. If executions using the same key repeatedly produce similar hardware behavior, that behavior may allow otherwise opaque FHE-as-a-Service computations to be linked to the same recurring evaluation context. We therefore treat this behavior as a metadata-deanonymization concern and ask whether such repeatable differences exist.

Motivated by these observations, we present JANUS-FHE, a regularity-oriented GPU framework for BFV multiplication and relinearization. PolyFHEmus [24] motivates our use of large-integer multiplication for HE polynomial arithmetic, but JANUS develops a distinct end-to-end GPU pipeline around this direction. The pipeline begins by reconstructing ciphertext coefficients from their RNS representation and converting them into a fixed-width digit representation whose size is determined by the public parameter set. JANUS then uses Kronecker substitution [49] to reformulate the BFV tensor-product multiplication as large-integer convolution and evaluates the resulting products using a DGT-based Schönhage–Strassen-style pipeline [6, 44]. The transforms use a Stockham organization [48], giving each stage a predetermined memory-access and execution structure. After the inverse transform, JANUS recovers the polynomial products through fixed-width carry propagation, performs BFV scale-and-round, and converts the results back into the ciphertext RNS basis. JANUS further extends the same execution model through method-I relinearization, including the key-switch products, accumulation, reconstruction, and final divide-and-round by the special modulus. Across these stages, transform dimensions, loop bounds, memory-access schedules, and execution geometry are determined by public parameters, while modular corrections and related arithmetic are expressed using branchless masked operations. Rather than relying on compiler optimization to regularize value-dependent

source constructs after implementation, JANUS is designed so that regular execution is explicit throughout the complete multiplication-and-relinearization path.

Our main contributions are summarized as follows:

- We present a case study of BFV relinearization in HEonGPU and examine how its GPU behavior changes when the computation is repeated with different relinearization keys. Although the same operation is executed each time, we find that some key-ciphertext configurations repeatedly produce different amounts of shared-L2 cache activity. We use hardware performance counters as a controlled measurement tool to characterize this behavior across repeated executions.
- We design JANUS-FHE, an end-to-end GPU framework for BFV multiplication and relinearization with execution regularity as a design objective. JANUS reconstructs RNS coefficients into a fixed-width representation, reformulates polynomial multiplication using Kronecker substitution and DGT-based large-integer convolution, and carries the same fixed execution structure through BFV scale-and-round and method-I relinearization. The sizes of these computations, their memory-access schedules, and their execution geometry are determined by public parameters rather than by the ciphertext or evaluation-key values being processed.
- We compare JANUS and HEonGPU using the same repeated-execution L2 measurement procedure. We test whether differences between two key configurations appear again in an independent set of measurements and remain after the ciphertext is changed. JANUS shows fewer such key pairs than HEonGPU at both evaluated stages: 4 versus 9 during key switching and 2 versus 5 during divide-and-round. At $N = 8192$, JANUS completes the full multiplication-and-relinearization path in 0.350 ms, compared with 0.153 ms for HEonGPU.

2 Background and Threat Model

This section introduces the concepts needed to understand the design and evaluation of JANUS. We begin with the basics of BFV multiplication and relinearization, the computation targeted by our framework. We go on to introduce the algorithmic foundations used in JANUS's formulation of polynomial multiplication, from Kronecker substitution and large-integer multiplication to DGT-based convolution and Stockham transform organization. Finally, we discuss and define the threat model considered in this work.

2.1 BFV Arithmetic and Relinearization

Fully Homomorphic Encryption (FHE) enables arithmetic to be performed directly on encrypted data. In lattice-based schemes such as BFV [18], messages are encoded and encrypted as polynomials in the ring $\mathcal{R}_Q = \mathbb{Z}_Q[X]/(X^N + 1)$, where N is typically a power of two and Q is the ciphertext modulus. Ciphertexts are represented as tuples of polynomials in this ring. In high-performance implementations, the ciphertext modulus is commonly represented using a Residue Number System (RNS),

$$Q = \prod_{i=0}^{r-1} q_i,$$

so that polynomial arithmetic can be carried out independently over several machine-word-sized moduli q_i.

Multiplying two ciphertext tuples (c_0, c_1) and (d_0, d_1) results in a larger tuple (e_0, e_1, e_2). The three components are $e_0 = c_0 d_0, \quad e_1 = c_0 d_1 + c_1 d_0, \quad e_2 = c_1 d_1$. BFV then applies scale-and-round to the multiplication result, producing a three-component ciphertext whose components we denote by $(\tilde{e}_0, \tilde{e}_1, \tilde{e}_2)$. Relinearization operates on this scaled ciphertext.

In practical HE pipelines, allowing this increase in ciphertext size to accumulate would increase the cost of subsequent computation, so an operation known as relinearization is applied to map the expanded ciphertext back to its original form.

Relinearization uses a public evaluation key, called the relinearization key, to fold the additional scaled component $\tilde{e}_2$ back into the first two components:

$$(\tilde{e}_0, \tilde{e}_1, \tilde{e}_2) \xrightarrow{\text{rlk}} (c'_0, c'_1).$$

Relinearization keys may be reused multiple times in the same evaluation context.

2.2 Algorithmic Foundations of JANUS

JANUS builds on RNS reconstruction into fixed-width integers, Kronecker substitution and large-integer multiplication, DGT-based convolution, and Stockham transform organization. These foundations support the complete BFV pipeline described in Section 3.

RNS Reconstruction and Fixed-Width Representation. In an RNS representation, an integer coefficient a is represented by its residues $(a \bmod q_0,\ a \bmod q_1,\ \ldots,\ a \bmod q_{r-1})$, with respect to a set of pairwise coprime moduli $q_0, \ldots, q_{r-1}$. Let $Q = \prod_{i=0}^{r-1} q_i$. The Chinese Remainder Theorem uniquely determines the coefficient modulo Q, and a reconstruction procedure such as Garner's algorithm recovers this value from its residues.

Kronecker Substitution. Polynomial multiplication can be viewed as convolution over coefficient vectors. Let

$$A(x) = \sum_{i=0}^{n-1} a_i x^i \qquad \text{and} \qquad B(x) = \sum_{i=0}^{n-1} b_i x^i.$$

Their product is

$$C(x) = A(x)B(x) = \sum_{k=0}^{2n-2} c_k x^k,$$

where

$$c_k = \sum_{i+j=k} a_i b_j.$$

Kronecker substitution [49] converts this convolution into integer multiplication by placing the polynomial coefficients at fixed positions using a sufficiently large radix R:

$$A(R) = \sum_{i=0}^{n-1} a_i R^i, \qquad B(R) = \sum_{i=0}^{n-1} b_i R^i.$$

Multiplying the resulting integers gives

$$A(R)B(R) = \sum_{k=0}^{2n-2} c_k R^k.$$

If R is large enough to prevent adjacent coefficient values from overlapping, the polynomial coefficients can be recovered exactly from the integer product.

Large Integer Multiplication via Schönhage–Strassen. The Schönhage–Strassen algorithm [44] multiplies large integers by decomposing them into fixed-size digits, interpreting the digit sequences as polynomial coefficients, transforming them, performing pointwise multiplication, and applying the inverse transform followed by carry propagation. Compared with NTT-based multiplication performed separately over the RNS moduli, this produces a structured large-integer convolution whose dimensions are fixed by the operand representation, matching the regular execution structure pursued by JANUS.

Discrete Galois Transform-Based Convolution. Transform-based convolution maps coefficient sequences to a domain where convolution becomes pointwise multiplication; high-performance HE implementations commonly use the Number Theoretic Transform (NTT) for this purpose [1, 38, 41].

The *Discrete Galois Transform* (DGT), originally proposed for fast negacyclic integer convolution [16] and later adapted to GPU-based homomorphic-encryption arithmetic [3, 6], provides an alternative. The DGT operates over a quadratic extension such as $\mathbb{F}_{p^2}$, represented using Gaussian-integer pairs modulo a supporting prime p, and uses folding and twisting to realize negacyclic convolution modulo $x^N + 1$.

A DGT requires supporting primes, transform roots, and corresponding twiddle factors for the selected transform length. These parameters can be generated in advance and reused across repeated transform computations.

Stockham Transform Organization. Transform algorithms can differ in how intermediate values are arranged and moved through memory. Conventional in-place formulations such as Cooley–Tukey and Gentleman–Sande [21] repeatedly update the same working buffer and may require a separate bit-reversal permutation to obtain the desired output ordering.

