Cinnamon: A Framework for Scale-Out Encrypted AI

Siddharth Jayashankar
sidjay@cmu.edu
Carnegie Mellon University
Pittsburgh, Pennsylvania, USA

Edward Chen
ejchen@cmu.edu
Carnegie Mellon University
Pittsburgh, Pennsylvania, USA

Tom Tang
yizet@andrew.cmu.edu
Carnegie Mellon University
Pittsburgh, Pennsylvania, USA

Wenting Zheng
wenting@cmu.edu
Carnegie Mellon University
Pittsburgh, Pennsylvania, USA

Dimitrios Skarlatos
dskarlat@cs.cmu.edu
Carnegie Mellon University
Pittsburgh, Pennsylvania, USA

Abstract

Fully homomorphic encryption (FHE) is a promising cryptographic solution that enables computation on encrypted data, but its adoption remains a challenge due to steep performance overheads. Although recent FHE architectures have made valiant efforts to narrow the performance gap, they not only have massive monolithic chip designs but also only target small ML workloads. We present Cinnamon, a framework for accelerating state-of-the-art ML workloads that are encrypted using FHE. Cinnamon accelerates encrypted computing by exploiting parallelism at all levels of a program, using novel algorithms, compilers, and hardware techniques to create a scale-out design for FHE as opposed to a monolithic chip design. Our evaluation of the Cinnamon framework on small programs shows a 2.3× improvement in performance compared to prior state-of-the-art designs. Further, we use Cinnamon to show for the first time the scalability of large ML models such as the BERT language model in FHE. Cinnamon achieves a speedup of 36, 600× compared to a CPU bringing down the inference time from 17 hours to 1.67 seconds thereby enabling new opportunities for privacy-preserving machine learning. Finally, Cinnamon's parallelization strategies and architectural extensions reduce the required resources per-chip leading to a 5× and 2.68× improvement in performance-per-dollar compared to state-of-the-art monolithic and chiplet architectures respectively.

CCS Concepts: • **Security and privacy → Cryptography**; • **Computer systems organization → Parallel architectures**; • **Computing methodologies → Parallel algorithms**.

Keywords: Fully Homomorphic Encryption, Encrypted AI, Parallelism, Accelerators

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike International 4.0 License.

ASPLOS '25, March 30-April 3, 2025, Rotterdam, Netherlands
© 2025 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0698-1/25/03
https://doi.org/10.1145/3669940.3707260

ACM Reference Format:
Siddharth Jayashankar, Edward Chen, Tom Tang, Wenting Zheng, and Dimitrios Skarlatos. 2025. Cinnamon: A Framework for Scale-Out Encrypted AI. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1 (ASPLOS '25), March 30-April 3, 2025, Rotterdam, Netherlands.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3669940.3707260

1 Introduction

Datacenters today provide the computational backbone for much of the world's processing needs and support for executing analytics and machine learning workloads on the cloud. Organizations often run such workloads on sensitive user data to extract valuable insights. Unfortunately, computing on sensitive data in the cloud is vulnerable to attacks from data breaches [12, 23, 61, 63] to side channels [11, 40, 41, 44]. Furthermore, as privacy becomes an important pillar of our society, restrictive regulations [20, 26] threaten to limit the use of private data. Case in point, Italy's temporary ban on ChatGPT [45, 46, 52]. Consequently, it is increasingly important to identify new methods of outsourcing data for computation without compromising sensitive data.

Fully homomorphic encryption (FHE) [10, 15] is a promising cryptographic technique that allows *generic computation on encrypted data.* With FHE, clients can securely offload their computation on a private input x to a cloud by encrypting x and sending the encryption $\mathsf{Enc}(x)$ to the untrusted cloud provider. The cloud can apply a function f on $\mathsf{Enc}(x)$ to obtain $\mathsf{Enc}(f(x))$ and return this result to the client. Finally, the client decrypts the result to get $f(x)$. Although FHE has many compelling applications, such as secure computational genomics and private database analytics [8, 29, 37, 54], privacy-preserving machine learning is a major drive behind its increasing popularity [25, 34, 42, 43, 65].

Despite its strong privacy guarantees, FHE's biggest drawback is its significant performance overhead. Today, computing FHE on CPUs with state-of-the-art libraries [1, 2, 5, 58] imposes steep overheads of over four orders of magnitude compared to its plaintext counterpart. In the space of machine learning FHE programs, there are two workload characteristics that affect performance. First, larger inputs increase

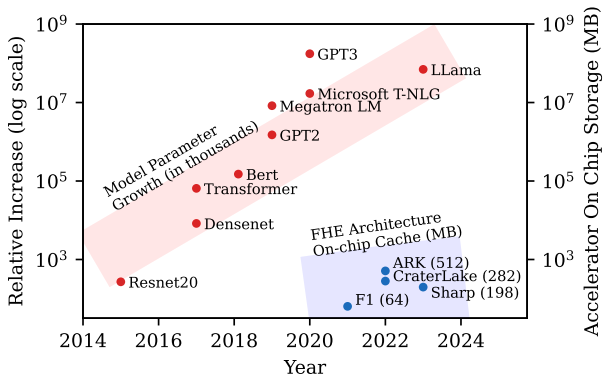

Figure 1. Growth of ML Models and FHE Architectures.

the number of ciphertexts required for encryption (i.e. wider programs). Second, advanced model architectures lead to complex FHE computation with higher multiplicative depth (i.e., deeper programs), which require more bootstrapping. Bootstrapping is an expensive ciphertext maintenance operation which can account for about 70-80% of the execution time of FHE programs.

Although prior work has proposed several solutions targeting FPGAs, GPUS, and ASICs that aim to alleviate some of the overheads[22, 33, 35, 36, 38, 39, 55, 56, 59, 64], they have only demonstrated the feasibility of FHE on small ML programs. For example, the largest workload accelerated by recent architectures [38, 56] is ResNet20 [31, 43] inference. This model is small enough for its inputs and activations to be encrypted using a single ciphertext, and its computation requires about fifty bootstraps. Despite targeting such a small application, FHE architectures have gravitated towards massive chip designs with a quarter to half a gigabyte of on-chip caches.

Looking into the landscape of ML models in recent years, the unsatisfied appetite for large models is pushing the limits of even plaintext architectures [51]. Figure 1 shows the scaling of ML models compared to the cache capacity on the chip of FHE architectures, which shows that FHE architectures cannot keep up with the growth in ML model parameters due to their significant on-chip cache requirements. In general, larger ML models have both larger input sizes and more complex computation. For example, a transformer-based model like BERT Base [19] is wider and requires 3 and 12 ciphertexts to encrypt 128 and 512 tokens, respectively. BERT is also significantly deeper with each BERT inference requiring thousands of bootstraps. For example, a 128 token encrypted BERT inference requires about *1,400* bootstraps. Sadly, even prior massive chip designs [35, 36, 56] only optimize for performing a bootstrap operation on one ciphertext at a time.

We present Cinnamon, a framework for scale-out encrypted computing that aims to accelerate large, state-of-the-art models. Instead of building even larger monolithic chips to meet the compute, cache, and memory demand, Cinnamon explores an alternative approach to scale-out encrypted computing by exploiting parallelism. From our observations about machine learning FHE workload characteristics, we identify two levels of parallelism that are suitable for a scale-out design: program-level and limb-level parallelism.

As larger models lead to wider programs requiring a larger number of ciphertexts, it opens up optimization opportunities via program-level parallelism, i.e. parallelism across ciphertexts. Cinnamon introduces a Python domain-specific language (DSL) for FHE that introduces concurrent execution streams, and users can use this abstraction to exploit program level parallelism. Cinnamon's compiler uses this information as well as information about the underlying hardware architecture to distribute streams across chips, enabling highly efficient parallel execution.

Larger models also lead to deeper programs and more bootstrap operations. As each ciphertext is very large, Cinnamon's second focus to parallelize bootrapping is to find parallelization opporunities within individual ciphertexts. Our first insight is to use a partitioning strategy called limb-level parallelism. Limbs are largely independent components of a ciphertext and introduce inherent parallelism for many FHE primitives. However, we find that *key switching* [28], a very common but expensive [36, 38, 55, 56] FHE sub-functionality, exhibits a large degree of dependencies across limbs. To resolve the communication bottleneck arising from these cross-limb dependencies, Cinnamon introduces novel parallel keyswitching algorithms and compiler optimizations that lead to a 32× reduction from 16TB/s to only 512GB/s in the bandwidth required to implement limb level parallelism. This algorithmic breakthrough enables Cinnamon to parallelize large FHE computation across multiple substantially smaller chips with smaller and fewer functional units and smaller on-chip caches (56MB). To further reduce the on-chip resources, Cinnamon also introduces a compact architectural implementation of a functional unit for base conversion.

We evaluate Cinnamon through simulations for performance and RTL synthesis with a commercial PDK and SRAM compiler for area and power. We first show how Cinnamon's parallelization techniques can be used to improve the performance of small ML models by 2.3× compared to the state-of-the-art design. We further show for the first time the scalability of Cinnamon on large ML models such as the BERT language model. Cinnamon achieves a speedup of 36,600× compared to a 48-core CPU bringing down inference time from 17 hours to 1.67 seconds enabling new opportunities for privacy-preserving machine learning. Cinnamon's limb-level parallelism and base conversion unit also reduce the per chip cache, compute, and communication resources by 4.82×, 8.3×, and 6× respectively compared to a monolithic chip design, thereby eliminating the need to architect large FHE architectures. Finally, we highlight the impact of area constraints on the manufacturing process and demonstrate that Cinnamon can lead to 5× and 2.68×

improvement in performance-per-dollar compared to state-of-the-art monolithic and chiplet architectures respectively. Overall, Cinnamon makes the following contributions:

- A compiler design for enabling program-level parallelism for FHE programs, including a Python DSL and a polynomial IR for expressing concurrent execution streams to scale-out performance across multiple chips.
- A novel partitioning strategy for limb-level parallelism and a limb-level IR that significantly reduces communication requirements across chips and enables FHE architectures with small cache sizes.
- A new base conversion unit architecture that shrinks the compute resources required by FHE architectures.
- The demonstration of practical inference time for language models such as BERT on FHE architectures.
- The evaluation of Cinnamon across multiple metrics including, performance, area, and cost.
- The first end-to-end open source framework for encrypted computing, which we will release after publication.

2 Background on Fully Homomorphic Encryption

In this paper, we focus on the CKKS [15] encryption scheme, a popular FHE scheme for encrypted computing on real numbers. CKKS can efficiently batch multiple plaintext values into a single ciphertext. Operations on this ciphertext will affect all of the underlying plaintext values. Figure 2 shows a high level overview of plaintexts and ciphertexts in CKKS. To encrypt, several plain text values are first batched into a vector (①) and then encrypted into a ciphertext $\mathfrak{C}$, composed of a pair of polynomials $(\mathbf{C^0}, \mathbf{C^1})$ (②) that are elements of a polynomial ring over an integer modulus with a ring dimension N. For a ring dimension N, the polynomial degree is $N-1$. CKKS supports three kinds of homomorphic operations on ciphertext values - addition, multiplication and rotation. These operations are composed of arithmetic over the polynomial ring.

Limbs: The polynomial rings in FHE require a very large integer modulus, alos known as the ciphertext modulus. This makes the coefficients of the ciphertext polynomials $\mathbf{C^0}, \mathbf{C^1}$ very large. Modular arithmetic over such large integers is extremely inefficient. However, if the ciphertext modulus is strategically chosen to be the product of a set of ℓ smaller moduli $\{q_0, q_1 \ldots q_{\ell-1}\}$, then, the Residue Number System (RNS) [6] can be used to uniquely represent a polynomial $\mathbf{C}$ as a tuple of ℓ polynomials $\{\mathbf{C} \mod q_0, \mathbf{C} \mod q_1, \ldots, \mathbf{C} \mod q_{\ell-1}\}$, each with much smaller coefficients. We call the set of moduli $Q = \{q_0 \ldots q_{\ell-1}\}$ the RNS basis and the set of decomposed polynomials $\mathbf{C}_Q = \{\mathbf{C}_{q_0}, \ldots, \mathbf{C}_{q_{\ell-1}}\} = \mathbf{C}_{\{q_0, q_1 \ldots q_{\ell-1}\}}$ limbs. In ③ from Figure 2, each column depicts a limb. The figure uses a 39 bit ciphertext modulus and 7 bit q_i's. In practice, the ciphertext modulus can be 1000s of bits wide and the q_i's are machine word sized.

