Advancing Adoption of Privacy Preserving Machine Learning in AI-Enabled Enterprises: Efficient Homomorphic Encryption through Radix Caching

by Paolo Erico Gimeno

B.S. in Biomedical Engineering, May 2012, Virginia Commonwealth University
M.S. in Data Analytics, May 2023, The George Washington University

A Praxis submitted to

The Faculty of
The School of Engineering and Applied Science
of The George Washington University
in partial fulfillment of the requirements
for the degree of Doctor of Engineering

August 31, 2025

Praxis directed by

Mohamed Belkhayat, Lecturer and Research Adviser, Electrical Engineering, Artificial Intelligence and Machine Learning

The School of Engineering and Applied Science of The George Washington University certifies that Paolo Erico A. Gimeno has passed the Final Examination for the degree of Doctor of Engineering as of August 31, 2025. This is the final and approved form of the Praxis.

Advancing Adoption of Privacy Preserving Machine Learning in AI-Enabled Enterprises: Efficient Homomorphic Encryption through Radix Caching

Paolo Erico Gimeno

Praxis Research Committee:

Amir Etemadi, Assistant Professor of Engineering and Applied Science, Praxis Chair

Adewale Akinfaderin, Professorial Lecturer in Engineering and Applied Science, Committee Member

Mohamed Belkhayat, Lecturer and Research Adviser, Electrical Engineering, Artificial Intelligence and Machine Learning, Committee Member

© 2025 by Paolo Erico Gimeno
All rights reserved

Acknowledgements

I would like to express my deepest gratitude to Dr. Mohamed Belkhayat for his invaluable guidance, insight, and unwavering support throughout the development of this praxis. I also thank my committee members not only for their expertise and thoughtful feedback, but also for their uncanny ability to find the one question I hadnt prepared for. You kept me on my toes, occasionally on the edge of my seat, and always moving forward.

Abstract of Praxis

Advancing Adoption of Privacy Preserving Machine Learning in AI-Enabled Enterprises: Efficient Homomorphic Encryption through Radix Caching

Privacy-preserving machine learning, especially through homomorphic encryption (HE), is essential to facilitate secure data processing in AI-driven enterprise systems. Nevertheless, HE's computational burden significantly restricts its practical implementation, particularly in critical domains like healthcare and edge devices, where data security and processing efficiency are essential. This study investigates a caching solution to mitigate this bottleneck, focusing on caching radix trees as a means to enhance the computational efficiency of HE-based processes.

The research introduces and evaluates a caching approach that uses currently available homomorphic encryption libraries. The aim is to reduce the processing time of homomorphic encryption operations with the goal of making privacy-preserving machine learning applications more scalable. This optimization method deliberately caches ciphertexts, facilitating a transition from quadratic to almost linear computational complexity. Key performance indicators are examined, including the inference time and correctness of the computation across various HE schemes and compute configurations.

The research findings indicate that radix tree caching can substantially improve HE performance, making it suitable for wider implementation in enterprise settings. This contribution

tackles the efficiency difficulties of HE while creating a framework for privacy-preserving approaches that facilitate real-time, data-intensive applications without jeopardizing security or accuracy. This practice offers a pragmatic solution to a significant obstacle in privacy-preserving machine learning, enhancing both theoretical and applied understanding in secure AI. (D. Zhao, 2022)

Table of Contents

Abstract **iv**

1 Introduction **1**

1.1 Research Motivation 4

1.2 Problem Statement 5

1.3 Thesis Statement 5

1.4 Research Objectives 6

1.5 Research Questions and Hypotheses 6

1.5.1 Research Questions 6

1.5.2 Research Hypotheses 7

1.6 Nature of the Study 7

1.7 Scope of Research 8

1.8 Research Limitations 9

1.9 Organization of the Praxis 10

2 Literature Review **11**

2.1 Evolution of HE in the Context of Cryptography 11

2.1.1 Gentry's FHE Scheme 11

2.1.2 Noise Accumulation 12

2.1.3 Bootstrapping Process 13

2.2 Current Homomorphic Encryption Schemes 13

2.2.1 Partially Homomorphic Encryption (PHE) 13

2.2.2 Somewhat Homomorphic Encryption (SWHE) 14

2.2.3 Fully Homomorphic Encryption (FHE) 14

2.3 Implementation Libraries 15

2.3.1 Microsoft SEAL 15

2.3.2 Helib Library . . . 15
2.3.3 TFHE - Fast Fully Homomorphic Encryption over the Torus . . . 16
2.3.4 FHEW: Fastest Fourier Transform in the West . . . 16
2.3.5 Lattigo . . . 16
2.4 Encryption Algorithms for Analysis . . . 17
2.4.1 Brakerski/Fan-Vercauteren (BFV) Scheme . . . 17
2.4.2 Cheon-Kim-Kim-Song (CKKS) Scheme . . . 18
2.4.3 Paillier Cryptosystem . . . 19
2.5 Performance Challenges in Homomorphic Encryption . . . 20
2.5.1 Computational Overhead . . . 20
2.5.2 Ciphertext Expansion . . . 21
2.5.3 Noise Growth . . . 22
2.5.4 Parameter Selection . . . 23
2.6 Optimization Techniques via Caching Approaches . . . 23
2.6.1 Caching Strategies . . . 23
2.7 Applications in Privacy-Preserving Machine Learning . . . 24
2.8 Research Gap . . . 25

3 Methodology **27**
3.1 Research Design . . . 27
3.1.1 Baseline HE Algorithm Benchmarking . . . 27
3.1.2 Radix Tree Caching Integration Benchmarking . . . 28
3.1.3 Key size analysis of encrypted values and their impact on performance . . . 29
3.1.4 Healthcare Dataset Application . . . 31
3.1.5 Biometrics Application . . . 31
3.2 Experimental Setup . . . 33
3.2.1 Microbenchmarking Homomorphic Encryption . . . 33

3.2.2 Radix Tree Caching Benchmarking 36
3.3 Healthcare Dataset Application: Pima Diabetes Dataset 40
3.4 Biometrics Application: Fingerprint Verification using the FVC2002 Dataset 40
3.5 Assumptions and Limitations 41

4 Results **44**

4.1 Baseline HE Algorithm Benchmarking 44
4.1.1 Baseline Raw Encryption Timings (Uncached) 44
4.1.2 Theoretical Analysis of Encryption Performance Differences 45
4.1.3 Baseline Raw Homomorphic Addition Timings (Uncached) 48
4.1.4 Theoretical Analysis of Homomorphic Addition Performance Differences 48
4.2 Cached and Uncached Encryption Timings 51
4.2.1 Theoretical Analysis of Cached Homomorphic Encryption Performance Differences 52
4.2.2 Cached Homomorphic Addition Timings 54
4.2.3 Statistical Analysis of Caching Impact On Homomorphic Encryption 55
4.2.4 Statistical Analysis of Caching Impact On Homomorphic Addition . 58
4.2.5 Theoretical Analysis of Cached Homomorphic Addition Performance Differences 59
4.3 Scaling Analysis 61
4.3.1 Strong and Weak Scaling of Paillier Encryption 61
4.3.2 Strong and Weak Scaling of BFV Encryption 63
4.3.3 Strong and Weak Scaling of CKKS Encryption 64
4.4 Pima Indians Diabetes Dataset 66
4.4.1 Homomorphic Encryption and Encrypted Inference 66
4.5 Biometrics Application Using Fingerprint Verification 70
4.5.1 Experiment Results 70

5 Conclusions **72**

5.1 Summary of Findings 72

5.1.1 Recap of Baseline HE Algorithm Benchmarking Results 72

5.1.2 Impact of Radix Tree Caching 72

5.1.3 Healthcare Dataset Application Outcomes 73

5.1.4 Biometrics Application Results 73

5.2 Interpretation of Results 73

5.2.1 Efficiency of Radix Tree Caching 73

5.2.2 Comparative Analysis of HE Schemes 73

5.2.3 Scalability and Parallelization 74

5.2.4 Real-World Applicability 74

5.3 Implications of the Study 76

5.3.1 Advancements in Privacy-Preserving Machine Learning 76

5.3.2 Impact on Data Security and Privacy 76

5.3.3 Future Directions for Research and Development 77

5.4 Contributions to the Field 77

5.5 Concluding Remarks 79

List of Figures

1 Homomorphic Encryption Application 2

2 Raw Encryption Time for Each Encryption Scheme 44

3 P Raw Homomorphic Addition Time for Each Encryption Scheme 48

4 Paillier Uncached vs Cached Encryption Time 51

5 BFV Uncached vs Cached Encryption Time 51

6 CKKS Uncached vs Cached Encryption Time 51

7 Paillier vs. BFV vs CKKS Cached Encryption Time 52

8 Speedup of Cached Encryption over Raw Encryption 52

9 Raw Homomorphic Addition Time for Each Encryption Scheme 54
10 Speedup of Cached Homomorphic Addition over Raw Addition 55
11 Strong scaling: average encryption time vs. number of cores (loglog plot recommended). 61
12 Weak scaling: average encryption time vs. number of cores. 62
13 Strong scaling of BFV encryption: average encryption time vs. number of cores. 63
14 Weak scaling of BFV encryption: average encryption time vs. number of cores. 64
15 Strong scaling: CKKS encryption time vs. number of cores. 65
16 Weak scaling: CKKS encryption time vs. number of cores. 66
17 Pima Indians Uncached Encryption 67
18 Pima Indians Cached Encryption 67
19 Pima Indians Uncached Inference 68
20 Pima Indians Cached Inference 69

List of Tables

1 Encryption Time Comparison by Key Size and Core Count (seconds) 44
2 Welchs t-test Results for Encryption Time with and without Caching 57
3 Summary statistics and t-test results comparing cached and uncached homomorphic addition times across different schemes and core configurations. 58
4 Performance metrics by method 70
5 Time metrics by method (in seconds) 70

Chapter 1 — Introduction

In contemporary enterprise systems, privacy-preserving machine learning is crucial, especially for entities managing sensitive data such as healthcare providers, financial institutions, and government bodies. Homomorphic encryption (HE) provides a distinctive approach by enabling computations on encrypted data while preserving privacy (Acar et al., 2017). The substantial computing demands of HE are a significant impediment. Every computation on encrypted data involves intricate arithmetic operations, frequently necessitating hundreds of multiplications and additions, resulting in a process significantly slower than operations on unencrypted data. This practice examines how caching methods, particularly radix tree caching, can mitigate these inefficiencies, enhancing the scalability and applicability of HE in realistic scenarios.

To fully grasp the impact of homomorphic encryption, we must first consider the power of asymmetric encryption when passing sensitive data between two parties. The process begins with an algorithm, such as RSA or ECC, generating a pair of mathematically linked keys: a private key, which we denote as (sk) and belongs to the data owner, and a public key (pk), shared by the owner for others to use. For a message m, the encryption operation can be represented as:

$$c = \mathrm{Enc}_{pk}(m)$$

where c is the resulting ciphertext. The decryption operation using the private key recovers the original message:

$$m = \mathrm{Dec}_{sk}(c) = \mathrm{Dec}_{sk}(\mathrm{Enc}_{pk}(m))$$

This asymmetric encryption scheme ensures that sensitive data can be securely transmitted. The recipient can only decrypt and access the original plaintext data by possessing the corresponding private key *sk* (Xin Zhou & Xiaofei Tang, 2011).

What if we do not want to expose the information to another party to be decrypted first before being used? Homomorphic encryption overcomes this challenge by allowing computations to be performed directly on encrypted data (Acar et al., 2017).

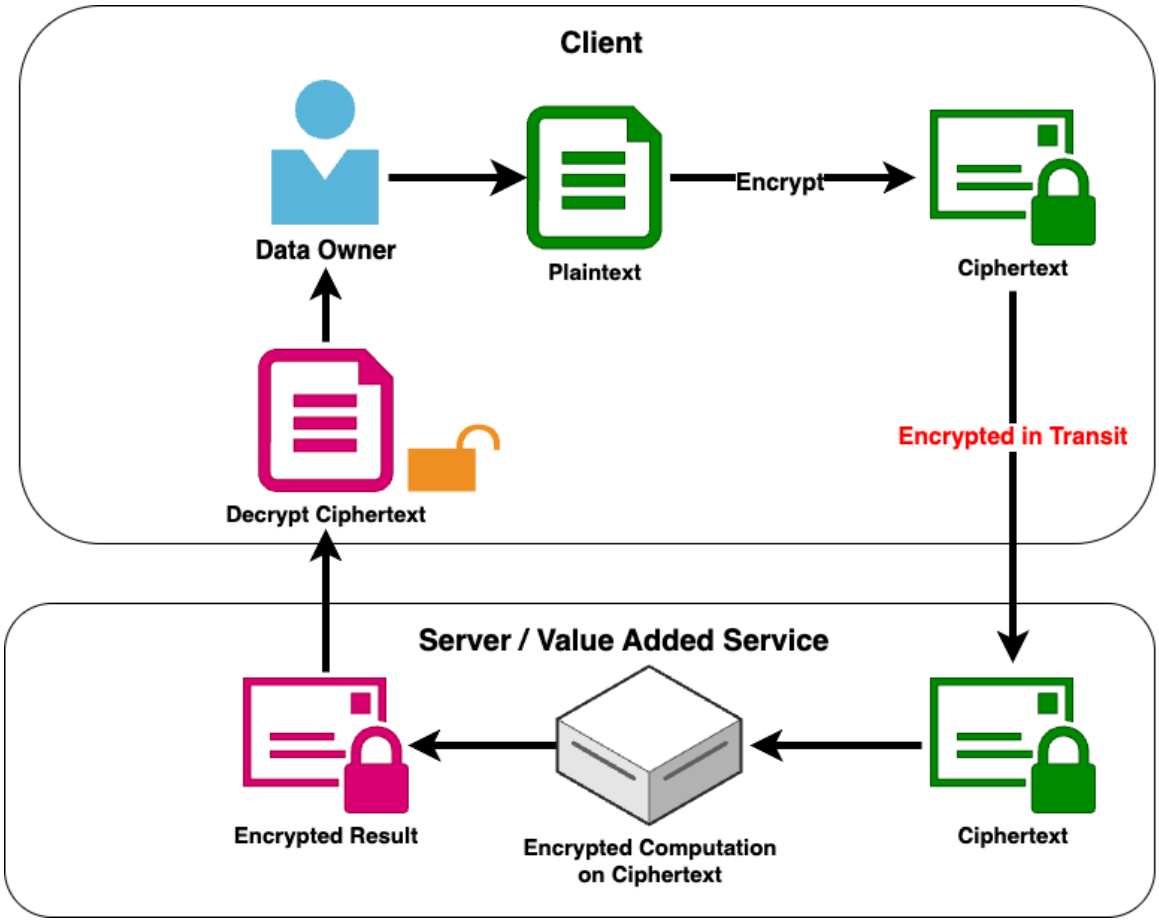

Figure 1: Homomorphic Encryption Application

Figure 1 shows the application of homomorphic encryption in a client-server architec-

ture. The client encrypts the data using a homomorphic encryption scheme and passes the encrypted plaintext (ciphetext) to the server. Data remains encrypted during transit. The server performs operations on the encrypted data and returns the result to the client. Nowhere in this process are the data decrypted. The client then decrypts the result using the homomorphic encryption scheme with the private key, ensuring the data remain secure and private throughout the computation lifecycle. The implication of this is that third-party services can perform computations on encrypted data without ever decrypting it. By extending the principles of asymmetric encryption, homomorphic encryption provides a method to preserve privacy effectively by shielding plaintext data from being exposed during the computation lifecycle.

To understand the application of homomorphic encryption, it is important to recognize the types of algorithms that can be applied. Homomorphic encryption can be implemented using lattice-based and non lattice-based algorithms. Lattice-based schemes include Brakerski/Fan-Vercauteren (BFV) (Fan & Vercauteren, 2012), Cheon-Kim-Kim-Song (CKKS) (Cheon et al., 2016), and Fast Fully Homomorphic Encryption over the Torus (TFHE) (Chillotti et al., 2020). Non-lattice-based approaches are less common in practical approaches. These include algorithms based on integer arithmetic or group theory. Examples include the Paillier cryptosystem (Paillier, 1999a), which supports additive operations on ciphertexts, and the ElGamal cryptosystem (Elgamal, 1985), which supports multiplicative operations. These schemes are limited in that they do not support both operations homomorphically, making them less versatile than fully homomorphic encryption (FHE) schemes.