The *Stockham* formulation [48] instead alternates between two buffers while progressively arranging the data into the required output order, avoiding a separate bit-reversal pass [20, 29]. Each stage follows a predetermined indexing rule, making the organization well suited to regular GPU execution [12, 25].

2.3 Microarchitectural Behavior on Shared GPUs

GPUs execute large numbers of threads concurrently using a Single Instruction, Multiple Thread (SIMT) model. Threads are grouped into warps and share several layers of hardware infrastructure, including caches, memory-system components, execution units, and scheduling resources. In shared or multi-tenant environments, workloads from different users may therefore interact with some of the same underlying microarchitectural resources.

The activity observed at this level can vary across executions. Differences may appear in timing, cache activity, memory traffic, contention, or other hardware events, even when the same high-level operation is being performed. These effects arise from the way a workload interacts with the GPU memory hierarchy and execution resources rather than from the cryptographic abstraction itself.

Prior work has shown that repeated observations of such GPU behavior can reveal recurring properties of workloads and execution contexts [5, 19, 28, 36, 50]. When these differences remain stable across repeated executions, they can form recognizable patterns that support workload or context fingerprinting. This motivates examining whether repeated GPU-based HE operations in an FHE-as-a-Service setting produce recurring microarchitectural behavior that could support metadata deanonymization through execution linkability.

2.4 Threat Model

We consider an FHE-as-a-Service setting in which workloads from mutually distrusting users may execute on the same physical GPU. The adversary is an unprivileged co-resident process that knows the public parameters and implementation of the HE scheme but does not modify the victim's inputs or kernels, inject faults, or interfere with correct execution. We do not consider plaintext recovery or secret-key recovery.

Our concern is metadata deanonymization through execution linkability. Relinearization keys are public evaluation keys and are commonly reused across many ciphertext operations within the same evaluation context. Thus, the same evaluation material may participate in computation repeatedly even as the ciphertext changes. If those executions consistently produce a similar microarchitectural pattern, an observer may be able to link otherwise opaque encrypted computations to the same recurring evaluation context or distinguish that context from previously observed ones.

A practical unprivileged cache observer operates through a substantially noisier channel than the privileged hardware counters available through NVIDIA Nsight Compute. We therefore use Nsight Compute as a high-fidelity diagnostic instrument rather than as an attacker capability. The profiler gives us a controlled

view of the underlying microarchitectural signal. For a given hardware signal, absence of recurring input-conditioned structure under this cleaner observation channel provides stronger evidence than measurements obtained through a noisier unprivileged channel.

We treat microarchitectural side-channel resilience as a structural implementation property. Relying on transient system noise to mask a reproducible input-conditioned hardware footprint is an engineering convenience rather than a security guarantee, because the amount of noise depends on the deployment environment and the observer's measurement capabilities. JANUS therefore aims to remove such execution structure at its source, independently of an adversary's ability to recover it from noisy observations.

3 JANUS-FHE Design and Implementation

3.1 JANUS Pipeline Overview

JANUS implements the complete BFV multiplication-and-relinearization path on the GPU. It takes two two-component ciphertexts $\mathbf{c} = (c_0, c_1)$ and $\mathbf{d} = (d_0, d_1)$ in RNS representation and returns a two-component relinearized ciphertext in the same representation. JANUS first reconstructs the ciphertext coefficients from the RNS basis and converts them to fixed-width base-2^{16} digits for Kronecker embedding. It then computes the BFV tensor-product components $e_0 = c_0 d_0$, $e_1 = c_0 d_1 + c_1 d_0$, and $e_2 = c_1 d_1$ using Stockham-organized DGT convolution.

After the inverse transform, JANUS performs carry reconstruction and BFV scale-and-round to obtain $(\tilde{e}_0, \tilde{e}_1, \tilde{e}_2)$ in the ciphertext RNS basis. Method-I relinearization then processes $\tilde{e}_2$ with the relinearization key, accumulates the key-switch products in the DGT domain, reconstructs the results, and performs the final divide-and-round by the special modulus P. The resulting corrections are added to $\tilde{e}_0$ and $\tilde{e}_1$, producing the final ciphertext (r_0, r_1). Figure 1 summarizes this complete pipeline.

3.2 RNS Reconstruction and Digitization

JANUS receives each ciphertext polynomial in the usual RNS representation. For a coefficient a, the input therefore consists of the residues

$$a_i = a \bmod q_i, \qquad 0 \le i < r,$$

under the ciphertext moduli $q_0, \ldots, q_{r-1}$. Before the coefficient can participate in the large-integer multiplication used by JANUS, these residues must be combined to reconstruct a modulo

$$Q = \prod_{i=0}^{r-1} q_i.$$

JANUS performs this conversion on the GPU using a Garner-style Chinese Remainder reconstruction. The residues are incorporated successively according to the fixed RNS basis until the coefficient has been reconstructed modulo Q. Since Q may be wider than a native machine word, the reconstructed coefficient is stored as a fixed-width multiword integer. The number of words required for this representation is determined from the public modulus set and is therefore the same for every coefficient processed under a given BFV parameter set. Modular products arising during reconstruction use Montgomery reduction [35], while modular corrections are expressed using masked arithmetic rather than value-dependent repetition.

The reconstructed coefficient is then decomposed into radix-2^{16} digits. With $\rho = 2^{16}$, JANUS represents each coefficient as

$$a = \sum_{j=0}^{L_{\mathrm{dig}}-1} a_j \rho^j, \qquad 0 \le a_j < \rho.$$

The digit count L_{dig} is selected from the maximum coefficient width implied by Q, rather than from the value of an individual coefficient. Every coefficient therefore occupies the same number of digit positions;