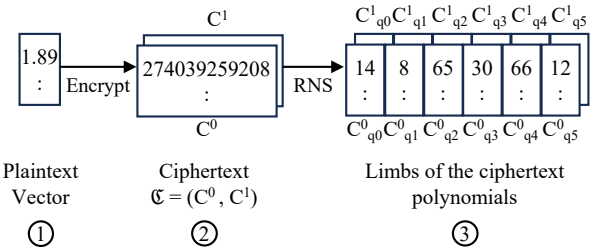

Figure 2. CKKS Plaintexts and Ciphertexts

Most polynomial operations are data parallel over the limbs. For example, polynomial additions work as shown: $\mathbf{X_Q} + \mathbf{Y_Q} = \{\mathbf{X}_{q_0} + \mathbf{Y}_{q_0}, \mathbf{X}_{q_1} + \mathbf{Y}_{q_1}, \ldots, \mathbf{X}_{q_{\ell-1}} + \mathbf{Y}_{q_{\ell-1}}\}$. Similarly, operations like NTT, multiplication and automorphism are data parallel across limbs. The number of limbs of a ciphertext is also called the level of the ciphertext.

NTT: The Number Theoretic Transform (NTT) is the analog of the Fast Fourier Transform (FFT) in a prime number field. Similar to FFT, NTT performs a convolution over the coefficients of a polynomial and speeds up modular polynomial multiplication by transforming polynomials from the coefficient domain to the evaluation domain. The inverse NTT (INTT) reverses this transformation. Unless explicitly stated, all polynomials are assumed to be in the evaluation domain by default. We use the superscript $\bar{\mathbf{X}}$ to represent a polynomial $\mathbf{X}$ in the coefficient domain.

Automorphism: The automorphism operation implements permutations over the polynomial coefficients. It is used to implement ciphertext rotation.

Base conversion: Base Conversion [6] converts a polynomial from one RNS basis $Q = \{q_0, \ldots, q_{\ell-1}\}$ to another RNS basis $P = \{p_0, \ldots, p_{m-1}\}$. Thus the polynomial $\bar{\mathbf{C}}_Q = \{\bar{\mathbf{C}}_{q_0}, \ldots, \bar{\mathbf{C}}_{q_{\ell-1}}\}$ is transformed to $\bar{\mathbf{C}}_P = \{\bar{\mathbf{C}}_{p_0}, \ldots, \bar{\mathbf{C}}_{p_{m-1}}\}$. The equation below shows the calculation of an output limb $\bar{\mathbf{C}}_{p_k}$. Here f_{jk} are scalar base conversion factors.

$$\bar{\mathbf{C}}_{p_k} = \sum_{j=0}^{\ell} (\bar{\mathbf{C}}_{q_j} \cdot f_{jk} \mod p_k)$$

Unlike other operations, base conversion is not data parallel across limbs. Also, base conversion can be performed only in the coefficient representation.

Mod Up and Mod Down: The mod up operation converts a polynomial $\mathbf{X_S}$ from a smaller RNS basis S to a larger RNS basis T. The mod down operation converts a polynomial $\mathbf{X_{S \cup E}}$ from a larger RNS Basis $S \cup E$ to a smaller basis S, followed by a multiplication by a scalar factor. The mod up and mod down operations are described in Figure 3.

Digits: Digits are disjoint partitions of the limbs of a polynomial. For example, consider a polynomial $\mathbf{C}_Q = \{\mathbf{C}_{q_0}, \mathbf{C}_{q_1}, \ldots, \mathbf{C}_{q_5}\}$. One possible 3 digit partition of $\mathbf{C}_Q$ is $\{\{\mathbf{C}_{q_0}, \mathbf{C}_{q_1}\}, \{\mathbf{C}_{q_2}, \mathbf{C}_{q_3}\}, \{\mathbf{C}_{q_4}, \mathbf{C}_{q_5}\}\}$. Each digit can be individually base converted to another RNS basis. For example, the digit $\{\mathbf{C}_{q_2}, \mathbf{C}_{q_3}\}$ can be converted

```
def modUp(X_S):
    X̄_S = INTT(X_S)
    Ȳ_T = BCONV(X̄_S)
    Y_T = NTT(Ȳ_T)
    return Y_T
```

```
def modDown(X_{S∪E}):
    X̄_E = INTT(X_E)
    Ȳ_S = BCONV(X̄_E)
    Y_S = NTT(Ȳ_S)
    return (X_S − Y_S) * f_S
```

Figure 3. Mod Up and Mod Down functions

```
# Inputs: input polynomial C_Q, the number of
# digits d and evaluation key EvalKey
def keyswitch(C_Q, d, EvalKey):
    # D can be indexed 0,...,d−1
    D = SplitIntoDigits(Q, d)
    E = GetExtensionBasis(Q, d)

    F0_{Q∪E}, F1_{Q∪E} = 0, 0
    for i in range(d):
        B_{Q∪E} = modUp(C_{D[i]})
        F0_{Q∪E} += B_{Q∪E} * EvalKey[i][0]_{Q∪E}
        F1_{Q∪E} += B_{Q∪E} * EvalKey[i][1]_{Q∪E}

    Ĉ0_Q, Ĉ1_Q = modDown(F0_{Q∪E}), modDown(F1_{Q∪E})
    return Ĉ0_Q, Ĉ1_Q
```

Figure 4. Keyswitch Routine

to a new basis $P = p_0, p_1, \ldots p_7$ to obtain: ${}^{q_2q_3}C_P = ({}^{q_2q_3}C_{p_0}, {}^{q_2q_3}C_{p_1}, \ldots, {}^{q_2q_3}C_{p_7})$. We use the label ^{q_iq_j} to indicate the digit to which a value corresponds. Note that ${}^{q_2q_3}C_P \neq C_P$. Instead, $C_P = {}^{q_0q_1}C_P \times {}^{q_0q_1}f + {}^{q_2q_3}C_P \times {}^{q_2q_3}f + {}^{q_4q_5}C_P \times {}^{q_4q_5}f$, where each ${}^{q_iq_j}f$ is a scalar factor.

Keyswitching: Keyswitching [28] homomorphically converts a ciphertext encrypted using secret key k to a ciphertext encrypted under a different key k' using an evaluation key $\texttt{EvalKey}_{k\rightarrow k'}$. It is a major ciphertext maintenance subroutine required to implement homomorphic rotation and multiplication. Figure 4 describes the keyswitching kernel and Figure 5 shows how homomorphic rotation and multiplication internally use keyswitching.

Keyswitching works by splitting the input polynomial C_Q into d digits and computing a mod up of each digit to the basis $Q \cup E$ to compute an inner product with the evalkey EvalKey. Here, E is a temporary *extension* basis such that $E \cap Q = \phi$. Finally, the inner product undergoes a mod down to the original basis Q. Keyswitching is a computationally expensive subroutine and accelerating keyswitching is the focus of most FHE accelerators [35, 36, 39, 56].

Multiplicative Budget: The multiplicative budget is an intrinsic property of a ciphertext. Every ciphertext has a finite multiplicative budget and every multiplication operation performed on a ciphertext consumes some of the available budget. Once the multiplicative budget has been exhausted, any further computation requires this budget to be refreshed using an expensive homomorphic procedure called bootstrapping.

```
# Inputs: Ciphertexts A, B
def HMultiply(A, B):
    C0 = A[0] * B[0]
    C1 = A[0] * B[1] + A[1] * B[0]
    C2 = A[1] * B[1]
    Ĉ0, Ĉ1 = keyswitch(C2, d, EvalKey)
    return C0 + Ĉ0, C1 + Ĉ1

# Inputs: Ciphertext A, rotation amount r
def HRotate(A, r):
    C0 = automorphism(A[0], r)
    C1 = automorphism(A[1], r)
    Ĉ0, Ĉ1 = keyswitch(C1, d, EvalKey)
    return C0 + Ĉ0, Ĉ1
```

Figure 5. Homomorphic Multiplication and Rotation

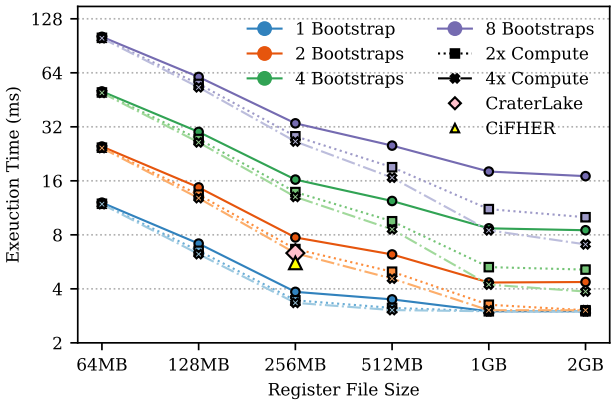

Figure 6. Performance scaling based on the number of bootstrap operations on-chip cache capacity and compute.

Bootstrapping [13, 24, 30]: The bootstrapping procedure raises a ciphertext to the highest multiplicative budget and then further processes it. This further processing involves several homomorphic matrix-vector multiplications and homomorphic polynomial evaluation. Bootstrapping itself consumes some part of the total multiplicative budget, leaving the remainder for application processing. Bootstrapping is an extremely expensive operation as it requires several keyswitch operations and often comprises upto 80% of the execution time of FHE programs. Thus, we pay special attention to bootstrapping and keyswitching in this work.

3 Motivation

In this section, we perform a detailed study of the challenges and opportunities of computing larger ML models in FHE.

3.1 Larger Models Lead to Wider FHE Programs

Large ML models lead to wider FHE programs requiring more chiphertexts. This is because larger ML models have a larger number of input and activation values which cannot fit into a single ciphertext and instead require multiple ciphertexts

to encrypt. This opens up the opportunity for *program-level parallelism*, or parallelism across multiple ciphertexts.

Each of these ciphertexts will have to be bootstrapped when its multiplicative budget is exhausted. As bootstrapping is the most expensive FHE operation, we focus on the bootstrapping of multiple ciphertexts in Figure 6, exploring how different hardware resources affect performance. The baseline is a single-chip Cinnamon with 1TB/s HBM, a 256MB register file and four compute clusters (details of Cinnamon are presented in section 5). This configuration is representative of prior FHE accelerators. To scale compute, we tune the number of compute clusters. We also show the results of recent proposals [38, 56] for one bootstrap.

For a single bootstrap, we see that contemporary 64MB and 128MB caches found in the cache-heavy GPUs and CPUs are insufficient. Instead, a 256MB cache, as adopted by most recent proposals, is a balanced design. Scaling cache capacity further to 1GB yields smaller performance gains of 28%. As the number of ciphertexts to be bootstrapped increases, we see three major trends. First, with the default compute, performance degrades linearly for the smaller cache capacities with the number of bootstraps. At 256MB cache size, the execution time of a single bootstrap is about 3.8ms but increases to 32ms for eight bootstraps. However, a larger cache capacity can improve the performance of parallel bootstraps. There is 5.63× improvement by increasing cache capacity from 256MB to 1GB for eight bootstraps, while the improvement is only 1.28× for a single bootstrap. Note that 1GB of cache represents a sweet spot in the design – increasing the cache capacity further does not yield additional benefits. This is because bootstraps share plaintext matrices and evaluation keys, and 1GB is enough to fit this metadata within the cache and avoid spills. Furthermore, increasing compute can lead to more performance benefits. For example, at eight bootstraps and 1GB cache capacity, there is a 1.62× speedup by doubling the number of clusters to eight. Since bootstrap operations on independent ciphertexts can be fully parallelized, replicating all the resources can potentially lead to linear speedups for FHE programs.