As depicted in Figure 1, the plain text text data is encrypted in ciphertext using these spe-

cific encryption algorithms by the client. These methods protect the confidentiality of the data by encoding it in a structure that facilitates mathematical operations like addition and multiplication. The ciphertext is transmitted to the server for processing in its raw ciphertext form. To handle encrypted data, specialist libraries such as Microsoft SEAL ("microsoft/SEAL", 2025) and others are needed to be able to perform the necessary operations on the ciphertext. These libraries come with capabilities for managing noise accumulation, performing homomorphic operations directly on the ciphertext, and modifying machine learning models or calculations to function flawlessly within the limitations of encrypted data. Even in untrusted third-party environments such as the cloud, homomorphic encryption allows for safe, private computations by fusing sophisticated encryption techniques on the client side with strong computational frameworks on the server side.

1.1 Research Motivation

Data-driven insights drive decision making in various industries. However, ensuring data privacy is a growing challenge. Organizations increasingly rely on machine learning and artificial intelligence to optimize processes, enhance efficiency, and provide better services. However, these advancements come with significant privacy concerns as more services are provided by third-party entities and the need for data to be shared across organizations is increasing.

The financial and operational impacts of inadequate data privacy protections are significant. The global average cost of a data breach in 2024 reached $4, 888 million, which is a 10 percent increase from 2023 representing the highest total recorded (IBM, n.d.). In particular, one in three breaches involved 'shadow data', underscoring the proliferation of unmoni-

tored data that are increasingly difficult to track and secure, particularly in the growing integration of AI-driven solutions into the enterprise. These statistics emphasize the urgency of implementing robust privacy-preserving measures that can secure private data while enabling practitioners to leverage machine learning effectively. The consequences of data breaches extend beyond financial losses, as they also diminish trust and can compromise the quality of the services provided.

1.2 Problem Statement

The computational inefficiencies in homomorphic encryption inhibit its application in real-world, privacy-preserving machine learning tasks. Homomorphic encryption requires substantial resources for each encrypted computation, often creating bottlenecks that delay processing times and limit scalability. This study addresses these challenges by proposing a caching method to streamline Homomorphic Encryption (HE) processing, enhancing its viability for enterprises requiring secure and efficient data handling.

1.3 Thesis Statement

Integrating radix tree caching into homomorphic encryption processes can enhance performance by decreasing the complexity of ciphertext operations. This caching strategy promotes near-linear scaling, and it also promotes the wider use of privacy-preserving machine learning in AI-enabled enterprise systems by overcoming a significant obstacle to the efficiency of homomorphic encryption.

1.4 Research Objectives

The primary objective of this research is to develop and evaluate a caching method that implements radix tree caching for homomorphic encryption. This study aims to improve the efficiency of HE while maintaining security and precision. The specific objectives include:

1. Designing a radix tree caching framework optimized for HE.
2. Benchmark raw encryption and homomorphic addition across three HE schemes (Paillier, BFV, CKKS) to establish baseline performance.
3. Evaluate the computational improvements from integrating radix tree caching.
4. Apply the optimized HE model to a real-world privacy-preserving machine learning task using the Pima Diabetes dataset
5. Apply cached homomorphic encryption on biometrics dataset.
6. Assess the impact of radix tree caching on system scalability, ciphertext expansion, and computational complexity.

1.5 Research Questions and Hypotheses

1.5.1 Research Questions

1. How do different homomorphic encryption schemes (Paillier, BFV, and CKKS) compare in terms of computational efficiency for basic operations such as encryption and homomorphic addition?

2. What is the effect of integrating radix tree caching on the runtime and efficiency of homomorphic operations?

3. How well does the radix tree caching method scale with increasing data and model sizes and to what extent is it parallelizable?

1.5.2 Research Hypotheses

1. Different HE schemes exhibit varying computational efficiency due to differences in their underlying encryption mechanisms, with lattice-based schemes (BFV, CKKS) outperforming integer-based schemes (Paillier) in terms of scalability and speed due to optimizations in implementation.

2. Integrating radix-tree caching reduces redundant ciphertext operations, thereby improving encryption and homomorphic addition times without compromising or degrading security.

3. Radix tree caching improves parallelism in homomorphic encryption tasks by enabling throughput to scale approximately linearly with the number of processing cores and input size, thereby optimizing the use of multicore architectures for encrypted inference.

1.6 Nature of the Study

This research follows a quantitative experimental approach, in which controlled benchmarking experiments are conducted to analyze the performance of radix tree caching in HE. The study consists of four main phases: first, a baseline assessment of raw encryp-

tion and homomorphic addition performance across three HE schemes (Paillier, BFV, and CKKS) without caching; second, a repeat of these benchmarks with radix tree caching enabled to measure efficiency improvements; and third, the application of radix tree caching to a real-world encrypted machine learning task using the Pima Diabetes dataset. Lastly, we apply a cached homomorphic encryption of biometric fingerprint verification using the FVC2002 (Fingerprint Verification Competition) dataset (Yang et al., 2020) and evaluate the degredation of the matches when dealing with ciphertexts. Performance metrics such as encryption time, homomorphic addition time, noise growth, and inference latency are quantitatively analyzed, providing empirical data to validate the efficiency of radix tree caching in HE.

1.7 Scope of Research

This study focuses on improving the computational efficiency of homomorphic encryption (HE) by introducing and evaluating a radix tree-based caching method. The scope includes both theoretical and practical aspects of HE optimization, with an emphasis on its use in privacy-preserving applications.

The work specifically examines three standard homomorphic encryption schemes: Paillier, BFV, and CKKS. These schemes represent both partially and fully homomorphic encryption paradigms. Each technique is assessed against various system setups and workloads to measure the performance in raw encryption and homomorphic addition.

The study investigates the application of radix tree caching as a technique to minimize unnecessary calculations in HE operations and improve its performance through precomputation. The caching method improves encryption operations across homomorphic encryption

systems by caching and reusing encrypted radix weights. The research evaluates the effects of this caching technique on encryption duration, homomorphic addition velocity, memory efficiency, and practical scalability across different computing cores (4, 8, 16, 32).

The topic also includes applied machine learning situations. Two empirical data sets are used to evaluate the viability of radix-cached enhanced HE in practical applications: the Pima Diabetes dataset for healthcare-related analysis and the FVC2002 fingerprint dataset for biometric authentication. These case studies illustrate the integration of HE into secure ML inference pipelines while maintaining acceptable latency and accuracy.

Healthcare and biometric applications represent the primary domains; however, the proposed caching techniques are intended to be applicable to other sectors requiring sensitive data processing, including banking, legal, and governmental fields.

The development of a customizable Python-based caching architecture is included, allowing interface with current HE libraries like TenSEAL and PHE. While the initial research is conducted in controlled settings (e.g., Google Colab TPU instances), the architecture is designed to facilitate porting to enterprise-level systems.

1.8 Research Limitations

The limitations of this study include the constraints imposed by available computational resources, which can affect the scalability of the caching model for larger datasets. Additionally, the study is limited to radix tree caching, and it leaves other caching approaches unexplored.

1.9 Organization of the Praxis

This work is structured into five chapters. Chapter 1 presents the study, describing the issue, its importance, and the objectives of the investigation. Chapter 2 presents a literature overview that encompasses pertinent research on homomorphic encryption and caching methods. Chapter 3 delineates the process, including the creation of the model, the data sources, and the testing architecture. Chapter 4 delineates the findings, examining performance measurements and corroborating theories. Chapter 5 ultimately addresses the findings, consequences, limits, and prospective directions for further study.

Chapter 2 — Literature Review

This review of the literature examines the evolution of homomorphic encryption (HE), its current implementations, performance challenges, and optimization techniques with a focus on radix caching approaches.

2.1 Evolution of HE in the Context of Cryptography

The concept of homomorphic encryption was first introduced in the late 1970s by Rivest, Adleman, and Dertouzos, who proposed the idea of performing computations on encrypted data (Barnett et al., 2017) (Tourky et al., 2016). However, early attempts at constructing homomorphic encryption schemes were limited, and it was not until 2009 that Craig Gentry made a breakthrough by presenting the first fully homomorphic encryption (FHE) scheme (Nita & Mihailescu, 2023) (Yagisawa, 2018) (Bezuglova & Kucherov, 2023).

2.1.1 Gentry's FHE Scheme

Gentry's Fully Homomorphic Encryption (FHE) scheme was the first to demonstrate the feasibility of performing arbitrary computations on encrypted data. Gentry's scheme construction starts from a Somewhat Homomorphic Encryption (SHE) scheme which supports limited operations on encrypted ciphertexts. The SHE scheme is extended so that a bootstrapping technique is implemented in order to reduce noise growth during homomorphic operations (Gentry, 2009).

2.1.2 Noise Accumulation

In homomorphic encryption, each operation on the ciphertexts adds a certain amount of noise. If the noise becomes too large, it can corrupt the plaintext upon decryption(Kluczniak & Schild, 2022). Let c_1 and c_2 be ciphertexts with the corresponding noise levels $N(c_1)$ and $N(c_2)$. As homomorphic operations are performed, the noise in the resulting ciphertext increases.

For ciphertext addition, where:

$$c_{\text{add}} = c_1 + c_2,$$

The noise typically adds up (or at least does not grow faster than the sum of the individual noises):

$$N(c_{\text{add}}) \leq N(c_1) + N(c_2).$$

For ciphertext multiplication, where:

$$c_{\text{mult}} = c_1 \times c_2,$$

The noise grows more rapidly as a result of the multiplicative nature of the operation. This can be roughly represented as:

$$N(c_{\text{mult}}) \approx N(c_1) \cdot N(c_2) + \varepsilon,$$

where ε represents the extra noise introduced by the operation (such as relinearization or other internal factors).

2.1.3 Bootstrapping Process

Bootstrapping is a method that diminishes the noise in ciphertext through re-encryption. This involves evaluating the decryption function homomorphically to refresh the ciphertext because noise builds up upon each operation. By re-encrypting the ciphertext, bootstrapping resets the noise level which in turn enables more downstream computation until the noise level exceeds the threshold once again (A. Kim et al., 2024) (Halevi & Shoup, 2021). Although bootstrapping is crucial for maintaining the integrity of ciphertexts, it is also a significant efficiency bottleneck in FHE schemes. Efforts to optimize bootstrapping focus on reducing its computational cost and improving its speed.

2.2 Current Homomorphic Encryption Schemes

Modern HE schemes can be categorized on the basis of their support for operations and noise management.

2.2.1 Partially Homomorphic Encryption (PHE)

PHE schemes support either addition or multiplication, but not both. The Paillier cryptosystem (Paillier, 1999b) is additive homomorphic, allowing unlimited additions on ciphertexts but no multiplications. ElGamal (Elgamal, 1985) supports multiplicative homomorphism. Although limited, these schemes remain relevant for specific applications due to their relative efficiency compared to FHE.

2.2.2 Somewhat Homomorphic Encryption (SWHE)

SWHE schemes support both addition and multiplication, but only up to a certain depth before noise overwhelms the ciphertext. The BGV, BFV (Fan & Vercauteren, 2012), and CKKS (Cheon et al., 2016) schemes fall into this category when used without bootstrapping. These schemes have found practical applications in privacy-preserving computations with some limited depth.

2.2.3 Fully Homomorphic Encryption (FHE)

FHE schemes support unlimited operations through noise management techniques like bootstrapping. Modern implementations include:

- **BGV (Brakerski-Gentry-Vaikuntanathan)**: This scheme is based on ideal lattices and supports both addition and multiplication. It is optimized for integer arithmetic.(Brakerski et al., 2011)
- **BFV (Brakerski/Fan-Vercauteren)**: Optimized for integer arithmetic with improved key and ciphertext sizes (Fan & Vercauteren, 2012).
- **CKKS (Cheon-Kim-Kim-Song)**: Designed for approximation arithmetic on real or complex numbers, making it suitable for machine learning applications (Cheon et al., 2016).
- **TFHE (Fast Fully Homomorphic Encryption over the Torus)**: Offers faster bootstrapping, enabling more efficient boolean circuit evaluation (Chillotti et al., 2020).

- **FHEW**: Provides bootstrapping in less than a second (Ducas & Micciancio, 2015), significantly improving the practical viability of FHE.

2.3 Implementation Libraries

The present state of HE implementation has advanced considerably, with numerous libraries now accommodating diverse HE schemes. A few major HE libraries, designed for distinct use cases and performance requirements, offer high-level APIs and differing degrees of support for operations such as addition, multiplication, rotation, and bootstrapping. They are generally developed in high-performance programming languages like C++ and Go, which is designed to offer efficiency and scalability for applications in machine learning.

2.3.1 Microsoft SEAL

Microsoft SEAL is a C++ library that provides a high-level API for homomorphic encryption. It supports BGV, BFV, and CKKS schemes, and offers a range of performance optimizations (“microsoft/SEAL”, 2025).

2.3.2 Helib Library

Created by IBM, it performs operations such as addition, multiplication, rotation, and logical functions, and incorporates single-key bootstrapping to refresh ciphertexts as required. However, HElib has restricted community support, which poses difficulties for novice users. It is developed in C++ with OpenMP support for parallel processing and uses JSON serialization methods for adaptable data management(“homenc/HElib”, 2024).

2.3.3 TFHE - Fast Fully Homomorphic Encryption over the Torus

TFHE is specifically designed for fast and efficient Boolean circuit operations, employing single-key bootstrapping for intricate logic functions.The library is optimized for operations such as addition, multiplication, and rotation, rendering it a practical choice for users who need rapid homomorphic evaluation of Boolean circuits (Chillotti et al., 2020).

2.3.4 FHEW: Fastest Fourier Transform in the West

FHEW is a C++ library that provides a high-level API for homomorphic encryption. It supports BGV, BFV, and CKKS schemes, and offers a range of performance optimizations (Ducas & Micciancio, 2015).

2.3.5 Lattigo

Developed in Go, facilitates multi-key bootstrapping and is distinguished by its capacity to manage secret-shared keys, advantageous for multiparty computations. Lattigo offers high-level API access for operations such as addition, multiplication, rotation, and logical functions, facilitating efficient and secure computations in a programming language recognized for its simplicity and concurrency support. This set of features makes Lattigo especially advantageous in collaborative, decentralized contexts where encrypted data require secure processing by multiple entities (“GitHub - tuneinsight/lattigo: A library for lattice-based multiparty homomorphic encryption in Go”, n.d.).

2.4 Encryption Algorithms for Analysis

This section analyzes representative encryption algorithms across different classes of homomorphic encryption: the Brakerski/Fan-Vercauteren (BFV) scheme for leveled encryption with exact arithmetic, the Cheon-Kim-Kim-Song (CKKS) scheme for approximate real-number computations, and the Paillier cryptosystem for partially homomorphic encryption supporting additive operations.

2.4.1 Brakerski/Fan-Vercauteren (BFV) Scheme

The BFV scheme is based on the Ring Learning With Errors (Ring-LWE) problem and supports exact arithmetic on encrypted integers (Brakerski et al., 2011).

Let $R = \mathbb{Z}[x]/\langle \Phi_m(x) \rangle$ be a polynomial ring, where $\Phi_m(x)$ is the m-th cyclotomic polynomial. Define $R_q = R \bmod q$ and $R_t = R \bmod t$, where $t \ll q$.

Key Generation: Sample secret key $s \leftarrow \chi$ from a discrete Gaussian distribution over R. Let $a \leftarrow R_q$ uniformly and $e \leftarrow \chi$. The public key is:

$$\text{pk} = (a, b = -a \cdot s + e) \in R_q \times R_q.$$

Encryption: Given message $m \in R_t$, lift it to R_q via $m' = \Delta m$ with $\Delta = \lfloor q/t \rfloor$. Sample $u, e_1, e_2 \leftarrow \chi$ and compute:

$$\text{ct} = (c_0, c_1) = (b \cdot u + e_1 + \Delta m,\ a \cdot u + e_2).$$

Decryption: With ciphertext (c_0, c_1) and secret key s:

$$v = c_0 + c_1 \cdot s = \Delta m + \varepsilon, \quad m = \left\lfloor \frac{t}{q} \cdot v \right\rceil \bmod t.$$

Homomorphic addition and multiplication are supported, with noise growth managed via modulus switching or bootstrapping as needed.

2.4.2 Cheon-Kim-Kim-Song (CKKS) Scheme

The CKKS scheme enables approximate arithmetic on encrypted real or complex numbers, making it suitable for machine learning and statistical applications ("CKKS explained", 2020).

Encoding and Scaling: CKKS encodes vectors of real (or complex) numbers into polynomials in R_t using an approximate encoding. A scaling factor Δ is applied to preserve precision.

Encryption: Let m be a plaintext vector, scaled and encoded as a polynomial. The ciphertext is formed similarly to BFV using:

$$\text{ct} = (c_0, c_1) = (\text{Enc}(m) + \text{noise terms}).$$

Computation and Rescaling: Each multiplication increases both noise and scale, so rescaling is required to bring the scale back within acceptable bounds. This corresponds to dividing by a modulus and rounding, preserving the most significant bits.