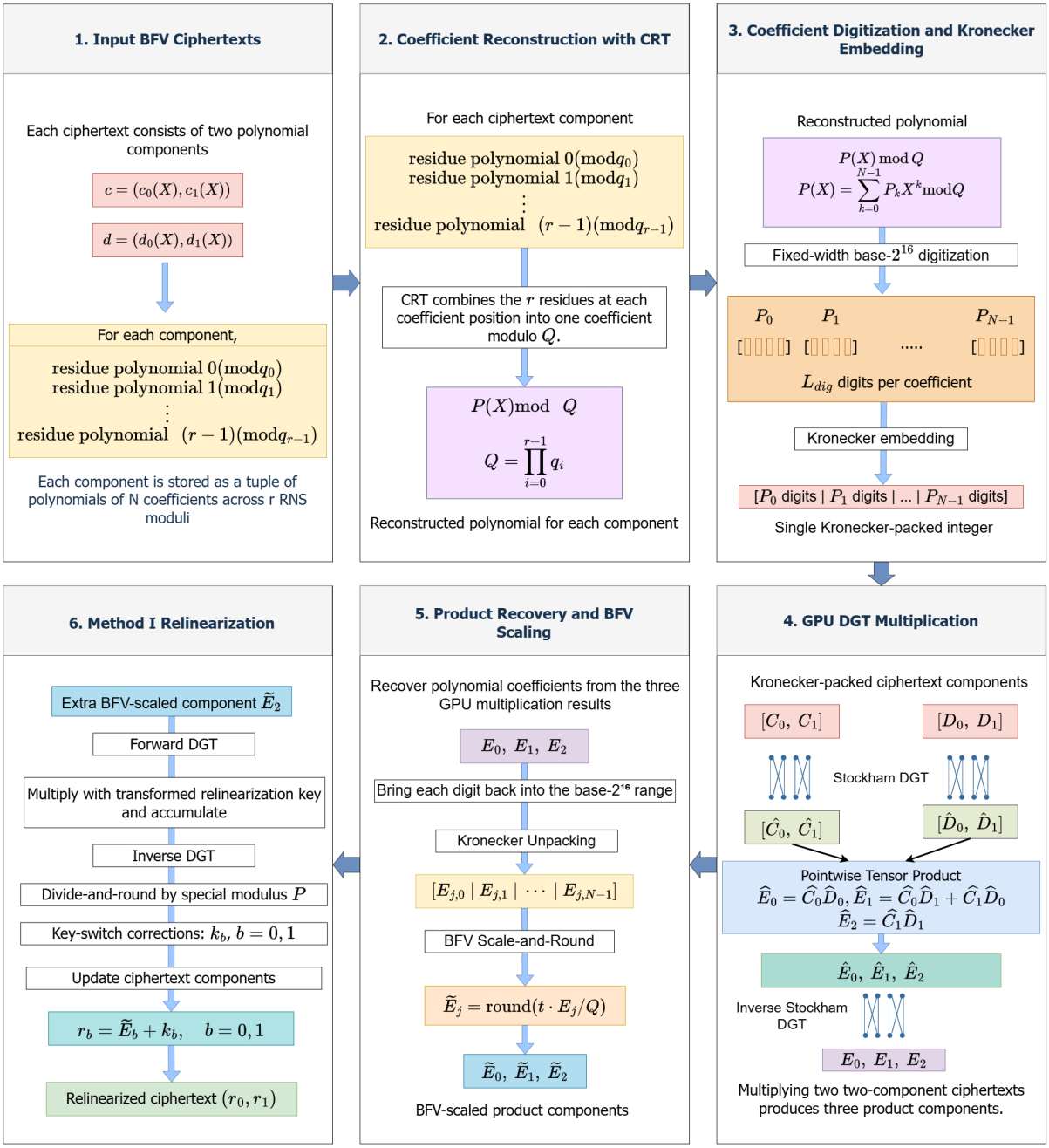


Fig. 1: End-to-end JANUS-FHE pipeline for BFV multiplication and relinearization. JANUS reconstructs the input ciphertext coefficients from their RNS representation, converts them to fixed-width digits for Kronecker embedding, and computes the three BFV tensor-product components using Stockham-organized DGT multiplication. The products are then recovered and scaled according to BFV before method-I relinearization produces the final two-component ciphertext.

coefficients with smaller numerical values are represented with zero-valued high-order digits instead of shorter arrays.

JANUS then places these fixed-radix digit representations into the layout used for Kronecker substitution. Each polynomial coefficient is assigned a fixed region large enough to accommodate the coefficient growth

produced during polynomial multiplication. The region width is selected from the public bounds on the coefficient size and polynomial degree, so adjacent coefficient products do not overlap when the digit sequences are convolved. At the end of this stage, each input polynomial has been converted from an array of RNS residues into a fixed-layout digit array ready for transform-based multiplication.

3.3 DGT-Based Tensor-Product Multiplication

JANUS computes three polynomial products required by BFV multiplication. The digit arrays are processed using the *Discrete Galois Transform* (DGT), which converts their convolution into pointwise multiplication in the transform domain. The DGT operates over the quadratic extension field $\mathbb{F}_{p^2}$, represented in our implementation using Gaussian-integer pairs modulo a supporting prime p. Folding and twisting are used to obtain the negacyclic convolution required for multiplication modulo $X^N + 1$.

JANUS applies the forward DGT to the digit representations of the four input ciphertext components. Let $\widehat{C}_0, \widehat{C}_1, \widehat{D}_0, \widehat{D}_1$ denote their transform-domain representations. The three BFV tensor-product components are then formed pointwise as

$$\widehat{E}_0 = \widehat{C}_0\widehat{D}_0, \quad \widehat{E}_1 = \widehat{C}_0\widehat{D}_1 + \widehat{C}_1\widehat{D}_0, \quad \widehat{E}_2 = \widehat{C}_1\widehat{D}_1.$$

The two products contributing to e_1 are accumulated in the transform domain before the inverse transform is applied.

For each supported transform length, JANUS uses supporting primes, transform roots, and twiddle factors selected in advance. These values depend on the public transform parameters and are reused across ciphertext operations. Their generation and preparation are described in the following subsection.

To map the DGT efficiently to the GPU, JANUS decomposes a large transform into smaller transforms that can be processed by individual thread blocks. The smaller transforms are applied across rows and columns, with predetermined twiddle-factor multiplications between them. This decomposition avoids requiring global synchronization within a transform stage and limits the amount of working data required by each thread block.

For each transform length, JANUS selects supporting primes p_i satisfying the required root conditions, verifies the root parameters h and ω, and precomputes the twiddle factors [3, 6]. The number of primes is determined by a public bound on the signed convolution coefficients: if their magnitudes are below $p/2$, one supporting prime suffices for exact recovery after the inverse DGT. All transform parameters are prepared in advance and reused across ciphertext operations.

The forward and inverse transforms use a Stockham organization. Each stage reads from one working buffer and writes to another, after which the two buffers exchange roles. Butterfly positions and twiddle-factor indices are determined by the transform stage, thread index, and public transform dimensions. The same stage structure is therefore executed for all ciphertext values under a given parameter set.

After pointwise multiplication, JANUS applies the inverse DGT to $\widehat{E}_0, \widehat{E}_1, \widehat{E}_2$. The resulting values represent the digit-level convolutions for the three tensor-product components. They are passed to the carry-reconstruction stage described next.

3.4 Reconstruction and BFV Scale-and-Round

After the inverse DGT, JANUS converts the three convolution results back into the fixed-radix representation used for BFV processing. The inverse-transform outputs are first returned from the DGT representation to the fixed-radix digit layout. Because the multiplication is negacyclic, the recovered convolution coefficients may be negative. JANUS therefore center-lifts each residue modulo the supporting DGT prime to recover its signed integer value. A public bias, chosen from the parameter bounds and as a multiple of Q, is then added so that every value is non-negative before carry propagation. The bias is selected to cover the maximum possible magnitude of a product coefficient under the chosen parameter set.

JANUS performs normalization through carry propagation. For radix $\rho = 2^{16}$, a value v_j at digit position j is separated into a normalized digit and a carry,

$$a_j = v_j \bmod \rho, \qquad \kappa_{j+1} = \left\lfloor \frac{v_j}{\rho} \right\rfloor .$$

The carry is added to the next digit position, and this process continues across the full fixed-width digit region. JANUS traverses the complete allocated width rather than terminating when the remaining carry becomes zero.

Once the digits have been normalized, they are regrouped according to the Kronecker layout established before multiplication. This recovers the integer coefficients of the three tensor-product components e_0, e_1, e_2.

JANUS then performs the BFV scale-and-round operation. Each reconstructed component is scaled by the plaintext modulus t and divided and rounded with respect to the ciphertext modulus Q. Conceptually,

$$\tilde{e}_j = \left\lfloor \frac{t\, e_j}{Q} \right\rceil , \qquad j \in \{0, 1, 2\}.$$

Because the bias added before carry propagation is a known multiple of Q, its contribution after scaling is known in advance and is removed when the result is returned to the ciphertext RNS basis. The resulting coefficients are then reduced under the ciphertext moduli $q_0, \ldots, q_{r-1}$. At the end of this stage, JANUS holds the three-component ciphertext $(\tilde{e}_0, \tilde{e}_1, \tilde{e}_2)$, which is then passed to relinearization.

3.5 Relinearization

The output of the multiplication stage is the three-component ciphertext $(\tilde{e}_0, \tilde{e}_1, \tilde{e}_2)$. JANUS next applies method-I relinearization to fold the third component $\tilde{e}_2$ back into the first two components. The relinearization key is a public evaluation key and contains the key-switching material required for this operation.

Relinearization begins from $\tilde{e}_2$ in the ciphertext RNS basis. JANUS separates this component across the ciphertext moduli, $q_0, \ldots, q_{r-1}$, so that each RNS component can be processed with the corresponding part of the relinearization key. The number of components processed at this stage is fixed by the ciphertext modulus set.

Each RNS component of $\tilde{e}_2$ is then converted into the representation required by the DGT and passed through the forward Stockham transform. The corresponding relinearization-key components are prepared in the transform domain in advance and reused across ciphertext operations within the same evaluation context. This avoids repeating the same key-side transform during every relinearization.

Let $\widehat{U}_i$ denote the transformed contribution from the i-th RNS component of $\tilde{e}_2$, and let $\widehat{\mathrm{RLK}}_{i,0}$ and $\widehat{\mathrm{RLK}}_{i,1}$ denote the corresponding transformed relinearization-key components. JANUS computes the two key-switching accumulations

$$\widehat{V}_0 = \sum_{i=0}^{r-1} \widehat{U}_i \widehat{\mathrm{RLK}}_{i,0},$$

and

$$\widehat{V}_1 = \sum_{i=0}^{r-1} \widehat{U}_i \widehat{\mathrm{RLK}}_{i,1}.$$

The products are accumulated directly in the transform domain, so the inverse transform is required only after each complete output accumulation has been formed.

JANUS then applies the inverse DGT to $\widehat{V}_0$ and $\widehat{V}_1$. The resulting values are returned through the same reconstruction process used after multiplication: the transform representation is reversed, the fixed-radix digits are reconstructed, and carries are propagated across the full allocated width to recover the corresponding integer coefficients.