This analysis reveals that FHE computing is stuck between a rock and a hard place. On the one hand, larger models incur linear slowdown on current architectures as independent bootstraps have to be computed sequentially. On the other hand, improving performance further requires even more hardware resources. Hence, we need to find ways to exploit the inherent parallelism of larger models to avoid building even larger monolithic chips.

3.2 Larger Models Lead to Deeper FHE Programs

Larger models also lead to deeper programs that require more bootstraps. Hence, improving the performance of bootstrapping is key to improving performance. Since ciphertexts are composed of polynomial limbs, there are two potential parallelization strategies: coefficient-level and limb-level. Prior architectures have primarily focused on coefficient-level parallelism and developed techniques requiring complex Networks-on-Chip with 10-20TB/s of bandwidth to support the all-to-all dependencies introduced by NTT and automorphism. Thus, scaling coefficient level parallelism further on a single chip is challenging and scaling it across chips or chiplets is even harder.

Fortunately, FHE ciphertexts exhibit a second kind of parallelism called *limb-level parallelism*. This is a good candidate for parallelism because most FHE operations do not have any dependencies at the limb level. However, keyswitching, an integral part of ciphertext multiplication, rotation and bootstrapping, introduces several dependencies across limbs. Due to the challenges associated with managing these dependencies, most prior accelerators either do not pursue this type of parallelism at all, limiting themselves to just coefficient level parallelism, or use large communication bandwidth on the order of multiple TB/s to deal with these dependencies. This requirement of very high communication bandwidth for both coefficient and limb level operations steered prior work towards very large monolithic architectures. As a result, monolithic FHE architectures require large 256MB-512MB caches and large areas for functional units since all cache and compute resources to process large ciphertexts (each about 20MB) have to exist within a single chip.

One prior work CiFHER [38], identified concerns with building large chips and proposed using limb-level parallelism to split the resources needed for FHE across multiple FHE chiplet cores on a single package. CiFHER resolves dependencies across chiplets by broadcasting the limbs to all chiplets and represents the current state of the art for limb level parallelism. This approach is feasible for a multicore design where the memory bandwidth on a per-core basis is limited, thereby becoming the limiting factor in performance instead of communication. However, scaling performance with this approach is not sustainable as merely increasing the memory bandwidth leads to an unbalanced design that is severely bottlenecked on network bandwidth. For instance, if the per-chip memory bandwidth is scaled to 2TB/s, as required by most FHE architectures, avoiding the network bottleneck would require an unattainable 16TB/s bandwidth.

While limb-level parallelism holds the promise of significantly improving the performance of bootstrapping by enabling the scaling out of FHE workloads to multiple chips while simultaneously reaping the benefits offered by smaller chips, properly extracting its full potential has been elusive. Further improvements require algorithmic innovations to reduce the high network bandwidth requirements.

4 Design

The goal of Cinnamon is to provide a framework for accelerating large ML workloads in FHE using a holistic cross-stack approach that optimizes for parallel computation across

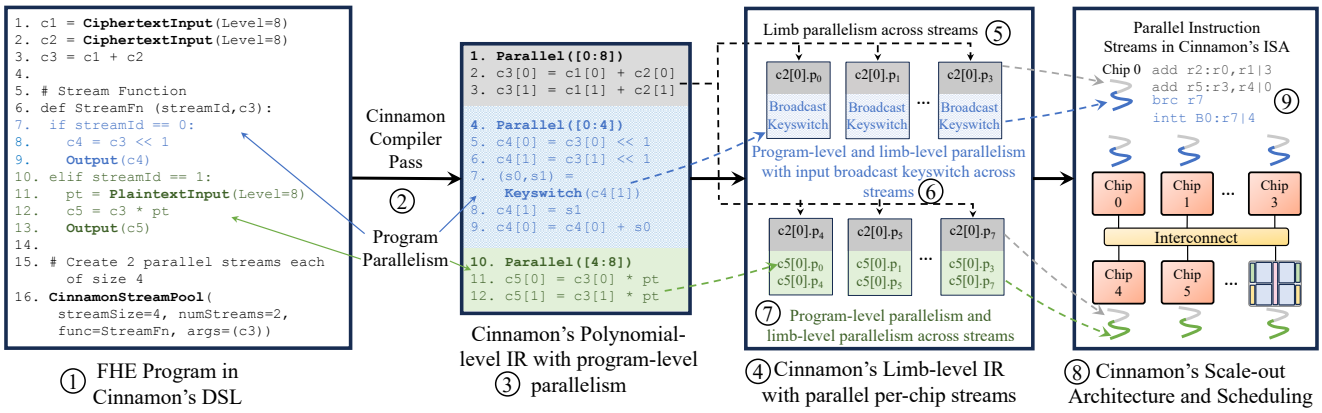

Figure 7. Cinnamon framework overview

FHE algorithms, the compiler, and the hardware architecture while minimizing cost.

4.1 Overview

Figure 7 shows an overview of the Cinnamon framework. Cinnamon is composed of four main components: 1. a Cinnamon DSL for ease of programming, 2. a polynomial-level IR designed for program-level parallelism, 3. a limb-level IR that facilitates program-level, limb-level, and keyswitch parallelism on top of per-chip streams, and 4. a scale-out hardware architecture and cycle-level scheduler that leverage parallelism to support efficient encrypted computing. We describe the components of Cinnamon below.

4.2 A DSL and IR for Parallel FHE

The Cinnamon DSL is embedded in Python. It implements FHE operations like add, multiply, rotate etc. as language constructs. ① from Figure 7 shows a sample program in the Cinnamon DSL. The Cinnamon DSL also offers programmers the capability to create concurrent execution streams, which are programmed similar to threads or python multiprocessing. Line 6 in ① shows an example of a stream function `StreamFn`. Within the function, the programmer provides the code for each stream indexed by the variable `streamId`. At line 16, the call `CinnamonStreamPool(..)` instantiates the streams.

In ② of Figure 7, the program from ① is lowered to Cinnamon's polynomial IR where the ciphertext representation is expanded to a polynomials. For example, a ciphertext addition (`c1 + c2`) is expanded to two polynomial additions (`c1[0] + c2[0]` and `c1[1] + c2[1]`). The Cinnamon compiler also automatically distributes the concurrent streams across the chips depending on the programmer specified stream size and number of streams (lines 4 and 10 in ③). The blue stream is placed on chips $0-4$ and the green stream is placed on chips $4-8$.

4.3 Efficient Limb-level Parallelism

In steps ④ – ⑦ of Figure 7, Cinnamon lowers polynomial-level representation into Cinnamon's *limb-level representation* and automatically provides parallelization at this level. Cinnamon is able to compose limb-level parallelism with user provided program-level parallelism. This is shown by ⑤ and ⑥ in Figure 7 where ciphertext operations in the blue and green streams are placed on the first and second groups of four chips according to user input, and then further parallelized by Cinnamon at the limb-level within each set of four chips using the data partitioning policies described below.

4.3.1 Data partitioning. Cinnamon uses limb-level partitioning as it enables data parallelism across the limbs for polynomial operations like addition, multiplication, NTT and automorphism. For a polynomial $\mathbf{C_Q}$ and a hardware architecture comprised of n chips with each chip having an identifier c, Cinnamon partitions $\mathbf{C_Q}$'s limbs across the n chips in a modular fashion. We define the set $Q_c = \{q_i \mid i \bmod n = c\}$, to be the subset of the RNS basis Q handled by chip c. For example, if $\mathbf{C_Q}$ has $\ell = 12$ limbs and $n = 4$, then chip $c = 0$ holds limbs $\mathbf{C}_{Q_0} = \mathbf{C}_{\{q_0,q_4q_8\}}$. For concurrent streams, the modular distribution is adjusted according to the stream size and placement. While limb-level partitioning works well for many FHE operations, keyswitching is difficult to parallelize as it introduces complex dependencies across limbs. In the rest of this section, we describe sequential keyswitching, the challenges with parallelizing keyswitching and our novel algorithms for parallelizing keyswitching.

Sequential keyswitching: Figure 8a gives a high level overview of sequential keyswitching with the help of an example. $Q = \{q_0, q_1, \ldots, q_{11}\}$ is a basis of 12 RNS moduli and $\mathbf{C}_Q$ are the limbs of the input polynomial $\mathbf{C}$ in the basis Q. We'll call Q the initial basis. In ②, the limbs $\mathbf{C}_Q$ are partitioned into $d = 3$ digits, represented by the three colors. Additionally, we pick $E = \{\tilde{q}_0, \tilde{q}_1, \ldots, \tilde{q}_3\}$ as the *extension basis*, which is used temporarily for keyswitching. In ③, the first digit ($\mathbf{C}_{\{q_0 \ldots q_3\}}$) undergoes a mod up operation and is expanded to the basis $Q \cup E$ resulting in ${}^{q_0 \cdots q_3}\mathbf{C}_{Q \cup E}$. In ④, the

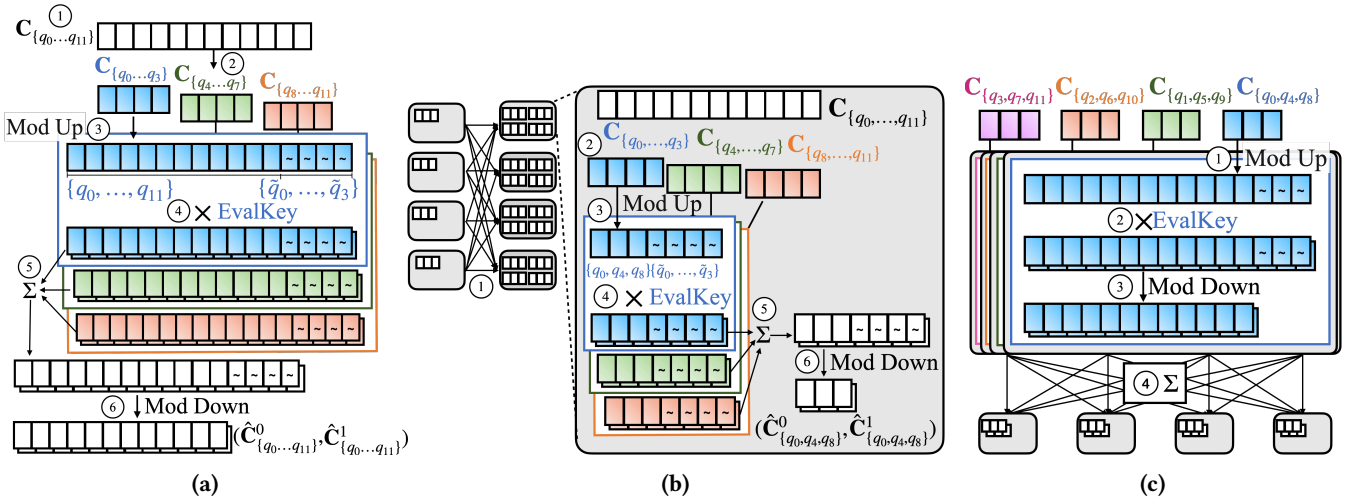

Figure 8. (a) Sequential keyswitching (b) Input Broadcast Keyswitching (c) Output aggregation keyswitching.

extended digit $^{q_0 \cdots q_3}C_{Q \cup E}$ is multiplied by the two polynomials of the evalkey to get two evalkey products. In ⑤, ②-④ are repeated for the other two digits and all evalkey products are aggregated. In ⑥, the two evalkey product polynomials under go a mod down to Q from $Q \cup E$.

Challenge of parallelizing keyswitching: The challenge in designing a limb-parallel keyswitching algorithm is minimizing cross chip communication, caused by the limb dependencies from the base conversion operations in mod up (③) and mod down (⑥). A simple approach to parallelize keyswitching [38] distributes all limbs across chips in a modular fashion and resolves cross limb dependencies by broadcasting the input limbs of each base conversion to all the chips. This method requires 3 broadcasts: 1 in ① and 2 in ⑥ and has a very high communication overhead.