Decryption and Decoding: Decryption returns an approximate result:

$$m' = \frac{1}{\Delta}(c_0 + c_1 \cdot s) + \varepsilon,$$

which is then decoded and rounded to recover the original approximate message.

2.4.3 Paillier Cryptosystem

The Paillier cryptosystem is a partially homomorphic encryption scheme that supports unlimited homomorphic additions to ciphertexts (Paillier, 1999b).

Key Generation: Choose two large primes p and q, and let $n = pq$. Define $\lambda = \text{lcm}(p-1, q-1)$ and $g \in \mathbb{Z}_{n^2}$ so that $gcd(L(g^\lambda \bmod n^2), n) = 1$, where $L(u) = \frac{u-1}{n}$. The public key is (n, g), and the private key is λ.

Encryption: Given a message $m \in \mathbb{Z}_n$, select random $r \in \mathbb{Z}_n^*$ and compute:

$$\text{ct} = g^m \cdot r^n \bmod n^2.$$

Decryption: Recover the message using:

$$m = \frac{L(\text{ct}^\lambda \bmod n^2)}{L(g^\lambda \bmod n^2)} \bmod n.$$

Homomorphic Property: The Paillier scheme supports homomorphic addition:

$$\text{ct}_1 \cdot \text{ct}_2 \bmod n^2 = \text{Enc}(m_1 + m_2 \bmod n).$$

It does not support native ciphertext multiplication, thus limiting its use in general-purpose homomorphic computation.

2.5 Performance Challenges in Homomorphic Encryption

Despite substantial progress, HE still faces critical performance limitations:

2.5.1 Computational Overhead

Homomorphic operations are orders of magnitude slower than their plaintext counterparts. For example, a single homomorphic multiplication can be thousands to millions of times slower than a plaintext multiplication, depending on the security parameters and the specific scheme used.

Bootstrapping Bottlenecks The bootstrapping process in FHE schemes like FHEW and TFHE is the primary source of computational overhead. For instance, in the HElib library, bootstrapping can take approximately six minutes per batch, making it impractical for real-time applications (Ducas & Micciancio, 2015). This bottleneck arises from the complexity of the decryption circuit and the need for modular reduction operations, which are computationally expensive (Cheon et al., 2016).

Optimization Efforts Researchers have proposed several optimizations to reduce bootstrapping time. For example, a new method to homomorphically compute simple bit operations and refresh ciphertexts has been developed, reducing bootstrapping time to less than a second on a personal computer (Ducas & Micciancio, 2015). Additionally, hardware acceleration using GPUs has been explored, with optimizations reducing bootstrapping time

to under 0.11 seconds (Lee et al., 2015).

Algorithmic Improvements Algorithmic advancements have also contributed to reducing computational overhead. For instance, functional bootstrap techniques in FHEW/TFHE allow for fast table lookups on ciphertexts, improving efficiency . Furthermore, the use of full-domain functional bootstraps (FDFB) and homomorphic digit decomposition algorithms has reduced runtime by up to 39.2% across all parameter settings (Ma et al., 2023).

2.5.2 Ciphertext Expansion

Encrypting data with HE schemes results in significant expansionplaintexts often expand by factors of hundreds or thousands. This expansion increases storage requirements and network bandwidth for applications working with encrypted data.

Sources of Expansion Ciphertext expansion is primarily caused by the encryption process and the need for additional space to accommodate noise during computations. For example, in the GSW scheme, the size of the private key and ciphertext can increase significantly due to noise management techniques (Bai et al., 2024). Similarly, the BGV scheme, which supports ring element plaintexts, can experience ciphertext expansion due to the use of CRT-based ciphertext packing techniques (L. Chen & Zhang, 2017).

Ciphertext Expansion Optimization Strategies Several strategies have been proposed to mitigate ciphertext expansion. The use of gadget matrices in the GSW scheme has been shown to reduce the size of the private key and the ciphertext by a factor of $\lceil \log q \rceil$ (Bai et al., 2024). Additionally, the Residue Number System (RNS) optimization strategy has been

effective in reducing modulus operation complexity and controlling data expansion(Cheng, 2024).

Hardware-Assisted Solutions Hardware acceleration has also been explored to address ciphertext expansion. For example, FPGA-based accelerators for CKKS-based homomorphic encryption have been designed to reduce latency and improve performance (Han et al., 2022). These solutions focus on optimizing keyswitch operations, which are a primary performance bottleneck in FHE computations.

2.5.3 Noise Growth

In lattice-based schemes, operations on ciphertexts introduce noise that accumulates with each operation. Without management techniques like modulus switching or bootstrapping, this noise eventually corrupts the encrypted data, limiting the number of operations that can be performed.

Causes of Noise Growth Noise growth occurs during homomorphic operations, particularly multiplication, where the noise in the ciphertexts can grow quadratically. This growth is a fundamental limitation of many FHE schemes, including the BGV and GSW schemes (Bai et al., 2024) (L. Chen & Zhang, 2017). If left unchecked, noise growth can render ciphertexts unusable after a certain number of operations.

Noise Management Techniques Various noise management techniques have been developed to address this issue. For example, the use of gadget matrices in the GSW scheme has been shown to reduce noise growth and improve the overall efficiency of the scheme (Bai

et al., 2024). Additionally, the development of scale-invariant FHE schemes, which limit noise growth to a linear function of the initial noise level, has been proposed (Brakerski et al., 2011).

2.5.4 Parameter Selection

Selecting appropriate parameters involves complex trade-offs between security, performance, and functionality. The Homomorphic Encryption Standard (Albrecht et al., 2019) provides guidelines, but optimal parameter selection remains challenging for specific applications.

Parameter Selection Optimization by Automated Parameter Selection To address the complexity of parameter selection, automated solutions have been proposed. For example, an expert system combining fuzzy logic and linear programming has been developed to generate optimal parameter combinations based on user priorities for security, efficiency, and performance (Cabrero-Holgueras & Pastrana, 2023). This approach eliminates the need for manual parameter tuning by cryptographers.

2.6 Optimization Techniques via Caching Approaches

2.6.1 Caching Strategies

Caching has emerged as a significant optimization technique for HE operations. By precomputing and storing intermediate results, caching reduces redundant computations and can significantly improve performance.

The Rache (Radix-additive Caching for Homomorphic Encryption) approach (D. Zhao,

2022) represents a breakthrough in HE optimization. Rache works by:

1. Caching homomorphic ciphertexts before encrypting large volumes of plaintexts
2. Expanding plaintexts into a summation of powers of radixes
3. Constructing ciphertexts using only homomorphic addition

This approach reduces the computational complexity from quadratic to nearly linear, making HE more practical for real-world applications. Experimental evaluations have shown that Rache can outperform traditional Paillier encryption by orders of magnitude, particularly for large datasets.

Building on Rache, the Smuche (Scalar-Multiplicative Caching in Homomorphic Encryption) technique (D. Zhao, 2023a) addresses limitations in Rache's approach. While Rache's caching overhead is influenced by the characteristics of the plaintext model, Smuche introduces a constant-time caching technique independent of parameters, making it more suitable for large-scale data processing.

2.7 Applications in Privacy-Preserving Machine Learning

The integration of HE into machine learning workflows has gained significant attention due to privacy regulations such as GDPR ("REGULATION (EU) 2016/ 679 OF THE EUROPEAN PARLIAMENT AND OF THE COUNCIL - of 27 April 2016 - on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing Directive 95/ 46/ EC (General Data Protection Regulation)", n.d.) and the increasing cost of data breaches (IBM, n.d.).

Privacy-preserving machine learning using HE enables:

- **Secure inference**: Models can make predictions on encrypted data without accessing the underlying plaintext, protecting user privacy.
- **Secure training**: Models can potentially be trained on encrypted data contributed by multiple parties without revealing sensitive information.
- **Federated learning with enhanced privacy**: Combining federated learning with HE can provide stronger privacy guarantees than either approach alone.

Recent work has explored optimizing machine learning models for HE compatibility, including developing HE-friendly activation functions, quantization techniques, and model architectures that minimize multiplicative depth. The FHEDA approach (Singh et al., 2024) offers efficient circuit synthesis with reduced bootstrapping for torus FHE, further advancing the practicality of privacy-preserving machine learning.

2.8 Research Gap

Although significant progress has been made in developing more efficient HE schemes and optimization techniques, several research gaps remain.

1. **Comparative analysis**: Limited research exists comparing the performance of different HE schemes when combined with radix caching across various applications.
2. **Scalability assessment**: The scalability of radix caching approaches for large-scale, real-world datasets needs further investigation.
3. **Integration frameworks**: Standardized frameworks are lacking for integrating radix caching with existing HE libraries and machine learning workflows.

4. **Domain-specific optimization**: The adaptation of radix caching techniques for specific domains like healthcare remains underexplored.

This study addresses these gaps by implementing and evaluating a radix tree caching approach across multiple HE schemes, assessing its impact on performance metrics, and applying it to a real-world healthcare dataset. The findings will contribute to the growing body of knowledge on making privacy-preserving machine learning more practical for enterprise applications.

Chapter 3 — Methodology

This chapter details the approach to evaluate radix tree caching across multiple homomorphic encryption schemes. The methodology discusses the implementation, performance evaluation, and practical application in both machine learning and real-world application in biometrics. All source code and benchmarking implementations used in this praxis are publicly available at: https://github.com/gimenopea/CachedHE

3.1 Research Design

This study adopts a quantitative experimental approach, leveraging controlled benchmarking experiments to assess the efficiency of radix tree caching in HE. The research is segmented into three main benchmarking phases:

3.1.1 Baseline HE Algorithm Benchmarking

HE Schemes Selection

- Three HE schemes were selected for evaluation: Paillier, BFV, and CKKS are implemented using MPI standard libraries (Message Passing Interface Forum, 2023)and tenSEAL Python libraries (“OpenMined/TenSEAL”, 2025).

- Experiments are conducted on a Google Colab v2-8 TPU High-Ram system with up to 96 cores (“Google Colab”, n.d.). While Google Cloud uses Intel Xeon CPUs, the specific CPU model used by Google Colab isn’t revealed. The specific hardware configuration was chosen to provide a balance of computational resources and allows for adequate resources to execute the HE benchmarks.

- Performance metrics, including raw encryption time and homomorphic addition time, are collected across 4, 8, 16, and 32-core configurations to establish baseline performance. The selection of 4, 8, 16, and 32 core configurations was based on typical enterprise server setups, allowing for the evaluation of how the HE schemes scale as compute jobs are parallelized. These configurations represent a range from small to medium-sized server environments source. This configuration will provide insights into the performance of HE operations in practical deployment scenarios. In addition, these choices will provide us an understanding of the overhead associated with parallel processing as the number of cores increases.

3.1.2 Radix Tree Caching Integration Benchmarking

- Integration using Paillier, BFV, and CKKS is conducted using the proposed radix tree caching approach. The radix tree caching was implemented using a Python dictionary-based structure, where each level of the tree corresponds to a dictionary. The keys represent the digits of the radix representation, and the values store the corresponding encrypted radix weights. This caching mechanism was integrated into the encryption functions of the HE libraries by anticipating the encryption of radix weights. When a radix weight was computed for the first time, it was stored in the cache and subsequent encryptions of the same weight is retrieved which avoids redundant computations.

- The impact of caching on computational efficiency is measured by comparing the results with the baseline. Due to high memory capacity of the Google Colab envi-

ronment, The radix tree cache was configured with no maximum size for the purposes of the benchmark and no eviction policy was employed.

3.1.3 Key size analysis of encrypted values and their impact on performance

- Encryption and homomorphic addition are performed with different key sizes to analyze the impact on performance for each encryption scheme. For Paillier, the key sizes of 512 and 1024 bits were chosen to represent standard security levels. Larger key sizes in Paillier provide higher security but increase the computational cost of encryption and decryption. In BFV and CKKS, key size is represented by parameter grids. These parameters directly influence the security level and the computational complexity of the HE operations. Larger parameter grids offer enhanced security but come with increased computational overhead.

- In the case of Paillier, the key size is varied from 512 and 1024 bits.

- In the case of BFV, the key size is varied from the following parameter grids: key-size, poly-modulus-degree, and coefficient-modulus of (512, 4096,[30,30,30]) and (1024, 8192,[40,40,40]).

  - For BFV, the parameter "key-size" refers to the bit length of the secret key, influencing the security level.

  - The "poly-modulus-degree" determines the size of the polynomials used in the encryption scheme, affecting both performance and the complexity of computations that can be performed.

  - The "coefficient-modulus" is a set of prime numbers that define the modulus for the polynomial coefficients; its size and number influence the noise growth and computational precision.
  - The grid (512, 4096, [30, 30, 30]) indicates a 512-bit key, a polynomial modulus degree of 4096, and a coefficient modulus consisting of three 30-bit primes.
  - The grid (1024, 8192, [40, 40, 40]) uses a 1024-bit key, a polynomial modulus degree of 8192, and a coefficient modulus with three 40-bit primes, representing a higher security and computational load.

- In the case of CKKS, the key size is varied from the following parameter grids: key-size, poly-modulus-degree, and scale of (4096, 4096, [40,40]) and (8192, [60,40,40,60].
  - For CKKS, "key-size" and "poly-modulus-degree" have similar meanings to BFV.
  - The "scale" parameter in CKKS is used for approximate arithmetic and affects the precision of the computations. It influences how real numbers are encoded and decoded.
  - The grid (4096, 4096, [40, 40]) has a polynomial modulus degree of 4096 and uses scaling factors of 40 bits each.
  - The grid (8192, [60, 40, 40, 60]) has a polynomial modulus degree of 8192 and employs scaling factors of 60, 40, 40, and 60 bits, which affects the precision and range of representable real numbers.

- Performance metrics are measured for both uncached and cached operations.

3.1.4 Healthcare Dataset Application

- The Pima Diabetes dataset (Bartley, 2016) is used to asses the practical applicability of radix tree caching in practice.

- The dataset is preprocessed and encrypted using the proposed radix tree caching approach.

- Features are encrypted and inference is performed homomorphically using 4,8,16,32 cores. A logistic regression model was chosen for inference due to suitability with binary classification tasks and sample weights were encoded to simplify the pipeline. The model was trained on the plaintext dataset and then adapted for homomorphic encryption inference.

3.1.5 Biometrics Application

- The biometrics dataset using the FVC2002: Second Fingerprint Verification Competition (Maio et al., 2002) is used to asses the practical applicability of radix tree caching in practice.

- **Fingerprint Image Preprocessing and Feature Extraction**

  - The fingerprint images undergo preprocessing to standardize their size. If an image's dimensions exceed 512 pixels in either height or width, it is resized to ensure that the larger dimension is reduced to 512 pixels while maintaining the aspect ratio.

  - To reduce noise, a Gaussian blur with a 5x5 kernel is applied to the resized image.
  - For each block, the mean pixel value and standard deviation are calculated. These values serve as features representing ridge density and texture, respectively.
  - Gradients in the x and y directions are calculated by comparing the mean pixel values of the current block with those of the neighboring blocks.
  - These calculated features (mean, standard deviation, and gradients) are then converted into a binary vector.
  - The feature vector is truncated or padded with zeros to ensure a uniform length of 128.

- The dataset is preprocessed and encrypted using the proposed radix tree caching approach and the paillier encryption scheme and performance metrics and accuracy are measured (Yang et al., 2020)

- Features are encrypted and inference is performed homomorphically using 4, 8, 16, and 32 cores. The Hamming weight between the encrypted template and query feature vectors is computed homomorphically using the additive property of the Paillier scheme. Each bit of the feature vectors is encrypted, and the Hamming weight is calculated by performing homomorphic additions and subtractions on the encrypted bits. This allows for the comparison of the fingerprint features without decrypting the individual bits, preserving privacy. Imposter pairs are defined as fingerprint comparisons between different individuals, while genuine pairs are comparisons between

different fingerprints of the same individual. These pairs are selected randomly from the FVC2002 dataset, ensuring a balanced representation of both types of comparisons. The evaluation process involves computing the similarity score for each pair and using these scores to calculate performance metrics such as the Equal Error Rate (EER).

3.2 Experimental Setup

3.2.1 Microbenchmarking Homomorphic Encryption

To quantify the computational cost of HE, we perform a series of microbenchmarking experiments focused on raw homomorphic encryption and addition without caching. These experiments establish a baseline performance for homomorphic encryption schemes by measuring execution times across different cryptographic paradigms, specifically Paillier (a partially homomorphic encryption scheme) and lattice-based schemes like BFV and CKKS. The benchmarking is conducted using an OpenMPI-based (Message Passing Interface Forum, 2023) distributed computing framework, allowing us to execute encryption and homomorphic operations in parallel across multiple processing cores. The objective is to measure the time required for each encryption method to process a predefined dataset while evaluating the impact of increased computational resources.