As in multiplication recovery, the negacyclic convolution may produce negative coefficients. JANUS center-lifts the inverse-DGT residues and adds a public bias before carry propagation. For relinearization,

the bias is chosen as a multiple of QP, so after the final division by P, its contribution is a multiple of Q and vanishes when the result is reduced to the ciphertext RNS basis.

The reconstructed key-switching values contain the contribution of the special modulus P. JANUS therefore performs the final divide-and-round by P, producing two correction components, which we denote by k_0 and k_1. These corrections are added to the first two components of the scaled ciphertext:

$$r_0 = \tilde{e}_0 + k_0, \qquad r_1 = \tilde{e}_1 + k_1,$$

with the results reduced back into the ciphertext RNS basis.

The output is therefore the standard two-component BFV ciphertext (r_0, r_1). For a fixed parameter set, the number of RNS components, transform stages, evaluation-key products, accumulation terms, and reconstruction steps is determined before ciphertext processing begins; the ciphertext and evaluation-key values supply the arithmetic operands within this fixed structure.

4 Case Study: Execution Regularity and Microarchitectural Leakage in HEonGPU

We examine BFV relinearization type I in HEonGPU [39], a high-performance GPU implementation of homomorphic encryption. We use this case study to understand how a conventional GPU HE implementation behaves when ciphertext-derived values and evaluation-key material move through the different steps of relinearization, namely key switching, modular arithmetic, and rounding. We examine both the execution of this path and the microarchitectural variation that appears when the same computation is repeated under different input configurations.

We begin with the implementation itself. We trace the relinearization path through the relevant CUDA kernels and inspect how the corresponding source code is translated into GPU machine instructions. This allows us to examine whether different processed values change the control flow or the amount of work executed by the GPU. We then profile repeated executions using different ciphertext and relinearization-key configurations and characterize the resulting GPU behavior. These measurements establish the HEonGPU baseline used later in our comparative evaluation of JANUS in Section 5.

4.1 HEonGPU Relinearization Pipeline

After BFV multiplication and scale-and-round, HEonGPU represents the three-component ciphertext as (c_0, c_1, c_2), corresponding to the scaled multiplication result introduced in Section 2.1. A standard ciphertext contains only two components. HEonGPU removes the additional component c_2 using its type-I key-switching procedure, which follows the same general approach as SEAL-based relinearization [40].

The relinearization begins by decomposing c_2 across the RNS basis and transforming the resulting components into the NTT domain. HEonGPU then invokes `keyswitch_multiply_accumulate_kernel`. For each decomposed component of c_2, the kernel multiplies it by the corresponding component of the relinearization key and accumulates the products into the intermediate buffer `temp2_relin`. This buffer therefore contains the result of the main key-switching computation before the final modulus conversion.

HEonGPU next applies an inverse NTT to `temp2_relin`, returning the accumulated values to the coefficient domain. The resulting coefficients are then processed by `divide_round_lastq_kernel`. This kernel performs the division and rounding required to remove the additional modulus introduced during key switching and folds the resulting values into the first two ciphertext components. The output is again a two-component ciphertext.

These two kernels capture the main stages that we examine in the remainder of this case study. The key-switch multiply-and-accumulate kernel processes c_2 together with the relinearization key, while the divide-and-round kernel processes the intermediate values produced by that computation. Within `divide_round_lastq_kernel`, HEonGPU uses the helper function `reduce_forced` to reduce intermediate coefficients modulo the current RNS modulus.

4.2 HEonGPU Source Code and SASS Analysis

We begin with `reduce_forced`, which is called inside `divide_round_lastq_kernel` when HEonGPU reduces an intermediate coefficient modulo the current RNS modulus. Its implementation is shown below.

```
static __device__ __forceinline__ T1 reduce_forced(
    const T1& input, const Modulus<T1>& modulus) {
    T1 result = input;
    while (result >= modulus.value)
        result = reduce(result, modulus);
    return result;
}
```

Listing 1.1: Forced modular reduction in HEonGPU.

At the CUDA source level, this function allows the amount of work to depend on the value being reduced. The loop continues while `result` remains larger than the modulus, so an input that requires another reduction can execute another iteration. This makes the source code worth examining further, but it does not by itself tell us what instructions ultimately execute on the GPU.

We therefore inspect the SASS generated for the optimized HEonGPU release build used in our experiments. SASS is the native instruction representation executed by the NVIDIA GPU after compilation. In the release build, the compiler transforms the value-dependent source construct into a branchless, predicated instruction sequence. The resulting binary therefore has a regular control-flow structure even though that regularity is not explicit in the CUDA implementation. Therefore, HEonGPU does not enforce this behavior at the source level; instead, the regular execution structure results from the compiler's optimization of the particular build we evaluate. We refer to this as *compiler-dependent regularity*: the compiled program can exhibit regular execution even when the corresponding source code permits value-dependent work.

We confirm this behavior by measuring the number of instructions executed by the principal kernels in the relinearization path. We evaluate six relinearization-key configurations with four ciphertexts, giving 24 key–ciphertext combinations. The dynamic instruction count remains identical across all 24 combinations for each measured kernel.

The instruction-count measurements therefore agree with the SASS analysis: in the optimized release build, changing the tested ciphertext or relinearization-key configuration does not change the amount of dynamic instruction work performed by these kernels.

However, this regularity is not guaranteed by the CUDA source itself. When the same source is compiled under an alternate debug-oriented configuration, the reduction condition is lowered to explicit conditional control flow. The execution structure of the same source code can therefore change with the compiler configuration. The HEonGPU case study section thus exposes a distinction between *regularity by construction* and *regularity by compilation*. In the evaluated release binary, compiler optimization produces regular control flow and invariant instruction counts. The source implementation itself does not require this behavior, making the resulting regularity dependent on how the code is compiled.

4.3 Execution Profiling of HEonGPU Relinearization

We next examine how HEonGPU's relinearization path behaves across repeated GPU executions, using hardware measurements to capture behavior that is not visible from the compiled instruction stream alone. We profile repeated relinearization workflow under different ciphertext and relinearization-key configurations and record the resulting execution activity for the kernels in this path.

Target Kernel. Our analysis focuses on `divide_round_lastq_kernel`, which processes the intermediate produced by key switching and the inverse NTT before the result is folded back into the two output ciphertext components. We profile this kernel using NVIDIA Nsight Compute and record the hardware counter `lts__t_sectors.sum`. This counter measures the number of sectors processed by the GPU's L2-cache subsystem during one kernel invocation. The L2 cache is the last-level cache shared across the GPU and services memory requests generated by executing kernels before accesses proceed to device memory. The counter

therefore provides a view of memory-system activity during relinearization that is not captured by the control-flow and instruction-count analysis above.

Each profiling run executes the same relinearization workflow 20 times in succession; we refer to each invocation of `divide_round_lastq_kernel` within that run as a *launch*. The L2-sector count is recorded separately for every launch.

Data collection. We construct a profiling dataset containing 32 relinearization-key configurations and four ciphertexts. Each launch loads one selected key configuration and one selected ciphertext and executes the same relinearization workflow while Nsight Compute records the kernel-level counter.

For every key–ciphertext configuration, we collect 100 profiling repetitions, with 20 launches recorded in each repetition. The complete dataset therefore contains $32 \times 4 \times 100 \times 20 = 256{,}000$ measured launches for the target kernel.

The four ciphertexts are generated from the plaintext patterns below. They are not intended to span the full plaintext space; instead, they provide a small set of distinct input organizations under which the same relinearization-key configurations can be measured.

- **Ciphertext class 0:** The same value repeated throughout, except for a few different values at the beginning.
- **Ciphertext class 1:** The same value repeated throughout, except for a few changes at scattered positions.
- **Ciphertext class 2:** A short increasing sequence at the beginning, followed by the same repeated value.
- **Ciphertext class 3:** A short alternating sequence at the beginning, followed by the same repeated value.

Interleaved profiling. The profiling runs are collected using an interleaved schedule. Rather than completing all repetitions for one key–ciphertext pair before moving to the next, the collection alternates among pairs throughout the experiment. This distributes longer-term effects such as thermal drift, dynamic voltage and frequency scaling (DVFS), and other temporal GPU variation across the dataset instead of allowing them to become associated with whichever configuration is measured earlier or later.

Each recorded measurement retains the corresponding key configuration, ciphertext class, repetition, and launch position. The resulting dataset therefore supports both aggregate analysis and comparison of the repeated-execution profiles associated with different experimental configurations.