We now describe Cinnamon's approach to keyswitching. Cinnamon designs two novel algorithms and compiler optimizations to reduce the overhead of keyswitching. The idea behind these new algorithms is to limit the inter chip communication required to just one point in the keyswitch: the beginning (mod up) or the end (mod down), and exploit reordering and batching across multiple keyswitches.

Input broadcast keyswitching: This is Cinnamon's first parallel keyswitching algorithm. It limits communication to just the mod up operation of the keyswitch. Intuitively, our insight is that by duplicating the extension limbs across chips, we can eliminate the need for the broadcasts required in the mod down (⑥) and complete the keyswitch with a single broadcast at the mod up (①).

We explain this algorithm via an example shown in Figure 8b. Initially, the limbs of the input polynomial C_Q are modularly distributed across the chips, with each chip c possessing C_{Q_c}. In ①, the limbs of the input polynomial C_Q are broadcast so that every chip has a copy of all limbs C_Q. We now zoom into chip $c = 0$. In ②, C_Q is split into three digits like in the sequential case. In ③, the first digit is expanded to the basis $Q_0 \cup E = \{q_0, q_4, q_8, \tilde{q}_0, \tilde{q}_1, \tilde{q}_2, \tilde{q}_3\}$. Thus, the output limbs of the base conversion operation in the initial basis are distributed but the output limbs in the extension basis are duplicated across the chips. In ④, the digits are multiplied by the evalkey. In ⑤, all the digits' evalkey products are aggregated and in ⑥, they undergo a mod down operation from the basis $Q_0 \cup E$ to $Q_0 = \{q_0, q_4, q_8\}$. The mod down operation in ⑥ requires all extension limbs. Since the extension limbs E were duplicated in steps ③ - ⑤, all chips already have all extension limbs and no broadcast is required in ⑥. As chip $c = 0$ ends up with limbs in the basis Q_0, no more communication is required. The same steps occur on the other chips in parallel.

This algorithm is equivalent to the sequential keyswitching algorithm. This is because ③, ④ and ⑤ are limb-wise operations (i.e., no dependencies across different limbs). In ⑥, the cross-limb dependencies are only in the extension limbs. Therefore, instead of splitting the extension limbs and then broadcasting them before ⑥, we locally compute the extension limbs at the beginning of ③. Since each chip retains all necessary basis from ③ onwards, ⑥ can be computed without further communication. Overall, this approach trades-off some duplication of compute and on chip storage across the chips but requires only a single broadcast at ①.

Output aggregation keyswitching: This is Cinnamon's second parallel keyswitching algorithm. This algorithm requires inter chip communication only at the end of the keyswitch. In this keyswitching algorithm, parallelization happens at the digit-level instead of at the limb-level. The idea is to treat the initial partition of limbs across the chips as the digits for keyswitching. We explain the algorithm via the example in Figure 8c. As we have $n = 4$ chips, we use $d = n = 4$ digits. We focus on chip $c = 0$, which has the first digit $C_{Q_0} = \{q_0, q_4, q_8\}$. Since we chose a digit partition such

that all chips already have the required limbs, no broadcast is required. In ①, the first digit undergoes a mod up to the basis $Q \cup E = \{q_0, q_1, \ldots q_{11}, \tilde{q}_0, \tilde{q}_1, \tilde{q}_2\}$ and in ②, it is multiplied by the evalkeys. In ③, the evalkey product undergoes a mod down to the basis Q. Since each digit's evalkey products reside across the chips, the chips perform an aggregate and scatter operation over the network to aggregate and distribute the sum's limbs in a modular fashion (④). This reordering of the mod down and aggregation is valid because these two operations are commutative.

While we show $d = 4$ digits here, it is possible to generalize this algorithm to any multiple of n digits by further partitioning the digits local to each chip. Overall, this keyswitching algorithm requires a communication of 2 aggregations. While this is strictly worse than both the previous algorithms if it is run a single time, we show how this can be batched across multiple keyswitches in the next section.

Note that Cinnamon's keyswitching algorithms only exploit reordering of operations and digit selection of keyswitching. The digit selection doesn't affect keyswitching as implementations with all possible choices of digits are interchangeable. Thus Cinnamon's keyswitching algorithms do not have any additional effects on noise or multiplicative levels of the keyswitching algorithms.

Cinnamon Keyswitch Pass: Our analysis of FHE programs shows that it is possible to reduce the inter chip communication from keyswitching by reordering and batching communication across multiple keyswitches. This is similar to the hoisting techniques described in [28]. In this section, we will show how Cinnamon's new keyswitching algorithms significantly reduce inter chip communication when coupled with Cinnamon's compiler optimizations. We focus our analysis on two very frequent patterns that appear in bootstrapping and several linear algebra kernels: multiple rotations on a single ciphertext and multiple rotations followed by aggregation.

Let's consider the first pattern: multiple rotations on a ciphertext. Assume there are r different rotations on the same ciphertext. In this case, we use Input Broadcast Keyswitching. Each of the r key switches would require a broadcast at ①. However, this can be optimized to just 1 broadcast for the whole batch of r keyswitches by exploting commutativity to perform reordering and batching. Thus, Input Broadcast Keyswitching requires just 1 broadcast for all r rotations.

Similarly, Output Aggregation Keyswitching optimizes the second pattern: multiple rotations followed by aggregation. Assume r different ciphertexts are rotated and then aggregated. Here, we use Output Aggregation Keyswitching as it only requires aggregations at ④. Batching and reordering optimizations can optimize this to just 2 aggregations across the whole batch of r keyswitches. Thus, Output Aggregation Keyswitching also requires just 2 aggregations for all r rotations.

Coupling the reordering and batching optimizations with Cinnamon's new keyswitching algorithms significantly reduces the communication overhead of keyswitching by amortizing it over multiple keyswitch operations. For example, the communication cost of the baby-step giant-step (BSGS) algorithm [14], which is used for large matrix of multiplication in FHE, can be reduced from $O(\sqrt{n})$ to $O(1)$ - just one broadcast and two aggregations. This is because the BSGS algorithm contains patterns that can be optimized using Input Broadcast and Output Aggregation Keyswitching.

In the Cinnamon compiler, we implement a pass to automatically detect program patterns where our keyswitching algorithms can significantly improve communication costs, choose the appropriate parallel keyswitching algorithm and perform reordering optimizations. As we will show in Section 7.3, Cinnamon's keyswitch compiler pass results in a 7× reduction in the data communication per bootstrap, which can be further improved to 9.81× by combining it with the Cinnamon compiler's program parallelism constructs.

4.4 Lowering to ISA and Scheduling

The Cinnamon compiler lowers the limb level representation to the Cinnamon ISA (described in section 4.6) using Belady's min [7] to allocate registers and it inserts loads and stores as early as possible.

4.5 Cinnamon Scale-out Architecture

We now describe Cinnamon's scale-out architecture for FHE that leverages our algorithmic and compiler techniques to parallelize ciphertext operations across multiple chips, significantly reducing per-chip compute and storage resources required. This helps Cinnamon achieve large speedups while keeping individual chip sizes relatively small.

4.5.1 Hardware Architecture Overview.

Cinnamon's scale out architecture is composable and horizontally scaled across multiple chips based on the needs of the encrypted program. Figure 9 shows an overview of the architecture.

Scale-out Architecture: Cinnamon presents two main topologies based on the number of chips. First, for configurations up to eight chips Cinnamon leverages a ring topology as shown in Figure 9(a). A ring topology is sufficient because most of the communication involves all the chips either for broadcast or aggregation operations. This means that none of the links on a chip need to transfer data that is not intended for it as a receiver. This allows us to use the full capacity of the network for useful work. Furthermore, Cinnamon introduces a switch architecture for scaling to twelve chips, as shown in Figure 9(b). Similar to recent multi-GPU designs using NVLink and NVSwitch [50], the switch topology allows any two pairs of chips to communicate simultaneously. Furthermore, the interconnect provides broadcast and aggregation primitives similar to [49].

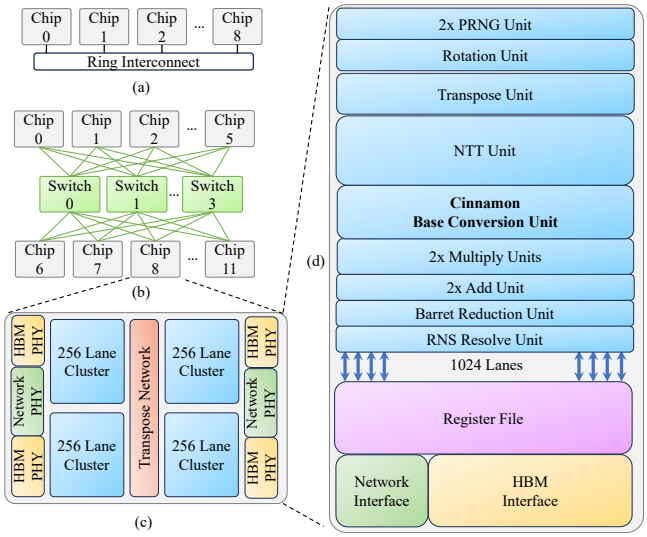

Figure 9. Cinnamon scale-out architecture shows: (a) Cinnamon with eight chips connected over a ring interconnect, (b) Cinnamon scaling to twelve chips with a switch interconnect, (c) Cinnamon's organization composed of four 256 lane clusters, (d) the logical organization of a Cinnamon chip.

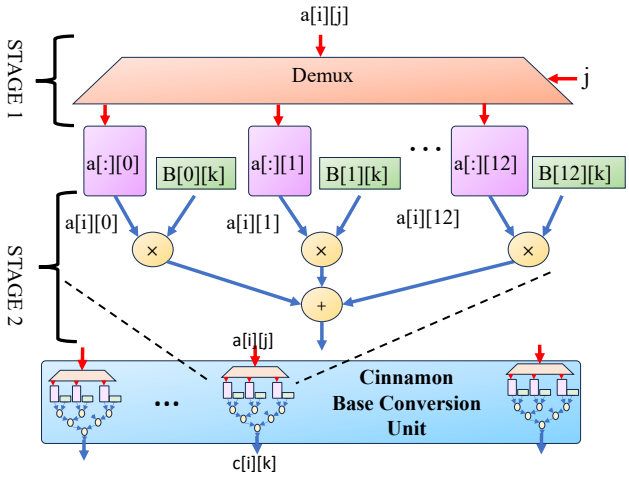

Figure 10. Cinnamon's base conversion unit.

Chip Architecture: The design of Cinnamon is based on the vector architecture from CraterLake [56]. However, Cinnamon introduces a novel base conversion unit (BCU) that significantly shrinks the size of the BCU while achieving the same performance, thus providing major end-to-end area, power and cost savings for the Cinnamon chips. Figure 9(c) shows the organization of a Cinnamon chip that is split into four compute clusters. Each compute cluster implements pipelined vector functional units. The vector width of the functional units is weighted according to the frequency of occurrence of the instructions in the programs. The chip further includes four HBM2E stacks of 256 GB/s and two network PHYs of 256 GB/s. Figure 9(d) show the logical organization, composed of the functional units along with the register file and the memory and network interfaces. The Cinnamon architecture contains the following functional units: NTT, transpose, add, multiply, PRNG, base conversion, Barrett Reduction and RNS Resolution. Cinnamon adopts a 28 bit data path similar to CraterLake and uses [47, 55] to design FHE friendly modular multipliers to reduce area and power.

4.6 Instruction Set Architecture

Cinnamon introduces a vector ISA that operates on limbs. Each register value stores a limb, that is, a 28-bit wide vector of 64K elements. This allows Cinnamon to standardize all ISA instructions and register file accesses to operate on a uniform vector size. Furthermore, the Cinnamon ISA also supports variants of add, subtract, and multiply instructions with one operand as a scalar value to eliminate the expansion of scalars to vector values that would otherwise consume valuable bandwidth and on chip storage. The ISA also contains instructions for inter-chip communication.