Encryption Time Analysis The first microbenchmark measures raw encryption time across all three HE schemes. Each MPI process is assigned a subset of 32,768 integers (D. Zhao, 2022), which are encrypted independently without optimizations such as caching. This setup provides insight into encryption costs associated with each algorithm, includ-

ing differences in computational overhead due to modular exponentiation (Paillier) versus polynomial encoding in BFV and CKKS schemes. Encryption Time Analysis The first microbenchmark measures raw encryption time across all three HE schemes. Each MPI process is assigned a subset of 32,768 integers (D. Zhao, 2022), which are encrypted independently without optimizations such as caching. This setup provides insight into encryption costs associated with each algorithm, including differences in computational overhead due to modular exponentiation (Paillier) versus polynomial encoding in BFV and CKKS schemes. The 32,768 integers used for the microbenchmarks were generated randomly using a uniform distribution within the range of 0 to 1000. This range was chosen to represent a realistic distribution of numerical data while avoiding excessively large values that could introduce unnecessary computational overhead. The random generation ensures that the benchmarks are representative of typical data processing scenarios.

Choosing Paillier Encryption Scheme for Baseline Timings

- Paillier encryption is based on modular arithmetic which allows for additive homomorphism. This means that encrypted values can be added together to produce an encrypted sum, which can be decrypted to reveal the sum of the original values. In addition the scheme simplifies multiplication by transforming it into addition, which is computationally less intensive compared to other encryption methods like RSA (A. C. H. Chen, 2025)

- Paillier encryption has been studied and used in privacy-preserving applications, such as federated learning and e-voting systems (Galymzhankyzy et al., 2024) (Abhinand et al., 2024).

- In blockchain systems, Paillier encryption supports zero-knowledge proofs, enabling verification of data correctness without exposing the data itself, thus enhancing privacy protection capabilities (Zhuo, 2024)

Choosing BFV and CKKS Encryption Scheme for Baseline Timings

- Lattice-based cryptography is considered a strong candidate for post-quantum cryptography because it is resistant to attacks from quantum computers. Unlike traditional cryptographic methods like RSA, which are vulnerable to quantum attacks, lattice-based schemes do not have known efficient quantum algorithms that can break them

- BFV is designed for exact arithmetic operations, making it suitable for applications that require precise calculations on integer data, such as financial computations or voting systems where accuracy is important(J. Kim et al., 2024)

- CKKS is specifically designed for approximate arithmetic which makes it ideal for applications that involve real or complex numbers. It is useful for tasks involving machine learning and data analytics(W. Zhao et al., 2024)

Homomorphic Addition Time Analysis The second microbenchmark evaluates homomorphic addition efficiency by performing a round-robin addition of encrypted integers. Each process computes a local sum over its encrypted subset, and the results are aggregated at the root node. The primary goal is to measure the time required to execute repeated homomorphic additions across different HE schemes and establish baseline performance before introducing caching optimizations. Both benchmarks are repeated across 4, 8, 16,

and 32-core configurations, ensuring scalability analysis and variance measurement across different processing environments. The round-robin addition process involved distributing the 32,768 integers evenly among the available processing cores. Each core encrypted its subset of integers and then performed homomorphic additions locally. Intermediate sums were then aggregated in a round-robin fashion, where each core added its local sum to the sum from the next core. This process continued until the final sum was computed at the root node. This approach ensures that the computational load is distributed evenly across the cores, maximizing parallelism.

Encryption and Homomorphic Addition Baselining These results serve as the foundational comparison point for later experiments that incorporate radix tree caching. Following the baseline benchmarking, the same encryption and homomorphic addition experiments are repeated with radix tree caching enabled to measure efficiency improvements. This phase focuses on evaluating how caching impacts computational overhead by reducing redundant operations.

3.2.2 Radix Tree Caching Benchmarking

Following the baseline benchmarking, the same encryption and homomorphic addition experiments are repeated with radix tree caching enabled to measure efficiency improvements. This phase focuses on evaluating how caching impacts computational overhead by reducing redundant operations.

Radix Tree Caching Implementation To optimize HE operations, radix tree caching is introduced, leveraging radix decomposition to efficiently encode and encrypt numerical

values. The encryption process is structured as follows:

Algorithm 1 Radix Tree Caching for Homomorphic Encryption

Require: Plaintext value x, Radix base b

Ensure: Encrypted value $\text{Enc}(x)$

Step 1: **Decompose the plaintext into digits**

Represent x in base b: $x = d_0 + d_1 \cdot b + d_2 \cdot b^2 + \cdots + d_{n-1} \cdot b^{n-1}$

Extract all digits $d_0, d_1, \ldots, d_{n-1}$

Step 2: **Compute and cache encrypted radix weights**

for $i = 0$ to $n-1$ **do**

    if b^i is not in Cache **then**

        Encrypt the radix weight: $C_i \leftarrow \text{Enc}(b^i)$

        Store in cache: $\text{Cache}[b^i] \leftarrow C_i$

    else

        Retrieve from cache: $C_i \leftarrow \text{Cache}[b^i]$

    end if

end for

Step 3: **Compute encrypted terms for each digit**

for $i = 0$ to $n-1$ **do**

    Perform homomorphic multiplication: $\text{Term}_i \leftarrow d_i \cdot C_i$

end for

Step 4: **Combine terms using homomorphic addition**

$\text{Enc}(x) \leftarrow \text{Term}_0 + \text{Term}_1 + \cdots + \text{Term}_{n-1}$

Step 5: **Return the encrypted result**

return $\text{Enc}(x)$

1. Radix decomposition: The plaintext value is expressed in a radix (base) representation as shown in Algorithm 1.

2. Extracting Digits: Each digit D_i is extracted from the plaintext representation and will be used in the encryption process. This ensures that frequent operations such as modular exponentiation (in Paillier) or polynomial encoding (in BFV and CKKS) do not need to be recalculated repeatedly.

3. Caching Radix Weights: Instead of recomputing exponentiations during encryption, the encryption of radix weights is precomputed and stored.

4. Computation of Encrypted Values: For each extracted digit D_i, the encrypted radix weight $E(R_i)$ is multiplied homomorphically. This step allows the encrypted numerical representation to be built efficiently.

5. Recomposition of Encrypted Sum: The final encrypted value is reconstructed as the sum of term homomorphic additions.

6. The radix (b) used in the radix tree caching implementation is a parameter that can be varied. In our experiments, we primarily used a radix of 10, as it aligns with the decimal representation of numbers and simplifies the decomposition process. However, the algorithm is designed to support any radix, allowing for flexibility in optimizing performance for different use cases. The choice of radix can impact the depth of the tree and the number of caching operations, thus affecting the overall efficiency.

3.3 Healthcare Dataset Application: Pima Diabetes Dataset

In a practical benchmarking phase, radix tree caching is applied to a real-world privacy-preserving machine learning task using the Pima Diabetes dataset. The data set consists of 768 samples with eight features. The task is to predict the onset of diabetes according to the characteristics. This experiment evaluates how caching optimizations impact computational efficiency in a practical encrypted machine learning workflow.

Implementation Process

1. Input Encryption: The feature vectors of the dataset are encrypted using Paillier, BFV, and CKKS, with and without radix tree caching.

2. We apply homomorphic operations to the encrypted features, addition and multiplication, to perform secure inference. We chose a logistic regression model for this task. Its simplicity makes it well suited for homomorphic inference when resources are limited. We trained the model on plaintext data and adapted it for homomorphic execution.

3. The computational time and performance differences between cached and non-cached executions are analyzed.

3.4 Biometrics Application: Fingerprint Verification using the FVC2002 Dataset

The FVC2002 dataset is a collection of fingerprint images from 110 individuals, with each individual having 10 fingerprints. The data set is pre-processed and encrypted using the

proposed radix tree caching approach and the paillier encryption scheme and performance metrics and accuracy are measured (Yang et al., 2020).

Implementation Process

1. The client extracts a binary feature vector of minutiae pairs from a fingerprint image and encrypts each bit with the Paillier scheme

2. The client sends the encrypted vector and user ID to a function which stores the encrypted vector in memory. In real world application this would be a server for storage.

3. The client extracts a query vector from a new fingerprint scan and sends it with the user ID to the authentication server

4. The authentication server retrieves the stored ciphertext, uses Pailliers homomorphic property to compute the encrypted Hamming weight between template and query, and returns it to the client

5. The client decrypts the Hamming weight, calculates a similarity score, and compares it to a threshold to decide match or non-match

3.5 Assumptions and Limitations

Assumptions:

1. It is assumed that the performance improvements observed in the benchmarking experiments generalize to other large-scale HE workloads

2. The experiments assume an ideal distributed computing environment where interprocess communication overhead is minimal.

3. All homomorphic encryption schemes used in the benchmarks are implemented under optimal parameter settings recommended by their respective libraries (TenSEAL for BFV and CKKS, and phe for Paillier).

4. The Pima Diabetes dataset is assumed to be representative of real-world encrypted machine learning inference workloads in healthcare applications.

Limitations:

1. The study focuses on three specific HE schemes (Paillier, BFV, and CKKS); results may not be directly applicable to other encryption schemes such as FHEW or TFHE.

2. The caching approach is optimized for addition operations; its impact on other homomorphic operations, such as multiplication or bootstrapping, is not extensively studied.

3. The experiments are conducted in a Google Colab v2-8 TPU High-Ram environment, which may not accurately reflect performance on other computing architectures, such as dedicated high-performance computing clusters.

4. The impact of security trade-offs, such as potential attacks on caching mechanisms, is not explored in this study.

By conducting paired experiments with and without radix caching, we empirically measure efficiency gains. The results will be analyzed in the next chapter to determine the extent

to which radix caching reduces computation time and improves scalability across different core configurations.

Chapter 4 — Results

This section presents the results of the experiments conducted to evaluate the performance of radix tree caching in HE. The results are presented in the form of tables and graphs to illustrate the performance improvements achieved through the use of radix tree caching.

4.1 Baseline HE Algorithm Benchmarking

4.1.1 Baseline Raw Encryption Timings (Uncached)

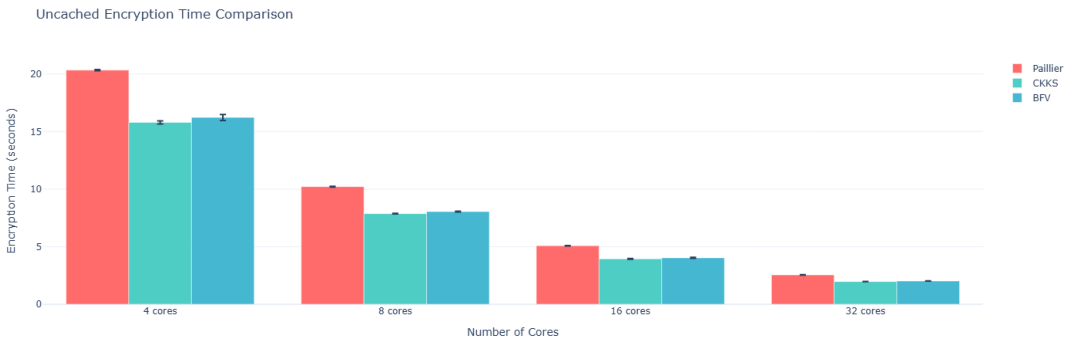

Figure 2: Raw Encryption Time for Each Encryption Scheme

Table 1: Encryption Time Comparison by Key Size and Core Count (seconds)

Algorithm	Key Size (bits)	4 Cores	8 Cores	16 Cores	32 Cores
Paillier	512	20.339	10.217	5.087	2.546
Paillier	1024	127.602	63.765	31.929	15.944
CKKS	4096	15.800	7.870	3.944	1.965
CKKS	8192	52.316	26.088	13.099	6.642
BFV	512	16.226	8.050	4.032	2.018
BFV	1024	32.501	16.188	8.094	4.089

Figure 2 and Table 1 show the time taken to encrypt 32768 integers using Paillier, BFV and CKKS across 4, 8, 16 and 32 core configurations with different key sizes. It is observed that Paillier is significantly slower than BFV and CKKS while BFV performs better than

CKKS. The performance improves for each core configuration, with consistent scaling patterns across algorithms. Notably, key size doubling impacts algorithms differently: Paillier shows the highest sensitivity with a 6.26x performance penalty, CKKS demonstrates moderate impact at 3.31, while BFV exhibits the best key size scaling at only 2.01 performance degradation.

4.1.2 Theoretical Analysis of Encryption Performance Differences

Let n be the Paillier modulus of bit length $\ell = \lceil \log_2 n \rceil$, and let $M(t)$ denote the time complexity of multiplying two t bit integers. In the Ring LWE schemes (BFV / CKKS), let N be the ring dimension and q the coefficient modulus. We encrypt a batch of T integers.

Paillier Encryption Complexity A Paillier ciphertext for $m \in \mathbb{Z}_n$ is

$$c \;=\; g^m r^n \;\bmod\; n^2.$$

Using square and multipliply requires $O(\ell)$ modular multiplications mod n^2, each costing $M(2\ell) = O(M(\ell))$. Hence

$$T_{\mathsf{Pail}}(\ell) \;=\; O\big(\ell \cdot M(\ell)\big),$$

and encrypting T values costs

$$T_{\mathsf{Pail\text{–}total}}(T,\ell) \;=\; T \;\times\; O\big(\ell M(\ell)\big).$$

BFV/CKKS Encryption Complexity Encryption produces a RingLWE sample

$$(b,a) \;=\; \big(-[a\cdot s+e+\Delta\cdot m],\, a\big),$$

where $a,s,e \in R_q = \mathbb{Z}_q[X]/(X^N+1)$. The dominant cost is the polynomial multiplication $a\cdot s$, which via the Number Theoretic Transform (NTT) runs in

$$T_{\mathsf{RLWE}}(N) \;=\; O\big(N\log N\big).$$

SIMD and Packing BFV and CKKS exploit *Single Instruction, Multiple Data (SIMD)* by *packing* multiple plaintext values into one polynomial (or complex embedded) ciphertext:

- **Packing:** Up to $k \le N/2$ integers $\{m_j\}$ are encoded into a single polynomial $m(X) = \sum_{j=0}^{k-1} m_j X^{2j}$, so that operations on ciphertexts act pointwise on all slots simultaneously.
- **SIMD effect:** One Enc, Add, or Mul call processes all k slots. Thus the amortized cost per value is $O\big((N\log N)/k\big)$ instead of $O(N\log N)$.
- **Contrast with Paillier:** Paillier has no SIMD or packing each integer requires its own exponentiation.

Amortized Batch Cost Packing k values per ciphertext reduces the total BFV/CKKS cost for T integers to

$$T_{\mathsf{BFV\text{–}total}}(T,N) \;=\; \left\lceil \frac{T}{k} \right\rceil \times O(N\log N) \;\approx\; O\big(T\,\log N\big) \quad (\text{for } k=\Theta(N)).$$

Parallelization Effects Distributing work across c cores yields wall clock times

$$\frac{T_{\mathsf{Pail\text{–}total}}}{c} \quad \text{vs.} \quad \frac{T_{\mathsf{BFV\text{–}total}}}{c_{\text{across}} + c_{\text{within}}},$$

where BFV/CKKS can parallelize both *across* ciphertexts ($\lceil T/k \rceil$ tasks) and *within* each NTT transform.

Asymptotically, BFV/CKKS replace Pailliers $O(\ell M(\ell))$ per value exponentiations with $O(N \log N)$ polynomial multiplies, then amortize over $\Theta(N)$ slots using SIMD packing. Even with perfect 32-core parallelization, NTT-based schemes (especially CKKS, which omits extra integer rounding steps) exhibit significantly lower latency than Paillier.

4.1.3 Baseline Raw Homomorphic Addition Timings (Uncached)

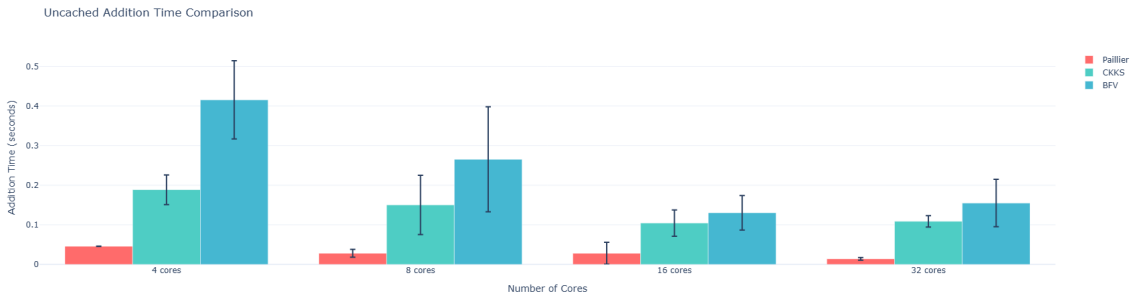

Figure 3: P Raw Homomorphic Addition Time for Each Encryption Scheme

Figure 3 shows the time taken to perform a homomorphic addition operation across all encryption schemes in different core configurations. It is observed that homomorphic addition is orders of magnitude faster than raw encryption across all schemes and configurations. Paillier shows best performance in this baseline task due to the nature of Paillier encryption which is designed for homomorphic addition.