Observed variations in L2 activity. The collected measurements show variation in the L2 activity of `divide_round_lastq_kernel`. Most launches cluster around a dominant value of approximately 70,581 L2 sectors, while a smaller fraction produce higher values. Across the complete dataset, approximately 13.72% of launches lie above this dominant value. We refer to these measurements as *above-baseline* observations.

The overall above-baseline rate remains similar across the four ciphertext classes. Ciphertext classes 0 through 3 produce rates of 13.67%, 13.68%, 13.87%, and 13.63%, respectively. We also observe differences among individual relinearization-key configurations. Representative key profiles K0, K12, K18, K24, and K27 produce aggregate above-baseline rates of 12.99%, 13.36%, 13.19%, 13.96%, and 13.79%, respectively.

These aggregate rates provide an initial description of the measured variation, but they do not capture how the observations are distributed across the repeated launch sequence. For each launch position, the dataset records how often its L2-sector count rises above the dominant value across repeated profiling runs. We retain these position-wise profiles for the comparative analysis in Section 5.

The measurements therefore establish that repeated executions of the evaluated HEonGPU relinearization path exhibit measurable variation in L2-cache activity. The analysis in Section 5 examines whether differences in these profiles reproduce across independent measurements and persist when the ciphertext changes, before comparing the corresponding behavior with JANUS.

5 Evaluation

The HEonGPU case study motivates two complementary questions for evaluating GPU-based HE implementations: whether the compiled program preserves a regular execution structure, and whether repeated

executions nevertheless produce distinguishable microarchitectural behavior. In the optimized build used in our experiments, the compiler transforms the source-level value-dependent reduction into a branchless, predicated instruction sequence. Under an alternate earlier compiler build, however, the same source produces explicit conditional control flow. The regularity observed in the release binary is therefore dependent on the compilation configuration rather than enforced by the source implementation. At the hardware level, repeated executions of the same release binary still exhibit measurable variation in L2-cache activity.

We evaluate JANUS along these two dimensions. We first examine whether the regular execution structure intended by its design is preserved in the compiled GPU implementation. We then return to the L2 measurements introduced in the case study and compare JANUS with HEonGPU under matched repeated-execution experiments. Finally, we evaluate the execution cost of the JANUS multiplication-and-relinearization pipeline.

5.1 Experimental Setup

All experiments are conducted on a single NVIDIA GeForce RTX 5090 GPU with 32 GB of device memory. The system uses NVIDIA driver version 580.97 with CUDA 13.0. Both JANUS and HEonGPU are evaluated on the same GPU, and no concurrent compute workload is executed during measurement. GPU kernels are compiled with `nvcc`, using `-O3` optimization and C++17 support.

JANUS supports polynomial degrees $N \in \{2^{12}, 2^{13}, 2^{14}, 2^{15}, 2^{16}, 2^{17}, 2^{18}\}$, with corresponding RNS prime counts $r \in \{2, 3, 7, 14, 29, 58, 117\}$. The individual ciphertext-modulus primes range from 40 to 59 bits. The comparative experiments in this section use $N = 8192$, $r = 3$, and plaintext modulus $t = 1{,}032{,}193$.

We use NVIDIA Nsight Compute as the diagnostic profiling instrument for the hardware-counter experiments. The profiler provides a high-fidelity view of kernel-level L2 activity and is used as a controlled measurement mechanism rather than as an observation capability available to the unprivileged attacker. This intentionally gives our experiments a cleaner observation channel than a practical co-resident cache attack. For the comparative experiments, measurements are collected repeatedly under multiple ciphertext and key-material configurations, with the collection procedure described alongside the corresponding L2 analysis below.

The evaluation proceeds from implementation structure to measured execution behavior. We first examine the memory-access schedule and compiled control flow of JANUS, establishing whether the regularity intended by the architecture is preserved after compilation. We then compare the systems using repeated L2 measurements and discuss performance.

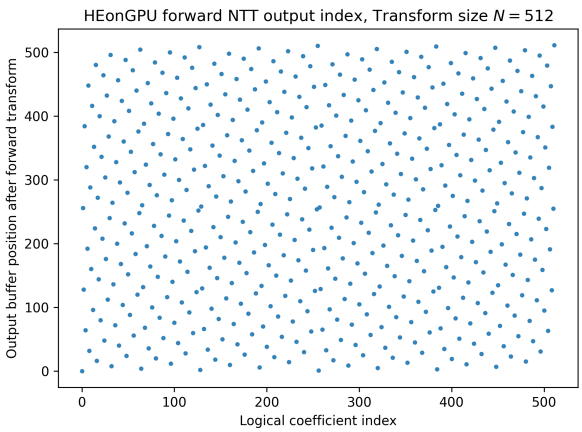
5.2 Execution Regularity of JANUS Transform Schedules

JANUS is designed so that the structure of the computation is determined by public parameters rather than by the ciphertext or evaluation-key values being processed. We first examine this property in the transform domain, where a substantial portion of the multiplication workload is carried out.

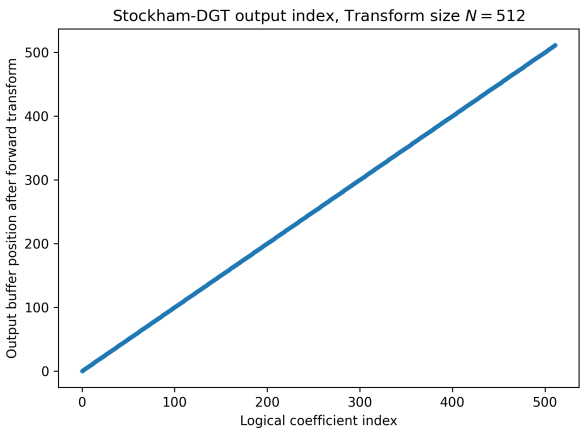
HEonGPU and JANUS both use staged transform computations, but they organize data movement differently. HEonGPU uses an in-place Cooley–Tukey NTT. As the transform progresses, each stage reads and writes the same working buffer under a stage-dependent indexing pattern, and a bit-reversal permutation is used to obtain the required output ordering. The locations accessed at this stage are therefore determined by the transform parameters and stage number.

JANUS instead performs its DGT using a Stockham-style schedule. Each stage reads from one buffer and writes to a second buffer, after which the roles of the buffers are exchanged. The indexing rule for each stage is determined from the thread index, stage number, and public transform dimensions, not from the values stored at those locations. The Stockham organization also performs the required reordering as part of the transform and therefore does not require a separate bit-reversal pass.

Figure 2 illustrates the difference in transform organization. The HEonGPU NTT exhibits the permutation structure associated with its in-place Cooley–Tukey schedule, whereas the JANUS Stockham-DGT produces its output through its alternating-buffer organization that performs the required reordering during the transform. The comparison shows how JANUS expresses its transform as a fixed sequence of stages whose data movement can be determined entirely from the public transform configuration.



(a) HEonGPU forward NTT output organization under the in-place Cooley–Tukey schedule.



(b) JANUS forward Stockham-DGT output organization under the alternating-buffer schedule.

Fig. 2: Output organization of the forward transforms. The two implementations use different transform schedules and data layouts. JANUS uses an autosorting Stockham organization in which the stage structure and index calculations are fixed by the public transform configuration.

We additionally examine the twiddle-table accesses generated by the two schedules. As shown in Figure 3, the HEonGPU NTT exhibits a stage-dependent twiddle-access pattern as its in-place schedule progresses, while the JANUS Stockham-DGT follows the predetermined twiddle-access pattern of its fixed stage organization. These differences reflect the respective transform algorithms; in JANUS, the important property is that the accessed twiddle locations are determined by the stage and transform dimensions rather than by the values being transformed.

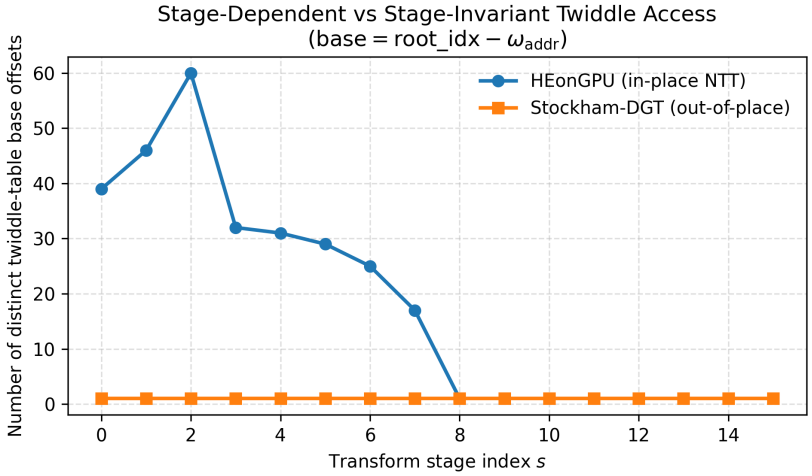


Fig. 3: Stage-wise twiddle-table access structure for the two transform implementations. JANUS follows a fixed Stockham access schedule determined by the public transform configuration.