4.7 Space-optimized Base Conversion Unit

Cinnamon introduces a microarchitectural design for a base conversion unit (BCU) to reduce area and power while retaining performance. The base conversion unit is by far one of the largest functional units in an FHE accelerator. For example, in CraterLake [56], the base conversion unit accounts more than a third of the total chip area ($158mm^2$ out of $472mm^2$). Therefore, reducing the size of the BCU results in significant reduction in overall chip area, cost, and power.

Recall the description of base conversion from Section 2. In general, a base conversion operation can be performed to convert any number of input bases to any number of output bases. Prior work [56] designs for this case. Thus, they implement a multiply accumulate buffer with the number of multipliers and SRAM buffer size proportional to the maximum number of output limbs.

However, FHE accelerators do not need to support general base conversion. This is because in FHE workloads, all base conversion operations are performed from a small number of input limbs to a much larger set of output limbs. We exploit this observation to design a base conversion unit in which the number of multipliers and SRAM buffer size is proportional to the number of input limbs. This design has two advantages: (i) the number of multipliers and SRAM buffers is significantly reduced. (ii) the SRAM buffers only need to be single ported as they are either only read from or written into, unlike an output buffered design that requires double ported buffers to read, accumulate and write back.

Figure 10 shows the Cinnamon BCU, which is a vector unit composed of several lanes. We zoom into one lane to

show the details of its microarchitecture. Each lane contains a set of limb buffers, (depicted by $a[:][0]\dots$) and a base conversion factor table (represented by $B[0]\dots$). The buffers are attached to modular multipliers and adders to perform a multiply accumulate operation. The Cinnamon base conversion unit operates in two stages. In stage one, the factor table is loaded with the factors for a particular base conversion and coefficients of the input limbs are written into their respective limb buffer. Once all the input limbs have been written, stage 1 is over. In stage 2, coefficients of an output limb are generated by performing a multiply and accumulate between the input limbs' coefficients and their respective base conversion factors. As the multipliers and adders are pipelined, each lane produces one output coefficient every cycle. This process is repeated for all the other output limbs. Once all the required output limbs are generated, the buffers and factor tables can be reset for the next base conversion.

In Cinnamon, we set the maximum number of input bases in our BCU to 13 as this allows Cinnamon to implement all keyswitching in up to four digits. Further, we reduce the number of base conversion lanes down from 256 to 128 per cluster. This approach trades off some throughput but leads to halving the logic area and power of the BCU. These optimizations allow our optimized BCU to reduce the per cluster multipliers and SRAM buffers required by [56] from 15K to 1.6K and 3.31MB to 0.71 MB respectively.

5 Implementation

Software We build the Cinnamon DSL in python and the Cinnamon compiler in C++. The frontend of the compiler is forked from EVA [18] while we built the entire backend (polynomial IR and onwards) in about 15,000 lines of code.
Hardware We implemented the components of Cinnamon in RTL and synthesised them in a commercial 22nm PDK. We use a commercial memory compiler to compile the SRAMs used. We target a clock frequency of 1GHz for all functional units. We use [32] to estimate the data of the PHY nodes. Table 1 lists the area for each of the components. Cinnamon implements fully pipelined automorphism and four step NTT functional units using the design in CraterLake [56] and uses the base conversion unit (BCU) from Section 4.7. Finally, Cinnamon incorporates 4 HBM2E stacks with bandwidth of 512GB/s providing a total bandwidth of 2TB/s. We assume a bandwidth of 256GB/s on each network interface. Each chip has a vector register file capacity of 56MB. Based on the synthesis results, a single Cinnamon chip requires a die area of 223.18mm^2 at 22nm and a total power of 190W.

6 Methodology

We use the Cinnamon DSL to program FHE machine learning models and leverage the Cinnamon compiler infrastructure to compile instruction streams for the Cinnamon scale-out architecture. We built a cycle-accurate simulator to model the Cinnamon hardware architecture using timing information from the synthesized Cinnamon RTL.

Component	Area (mm^2)
NTT	34.08
Base Conversion Unit	14.12
Rotation	2.48
Addition	0.4
Multiply	2.55
Transpose	3.56
PRNG	5.72
Barrett Reduction	1.04
RNS Resolve	1.33
Total FU Area (2xAdd, 2xMul, 2xPRNG, + 1x Remaining FUs)	82.55
Base Conversion Unit Buffers (2.85MB)	11.44
Register File (56 MB)	80.9
4x HBM PHY Nodes	38.64
2x Network PHY Nodes	9.66
Total Chip Area	**223.18**

Table 1. Component wise Area breakdown.

6.1 Configurations

We evaluated the following variants of Cinnamon: **Cinnamon-4** and **Cinnamon-8**, a scale-out architecture with 4 and 8 Cinnamon chips with a ring interconnect and **Cinnamon-12**, a 12 chip configuration with a switch interconnect. These configurations are based on the Cinnamon microarchitecture described in Section 5 and Table 1. We also evaluate a single large monolithic chip design **Cinnamon-M**, which is a Cinnamon chip scaled up to a register file of 224MB, 8 clusters, 2 NTT units, 2 Transpose Units, 2 BCU buffers, 5 multiply and 5 add units, and an increased BCU block size of 32. This chip has an area of about 719.78mm^2 and represents a single monolithic chip that is similar in resources as **Cinnamon-4**. All benchmarks evaluated on Cinnamon-M are independently optimized to run on a single chip. We compare Cinnamon with recent state-of-the-art FHE architectures CraterLake [56], ARK [36], and CiFHER [38] using their best reported performance results.

6.2 Benchmarks and Applications

We evaluate Cinnamon with the following benchmarks and applications. All benchmarks are implemented at a 128-bit security and use a ring dimension of $N = 64K$. To test the correctness of our compiler and benchmark implementation, we built a CPU emulator for the Cinnamon ISA and used it to run all the benchmarks.
Bootstrapping [30] takes in a single ciphertext at level $l = 2$, raises it to level $l = 51$ and consumes 36 levels to bootstrap the ciphertext, leaving $l_{eff} = 13$ levels for the application.
Resnet implements a ResNet-20 CNN in FHE [43]. We evaluated the time taken to complete an inference on a single 32x32 encrypted image from the CIFAR-10 dataset.

HELR implements logistic regression training [42] with a mini batch block size of 256 on the MNIST data set. We perform training for 30 iterations.

BERT implements a BERT-Base transformer model inference. We use the techniques presented in [65] to implement softmax, gelu, and tanh functions and Newton-Raphson for division and inverse square roots. We report the time taken for a single encrypted inference on a 128 token input. This benchmark is much larger than any of the other benchmarks and highlights the opportunities for program-level parallelism. Cinnamon is the first work that explores the performance of large transformer models on FHE architectures.

7 Evaluation

Our evaluation sheds light on the following aspects:

- The performance scaling of Cinnamon on ML models
- The cost and performance-per-dollar of Cinnamon
- The impact of program-level and limb-level parallelism
- The scalability of our limb-level parallelism techniques
- The utilization of Cinnamon and how different parameters affect its performance

7.1 Performance Results

Table 2 reports the execution time of Cinnamon and Figure 11 shows the normalized speedups for different architectures and benchmarks. As the results show, Cinnamon's parallelization techniques and scale-out architecture is highly effective in improving the performance of bootstrap as well as real world machine learning workloads.

Comparing Cinnamon-4 to Cinnamon-M, we see that Cinnamon achieves its goal to match the performance of a large monolithic chip using 4 smaller Cinnamon chips. Furthermore, we see that the parallelization techniques introduced by Cinnamon lead to performance improvements. On average Cinnamon improves the performance of bootstrap and small models by 2.93×, 1.75×, and 2.45× compared to state-of-the-art architectures CraterLake, ARK, and CiFHER.

Cinnamon is the first work to target larger models such as BERT and therefore we only compare the results based on Cinnamon configurations. We see that the performance of Cinnamon is scalable as Cinnamon-12 achieves a speedup of 36, 600× compared to a 48-core Intel Xeon with a 256GB Memory CPU bringing down inference time from 17 hours to 1.67 seconds, thus enabling new opportunities for privacy-preserving machine learning. Using the Cinnamon framework to create parallel streams to parallelize the attention layer (6 parallel ciphertexts) and the GELU layer (12 parallel ciphertexts) enables further scaling. Together these represent about 85% of the program. Cinnamon uses groups of four chips for parallel bootstrap and then creates two and three independent streams for each group for the Cinnamon-8 and Cinnamon-12 configurations respectively. The results show that Cinnamon-8 achieves a 1.8× speedup and Cinnamon-12

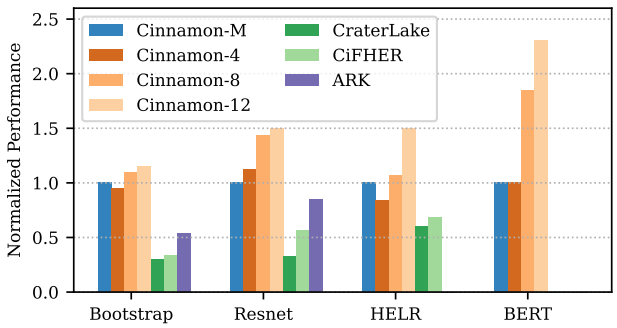

Figure 11. Normalized speedup of Cinnamon.

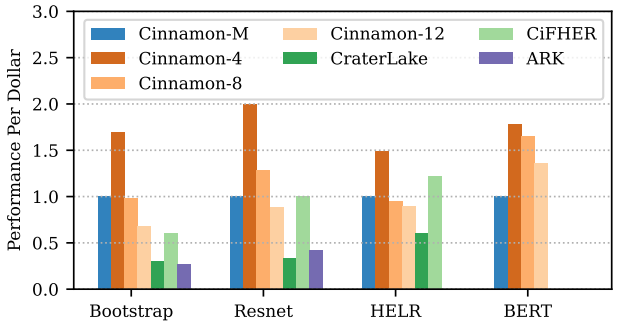

Figure 12. Relative Performance per Dollar.

achieves a 2.28× speedup compared to a large monolithic Cinnamon-M chip inline, with the available parallelism.

7.2 Cost and performance-per-dollar analysis

In this section we explore how Cinnamon reduces cost and achieves high performance-per-dollar for FHE programs. Actual manufacturing yield is a closely guarded industry secret and depends on several factors such as process technology maturity and ASIC design. However, we make a very optimistic assumption of defect density $D_0 = 0.2cm^{-2}$ and defect clustering $\alpha = 3$, leveraging prior work [60]. We assume a $300mm$ wafer and use the formula from [60] and publicly available data from semiconductor companies [21, 48] to estimate yield and wafer costs. We report the results in Table 3.

The results show that Cinnamon's strategy of splitting resources across chips enables Cinnamon to reduce the size of each individual chip to obtain a higher yield of 66% compared to the monolithic chip accelerator Cinnamon-M's 30%. Cinnamon chips also achieve a much higher yield compared to Craterlake and ARK. CiFHER leverages chiplets and a 7nm technology resulting in a high per-chiplet yield. Note that the cost of CiFHER is a single chiplet and thus underestimated. It does not include the cost of the chiplet interposer, as we were unable to obtain any publicly available data.

We use the performance data and tape-out costs to estimate performance-per-dollar. Figure 12 shows the results.

Benchmark	Cinnamon-M 1 Large Monolithic Chip	Cinnamon-4 4 Chips	Cinnamon-8 8 Chips	Cinnamon-12 12 Chips	CraterLake	CiFHER	ARK	CPU
Bootstrap (ms)	1.87	1.98	1.71	1.63	6.33	5.58	3.5	33s
Resnet (ms)	105.94	94.52	73.85	70.57	321.26	189	125	17.5min
HELR (ms)	73.20	87.61	68.74	48.76	121.91	106.88	-	14.9min
BERT (s)	3.83	3.83	2.07	1.67	-	-	-	1037.5min

Table 2. Execution time

Accelerator	Die area (mm^2)	Process	Yield (%)	Wafer Price ($/$mm^2$)	Yield Normalized Cost($)
ARK	418.3	7nm	48%	57500	50M
Cifher	47.08	7nm	90%	57500	3.5M
Craterlake	472	14nm	44%	23000	25M
Cinnamon-M	719.78	22nm	31%	10500	25M
Cinnamon	223.18	22nm	66%	10500	3.5M

Table 3. Manufacturing yield and estimated tape-out cost of FHE architectures.