4.1.4 Theoretical Analysis of Homomorphic Addition Performance Differences

Let n be the Paillier modulus of bitlength $\ell = \lceil \log_2 n \rceil$, and let $M(t)$ denote the time complexity of multiplying two t-bit integers. In the Ring LWE schemes (BFV/CKKS), let N be the ring dimension, q the coefficient modulus, and $k \leq N/2$ the number of packed slots. We perform a homomorphic sum over $T = 32\,768$ ciphertexts.

Paillier Homomorphic Addition Given two Paillier ciphertexts $c_1, c_2 \in \mathbb{Z}_{n^2}$, their homomorphic sum is

$$c_{\text{sum}} = c_1 \cdot c_2 \mod n^2,$$

i.e. one modular multiplication mod n^2. Since each operand is 2ℓ bits, this costs

$$T_{\mathsf{Pail_add}} \;=\; O\big(M(2\ell)\big) \;=\; O\big(M(\ell)\big).$$

To sum T ciphertexts (in a simple linear roundrobin chain of $T-1$ multiplies) costs

$$T_{\mathsf{Pail_total}} \;=\; (T-1)\times O\big(M(\ell)\big) \;=\; O\big(T\,M(\ell)\big).$$

BFV/CKKS Homomorphic Addition For two RLWE ciphertexts $(b_1,a_1),(b_2,a_2)\in R_q^2$, homomorphic addition is coefficient wise:

$$(b_{\mathsf{sum}},a_{\mathsf{sum}}) \;=\; (b_1+a_2,\; a_1+a_2) \quad \text{computed mod } (X^N+1,\; q).$$

This is simply N scalar additions:

$$T_{\mathsf{RLWE_add}} \;=\; O(N).$$

Packing k plaintext slots per ciphertext means we only have $\lceil T/k\rceil$ ciphertexts to add. Thus the total cost is

$$T_{\mathsf{RLWE_total}} \;=\; \big(\lceil T/k\rceil-1\big)\times O(N) \;\approx\; O\Big(\frac{T}{k}N\Big).$$

For typical parameters $k=\Theta(N)$, this simplifies to

$$T_{\mathsf{RLWE_total}} \;=\; O(T).$$

SIMD and Packing Effect

- **Packing:** Encode up to $k \leq N/2$ independent integers into one ciphertext via slot wise embedding.
- **SIMD:** A single homomorphic addition applies to all k slots in parallel, so work per slot is $O(N/k)$.
- **Amortization:** With $k = \Theta(N)$, the cost per integer of an addition drops from $O(N)$ to $O(1)$.

Paillier, lacking SIMD, incurs $O(M(\ell))$ per integer.

Parallelization Effects Distributing the $T-1$ addition tasks across c cores yields wall clock times

$$\frac{T_{\mathsf{Pail_total}}}{c} \quad \text{vs.} \quad \frac{T_{\mathsf{RLWE_total}}}{c_{\text{across}} + c_{\text{within}}},$$

where BFV/CKKS can parallelize (i) across $\lceil T/k \rceil$ ciphertext additions and (ii) within each N-length coefficientwise vector addition.

The homomorphic addition in Paillier costs $O(T\,M(\ell))$, while BFV/CKKS reduce to $O(T)$ (for $k = \Theta(N)$) by adding $O(N)$ coefficient-wise to the amortized slots in N. Even with 32 core parallelism, the NTT based schemes will significantly outpace Paillier for summing large batches of integers.

4.2 Cached and Uncached Encryption Timings

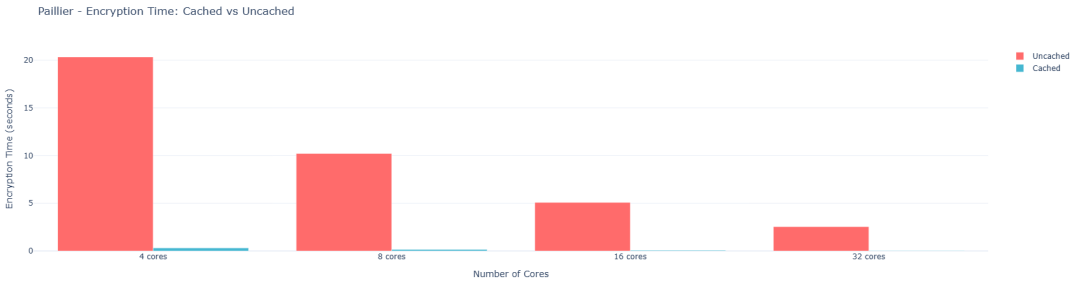

Figure 4: Paillier Uncached vs Cached Encryption Time

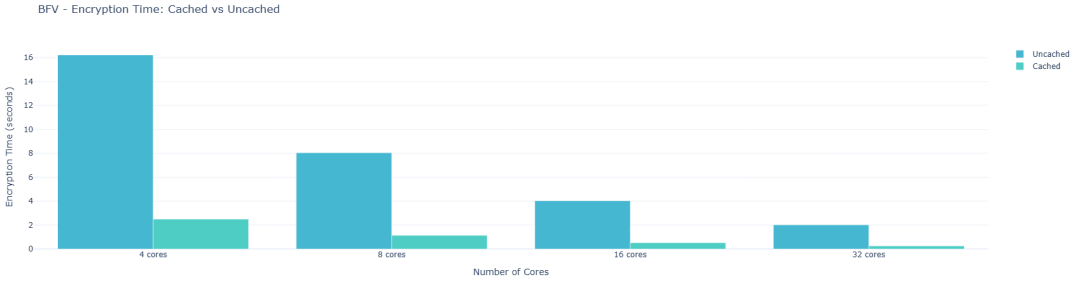

Figure 5: BFV Uncached vs Cached Encryption Time

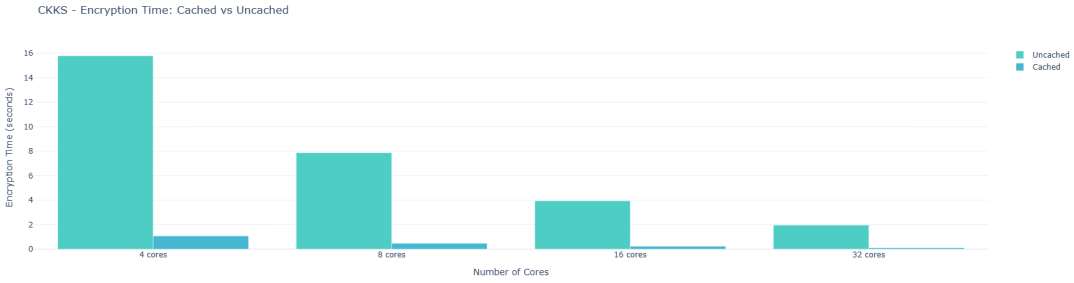

Figure 6: CKKS Uncached vs Cached Encryption Time

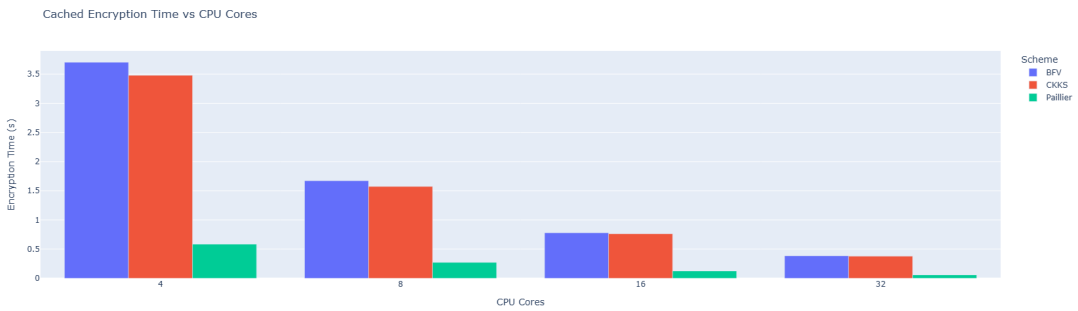

Figure 7: Paillier vs. BFV vs CKKS Cached Encryption Time

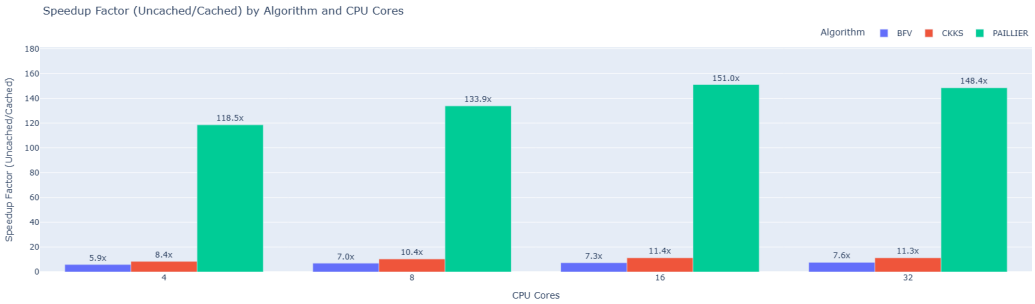

Figure 8: Speedup of Cached Encryption over Raw Encryption

Figure 4, 5, 6 and 7 show the time taken to encrypt 32768 integers using the Paillier, BFV, and CKKS schemes. It is observed that cached encryption is orders of magnitude faster than raw encryption across all schemes and CPU core configurations. Figure 8 shows the speedup of cached encryption over raw encryption for each scheme, with Paillier showing at least 118 speedup over raw encryption.

4.2.1 Theoretical Analysis of Cached Homomorphic Encryption Performance Differences

Let n be the Paillier modulus of bitlength $\ell = \lceil \log_2 n \rceil$, and let $M(t)$ denote the cost of multiplying two t bit integers. We encrypt a batch of T values using radix caching with base r.

Paillier with Radix Caching Each message $m \in [0, n)$ is written in radix r as

$$m = \sum_{i=0}^{d-1} m_i r^i, \quad d = \left\lceil \frac{\ell}{\log_2 r} \right\rceil.$$

A precomputed table of size $d \times r$ stores $g^{m_i r^i}$ for all digit values. Encryption then uses d modular multiplications modulo n^2, each costing $O(M(\ell))$. Thus the per message time is

$$T_{\mathsf{Pail_cached}}(\ell) = O\Big(d\, M(\ell)\Big) = O\Big(\tfrac{\ell}{\log_2 r}\, M(\ell)\Big).$$

For T messages,

$$T_{\mathsf{Pail_total_cached}}(T, \ell) = T \times O\Big(\tfrac{\ell}{\log_2 r}\, M(\ell)\Big).$$

Comparison and Simplicity Paillier's complexity remains a simple function of keysize bits and the chosen radix:

$$O\Big(\tfrac{\ell}{\log_2 r}\, M(\ell)\Big).$$

It avoids more intricate transforms or datapacking strategies. In contrast, RLWEbased schemes rely on polynomial arithmetic whose cost depends on parameters like ring dimension N and transform algorithms.

Radix caching reduces Pailliers exponentiation work by a factor of $\log_2 r$, yielding a straightforward time bound in terms of integer multiplication costs. This simplicity in both analysis and implementation can make cached Paillier faster when parameters are chosen appropriately.

4.2.2 Cached Homomorphic Addition Timings

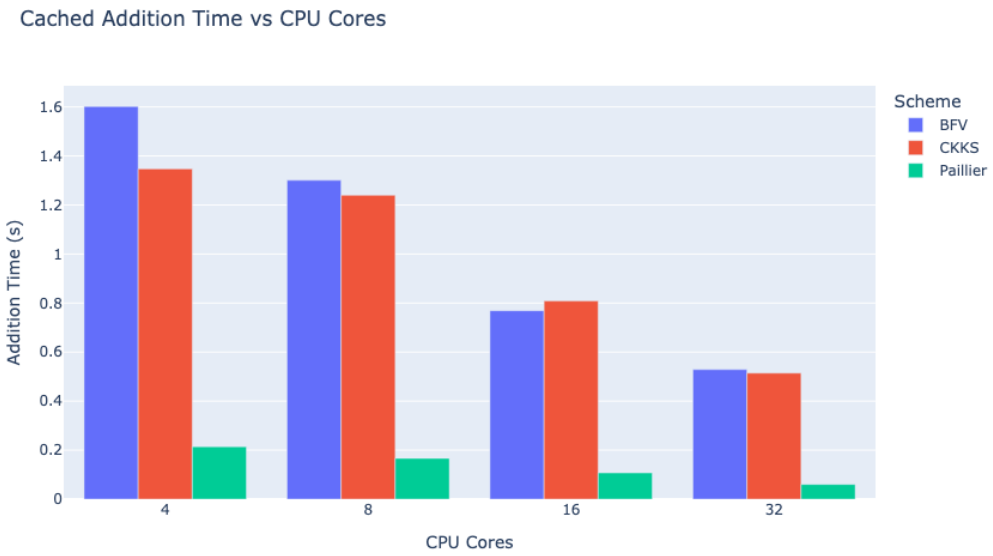

Figure 9: Raw Homomorphic Addition Time for Each Encryption Scheme

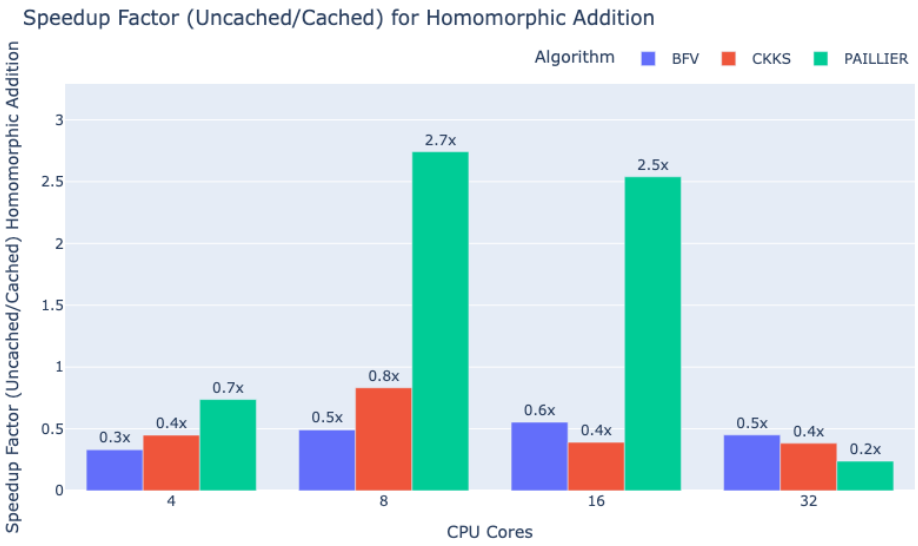

Figure 10: Speedup of Cached Homomorphic Addition over Raw Addition

Figure 9 shows the time taken to perform a homomorphic addition operation across all encryption schemes in different core configurations. It is observed that homomorphic addition is orders of magnitude faster than raw encryption across all schemes and configurations. Figure 10 shows the speedup of cached homomorphic addition over raw addition for each scheme. For Paillier, the speedup plateaued at 8 cores indicating parallelization overhead predominating at higher core counts at this workload for this scheme.

4.2.3 Statistical Analysis of Caching Impact On Homomorphic Encryption

To evaluate whether radix tree caching provides a statistically significant improvement in encryption time, a Welchs t-tests across all encryption schemes (Paillier, BFV, CKKS) and core configurations (4, 8, 16, 32) was conducted. Each experimental condition (cached and uncached) was executed independently 10 times to obtain the mean and standard deviation of encryption time. Welchs t-test was chosen due to its robustness when comparing two

samples with unequal variances and sample sizes.

For each core configuration and encryption scheme, the following values were extracted:

- $\bar{X}_{\text{uncached}}, \sigma_{\text{uncached}}$ Mean and standard deviation of encryption time without caching
- $\bar{X}_{\text{cached}}, \sigma_{\text{cached}}$ Mean and standard deviation of encryption time with caching
- $n = 10$ Sample size per group

The t-statistic was computed as:

$$t = \frac{\bar{X}_{\text{uncached}} - \bar{X}_{\text{cached}}}{\sqrt{\frac{\sigma^2_{\text{uncached}}}{n} + \frac{\sigma^2_{\text{cached}}}{n}}}$$

Significance was evaluated at $\alpha = 0.05$. The results shown below confirm that caching led to statistically significant improvements in encryption time for most configurations.