The same design principle extends beyond the transform itself. The digit representation, transform dimensions, loop bounds, and kernel launch geometry used by JANUS are determined by polynomial degree, modulus set, digit width, and other public configuration parameters. Ciphertext and evaluation-key values determine the operands processed by these stages, but not the number of stages or the dimensions over which they execute. We next examine whether this source-level regularity remains present after CUDA compilation.

5.3 SASS-Level Control-Flow Analysis of JANUS Kernels

The fixed schedules described above establish regularity at the source and algorithmic levels, but we also need to determine whether that structure is preserved in the executable code. As the HEonGPU case study demonstrated, CUDA source and compiled GPU instructions need not have the same control-flow structure. We therefore inspect the SASS generated for JANUS and examine how its masked arithmetic and execution conditions are represented in the compiled binary.

We compile the kernels into native device binaries and disassemble them using `nvdisasm`. For each examined kernel, we inspect explicit control-transfer instructions such as `BRA`, predicate-generating instructions such as `ISETP`, predicated selection instructions including `SEL` and `SELP`, and trap instructions such as `BPT.TRAP`. When a branch is present, we trace the predicate that controls it to determine whether the branch condition is derived from processed data or from public execution parameters.

Our audit includes representative kernels from the JANUS transform and arithmetic pipeline: the forward Stockham transform (`k_stockham_fwd_gf2_mont`), fold-and-twist (`k_fold_twist_gf2_mont`), pointwise multiplication (`k_pointwise_gf2_mont`), and transform scaling (`k_gf2_scale_mont`). These kernels exercise the principal arithmetic and data-movement operations used by the DGT pipeline.

Across the audited kernels, arithmetic corrections are implemented in the compiled code through predicated operations rather than value-dependent control transfers. Comparisons generate predicates that affect the value selected by subsequent instructions without changing the path of execution.

Branches are present in the compiled code, but the branches we observe are associated with execution conditions that are independent of the processed ciphertext or evaluation-key values. These include thread bounds checks derived from the launch dimensions and compiler-generated guards associated with the generated binary. We do not observe a control-transfer instruction in the audited arithmetic whose predicate is derived from the value being processed.

This differs from the situation observed in HEonGPU. There, the CUDA implementation of `reduce_forced` explicitly permits value-dependent repetition, while the compiler transforms this into a branchless, predicated instruction sequence in the optimized release build. An alternate build can lower the same source construct to explicit conditional control flow. JANUS instead expresses its arithmetic corrections using fixed-bound computation and masked selection at the source level. For the kernels examined here, the compiled SASS preserves that intended execution structure.

The SASS analysis therefore provides the compiled-code counterpart to the structural analysis above. JANUS does not rely on a particular compiler optimization to remove an iterative value-dependent source construct in these kernels. The next experiment asks a separate question: even when the execution structure is regular at both the source and compiled-instruction levels, do repeated executions still exhibit configuration-dependent variation in L2-cache activity?

5.4 Measuring Variation of L2 Behavior in HEonGPU and JANUS

We next examine whether different key-material configurations produce recurring differences in L2-cache activity, and whether those differences remain observable when the ciphertext changes. We perform the same analysis for HEonGPU and JANUS so that the two implementations are evaluated under the same measurement procedure.

We examine two corresponding stages of relinearization. For key switching, we compare HEonGPU's `keyswitch_multiply_accumulate_kernel` with JANUS `k_bfv_keyswitch_acc`. For the subsequent divide-and-round stage, we compare HEonGPU's `divide_round_lastq_kernel` with the final `k_bfv_divround` in JANUS. Although the internal implementations differ, these kernels perform the corresponding stages of the two relinearization pipelines.

For both systems, we record the Nsight Compute counter `lts__t_sectors.sum`. We evaluate 32 key-material configurations using ciphertext classes 0 and 3. For each key–ciphertext pair, we collect 72 profiling repetitions. Each repetition contains 20 launches of the measured kernel, with the L2-sector count recorded for every launch.

An L2 difference between two keys is meaningful only if it repeats across independent measurements. We therefore divide the 72 repetitions for each key–ciphertext pair into two non-overlapping sets of 36. Within each set, we compute the mean L2-sector count separately for each of the 20 launch positions. This gives two independently measured 20-launch L2 profiles for the same key–ciphertext pair.

For a fixed ciphertext, we then compare the 32 key-material configurations two at a time; we refer to each such comparison as a *key pair*. For each key pair, we compare the difference between the two keys across the 20 launch positions in the first set of measurements with the corresponding difference in the second set. We use Spearman correlation to measure how well the same difference repeats. A higher positive correlation indicates that the relative L2 behavior of the two keys recurs across the two independent measurement sets.

We perform this test separately under ciphertext 0 and ciphertext 3. We retain a key pair when its correlation satisfies

$$\rho \geq 0.3$$

for both ciphertexts. Thus, the same difference between two keys must repeat across independent measurements under ciphertext 0 and continue to repeat after the ciphertext changes to ciphertext 3.

5.5 L2 Behavior Across Keys and Ciphertexts

We now apply the analysis above to HEonGPU and JANUS. Across the 32 keys, there are 496 possible key pairs for each measured stage. We test each key pair under ciphertext 0 and ciphertext 3 and retain it only when the correlation satisfies $\rho \geq 0.3$ for both ciphertexts.

To characterize the retained-pair counts under randomized key association, we repeat the analysis over 100,000 random shuffles of the key labels. Shuffling removes the consistent association between a particular key and its measured L2 behavior. We report the mean of the number of retained pairs across the shuffled trials.

For key switching, HEonGPU retains 9 key pairs in the measured data compared with a shuffled mean of 4.91 pairs. JANUS retains 4 key pairs, compared with a shuffled mean of 4.68 pairs. For divide-and-round, HEonGPU retains 5 key pairs, compared with a shuffled mean of 4.80 key pairs. JANUS retains 2 key pairs, compared with a shuffled mean of 4.90 pairs.

Overall, HEonGPU retains more recurring key pairs than JANUS at both measured stages. During key switching, HEonGPU retains 9 pairs, placing it above its randomized mean of 4.91. JANUS successfully suppresses this recurring behavior, retaining only 4 pairs, which drops below its corresponding shuffled mean. This diagnostic confirms that JANUS effectively bounds the data-dependent microarchitectural variance within the expected random noise floor.

5.6 Performance

We measure the latency of BFV multiplication followed by relinearization on the same RTX 5090 used throughout the evaluation. We use $N = 8192$, three ciphertext moduli, and plaintext modulus $t = 1{,}032{,}193$. For JANUS, the measured path begins with ciphertext residues, includes the conversion to the fixed-width digit representation used by the DGT pipeline, and ends with the two-component relinearized ciphertext. CUDA graph capture is used for the JANUS pipeline.

Across five independent measurements, JANUS requires an average of 0.350 ms for the complete multiplication-and-relinearization operation. HEonGPU requires an average of 0.153 ms under the corresponding BFV workload. JANUS thus incurs approximately 2.29× the execution time of HEonGPU under these conditions.

To reduce the overhead of its regular execution structure, JANUS incorporates three optimizations. First, JANUS derives a tight bound on the digit-level convolution coefficients to minimize the number of supporting DGT primes. For the evaluated parameter set, the bound fits within a single 63-bit supporting prime ($K = 1$), allowing the complete Stockham DGT convolution to execute without a multi-prime CRT basis. This eliminates the corresponding CRT reconstruction and avoids repeating the forward transforms, pointwise products, and inverse transforms across multiple supporting primes. Second, carry propagation is implemented as a branchless parallel prefix scan rather than a serial carry chain, allowing the GPU to

process carry dependencies in parallel while preserving a fixed execution schedule. Finally, the complete multiplication-and-relinearization path is captured in a single CUDA graph, reducing host-side launch overhead across the sequence of GPU kernels.