For bootstrap and small ML programs we see that Cinnamon-4 provides significant improvements in performance-per-dollar of 5× and 2.68× on average compared to state-of-the-art monolithic designs like CraterLake and chiplet architectures like CiFHER. Smaller programs do not exhibit enough parallelism for larger architectures such as Cinnamon-8 and Cinnamon-12 and hence they provided limited benefits. However, looking into larger models such as BERT we see that all configurations provide major improvements in performance-per-dollar compared to a large monolithic Cinnamon-M chip.

7.3 Program and Limb-level Parallelism

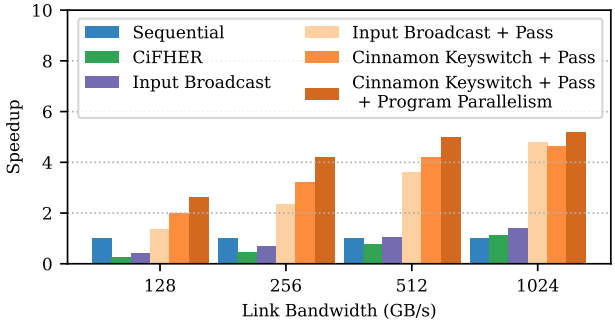

Figure 13. Comparison of various keyswitching techniques for bootstrapping on Cinnamon-4

To address the communication bottleneck of limb-level parallelism, Cinnamon proposes two new parallel keyswitching algorithms: *input broadcast* and *output aggregation* along with compiler *keyswitch passes* that perform reordering and batching optimizations to minimize inter-chip communication (Section 4.3.1). To better understand the effectiveness of these techniques, we show their speedups over a single-chip implementation.

Speedup Breakdown: Figure 13 shows the following configurations for bootstrap over different link bandwidth configurations: *Sequential* is the standard implementation of keyswitching running on a single Cinnamon chip. This is the baseline configuration. *CiFHER* uses the parallel keyswitching presented in [38]. *Input Broadcast* uses Input Broadcast Keyswitching to parallelize keyswitching. *Input Broadcast + Pass* uses Input Broadcast Keyswitching coupled with Cinnamon's compiler pass for reordering and communication batching (*Pass*). *Cinnamon Keyswitch + Pass* relies on the Cinnamon compiler to select between Cinnamon's algorithms of input broadcast and output aggregation keyswitching and perform reordering operations to minimize communication. Finally, *Cinnamon Keyswitch + Pass + Program parallelism* uses Cinnamon's program-level parallelism stream features on top of the *Cinnamon Keyswitch + Pass* to create two parallel streams mapped to two chips each to compute the two homomorphic mod computations in parallel.

At a 256 GB/s link bandwidth, we observe that CiFHER results in a 2.14× slowdown over *Sequential*. *Input Broadcast Keyswitching + Pass* provides a 2.34× improvement over sequential. As explained in 4.3.1, Input Broadcast Keyswitching reduces inter-chip communication and provides significant benefits when combined with *Pass* as it is able to effectively batch communication across program patterns. *Cinnamon Keyswitch + Pass* achieves an improvement of 3.22× over sequential. This further speedup is because of the introduction of output aggregation keyswitching that is complementary to input broadcast keyswitching and enables optimizing program patterns that input broadcast keyswitching cannot optimize. Finally, *Cinnamon Keyswitch +Pass + Program Parallelism* provides a 4.18× speed up over sequential by adding the Cinnamon compiler's parallel stream feature. Increasing the link bandwidth to $512GB/s$, we see that our parallelization techniques result in a 5× speedup over the sequential implementation. At $1024GB/s$, we see little improvement for Cinnamon, as it has been able to completely eliminate the network bottleneck at $512GB/s$ and the design becomes compute-bound.

Overall, the results show that the scale-out approach of Cinnamon with our proposed keyswitching algorithms, together with our compiler optimizations across multiple

keyswitching operations, lead to effective parallelization even at relatively low bandwidth.

7.4 Comparing Keyswitching Implementations

In this section, we compare Cinnamon's parallel keyswitching algorithms: input broadcast and output aggregation keyswitching to CiFHER's parallel keyswitching.

CiFHER's keyswitching algorithm requires broadcasts at both the mod up and mod down steps. Whereas Cinnamon's keyswitching algorithms limit the communication in parallel keyswitching to only one point: either the beginning or the end of keyswitching. This is done because reordering and batching optimizations can only batch one of the communication operations. We show through algorithmic and empirical evaluation how Cinnamon's algorithms provide significant gains over CiFHER.

Algorithmic Analysis: Let us consider the first pattern from Section 4.3.1: r different rotations on the same ciphertext. In this case, CiFHER requires 3 broadcasts for each of the r keyswitches. Batching can optimize the broadcast at the mod up step, but each keyswitch would still require the 2 broadcasts at the mod down step. Thus, CiFHER requires $O(r)$ broadcasts for r keyswitches. In contrast, Input Broadcast Keyswitching requires just one broadcast at the beginning of each keyswitch, which can be batched to just one broadcast for the whole batch of r rotations. As a result, Cinnamon requires 1 broadcast for r rotations.

Similarly, for the second program pattern: r different rotations on r different ciphertexts followed by aggregation, CiFHER would require 3 broadcasts for each of the r keyswitches. Batching can optimize the broadcast at the mod down step, but each keyswitch would still require the broadcasts at the mod up step. Thus, CiFHER requires $O(r)$ broadcasts for r keyswitches. In contrast, using output aggregation keyswitching, requires 2 aggregations at the end of each keyswitch. This can be batched to just two aggregations for the whole batch. Thus, Cinnamon only requires 2 aggregations for the r rotations.

Overall, Cinnamon provides a significant algorithm improvement over prior CiFHER.

Empirical Analysis: We evaluate how CiFHER and Cinnamon's keyswitching techniques perform with batching enabled on the Cinnamon-4 configuration for the bootstrapping benchmark. As expected by the algorithmic analysis above, Cinnamon's keyswitching algorithms reduce the inter-chip communication by 2.25× over CiFHER with batching, resulting in a 1.94× speedup. When program parallelism is introduced to both Cinnamon and CiFHER, the speedup of Cinnamon improves to 2.11× over CiFHER.

7.5 Scalability of Limb Level Parallelism

The current bootstrap implementation of Cinnamon refreshes 13 levels of compute. This configuration aligns with prior work [36, 56] and simplifies the direct comparison. In practice, different bootstrap implementations can tune the refresh levels by trading-off compute cost. This creates an interesting trade-off between the cost of each bootstrap and the frequency of bootstrapping. DaCapo [16] is a recent compiler that explores this trade-off. To this end, in this paper, we show how Cinnamon's limb level parallelism techniques scale with bootstrap implementations that refresh more levels. Specifically, we consider **Bootstrap-21**, a configuration of bootstrapping that refreshes 21 levels and compare it to **Bootstrap-13** in Figure 14.

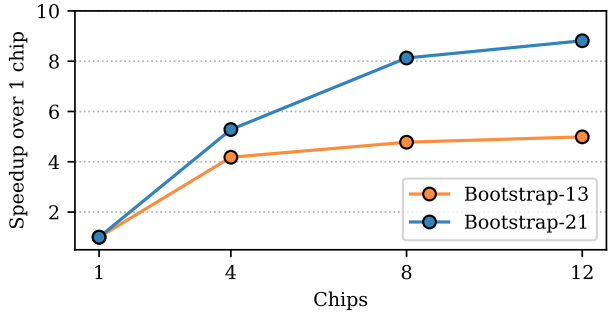

Figure 14. Speedup for Bootstrap-13 and Bootstrap-21

As we can see in the figure, Bootstrap-13 results in a 4.18× speedup on Cinnamon-4. It increases modestly to 4.78× on Cinnamon-8 and 4.98× on Cinnamon-12. This is because communication costs become the limiting factor and additional parallelism is not beneficial. However, Bootstrap-21 yields speedups of 5.28× on Cinnamon-4, 8.12× on Cinnamon-8, and 8.81× on Cinnamon-12. The speedups increase because this configuration has almost 2× the compute of Bootstrap-13 and benefits more from the extra parallelism and resources provided by scaling to Cinnamon-8. This shows that Cinnamon's limb level parallelism techniques can be used to speedup bootstrap implementations that refresh more levels and opens the door for new research that explores the frequency-cost trade-off of bootstrapping through parallelism.

7.6 Hardware Utilization and Sensitivity Study

Figure 15 shows the utilization of Cinnamon across different architectures and compute, memory bandwidth, and network bandwidth resources. Compute utilization is reported as the number of cycles in which a unit is actively processing data. We report the area-weighted average of all FUs. Memory and Network utilization is the cycles memory or the network is active. For Cinnamon-4, we show the average results of all the benchmarks and for Cinnamon-8 and 12, we show the results on BERT. As the results show, Cinnamon-4 is able to achieve a high utilization close to 60% for compute, memory bandwidth and network bandwidth resources across benchmarks. Cinnamon-8 also achieves a high utilization of compute, memory and bandwidth for BERT. Cinnamon-12 starts to witness lower compute and memory utilization

as the program has been parallelized and the narrower sections that have less parallelism become a larger part of the program.

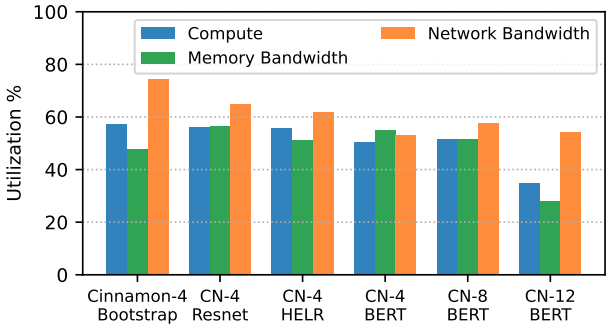

Figure 15. Cinnamon Utilization Results

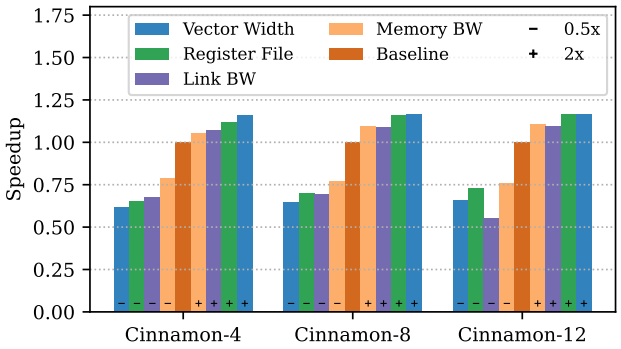

Figure 16. Cinnamon Sensitivity Results

Figure 16 shows the sensitivity of different Cinnamon configurations to halving and doubling the register file size, link and memory bandwidths, and vector width. For Cinnamon-4, we take the geomean of speedups across the 4 benchmarks. For Cinnamon-8 and Cinnamon-12, we report the results of the BERT benchmark. We observe that halving any of the resources of a configuration results in a slowdown of about $20-40\%$, with a geomean of 32%. However, on doubling any resource, we observe modest speedups ranging from $2-20\%$ with a geomean of 10%. This shows that Cinnamon chips are appropriately sized to deliver high performance in all configurations and applications.

8 Other Related Work

FHE Accelerators: FAB [4] and F1[55] were some of the first accelerators, but they did not accelerate FHE computations that require bootstrapping. HEAP [3] is an FPGA accelerator that uses scheme switching between CKKS and TFHE [17] for bootstrapping but can't efficiently support large models due to quadratic growth in the cost of scheme switching.