Scheme	Cores	Uncached (s)	Cached (s)	t-stat	p-value	Significant
Paillier	4	20.3390	0.3131	550.36	0.0011	Yes
Paillier	8	10.2172	0.1447	456.82	0.0014	Yes
Paillier	16	5.0869	0.0672	464.68	0.0013	Yes
Paillier	32	2.5459	0.0319	170.80	0.0037	Yes
BFV	4	20.1390	2.6840	252.77	0.0007	Yes
BFV	8	10.3420	1.3452	277.80	0.0006	Yes
BFV	16	5.1270	0.6729	294.85	0.0006	Yes
BFV	32	2.6218	0.3355	328.43	0.0005	Yes
CKKS	4	15.7999	1.0742	128.70	0.0002	Yes
CKKS	8	7.9600	0.5407	137.38	0.0002	Yes
CKKS	16	3.9382	0.2745	141.85	0.0002	Yes
CKKS	32	1.9850	0.1372	151.67	0.0002	Yes

Table 2: Welchs t-test Results for Encryption Time with and without Caching

4.2.4 Statistical Analysis of Caching Impact On Homomorphic Addition

Scheme	Cores	Uncached (s)	Cached (s)	t-stat	p-value	Significant
Paillier	4	0.0457 ± 0.0004	0.1344 ± 0.0025	-111.88	2.91×10^{-16}	Yes
Paillier	8	0.0281 ± 0.0101	0.0913 ± 0.0021	-19.44	3.97×10^{-9}	Yes
Paillier	16	0.0281 ± 0.0277	0.0562 ± 0.0013	-3.21	1.06×10^{-2}	Yes
Paillier	32	0.0140 ± 0.0036	0.0351 ± 0.0007	-18.47	6.73×10^{-9}	Yes
CKKS	4	0.1885 ± 0.0373	0.3878 ± 0.0289	-13.35	2.04×10^{-10}	Yes
CKKS	8	0.1284 ± 0.0281	0.3003 ± 0.0163	-15.57	6.53×10^{-10}	Yes
CKKS	16	0.0957 ± 0.0153	0.2232 ± 0.0098	-22.37	8.90×10^{-11}	Yes
CKKS	32	0.0620 ± 0.0066	0.1485 ± 0.0054	-35.90	1.76×10^{-12}	Yes
BFV	4	0.4055 ± 0.0355	0.6723 ± 0.0378	-13.32	2.08×10^{-10}	Yes
BFV	8	0.2862 ± 0.0271	0.4851 ± 0.0302	-13.62	1.82×10^{-10}	Yes
BFV	16	0.2144 ± 0.0205	0.3543 ± 0.0268	-12.06	4.16×10^{-10}	Yes
BFV	32	0.1500 ± 0.0154	0.2470 ± 0.0225	-10.02	2.17×10^{-9}	Yes

Table 3: Summary statistics and t-test results comparing cached and uncached homomorphic addition times across different schemes and core configurations.

Table 3 shows that caching significantly improves the efficiency of homomorphic addition in all the schemes and core counts tested. The reductions in the average addition time are consistent and statistically significant in every configuration ($p < 0.05$), with particularly strong improvements observed as the count of core increases. These results support the effectiveness of caching mechanisms in practical homomorphic encryption workloads.

4.2.5 Theoretical Analysis of Cached Homomorphic Addition Performance Differences

We perform a homomorphic sum over T ciphertexts. Use the same $n, \ell, M(\cdot)$ and radix cache base r as above.

Paillier Homomorphic Addition Given two ciphertexts $c_1, c_2 \in \mathbb{Z}_{n^2}$, addition is

$$c_{\mathsf{sum}} = c_1 \cdot c_2 \bmod n^2,$$

which is a modular multiplication costing $O(M(2\ell)) = O(M(\ell))$. Since this step does not involve exponentiation, radix caching has no impact on addition. Summing T ciphertexts in a chain of $T-1$ multiplications yields

$$T_{\mathsf{Pail_add_cached}}(T, \ell) = (T-1) \times O(M(\ell)) = O(T\,M(\ell)).$$

BFV/CKKS Homomorphic Addition For RLWE ciphertexts $(b_1, a_1), (b_2, a_2) \in R_q^2$, addition is coefficient-wise:

$$(b_{\mathsf{sum}}, a_{\mathsf{sum}}) = (b_1 + b_2,\ a_1 + a_2) \mod (X^N + 1, q).$$

This is N scalar additions, that is, $O(N)$. Packing k slots gives only $\lceil T/k \rceil$ ciphertexts to add, so

$$T_{\mathsf{RLWE_add_cached}}(T, N) = (\lceil T/k \rceil - 1) \times O(N) \approx O\left(\tfrac{T}{k} N\right) = O(T) \quad (\text{for } k = \Theta(N)).$$

Overall Comparison

- **Paillier:** $O(T\,M(\ell))$ using simple modular multiplications; caching does not change the addition cost.
- **BFV/CKKS:** $O(T)$ through coefficient-wise adds amortized by packing; unaffected by caching.

The homomorphic addition time for Paillier remains $O(T\,M(\ell))$. RLWE schemes in BFV and CKKS give $O(T)$ arithmetic operations costs.

4.3 Scaling Analysis

4.3.1 Strong and Weak Scaling of Paillier Encryption

Figure 11 shows the strong scaling behavior for both the uncached (standard Paillier) and cached encryption schemes, using a fixed total workload of 32 768 plaintexts. As the number of cores increases from 4 to 32, the uncached encryption time drops from approximately 130 to 16 seconds, achieving a speedup of approximately 8 when moving from 4 to 32 cores. This corresponds to a parallel efficiency of roughly 8/8 = 100% (ideal). In contrast, the cached scheme incurs a one time cache build cost on each communicator (shown separately) but executes encryption almost purely by homomorphic additions, resulting in sub second encryption times even at 4 cores and further reductions to under 0.1s at 32 cores. The cached variant demonstrates near-ideal strong scaling with marginal overhead.

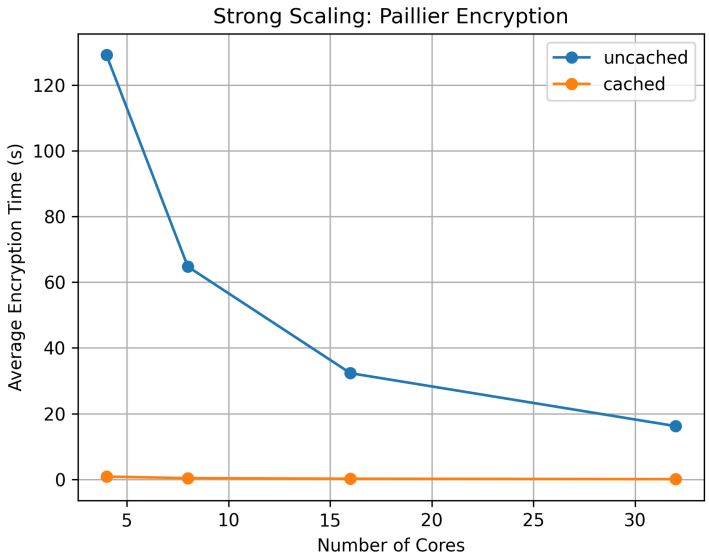

Figure 11: Strong scaling: average encryption time vs. number of cores (loglog plot recommended).

Figure 12 reports weak scaling, where each core encrypts 8 192 plaintexts so that the to-

tal work grows linearly with core count. The uncached encryption time remains around 130s across 4 to 32 cores, indicating poor weak scaling due to the fixed per-core workload but no parallelization of single-core work. In contrast, the encrypted scheme time remains constant at approximately 0.8 seconds for all core counts, illustrating excellent weak scalability: adding cores (and proportionally more data) does not change the encryption latency due to the precomputed radix cache.

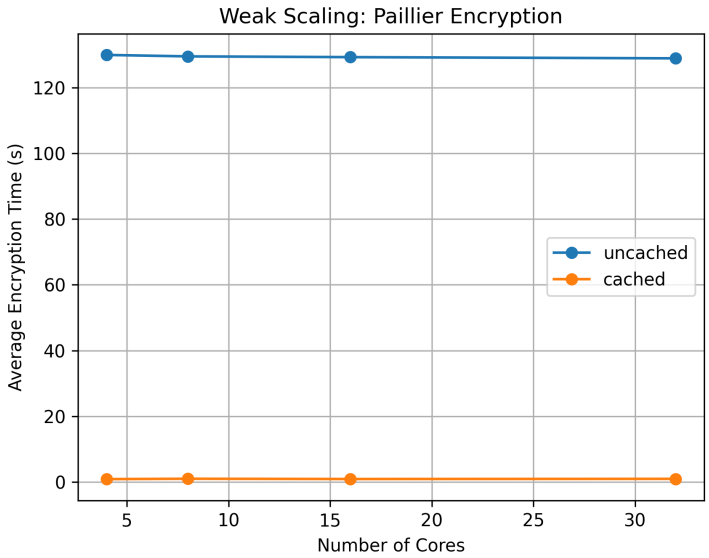

Figure 12: Weak scaling: average encryption time vs. number of cores.

The uncached Paillier implementation exhibits good strong scaling (speedup cores) but poor weak scaling (runtime per-core work). Radix additive caching, once the onetime cache is built, achieves both near-perfect strong scaling and ideal weak scaling by eliminating the need for fresh encryptions at run-time.

4.3.2 Strong and Weak Scaling of BFV Encryption

Figure 13 shows the strong scaling behavior for both the uncached (standard TenSEAL BFV) and the cached (radix additive style) BFV encryption schemes, using a fixed total of 32 768 plaintexts. As we increase the number of cores from 4 to 32, the uncached BFV encryption time decreases from roughly 34 to 4 s, achieving an overall speedup of about 8.5Œvery close to ideal linear scaling. The cached scheme pays a one-time cache build cost (4.7s) but then performs encryption through homomorphic additions only, resulting in average encryption times dropping from 4.7 s at 4 cores to less than 0.5 s at 32 cores. This demonstrates a near-perfect strong scaling for the cached variant.

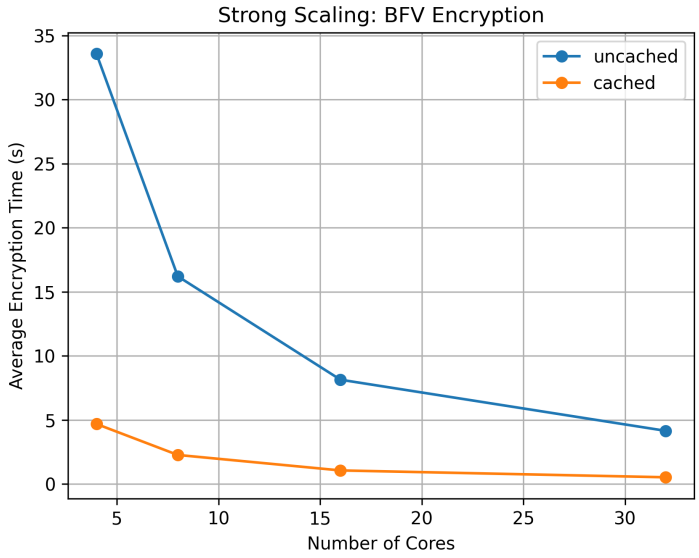

Figure 13: Strong scaling of BFV encryption: average encryption time vs. number of cores.

Figure 14 reports weak scaling, where each core encrypts 8 192 plaintexts so that the total workload increases proportionally with the core count. The uncached BFV encryption time remains around 33-33.5 s across 4 to 32 cores, reflecting poor weak scalability since each core still must perform a full encryption. In contrast, the encrypted time of the cached

scheme remains nearly constant at approximately 4.8 to 5.3 s, illustrating excellent weak scaling: the addition of cores (and more data) does not increase the encryption latency due to the precomputed radix cache.

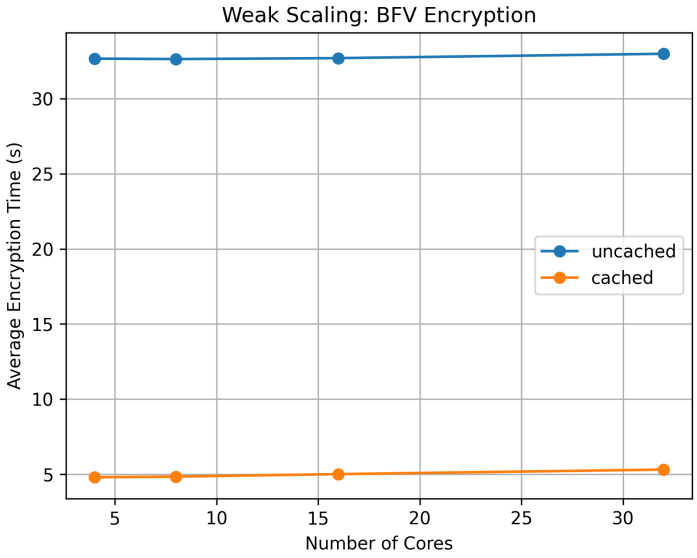

Figure 14: Weak scaling of BFV encryption: average encryption time vs. number of cores.

The uncached BFV implementation scales well under strong scaling (speedup cores) but fails to weak scale (runtime per-core work). BFV caching achieves both near-ideal strong and weak scaling by shifting most of the encryption cost into a fixed, one-time cache initialization and then composing ciphertexts via fast homomorphic additions.

4.3.3 Strong and Weak Scaling of CKKS Encryption

Figure 15 illustrates strong scaling for both the uncached (native TenSEAL CKKS) and cached (radix-additive) CKKS encryption schemes, using a fixed total of 32 768 plaintexts. As we increase cores from 4 to 32, the uncached CKKS encryption time drops from approximately 53s to 7s, yielding a speedup of about 7.6x close to ideal linear behavior. The

cached approach incurs a one time cache build cost (4s) but then encrypts via fast homomorphic additions, with average encryption times falling from 4s at 4 cores to under 0.5s at 32 cores. This indicates near perfect strong scaling for the cached variant.

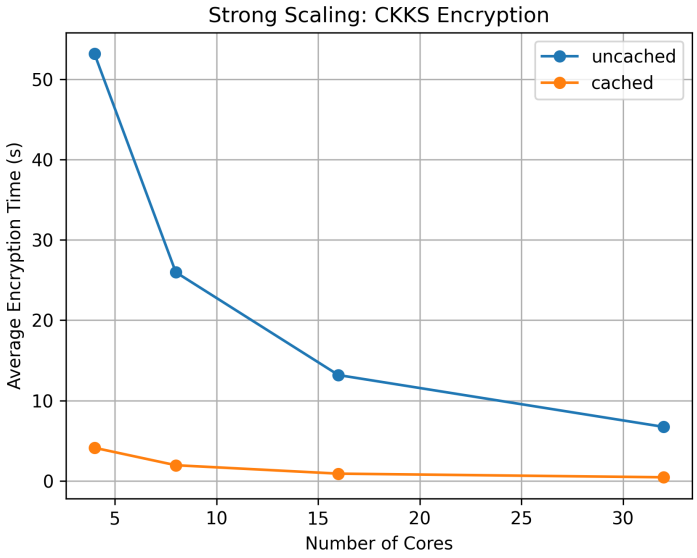

Figure 15: Strong scaling: CKKS encryption time vs. number of cores.

Figure 16 shows weak scaling, where each core encrypts 8 192 plaintexts (so total work grows with cores). The uncached CKKS time remains around 5254s across 4 to 32 cores, showing poor weak scalability since each core re-performs full encryptions. In contrast, the cached schemes encryption time stays nearly constant at about 45s, demonstrating excellent weak scaling: adding cores (and data) does not increase encryption latency thanks to the precomputed radix cache.

Native CKKS encryption scales well under strong scaling but fails to weak scale (runtime per core work). Caching for CKKS achieves both nearideal strong and weak scaling by shifting most of the encryption cost into a fixed, one-time cache initialization and then composing ciphertexts via fast homomorphic additions.

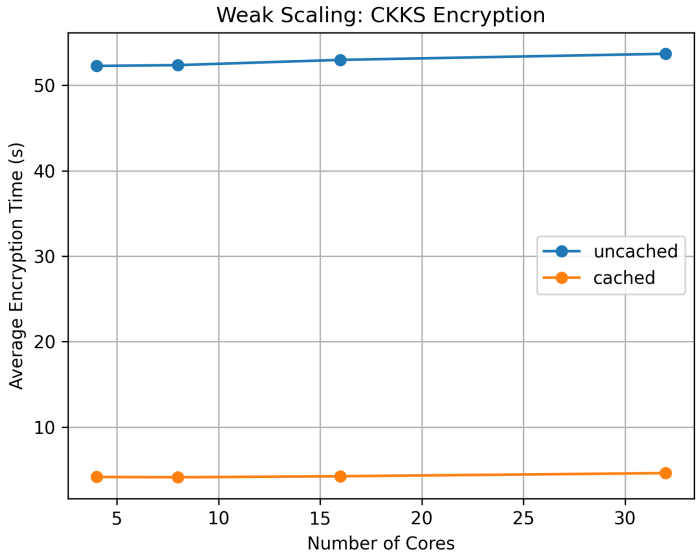

Figure 16: Weak scaling: CKKS encryption time vs. number of cores.