The remaining overhead is primarily a consequence of how JANUS enforces regular execution. Its DGT-based multiplication is organized as a sequence of fixed Stockham transform stages, with data movement and execution structure determined by public parameters rather than by the values being processed. This requires more transform work and more short kernel launches than HEonGPU's highly optimized NTT-based pipeline, but it also gives JANUS a substantially more explicit and predictable execution structure. The resulting 2.29× latency therefore represents a practical performance trade-off for stronger execution regularity: the complete multiplication-and-relinearization operation remains well below one millisecond while avoiding reliance on compiler-dependent regularization and reducing the recurring L2 differences observed in our comparative measurements.

5.7 Summary of Results

Our evaluation shows that JANUS differs from HEonGPU both in how execution regularity is established and in the recurring microarchitectural behavior observed under repeated measurement. In HEonGPU, the optimized release binary exhibits regular control flow because the compiler transforms a source-level value-dependent reduction into predicated instructions; an alternate build can produce explicit conditional control flow from the same source. JANUS instead expresses its transform schedule and arithmetic corrections through fixed, parameter-determined execution structure, and the audited SASS preserves this organization after compilation.

Under the same cross-ciphertext L2 analysis, HEonGPU retains 9 key pairs during key switching and 5 during divide and round, whereas JANUS retains 4 and 2, respectively. The clearest departure from the shuffled baseline occurs in HEonGPU key switching, while the JANUS counts remain at or below their corresponding shuffled baselines. These measurements do not imply that JANUS eliminates all microarchitectural variation, but they show that fewer recurring key-pair differences remain under our diagnostic.

This stronger execution regularity incurs a measurable performance cost. At $N = 8192$, JANUS completes multiplication and relinearization in 0.350 ms, compared with 0.153 ms for HEonGPU, corresponding to a 2.29× latency overhead. This overhead reflects the more explicit, parameter-determined execution structure of the JANUS pipeline, while the complete operation remains sub-millisecond.

6 Related Work

Homomorphic encryption (HE) allows computation directly on encrypted data, and modern lattice-based schemes such as BFV [18], BGV [11], and CKKS [13] have made it practical for a growing range of applications. Their security is based on hard lattice problems, but performance rests on heavy polynomial arithmetic (e.g., high-degree multiplication and key switching) that dominates HE workloads. A substantial body of work reduces this cost algorithmically through residue-number-system (RNS) decomposition [9], fast modular reduction [35], and transform-based polynomial multiplication built on the Number-Theoretic Transform (NTT) [15, 33]. These techniques form the computational backbone shared by essentially all high-performance HE implementations. These algorithms are massively parallel, and so GPUs have become an attractive platform for HE, with particular emphasis on polynomial multiplication and key switching. GPU-based implementations of BFV, BGV, and CKKS schemes achieve substantial performance gains by exploiting data parallelism and fast modular arithmetic [8, 17, 30, 42, 47]. However, the majority of this work prioritizes throughput and latency, paying little attention to constant-time execution and side-channel resilience at the microarchitectural level.

HEonGPU represents one of the most mature GPU-accelerated HE libraries, providing optimized CUDA implementations of core operations including NTT-based multiplication, relinearization, and modulus switching. Other GPU frameworks, such as Cheddar [14] and FIDESlib [2], explore similar design spaces with an emphasis on throughput and scalability. These systems demonstrate the performance potential of GPUs

for HE, but execution regularity and recurring microarchitectural behavior are not their primary design objectives. Our work considers these properties explicitly and studies how a high-performance GPU HE implementation behaves across repeated executions.

A distinct line of work rethinks polynomial multiplication itself. PolyFHEmus [24] reduces polynomial multiplication to large-integer multiplication through Kronecker substitution [26] and uses the Schönhage–Strassen algorithm [44] together with the Discrete Galois Transform (DGT). This work motivates our use of large-integer arithmetic for HE polynomial multiplication. JANUS, however, develops a different end-to-end GPU pipeline around this direction: it begins from ciphertexts in RNS representation, performs fixed-width reconstruction and DGT-based tensor-product multiplication, and carries the computation through BFV scale-and-round and method-I relinearization with execution regularity as a design objective.

The DGT has also been studied more broadly as an alternative to the NTT for HE polynomial arithmetic on GPUs [3, 6]. These works show how DGT-based convolution can be mapped efficiently to GPU architectures and explore transform organizations intended to improve performance. JANUS builds on this algorithmic foundation but uses a Stockham-organized DGT together with fixed-width representations, fixed execution geometry, and branchless masked arithmetic so that the structure of the computation is determined by public parameters rather than by the values being processed.

A separate body of work has shown that shared execution hardware can expose information through timing, cache activity, contention, and performance counters. Ristenpart et al. [43] demonstrated cross-tenant side channels in shared cloud infrastructure, while GPU studies have shown that repeated hardware observations can reveal information about workloads and execution contexts. Naghibijouybari et al. [36] used GPU performance counters and contention to infer properties of co-resident workloads, and Jiang et al. [27] demonstrated cryptographic leakage through GPU timing. Related side-channel work has likewise shown how timing and cache behavior can expose information that is not visible at the algorithmic level [10, 23, 31].

JANUS lies at the intersection of these two lines of work. Rather than treating GPU HE only as a throughput problem, we examine whether repeated HE executions produce recognizable microarchitectural behavior and design the underlying arithmetic pipeline so that its execution structure is explicit and parameter-determined. Our HEonGPU case study uses repeated shared-L2 measurements to characterize differences associated with relinearization-key configurations, while JANUS restructures the complete BFV multiplication-and-relinearization path around fixed-width computation and regular GPU execution.

7 Conclusion

This work examines the microarchitectural behavior of GPU-accelerated homomorphic encryption and explores execution regularity as a design objective for GPU HE implementations. Our case study of HEonGPU BFV relinearization shows that repeated executions can exhibit reproducible differences in shared-L2 cache activity across relinearization-key configurations. We also find that the regular control flow observed in the optimized HEonGPU binary is not enforced directly by the source implementation, illustrating the importance of examining both implementation structure and compiled execution behavior.

We presented JANUS-FHE, an end-to-end GPU framework for BFV multiplication and relinearization designed around a fixed, parameter-determined execution structure. JANUS reconstructs RNS coefficients into a fixed-width representation, performs Kronecker-based large-integer multiplication using a Stockham-organized DGT, and carries this structure through product recovery, BFV scale-and-round, and method-I relinearization using fixed-width computation and branchless masked arithmetic. SASS-level analysis confirms that the intended execution structure is preserved in the audited kernels.

Under the same repeated-execution shared-L2 methodology, JANUS retains fewer recurring key-pair differences than HEonGPU at both evaluated relinearization stages. These results do not imply that JANUS eliminates all microarchitectural variation, but they show that restructuring the computation around regular execution can reduce the recurring differences observed in our experiments. At $N = 8192$, JANUS completes the full multiplication-and-relinearization path in 0.350 ms, compared with 0.153 ms for HEonGPU, corresponding to a 2.29× latency overhead. Overall, our results show that execution regularity can be treated as an explicit design dimension in GPU-based HE alongside correctness and performance.