Sharp [35] proposes modifying FHE application to a precision of 36 bits to improve the utilization of the multiplicative budget. This enables applications to reduce the frequency of bootstrapping. However, Sharp has been tested only on small ML models that could fit in 36 bits. For BERT, we required about 50-80 bits of precision, and also as BitPacker [57] notes, some other models too require scales higher than 36 bits. Nevertheless, the optimization techniques presented in Sharp are orthogonal to the optimizations presented in Cinnamon. Cinnamon proposes speeding applications by exploiting parallelism to scale, and all the optimizations presented in Sharp can be used in conjunction with our techniques.

Precision optimizations of FHE programs: BitPacker [57] enables decoupling of scaling factors in an FHE programs from the arithmetic bit width of the underlying FHE accelerator to better utilize the multiplicative budget and reduce bootstrapping frequency. These techniques can also be used in conjunction with Cinnamon.

CPU/GPU Acceleration of FHE: [1, 2, 5, 58] are popular open source libraries for running FHE programs on CPUs. [9] optimizes FHE using AVX instructions. [22, 33] propose GPU acceleration of FHE.

Compiler Infrastructure: HEIR [27] is a compiler toolchain for FHE. It aims to standardize intermediate representations for FHE. Cinnamon's IR, compiler optimizations, and keyswitching algorithms can be used to extend HEIR. Further, the Cinnamon ISA can serve as a compilation target for the HEIR framework.

Hybrid Protocols: Cheetah [53] is an accelerator for Hybrid HE-MPC [34]. Such schemes avoid bootstrapping but incur high server-client communication costs. CHOCO-TACO [62] accelerates the generation of ciphertexts and evalkeys on the client side.

9 Conclusion

This paper presented Cinnamon, a framework for scale-out encrypted computing of state-of-the-art ML workloads in FHE. Cinnamon focused on exploiting parallelism along two dimensions, program and limb level, and composing them together. By cutting across algorithms, compilers, and architecture Cinnamon was able to split large FHE computation over multiple small chips. As a result, Cinnamon is able to achieve the twin benefits of fast FHE computation and reduced costs. Additionally, Cinnamon demonstrates for the first time that a language model like BERT can be practically implemented in FHE enabling new opportunities for privacy preserving machine learning.

Acknowledgments

We would like to thank the anonymous reviewers; shepherd Mingyu Gao; Qi Pang and Trevor Leong for help with benchmarks; and Siddharth Das, Ken Mai and Tathagatha Srimani for help with the synthesis. This research was funded by a CMU CyLab grant and NSF grant CCF 2217016.

References

[1] 2023. HEAAN. Online: https://github.com/snucrypto/HEAAN.

[2] 2023. Lattigo v5. Online: https://github.com/tuneinsight/lattigo. EPFL-LDS, Tune Insight SA.

[3] Rashmi Agrawal, Anantha Chandrakasan, and Ajay Joshi. 2024. HEAP: A Fully Homomorphic Encryption Accelerator with Parallelized Bootstrapping. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. 756–769. https://doi.org/10.1109/ISCA59077.2024.00060

[4] R. Agrawal, L. de Castro, G. Yang, C. Juvekar, R. Yazicigil, A. Chandrakasan, V. Vaikuntanathan, and A. Joshi. 2023. FAB: An FPGA-based Accelerator for Bootstrappable Fully Homomorphic Encryption. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 882–895.

[5] Ahmad Al Badawi, Jack Bates, Flavio Bergamaschi, David Bruce Cousins, Saroja Erabelli, Nicholas Genise, Shai Halevi, Hamish Hunt, Andrey Kim, Yongwoo Lee, Zeyu Liu, Daniele Micciancio, Ian Quah, Yuriy Polyakov, Saraswathy R.V., Kurt Rohloff, Jonathan Saylor, Dmitriy Suponitsky, Matthew Triplett, Vinod Vaikuntanathan, and Vincent Zucca. 2022. OpenFHE: Open-Source Fully Homomorphic Encryption Library. Cryptology ePrint Archive, Paper 2022/915. https://eprint.iacr.org/2022/915

[6] Jean-Claude Bajard, Julien Eynard, Anwar Hasan, and Vincent Zucca. 2016. A Full RNS Variant of FV like Somewhat Homomorphic Encryption Schemes. Cryptology ePrint Archive, Paper 2016/510. https://eprint.iacr.org/2016/510

[7] L. A. Belady. 1966. A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal* 5, 2 (1966), 78–101. https://doi.org/10.1147/sj.52.0078

[8] Song Bian, Zhou Zhang, Haowen Pan, Ran Mao, Zian Zhao, Yier Jin, and Zhenyu Guan. 2023. HE3DB: An Efficient and Elastic Encrypted Database Via Arithmetic-And-Logic Fully Homomorphic Encryption. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. 2930–2944.

[9] Fabian Boemer, Sejun Kim, Gelila Seifu, Fillipe D.M. de Souza, and Vinodh Gopal. 2021. Intel HEXL: Accelerating Homomorphic Encryption with Intel AVX512-IFMA52. In *Proceedings of the 9th on Workshop on Encrypted Computing & Applied Homomorphic Cryptography* (Virtual Event, Republic of Korea) *(WAHC '21)*. Association for Computing Machinery, New York, NY, USA, 57–62. https://doi.org/10.1145/3474366.3486926

[10] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. 2012. (Leveled) Fully Homomorphic Encryption without Bootstrapping. In *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference* (Cambridge, Massachusetts) *(ITCS '12)*. Association for Computing Machinery, New York, NY, USA, 309–325. https://doi.org/10.1145/2090236.2090262

[11] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. 2019. A Systematic Evaluation of Transient Execution Attacks and Defenses. In *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, Santa Clara, CA, 249–266. https://www.usenix.org/conference/usenixsecurity19/presentation/canella

[12] Capital One. [n. d.]. Information on the Capital One cyber incident. https://www.capitalone.com/digital/facts2019/.

[13] Jung Cheon, Han Kyoohyung, Andrey Kim, Miran Kim, and Yongsoo Song. 2018. *Bootstrapping for Approximate Homomorphic Encryption*. 360–384. https://doi.org/10.1007/978-3-319-78381-9_14

[14] Jung Cheon, Han Kyoohyung, Andrey Kim, Miran Kim, and Yongsoo Song. 2018. *Bootstrapping for Approximate Homomorphic Encryption*. 360–384. https://doi.org/10.1007/978-3-319-78381-9_14

[15] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. 2017. Homomorphic encryption for arithmetic of approximate numbers. In *Advances in Cryptology–ASIACRYPT 2017: 23rd International Conference on the Theory and Applications of Cryptology and Information Security, Hong Kong, China, December 3-7, 2017, Proceedings, Part I 23*. Springer, 409–437.

[16] Seonyoung Cheon, Yongwoo Lee, Dongkwan Kim, Ju Min Lee, Sunchul Jung, Taekyung Kim, Dongyoon Lee, and Hanjun Kim. 2024. DaCapo: Automatic Bootstrapping Management for Efficient Fully Homomorphic Encryption. In *33rd USENIX Security Symposium (USENIX Security 24)*. USENIX Association, Philadelphia, PA, 6993–7010. https://www.usenix.org/conference/usenixsecurity24/presentation/cheon

[17] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. August 2016. TFHE: Fast Fully Homomorphic Encryption Library. https://tfhe.github.io/tfhe/

[18] Roshan Dathathri, Blagovesta Kostova, Olli Saarikivi, Wei Dai, Kim Laine, and Madan Musuvathi. 2020. EVA: an encrypted vector arithmetic language and compiler for efficient homomorphic computation. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '20)*. ACM. https://doi.org/10.1145/3385412.3386023

[19] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. arXiv:1810.04805

[20] European Parliament and Council of the European Union. [n. d.]. Regulation (EU) 2016/679 of the European Parliament and of the Council. https://data.europa.eu/eli/reg/2016/679/oj

[21] EuroPracticeIC. 2024. EUROPRACTICE | Schedules 2024. https://europractice-ic.com/schedules-prices-2024/

[22] Shengyu Fan, Zhiwei Wang, Weizhi Xu, Rui Hou, Dan Meng, and Mingzhe Zhang. 2023. TensorFHE: Achieving Practical Computation on Encrypted Data Using GPGPU. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 922–934. https://doi.org/10.1109/HPCA56546.2023.10071017

[23] Federal Trade Commission. [n. d.]. Equifax Data Breach Settlement. https://www.ftc.gov/enforcement/refunds/equifax-data-breach-settlement.

[24] Craig Gentry. 2009. Fully Homomorphic Encryption Using Ideal Lattices. In *Proceedings of the Forty-First Annual ACM Symposium on Theory of Computing* (Bethesda, MD, USA) *(STOC '09)*. Association for Computing Machinery, New York, NY, USA, 169–178. https://doi.org/10.1145/1536414.1536440

[25] Ran Gilad-Bachrach, Nathan Dowlin, Kim Laine, Kristin Lauter, Michael Naehrig, and John Wernsing. 2016. CryptoNets: Applying Neural Networks to Encrypted Data with High Throughput and Accuracy. In *Proceedings of The 33rd International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 48)*. PMLR, New York, New York, USA, 201–210. https://proceedings.mlr.press/v48/gilad-bachrach16.html

[26] Eric Goldman. 2020. An introduction to the california consumer privacy act (ccpa). *Santa Clara Univ. Legal Studies Research Paper* (2020).

[27] Google. [n. d.]. HEIR: Homomorphic Encryption Intermediate Representation. https://heir.dev/

[28] Shai Halevi and Victor Shoup. 2020. Design and implementation of HElib: a homomorphic encryption library. Cryptology ePrint Archive, Paper 2020/1481. https://eprint.iacr.org/2020/1481

[29] Kyoohyung Han, Seungwan Hong, Jung Hee Cheon, and Daejun Park. 2018. Efficient Logistic Regression on Large Encrypted Data. Cryptology ePrint Archive, Paper 2018/662. https://eprint.iacr.org/2018/662

[30] Kyoohyung Han and Dohyeong Ki. 2019. Better Bootstrapping for Approximate Homomorphic Encryption. Cryptology ePrint Archive, Paper 2019/688. https://eprint.iacr.org/2019/688

[31] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 770–778. https://doi.org/10.1109/CVPR.2016.90

[32] Sangyun Hwang, Taekyung Yeo Kwanyeob Chae, Sangsoo Park, Won Lee, Shinyoung Lee, Soo-Min Lee, Kihwan Seong, Eunkyoung Ha, Eunsu Kim, Jihun Oh, Kyoung-Hoi Koo, Sanghune Park, and Jongshin Shin. 2024. A 3.2 Gbps/pin HBM2E PHY with Low Power I/O and Enhanced Training Scheme for 2.5D System-in-Package Solutions. In *HotChips'33*.