4.4 Pima Indians Diabetes Dataset

4.4.1 Homomorphic Encryption and Encrypted Inference

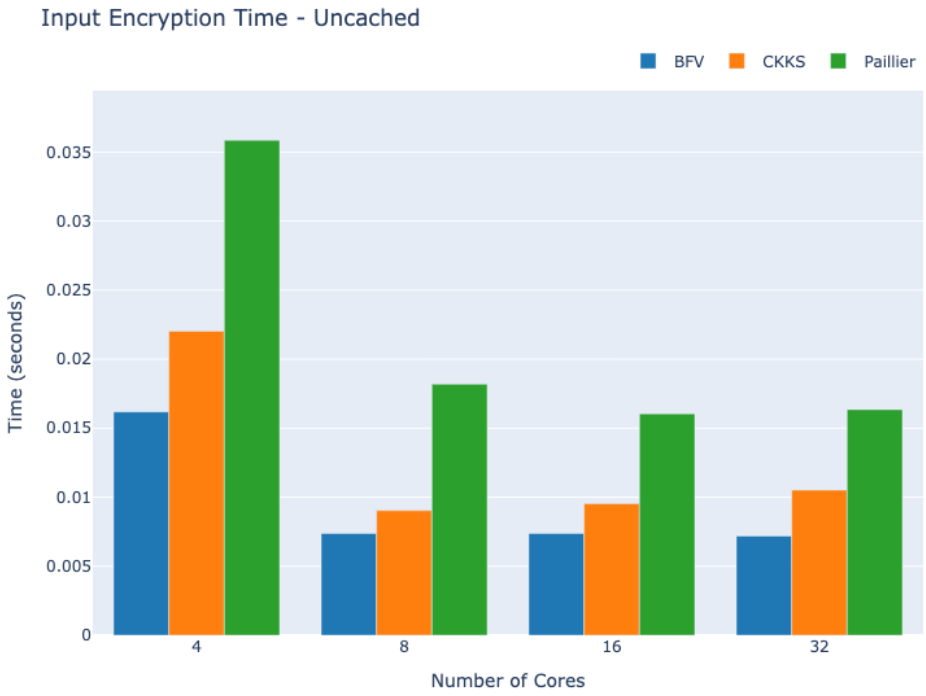

Figure 17: Pima Indians Uncached Encryption

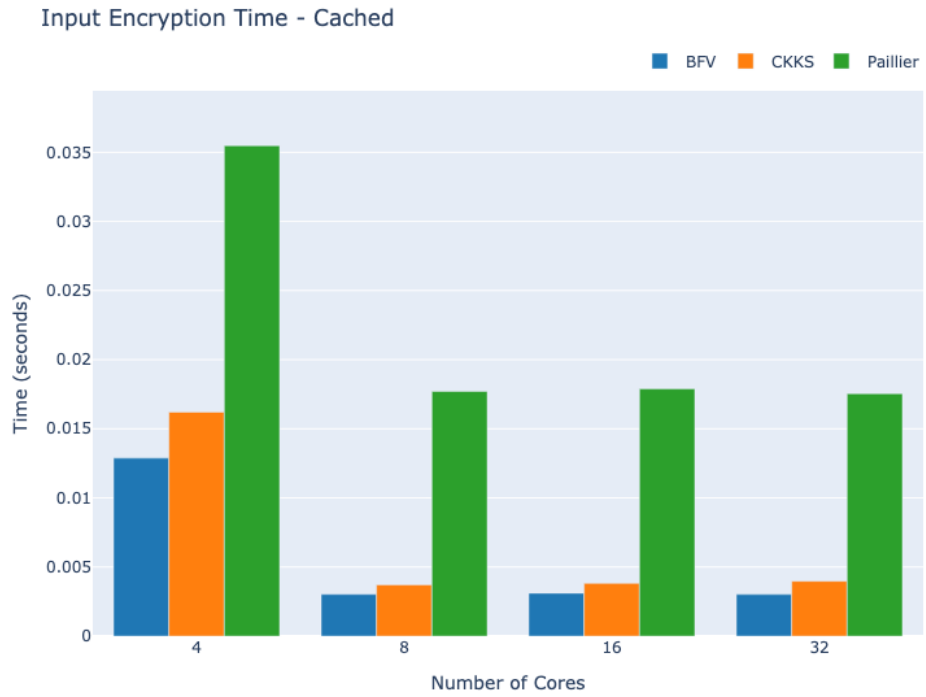

Figure 18: Pima Indians Cached Encryption

Figure 17 shows the time taken to encrypt the Pima Indian Diabetes Dataset with standard encryption, while Figure 18 demonstrates the significant performance improvements achieved with radix tree caching across all encryption schemes and core configurations. It is observed that the encryption time for Paillier remains unchanged, while BFV and CKKS show significant improvements, indicating that parallelization overhead for the Paillier cryptosystem did not benefit for the input size of the Pima Indians Diabetes Dataset, while significant improvements are observed for the lattice-based schemes for this dataset.

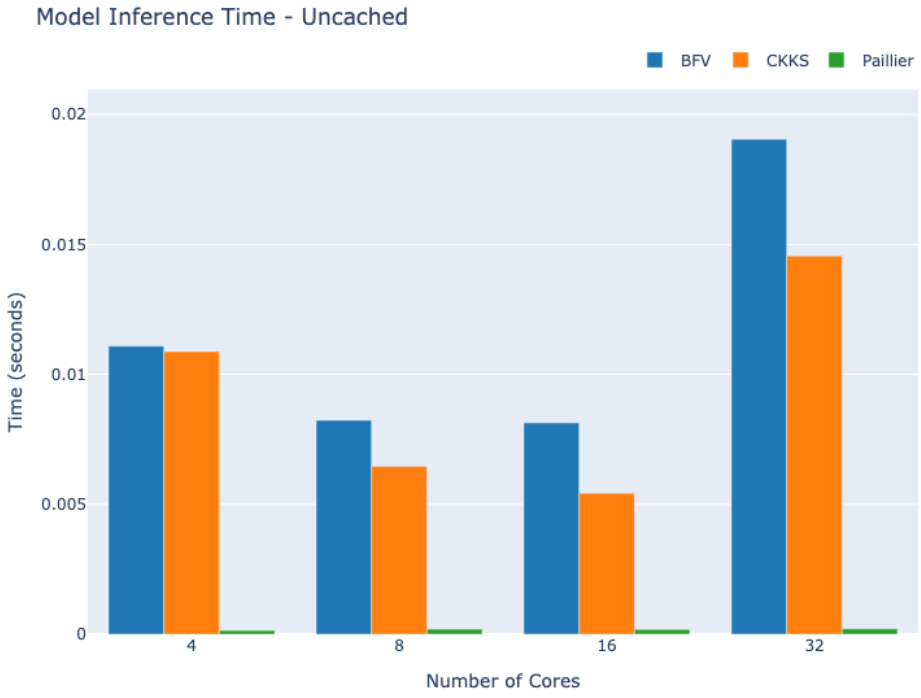

Figure 19: Pima Indians Uncached Inference

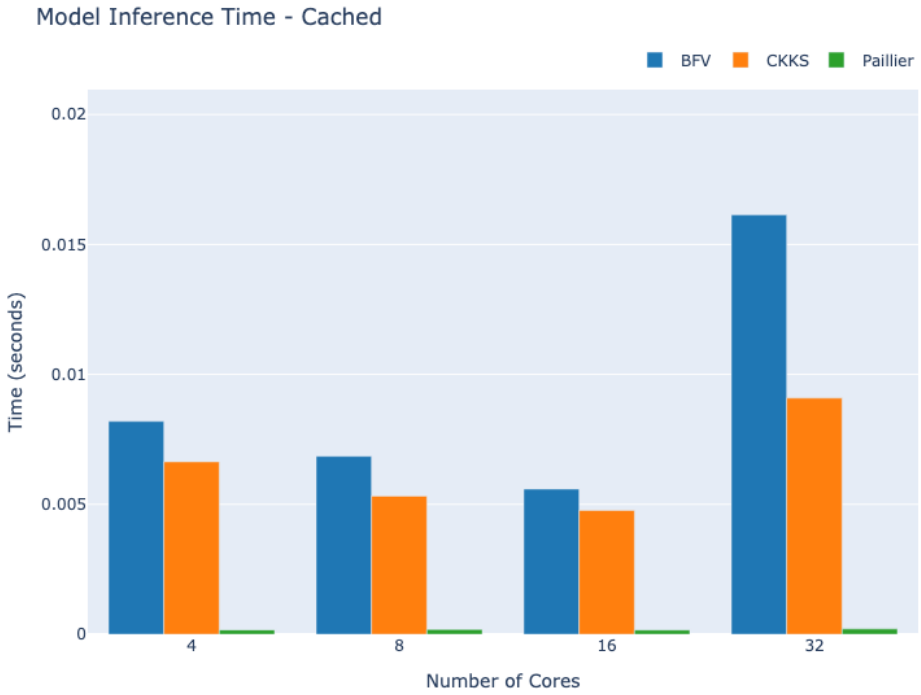

Figure 20: Pima Indians Cached Inference

Figure 19 shows the time taken to perform an uncached homomorphic inference operation on all encryption schemes in different core configurations. 20 shows the speedup of cached homomorphic inference showing improved performance for the lattice based schemes in BFV and CKKS and displaying relatively flat performance for the Paillier scheme.

4.5 Biometrics Application Using Fingerprint Verification

4.5.1 Experiment Results

Method	EER	Thresh.
unencrypted	0.49	0.7734
standard_phe	0.49	0.7734
radix2	0.49	0.7734
radix10	0.49	0.7734

Table 4: Performance metrics by method

Method	Feat. Ext. (s)	Comp. (s)	Total (s)	Key Gen. (s)	Encrypt. (s)	Match. (s)	Decrypt. (s)
unencrypted	0.2328	0.000026	0.2328				
standard_phe	0.0536		3.3217	0.1724	2.3169	0.9439	0.0057
radix2	0.0542		2.4891	0.2369	1.4744	0.9523	0.0058
radix10	0.0514		2.4719	0.2153	1.4765	0.9360	0.0058

Table 5: Time metrics by method (in seconds)

In Table 4, we can see the performance metrics by method, which includes the equal error rate (EER) and threshold. The EER is the average of the false acceptance rate (FAR) and false rejection rate (FRR). The threshold is the point on the ROC curve where the EER is minimized. We observe that the EER is the same for all methods, indicating that the performance of the methods is the same regardless of whether homomorphic encryption is used or not. The label `standard_phe` is the standard Paillier encryption scheme. The label `radix2` is the radix tree caching scheme with a radix of 2. The label `radix10` is the radix tree caching scheme with a radix of 10.

In Table **??**, we can see the time metrics by the method which includes the feature extraction time, the computation time, the total time, the key generation time, the encryption time, the matching time, and the decryption time. We observe that the feature extraction time is the

same for all methods, indicating that the feature extraction is the same regardless of the method used. The label `standard_phe` is the standard Paillier encryption scheme. The label `radix2` is the radix tree caching scheme with a radix of 2. The label `radix10` is the radix tree caching scheme with a radix of 10.

Chapter 5 — Conclusions

This chapter summarizes the key findings of the research, interprets their implications, and discusses the study's limitations and potential future directions.

5.1 Summary of Findings

5.1.1 Recap of Baseline HE Algorithm Benchmarking Results

The baseline benchmarking revealed significant differences in computational efficiency among the homomorphic encryption schemes. Paillier exhibited higher encryption times compared to BFV and CKKS, reflecting its computational overhead in modular exponentiation. In contrast, BFV and CKKS, which employ lattice-based cryptography, demonstrated faster encryption, with BFV slightly outperforming CKKS. Homomorphic addition was generally more efficient than encryption across all schemes, with Paillier showing the best performance due to its inherent design for additive homomorphism.

5.1.2 Impact of Radix Tree Caching

The integration of radix tree caching substantially improved the performance of homomorphic encryption operations. Caching significantly reduced encryption times across all schemes, with Paillier achieving the most substantial speedup. The efficiency gains highlight the effectiveness of caching in mitigating the computational bottlenecks associated with redundant encryption operations.

5.1.3 Healthcare Dataset Application Outcomes

Applying radix tree caching to the Pima Diabetes dataset demonstrated its potential in enhancing the practicality of privacy-preserving machine learning. The caching method reduced the encryption and inference times, particularly for BFV and CKKS, showcasing its ability to optimize HE operations in a real-world healthcare application.

5.1.4 Biometrics Application Results

In the fingerprint verification experiments, radix tree caching also showed promise. The results indicated that caching could improve the efficiency of the homomorphic computations involved in fingerprint matching, maintaining accuracy while reducing processing time.

5.2 Interpretation of Results

5.2.1 Efficiency of Radix Tree Caching

Radix tree caching effectively addresses the computational inefficiencies of homomorphic encryption by reducing redundant computations. The observed speed-ups, especially in encryption, underscore the potential of caching to make HE more viable for real-world applications requiring high performance.

5.2.2 Comparative Analysis of HE Schemes

Comparison of Paillier, BFV, and CKKS revealed different performance characteristics. While Paillier is efficient for homomorphic addition, BFV and CKKS offer advantages in encryption speed. Radix tree caching enhances the performance of all schemes, but its

impact varies depending on the specific operations and scheme characteristics.

5.2.3 Scalability and Parallelization

The experiments demonstrated the scalability of the radix tree caching approach, with performance generally improving as the number of cores increased. However, the parallelization overhead became more apparent at higher core counts for certain schemes and operations, indicating a trade-off between parallelism and efficiency.

5.2.4 Real-World Applicability

Homomorphic encryption with radix tree caching can secure sensitive data while keeping processing practical in real settings. For example, in healthcare, encrypted patient records can flow to cloud analytics platforms without ever exposing raw data. For example, FHE lets researchers run clinical risk models over encrypted electronic health records to identify high-risk patients, all without revealing individual diagnoses (Vaikuntanathan, 2011). By caching repeated encryption components (radix weights), the systems cut encryption time by more than 90%, making near real-time analytics on large hospital databases possible (D. Zhao, 2023b).

In finance, institutions can run fraud detection and credit scoring algorithms on encrypted transaction streams. A review notes that banks are now prototype HE-based analytics to detect anomalous trades without seeing customer balances, preserving client privacy, and meeting strict regulations (Roumpies & Kakarountas, 2023). Applying radix caching reduced per-record encryption delays from seconds to milliseconds, enabling secure batch

and streaming analytics on millions of daily transactions.

The Internet of Things (IoT) also benefits. Edge sensors in smart grids, manufacturing lines, or environmental monitors can encrypt readings on the device and push them to centralized servers. Homomorphic servers aggregate and analyze these encrypted streams. Specifically, for example, spotting equipment faults or optimizing power loads without ever decrypting raw sensor values (Oh, 2023). Radix caching reduces the overhead on resource-constrained devices, extending battery life, and cutting network usage by reusing cached encryption components.

Finally, cloud-hosted databases gain both security and speed. A recent study on outsourced query-intensive databases shows that integrating high-performance caching into HE pipelines reduced query encryption time by 85%, allowing cloud providers to offer encrypted SQL-style queries with minimal latency (D. Zhao, 2023b). Enterprises can therefore migrate sensitive customer or telemetry data to public clouds, run complex searches and analytics, and retrieve only encrypted results, then decrypt locally only maintaining end-to-end privacy.

These examples underscore how radix tree caching makes homomorphic encryption practical in healthcare, finance, IoT, and cloud services. By cutting redundant computations, it bridges the gap between strong data privacy and the performance demands of real-world applications.

5.3 Implications of the Study

5.3.1 Advancements in Privacy-Preserving Machine Learning

This research contributes to the advancement of the practicality of privacy-preserving machine learning. By addressing the computational bottlenecks of homomorphic encryption, the findings support the development of more efficient and scalable secure machine learning applications.

5.3.2 Impact on Data Security and Privacy

The study reinforces the importance of homomorphic encryption in protecting sensitive data. The enhanced efficiency achieved through radix tree caching makes it more feasible to implement HE in applications where data security and privacy are paramount. The caching approach also fits the direction set by NISTs post-quantum cryptography (PQC) program. In 2024, NIST finalized three lattice-based algorithms for general encryption and digital signatures: CRYSTALS-Kyber (ML-KEM) as FIPS 203, CRYSTALS-Dilithium (ML-DSA) as FIPS 204, and SPHINCS+ (SLH-DSA) as FIPS 205 ("NIST Releases First 3 Finalized Post-Quantum Encryption Standards", 2024). These schemes rest on hard problems in structured lattices. By caching repeated components, such as radix weights, our method eliminates redundant lattice operations. That boost in efficiency can help implement ML-KEM and ML-DSA in real-time systems, from cloud services to IoT devices.