Acknowledgment

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

References

1. S. Adhikary et al. cuTraNTT: GPU-based transposed number theoretic transform for homomorphic encryption. Cryptology ePrint Archive, Paper 2024/1729, 2024.
2. Carlos Agulló-Domingo et al. FIDESlib: A fully-fledged open-source FHE library for efficient CKKS on GPUs. arXiv:2507.04775, 2025.
3. Pedro Geraldo M. R. Alves, Jheyne N. Ortiz, and Diego F. Aranha. Faster homomorphic encryption over GPGPUs via hierarchical DGT. *Cryptology ePrint Archive*, 2020:752, 2020.
4. Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. A view of cloud computing. *Communications of the ACM*, 53(4):50–58, 2010.
5. Furkan Aydin, Emre Karabulut, Seetal Potluri, Erdem Alkim, and Aydin Aysu. RevEAL: Single-trace side-channel leakage of the SEAL homomorphic encryption library. Cryptology ePrint Archive, Report 2022/204, 2022.
6. Ahmad Al Badawi, Bharadwaj Veeravalli, and Khin Mi Mi Aung. Efficient polynomial multiplication via modified discrete galois transform and negacyclic convolution. In *Future of Information and Communication Conference*, volume 886 of *Advances in Intelligent Systems and Computing*, pages 666–682. Springer, 2019.
7. Ahmad Al Badawi, Bharadwaj Veeravalli, Jinguang Lin, Nan Xiao, Kazuaki Matsumura, and Khin Mi Mi Aung. Multi-GPU design and performance evaluation of homomorphic encryption on GPU clusters. *IEEE Transactions on Parallel and Distributed Systems*, 32(2):379–391, 2021.
8. Ahmad Al Badawi, Bharadwaj Veeravalli, Chan Fook Mun, and Khin Mi Mi Aung. High-performance FV somewhat homomorphic encryption on GPUs: An implementation using CUDA. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 70–95, 2018.
9. Jean-Claude Bajard, Julien Eynard, Anwar Hasan, and Vincent Zucca. A full RNS variant of FV-like somewhat homomorphic encryption schemes. Cryptology ePrint Archive, Report 2016/510, 2016.
10. Daniel J. Bernstein. Cache-timing attacks on AES. Preprint, 2005.
11. Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (leveled) fully homomorphic encryption without bootstrapping. In *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*, pages 309–325, 2012.
12. Liangyu Chen, Svyatoslav Covanov, Davood Mohajerani, and Marc Moreno Maza. Big prime field FFT on the GPU. In *Proceedings of the 2017 ACM on International Symposium on Symbolic and Algebraic Computation*, ISSAC '17, pages 85–92. ACM, 2017.
13. Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. Homomorphic encryption for arithmetic of approximate numbers. In *Advances in Cryptology – ASIACRYPT 2017*, volume 10624 of *Lecture Notes in Computer Science*, pages 409–437. Springer, 2017.
14. Wonseok Choi, Jongmin Kim, and Jung Ho Ahn. Cheddar: A swift fully homomorphic encryption library designed for GPU architectures. In *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 35–49. ACM, 2026.
15. James W. Cooley and John W. Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematics of Computation*, 19(90):297–301, 1965.
16. Richard E. Crandall. Integer Convolution via Split-Radix Fast Galois Transform. Technical report, Center for Advanced Computation, Reed College, 1999.
17. Wei Dai and Berk Sunar. cuHE: A homomorphic encryption accelerator library. In *Cryptography and Information Security in the Balkans (BalkanCryptSec 2015)*, volume 9540 of *Lecture Notes in Computer Science*, pages 169–186. Springer, 2015.
18. Junfeng Fan and Frederik Vercauteren. Somewhat practical fully homomorphic encryption. Cryptology ePrint Archive, Report 2012/144, 2012.
19. Ethan Ferguson, Adam Wilson, and Hoda Naghibijouybari. WebGPU-SPY: Finding fingerprints in the sandbox through GPU cache attacks. In *Proceedings of the 19th ACM Asia Conference on Computer and Communications Security*, 2024.
20. Matteo Frigo and Steven G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005.

21. W. Morven Gentleman and Gordon Sande. Fast fourier transforms: for fun and profit. In *Proceedings of the November 7–10, 1966, Fall Joint Computer Conference*, AFIPS '66 (Fall), pages 563–578. ACM, 1966.
22. Craig Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the 41st Annual ACM Symposium on Theory of Computing*, pages 169–178. ACM, 2009.
23. Baraq Ghaleb and William J. Buchanan. Side channel analysis in homomorphic encryption. Cryptology ePrint Archive, Paper 2025/867, 2025.
24. Charles Gouert and Nektarios Georgios Tsoutsos. PolyFHEmus: Rethinking multiplication in fully homomorphic encryption. Cryptology ePrint Archive, Paper 2024/1090, 2024.
25. Naga K. Govindaraju, Brandon Lloyd, Yuri Dotsenko, Burton Smith, and John Manferdelli. High performance discrete fourier transforms on graphics processors. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, 2008.
26. David Harvey. Faster polynomial multiplication via multipoint kronecker substitution. *Journal of Symbolic Computation*, 44(10):1502–1510, 2009.
27. Zhen Hang Jiang, Yunsi Fei, and David Kaeli. A complete key recovery timing attack on a GPU. In *Proceedings of the 2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 394–405. IEEE, 2016.
28. Hongyue Jin, Yanan Guo, and Zhenkai Zhang. Exploiting TLBs in virtualized GPUs for cross-vm side-channel attacks. In *Network and Distributed System Security Symposium (NDSS)*, 2026.
29. Steven G. Johnson and Matteo Frigo. Implementing FFTs in practice. In *Fast Fourier Transforms*. Connexions / Rice University, 2008.
30. Wonkyung Jung, Sangpyo Kim, Jung Ho Ahn, Jung Hee Cheon, and Younho Lee. Over 100x faster bootstrapping in fully homomorphic encryption through memory-centric optimization with GPUs. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2021(4):114–148, 2021.
31. Paul C. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *Advances in Cryptology – CRYPTO '96*, volume 1109 of *Lecture Notes in Computer Science*, pages 104–113. Springer, 1996.
32. Neal Livesay, Gilbert Jonatan, Evelio Mora, Kaustubh Shivdikar, Rashmi Agrawal, Ajay Joshi, José L. Abellán, John Kim, and David R. Kaeli. Accelerating finite field arithmetic for homomorphic encryption on GPUs. *IEEE Micro*, 43(5):55–63, 2023.
33. Patrick Longa and Michael Naehrig. Speeding up the number theoretic transform for faster ideal lattice-based cryptography. In *Cryptology and Network Security (CANS 2016)*, volume 10052 of *Lecture Notes in Computer Science*, pages 124–139. Springer, 2016.
34. Peter Mell and Timothy Grance. The NIST definition of cloud computing. Technical Report Special Publication 800-145, National Institute of Standards and Technology, 2011.
35. Peter L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–521, 1985.
36. Hoda Naghibijouybari, Ajaya Neupane, Zhiyun Qian, and Nael Abu-Ghazaleh. Rendered insecure: GPU side channel attacks are practical. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 2139–2153, 2018.
37. Ali Şah Özcan, Can Ayduman, Enes Recep Türkoğlu, and Erkay Savaş. Homomorphic encryption on gpu. *IEEE Access*, 11:84168–84186, 2023.
38. Ali Şah Özcan and Erkay Savaş. Two algorithms for fast GPU implementation of NTT. Cryptology ePrint Archive, Paper 2023/1410, 2023.
39. Ali Şah Özcan and Erkay Savaş. HEonGPU: A GPU-based fully homomorphic encryption library. Cryptology ePrint Archive, Paper 2024/1543, 2024.
40. Ali Şah Özcan and Erkay Savaş. GPU Implementations of Three Different Key-Switching Methods for Fully Homomorphic Encryption. Cryptology ePrint Archive, Paper 2025/124, 2025.
41. Özgün Özerk, Can Elgezen, Ahmet Can Mert, Erdinç Öztürk, and Erkay Savaş. Efficient number theoretic transform implementation on GPU for homomorphic encryption. *The Journal of Supercomputing*, 2022.
42. M. Sadegh Riazi, Kim Laine, Blake Pelton, and Wei Dai. HEAX: An architecture for computing on encrypted data. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 1295–1309. ACM, 2020.
43. Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, you, get off of my cloud: Exploring information leakage in third-party compute clouds. In *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS)*, pages 199–212. ACM, 2009.
44. Arnold Schönhage and Volker Strassen. Schnelle multiplikation großer zahlen. *Computing*, 7(3–4):281–292, 1971.
45. Shiyu Shen et al. CARM: CUDA-accelerated RNS multiplication in word-wise homomorphic encryption. *Cryptology ePrint Archive*, 2022:633, 2022.

46. Kaustubh Shivdikar, Yuhui Bao, Rashmi Agrawal, Michael Shen, Gilbert Jonatan, Evelio Mora, Alexander Ingare, Neal Livesay, José L. Abellán, John Kim, Ajay Joshi, and David Kaeli. GME: GPU-based microarchitectural extensions to accelerate homomorphic encryption. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, 2023.
47. Kaustubh Shivdikar, Gilbert Jonatan, Evelio Mora, Neal Livesay, Rashmi Agrawal, Ajay Joshi, José L. Abellán, John Kim, and David Kaeli. Accelerating polynomial multiplication for homomorphic encryption on GPUs. In *2022 IEEE International Symposium on Secure and Private Execution Environment Design (SEED)*, pages 61–72, 2022.
48. Jr. Thomas G. Stockham. High-speed convolution and correlation. In *Proceedings of the Spring Joint Computer Conference*, volume 28, pages 229–233, 1966.
49. Joachim von zur Gathen and Jürgen Gerhard. Fast algorithms for polynomial and integer arithmetic. In *Modern Computer Algebra*. Cambridge University Press, 1999.
50. Zhenkai Zhang, Kunbei Cai, Yanan Guo, Fan Yao, and Xing Gao. Invalidate+compare: A timer-free GPU cache attack primitive. In *33rd USENIX Security Symposium (USENIX Security 24)*, 2024.