[33] Wonkyung Jung, Sangpyo Kim, Jung Ho Ahn, Jung Hee Cheon, and Younho Lee. 2021. Over 100x Faster Bootstrapping in Fully Homomorphic Encryption through Memory-centric Optimization with GPUs. Cryptology ePrint Archive, Paper 2021/508. https://eprint.iacr.org/2021/508

[34] Chiraag Juvekar, Vinod Vaikuntanathan, and Anantha Chandrakasan. 2018. GAZELLE: A Low Latency Framework for Secure Neural Network Inference. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, Baltimore, MD, 1651–1669. https://www.usenix.org/conference/usenixsecurity18/presentation/juvekar

[35] Jongmin Kim, Sangpyo Kim, Jaewan Choi, Jaiyoung Park, Donghwan Kim, and Jung Ho Ahn. 2023. SHARP: A Short-Word Hierarchical Accelerator for Robust and Practical Fully Homomorphic Encryption. In *Proceedings of the 50th Annual International Symposium on Computer Architecture* (Orlando, FL, USA) *(ISCA '23)*. Association for Computing Machinery, New York, NY, USA, Article 18, 15 pages. https://doi.org/10.1145/3579371.3589053

[36] Jongmin Kim, Gwangho Lee, Sangpyo Kim, Gina Sohn, Minsoo Rhu, John Kim, and Jung Ho Ahn. 2022. ARK: Fully Homomorphic Encryption Accelerator with Runtime Data Generation and Inter-Operation Key Reuse. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1237–1254. https://doi.org/10.1109/MICRO56248.2022.00086

[37] Miran Kim, Yongsoo Song, Baiyu Li, and Daniele Micciancio. 2020. Semi-Parallel logistic regression for GWAS on encrypted data. *BMC Medical Genomics* 13, 7 (2020), 99. https://doi.org/10.1186/s12920-020-0724-z

[38] Sangpyo Kim, Jongmin Kim, Jaeyoung Choi, and Jung Ho Ahn. 2024. CiFHER: A Chiplet-Based FHE Accelerator with a Resizable Structure. arXiv:2308.04890 [cs.AR]

[39] Sangpyo Kim, Jongmin Kim, Michael Jaemin Kim, Wonkyung Jung, John Kim, Minsoo Rhu, and Jung Ho Ahn. 2022. BTS: an accelerator for bootstrappable fully homomorphic encryption. In *Proceedings of the 49th Annual International Symposium on Computer Architecture (ISCA '22)*. ACM. https://doi.org/10.1145/3470496.3527415

[40] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2019. Spectre Attacks: Exploiting Speculative Execution. In *IEEE Symposium on Security and Privacy (S&P)*.

[41] Esmaeil Mohammadian Koruyeh, Khaled N. Khasawneh, Chengyu Song, and Nael B. Abu-Ghazaleh. 2018. Spectre Returns! Speculation Attacks using the Return Stack Buffer. In *12th USENIX Workshop on Offensive Technologies, WOOT 2018, Baltimore, MD, USA, August 13-14, 2018*. USENIX Association.

[42] Han Kyoohyung, Seungwan Hong, Jung Cheon, and Daejun Park. 2019. Logistic Regression on Homomorphic Encrypted Data at Scale. *Proceedings of the AAAI Conference on Artificial Intelligence* 33 (07 2019), 9466–9471. https://doi.org/10.1609/aaai.v33i01.33019466

[43] Joon-Woo Lee, HyungChul Kang, Yongwoo Lee, Woosuk Choi, Jieun Eom, Maxim Deryabin, Eunsang Lee, Junghyun Lee, Donghoon Yoo, Young-Sik Kim, and Jong-Seon No. 2021. Privacy-Preserving Machine Learning with Fully Homomorphic Encryption for Deep Neural Network. *CoRR* abs/2106.07229 (2021). arXiv:2106.07229 https://arxiv.org/abs/2106.07229

[44] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown: Reading Kernel Memory from User Space. In *USENIX Security'18*.

[45] Natasha Lomas. [n. d.]. Italy orders ChatGPT blocked citing data protection concerns. https://techcrunch.com/2023/03/31/chatgpt-blocked-italy/. Last accessed: 05/28/2023.

[46] Cecily Mauran. [n. d.]. Whoops, Samsung workers accidentally leaked trade secrets via ChatGPT. https://mashable.com/article/samsung-chatgpt-leak-details. Last accessed: 05/28/2023.

[47] Ahmet Can Mert, Erdinç Öztürk, and Erkay Savaş. 2019. Design and Implementation of a Fast and Scalable NTT-Based Polynomial Multiplier Architecture. In *2019 22nd Euromicro Conference on Digital System Design (DSD)*. 253–260. https://doi.org/10.1109/DSD.2019.00045

[48] MuseSemi. 2024. Muse Semiconductor TSMC University FinFET Program Service and Price. https://www.musesemi.com/university-finfet-program

[49] NVIDIA. [n. d.]. NVIDIA Scalable Hierarchical Aggregation and Reduction Protocol (SHARP). https://docs.nvidia.com/networking/display/sharpv300

[50] NVIDIA. [n. d.]. Nvlink. https://www.nvidia.com/en-us/data-center/nvlink/

[51] NVIDIA. 2024. NVIDIA Blackwell Platform Arrives to Power a New Era of Computing. https://nvidianews.nvidia.com/news/nvidia-blackwell-platform-arrives-to-power-a-new-era-of-computing

[52] Kate Park. [n. d.]. Samsung bans use of generative AI tools like ChatGPT after April internal data leak. https://techcrunch.com/2023/05/02/samsung-bans-use-of-generative-ai-tools-like-chatgpt-after-april-internal-data-leak. Last accessed: 05/28/2023.

[53] Brandon Reagen, Wooseok Choi, Yeongil Ko, Vincent Lee, Gu-Yeon Wei, Hsien-Hsin S. Lee, and David Brooks. 2020. Cheetah: Optimizing and Accelerating Homomorphic Encryption for Private Inference. arXiv:2006.00505

[54] Xuanle Ren, Le Su, Zhen Gu, Sheng Wang, Feifei Li, Yuan Xie, Song Bian, Chao Li, and Fan Zhang. 2022. HEDA: multi-attribute unbounded aggregation over homomorphically encrypted database. *Proceedings of the VLDB Endowment* 16, 4 (2022), 601–614.

[55] Nikola Samardzic, Axel Feldmann, Aleksandar Krastev, Srinivas Devadas, Ronald Dreslinski, Christopher Peikert, and Daniel Sanchez. 2021. F1: A Fast and Programmable Accelerator for Fully Homomorphic Encryption. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture* (Virtual Event, Greece) *(MICRO '21)*. Association for Computing Machinery, New York, NY, USA, 238–252. https://doi.org/10.1145/3466752.3480070

[56] Nikola Samardzic, Axel Feldmann, Aleksandar Krastev, Nathan Manohar, Nicholas Genise, Srinivas Devadas, Karim Eldefrawy, Chris Peikert, and Daniel Sanchez. 2022. CraterLake: A Hardware Accelerator for Efficient Unbounded Computation on Encrypted Data. In *Proceedings of the 49th Annual International Symposium on Computer Architecture* (New York, New York) *(ISCA '22)*. Association for Computing Machinery, New York, NY, USA, 173–187. https://doi.org/10.1145/3470496.3527393

[57] Nikola Samardzic and Daniel Sanchez. 2024. BitPacker: Enabling High Arithmetic Efficiency in Fully Homomorphic Encryption Accelerators. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (La Jolla, CA, USA) *(ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 137–150. https://doi.org/10.1145/3620665.3640397

[58] SEAL 2023. Microsoft SEAL (release 4.1). https://github.com/Microsoft/SEAL. Microsoft Research, Redmond, WA..

[59] Deepraj Soni, Negar Neda, Naifeng Zhang, Benedict Reynwar, Homer Gamil, Benjamin Heyman, Mohammed Nabeel, Ahmad Al Badawi, Yuriy Polyakov, Kellie Canida, Massoud Pedram, Michail Maniatakos, David Bruce Cousins, Franz Franchetti, Matthew French, Andrew Schmidt, and Brandon Reagen. 2023. RPU: The Ring Processing Unit. In *2023 IEEE International Symposium on Performance Analysis of Systems*

and Software (ISPASS). 272–282. https://doi.org/10.1109/ISPASS57527.2023.00034

[60] Dylan Stow, Yuan Xie, Taniya Siddiqua, and Gabriel H. Loh. 2017. Cost-effective design of scalable high-performance systems using active and passive interposers. In *Proceedings of the 36th International Conference on Computer-Aided Design* (Irvine, California) *(ICCAD '17)*. IEEE Press, 728–735.

[61] The Verge. [n. d.]. T-Mobile discloses its second data breach so far this year. https://www.theverge.com/2023/5/2/23707894/tmobile-data-breach-april-personal-data-pin-hack-security.

[62] McKenzie van der Hagen and Brandon Lucia. 2022. Client-optimized algorithms and acceleration for encrypted compute offloading. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3503222.3507737

[63] Wired. [n. d.]. The Snowflake Attack May Be Turning Into One of the Largest Data Breaches Ever. https://www.wired.com/story/snowflake-breach-advanced-auto-parts-lendingtree/.

[64] Yinghao Yang, Huaizhi Zhang, Shengyu Fan, Hang Lu, Mingzhe Zhang, and Xiaowei Li. 2023. Poseidon: Practical Homomorphic Encryption Accelerator. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 870–881. https://doi.org/10.1109/HPCA56546.2023.10070984

[65] Jiawen Zhang, Jian Liu, Xinpeng Yang, Yinghao Wang, Kejia Chen, Xiaoyang Hou, Kui Ren, and Xiaohu Yang. 2024. Secure Transformer Inference Made Non-interactive. Cryptology ePrint Archive, Paper 2024/136. https://eprint.iacr.org/2024/136

A Artifact Appendix

A.1 Abstract

Our artifact provides the simulation tool and compiled programs that we used to evaluate the Cinnamon Framework. We provide scripts to run the simulations and reproduce the results of our experiments.

A.2 Artifact check-list (meta-information)

- **Run-time environment:** Linux with Docker Containers
- **Hardware:** x86_64 server
- **Output:** Table 2 and Figures 11, 12, 13 , 14 of the paper
- **Experiments:** Please refer to section appendix A.5
- **Disk space required:** 50GB
- **Time needed to prepare workflow:** 20 minutes
- **Time needed to complete experiments 24 hours:**
- **Code licenses: CC**
- **Archived** : https://doi.org/10.1184/R1/27635106. However, we recommend using the latest version from docker

A.3 Description

A.3.1 How to access. The container image is available at https://hub.docker.com/repository/docker/sidjay10/asplos25_cinnamon_artifact/general. The code is available at https://github.com/sidjay10/asplos25_cinnamon_artifact/

A.3.2 Software dependencies. We use Docker and provide a complete docker image that captures all the software dependencies required to build our simulation infrastructure

A.4 Installation

Build Time: 20 minutes. Please follow the steps in A.5

A.5 Experiment workflow

A.5.1 Overview. We are releasing the Cinnamon simulator along with the benchmarks we compiled for the evaluation. To reproduce our results, follow the steps below.

Step 1: Pull the container image

```
docker pull sidjay10/asplos25_cinnamon_artifact:v1
```

Step 2: Run the container.

```
mkdir -p asplos25_cinnamon_artifact/outputs
cd asplos25_cinnamon_artifact
docker run --rm -it \
-v $(pwd)/outputs:/cinnamon_artifact/outputs \
--name cinnamon \
sidjay10/asplos25_cinnamon_artifact:v1
```

Step 3: Build the Cinnamon Simulator

```
docker exec -it cinnamon ./build_cinnamon.sh
```

This command builds the cinnamon element within the SST simulator. The rest of the SST simulator is pre-built in the container image. This command should take about 5 minutes.

Step 4: Run the simulations to generate Figure 13

```
docker exec -it cinnamon \
    ./run_keyswitch_comparison.sh
```

This command should take about 10 minutes. When it completes, it will produce keyswitch_comparison.pdf under the output folder.

Step 5: Run the simulations to generate Figure 14

```
docker exec -it cinnamon \
    ./run_bootstrap_comparison.sh
```

This command should take about 10 minutes. When it completes, it will produce bootstrap_comparison.pdf under the output folder.

Step 6: Run the simulations to generate Figure 11, Figure 12 and Table 2

```
docker exec -it cinnamon ./run_performance.sh
```

This command should take about a day. This is because it runs the simulations for long-running benchmarks. When it completes, it will produce performance.pdf, performance_per_dollar.pdf and performance_table.txt under the outputs folder.

Step 7: Cleanup

```
docker stop cinnamon
```

A.6 Evaluation and expected results

The collected plots and tables for the simulations should match Figures 11, 12, 13 , 14 and Table 2 of the paper.

A.7 Methodology

Submission, Review, and Badging Methodology:

- https://www.acm.org/publications/policies/artifact-review-badging
- http://cTuning.org/ae/submission-20201122.html
- http://cTuning.org/ae/reviewing-20201122.html