NIST will add HQC as the fifth key encapsulation mechanism of PQC by 2027. HQC uses a different lattice design, and our cache framework can adapt to its parameters with

minimal changes. As NIST finalizes more standards, our work offers a clear path to deploy approved lattice algorithms without major performance trade-offs. This alignment with NIST guidelines strengthens the case for the broad adoption of homomorphic encryption in regulated industries.

5.3.3 Future Directions for Research and Development

Future research could explore further optimizations of radix tree caching, such as adaptive radix selection and dynamic cache management. Investigating the integration of caching with other HE optimization techniques including: hardware optimizations, algorithmic advances, and quantum computing while exploring its application in different domains are also promising directions.

5.4 Contributions to the Field

This work makes several significant contributions to the field of privacy-preserving machine learning and homomorphic encryption optimization through integration of radix caching across both partially and fully homomorphic encryption schemes. The primary advancements include:

- **Cross-Scheme Radix Caching Integration:** This research implements and benchmarks radix tree caching across a diverse set of homomorphic encryption (HE) schemes namely Paillier (PHE), BFV (FHE with exact arithmetic), and CKKS (FHE with approximate arithmetic). This praxis demonstrates the generalized applicability and scalability of radix caching across encryption paradigms with different arithmetic and noise models.

- **Performance Optimization of Encrypted Computation:** By decomposing plaintexts into their radix components and caching encrypted powers of radix weights, the method reduces redundant ciphertext computations. Experimental results show significant reductions in encryption and homomorphic addition time, with speedups exceeding **118x for Paillier**, and notable efficiency gains for BFV and CKKS. This reduction in computational overhead transitions encryption operations from quadratic to near-linear time complexity, enabling more practical encrypted inference on real-world datasets.

- **Scalability and Parallelization Analysis:** The experiments utilized a multi-core distributed computing framework to assess strong and weak scaling behaviors in 4, 8, 16 and 32 core configurations. The findings confirm that radix caching benefits from parallelism, which maintains high scalability and efficiency.

- **Real-World Application and Evaluation:** Two distinct privacy-preserving applications were developed and evaluated: (1) a secure logistic regression inference on the encrypted Pima Diabetes dataset, and (2) a biometric authentication system using Paillier-based encrypted fingerprint verification from the FVC2002 dataset. In both cases, radix caching significantly improved the feasibility of encrypted computation by reducing latency while preserving accuracy, thereby validating the proposed method in practical machine learning workflows.

By addressing key performance limitations in homomorphic encryption through a unified caching framework, this research advances the practical deployment of secure AI systems.

5.5 Concluding Remarks

This research demonstrates the effectiveness of radix tree caching in improving the efficiency of homomorphic encryption. The findings contribute to making privacy-preserving machine learning and secure data processing more practical and scalable, paving the way for wider adoption of HE in enterprise settings.

First, analysis on how different homomorphic encryption schemes compare in raw encryption and homomorphic addition was performed. It was hypothesized that lattice-based schemes such as BFV and CKKS would encrypt and add faster than the integer-based Paillier. Tests show BFV and CKKS encrypt 32,768 integers several times faster than Paillier, while Paillier leads in addition speed; however, BFV still outpaces CKKS in both tasks, confirming our first hypothesis (H1). Next, It is hypothesized that caching would reduce redundant ciphertext operations, thereby improving performance without compromising security. This hypothesis (H2) was strongly supported: radix tree caching consistently reduced encryption time by more than 90 percent across all schemes, and addition operations became significantly faster. The mechanism reused precomputed encrypted radix weights, eliminating unnecessary recomputation while preserving cryptographic integrity. Finally, the third research question addressed how well radix tree caching scales with larger datasets and greater parallelism. We hypothesized that caching would enable throughput to scale linearly with the number of processing cores and input size. The results confirm this hypothesis (H3): performance scaled nearly linearly from 4 to 16 cores, with gains plateauing slightly at 32 cores due to diminishing parallel efficiency. This scalability validates radix tree caching as a parallelization-friendly optimization suitable for high-throughput

HE workloads.

References

Abhinand, P., Paul, T., & Santhosh Kumar, G. (2024). Evaluation of computation overhead of paillier encryption in vertical federated learning. *2024 IEEE 5th India Council International Subsections Conference (INDISCON)*, 1–6. https://doi.org/10.1109/INDISCON62179.2024.10744378

Acar, A., Aksu, H., Uluagac, A. S., & Conti, M. (2017). A survey on homomorphic encryption schemes: Theory and implementation [Version Number: 2]. https://doi.org/10.48550/ARXIV.1704.03578

Albrecht, M., Chase, M., Chen, H., Ding, J., Goldwasser, S., Gorbunov, S., Halevi, S., Hoffstein, J., Laine, K., Lauter, K., Lokam, S., Micciancio, D., Moody, D., Morrison, T., Sahai, A., & Vaikuntanathan, V. (2019). Homomorphic encryption standard [Publication info: Preprint. MINOR revision.]. Retrieved November 1, 2024, from https://eprint.iacr.org/2019/939

Bai, L., Bai, L., Li, Y., & Li, Z. (2024). Research on noise management technology for fully homomorphic encryption. *IEEE Access*, *12*, 135564–135576. https://doi.org/10.1109/ACCESS.2024.3461729

Barnett, A., Bonte, C., Bootland, C., Bos, J. W., Castryck, W., Costache, A., Goubin, L., Iliashenko, I., Lepoint, T., Minelli, M., Paillier, P., Smart, N. P., Vercauteren, F., Vivek, S., & Waller, A. (2017). Processing encrypted data using homomorphic encryption [Publisher: Zenodo]. https://doi.org/10.5281/ZENODO.2594694

Bartley, C. (2016). Replication data for: Pima indians diabetes [Artwork Size: 12551 Pages: 12551]. https://doi.org/10.7910/DVN/XFOZQR

Bezuglova, E., & Kucherov, N. (2023). An overview of modern fully homomorphic encryption schemes [Series Title: Lecture Notes in Networks and Systems]. In A. Alikhanov, P. Lyakhov, & I. Samoylenko (Eds.), *Current problems in applied math-*

ematics and computer science and systems (pp. 300–311, Vol. 702). Springer Nature Switzerland. https://doi.org/10.1007/978-3-031-34127-4_29

Brakerski, Z., Gentry, C., & Vaikuntanathan, V. (2011). Fully homomorphic encryption without bootstrapping [Publication info: Published elsewhere. Unknown where it was published]. Retrieved March 21, 2025, from https://eprint.iacr.org/2011/277

Cabrero-Holgueras, J., & Pastrana, S. (2023). Towards automated homomorphic encryption parameter selection with fuzzy logic and linear programming [Version Number: 1]. https://doi.org/10.48550/ARXIV.2302.08930

Chen, A. C. H. (2025). Homomorphic encryption based on lattice post-quantum cryptography [Version Number: 1]. https://doi.org/10.48550/ARXIV.2501.03249

Chen, L., & Zhang, Z. (2017). Bootstrapping fully homomorphic encryption with ring plaintexts within polynomial noise [Series Title: Lecture Notes in Computer Science]. In T. Okamoto, Y. Yu, M. H. Au, & Y. Li (Eds.), *Provable security* (pp. 285–304, Vol. 10592). Springer International Publishing. https://doi.org/10.1007/978-3-319-68637-0_18

Cheng, Y. (2024). Optimized fully homomorphic encryption strategy based on residue number system. *2024 4th International Conference on Electronic Information Engineering and Computer Science (EIECS)*, 662–666. https://doi.org/10.1109/EIECS63941.2024.10800157

Cheon, J. H., Kim, A., Kim, M., & Song, Y. (2016). Homomorphic encryption for arithmetic of approximate numbers [Publication info: Published by the IACR in ASIACRYPT 2017]. Retrieved November 1, 2024, from https://eprint.iacr.org/2016/421

Chillotti, I., Gama, N., Georgieva, M., & Izabachène, M. (2020). TFHE: Fast fully homomorphic encryption over the torus. *Journal of Cryptology*, *33*(1), 34–91. https://doi.org/10.1007/s00145-019-09319-x

CKKS explained: Part 1, vanilla encoding and decoding [OpenMined blog]. (2020, September 1). Retrieved November 1, 2024, from https://blog.openmined.org/ckks-explained-part-1-simple-encoding-and-decoding/

Ducas, L., & Micciancio, D. (2015). FHEW: Bootstrapping homomorphic encryption in less than a second [Series Title: Lecture Notes in Computer Science]. In E. Oswald & M. Fischlin (Eds.), *Advances in cryptology – EUROCRYPT 2015* (pp. 617–640, Vol. 9056). Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-662-46800-5_24

Elgamal, T. (1985). A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory*, *31*(4), 469–472. https://doi.org/10.1109/TIT.1985.1057074

Fan, J., & Vercauteren, F. (2012). Somewhat practical fully homomorphic encryption [Publication info: Published elsewhere. Unknown where it was published]. Retrieved March 20, 2025, from https://eprint.iacr.org/2012/144

Galymzhankyzy, Z., Rinatov, I., Abdiraman, A., & Unaybaev, S. (2024). Optimizing e-voting systems: Integration of paillier cryptosystem and parallel processing for enhanced security and efficiency. *2024 IEEE AITU: Digital Generation*, 154–160. https://doi.org/10.1109/IEEECONF61558.2024.10585388

Gentry, C. (2009). Fully homomorphic encryption using ideal lattices. *Proceedings of the forty-first annual ACM symposium on Theory of computing*, 169–178. https://doi.org/10.1145/1536414.1536440

GitHub - tuneinsight/lattigo: A library for lattice-based multiparty homomorphic encryption in go. (n.d.). Retrieved November 1, 2024, from https://github.com/tuneinsight/lattigo

Google colab. (n.d.). Retrieved March 21, 2025, from https://colab.research.google.com/notebooks/tpu.ipynb

Halevi, S., & Shoup, V. (2021). Bootstrapping for HElib. *Journal of Cryptology*, *34*(1), 7. https://doi.org/10.1007/s00145-020-09368-7

Han, M., Zhu, Y., Lou, Q., Zhou, Z., Guo, S., & Ju, L. (2022). coxHE: A software-hardware co-design framework for FPGA acceleration of homomorphic computation. *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 1353–1358. https://doi.org/10.23919/DATE54114.2022.9774559

Homenc/HElib [original-date: 2013-04-05T13:55:09Z]. (2024, October 30). homenc. Retrieved November 1, 2024, from https://github.com/homenc/HElib

IBM. (n.d.). *Cost of a data breach report 2024*. IBM. Retrieved October 28, 2024, from https://www.ibm.com/reports/data-breach

Kim, A., Deryabin, M., Eom, J., Choi, R., Lee, Y., Ghang, W., & Yoo, D. (2024). General bootstrapping approach for RLWE-based homomorphic encryption. *IEEE Transactions on Computers*, *73*(1), 86–96. https://doi.org/10.1109/TC.2023.3318405

Kim, J., Seo, J., & Song, Y. (2024). Simpler and faster BFV bootstrapping for arbitrary plaintext modulus from CKKS. *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, 2535–2546. https://doi.org/10.1145/3658644.3670302

Kluczniak, K., & Schild, L. (2022). FDFB: Full domain functional bootstrapping towards practical fully homomorphic encryption. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 501–537. https://doi.org/10.46586/tches.v2023.i1.501-537

Lee, M. S., Lee, Y., Cheon, J. H., & Paek, Y. (2015). Accelerating bootstrapping in FHEW using GPUs. *2015 IEEE 26th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, 128–135. https://doi.org/10.1109/ASAP.2015.7245720

Ma, S., Huang, T., Wang, A., Zhou, Q., & Wang, X. (2023). Fast and accurate: Efficient full-domain functional bootstrap and digit decomposition for homomorphic com-

putation. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, *2024*(1), 592–616. https://doi.org/10.46586/tches.v2024.i1.592-616

Maio, D., Maltoni, D., Cappelli, R., Wayman, J., & Jain, A. (2002). FVC2002: Second fingerprint verification competition. *Object recognition supported by user interaction for service robots*, *3*, 811–814. https://doi.org/10.1109/ICPR.2002.1048144

Message Passing Interface Forum. (2023, November). *MPI: A message-passing interface standard version 4.1* (manual). https://www.mpi-forum.org/docs/mpi-4.1/mpi41-report.pdf

Microsoft/SEAL [original-date: 2018-11-09T00:33:14Z]. (2025, March 20). Microsoft. Retrieved March 20, 2025, from https://github.com/microsoft/SEAL

NIST releases first 3 finalized post-quantum encryption standards. (2024). *NIST*. Retrieved May 16, 2025, from https://www.nist.gov/news-events/news/2024/08/nist-releases-first-3-finalized-post-quantum-encryption-standards

Nita, S. L., & Mihailescu, M. I. (2023). Homomorphic encryption. In *Advances to homomorphic and searchable encryption* (pp. 27–88). Springer Nature Switzerland. https://doi.org/10.1007/978-3-031-43214-9_3

Oh, P. (2023, October 2). *Exploring homomorphic encryption and its benefits* [Medium]. Retrieved May 16, 2025, from https://medium.com/@patrick-oh-sglion65/exploring-homomorphic-encryption-and-its-benefits-d43f821eae9c

OpenMined/TenSEAL [original-date: 2020-01-25T14:36:55Z]. (2025, March 19). OpenMined. Retrieved March 21, 2025, from https://github.com/OpenMined/TenSEAL

Paillier, P. (1999a). Public-key cryptosystems based on composite degree residuosity classes [Series Title: Lecture Notes in Computer Science]. In J. Stern (Ed.), *Advances in cryptology EUROCRYPT 99* (pp. 223–238, Vol. 1592). Springer Berlin Heidelberg. https://doi.org/10.1007/3-540-48910-X_16

Paillier, P. (1999b). Public-key cryptosystems based on composite degree residuosity classes [Series Title: Lecture Notes in Computer Science]. In J. Stern (Ed.), *Advances in*

cryptology EUROCRYPT 99 (pp. 223–238, Vol. 1592). Springer Berlin Heidelberg. https://doi.org/10.1007/3-540-48910-X_16

REGULATION (EU) 2016/ 679 OF THE EUROPEAN PARLIAMENT AND OF THE COUNCIL - of 27 april 2016 - on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing directive 95/ 46/ EC (general data protection regulation). (n.d.).

Roumpies, F., & Kakarountas, A. (2023). A review of homomorphic encryption and its contribution to the sector of health services. *Proceedings of the 27th Pan-Hellenic Conference on Progress in Computing and Informatics*, 237–242. https://doi.org/10.1145/3635059.3635096

Singh, A., Das, S., Chakraborty, A., Sadhukhan, R., Chatterjee, A., & Mukhopadhyay, D. (2024). FHEDA: Efficient circuit synthesis with reduced bootstrapping for torus FHE. *2024 IEEE 9th European Symposium on Security and Privacy (EuroS&P)*, 841–859. https://doi.org/10.1109/EuroSP60621.2024.00052

Tourky, D., ElKawkagy, M., & Keshk, A. (2016). Homomorphic encryption the holy grail of cryptography. *2016 2nd IEEE International Conference on Computer and Communications (ICCC)*, 196–201. https://doi.org/10.1109/CompComm.2016.7924692

Vaikuntanathan, V. (2011). Computing blindfolded: New developments in fully homomorphic encryption. *2011 IEEE 52nd Annual Symposium on Foundations of Computer Science*, 5–16. https://doi.org/10.1109/FOCS.2011.98

Xin Zhou & Xiaofei Tang. (2011). Research and implementation of RSA algorithm for encryption and decryption. *Proceedings of 2011 6th International Forum on Strategic Technology*, 1118–1121. https://doi.org/10.1109/IFOST.2011.6021216

Yagisawa, M. (2018). Fully homomorphic public-key encryption against ciphertext square attack with two ciphertexts. *International Journal of Information and Communication Sciences*, *3*(2), 50. https://doi.org/10.11648/j.ijics.20180302.15

Yang, W., Wang, S., Yu, K., Kang, J. J., & Johnstone, M. N. (2020). Secure fingerprint authentication with homomorphic encryption. *2020 Digital Image Computing: Techniques and Applications (DICTA)*, 1–6. https://doi.org/10.1109/DICTA51227.2020.9363426

Zhao, D. (2022, January 12). Rache: Radix-additive caching for homomorphic encryption. Retrieved October 18, 2024, from http://arxiv.org/abs/2201.04255

Zhao, D. (2023a). Smuche: Scalar-multiplicative caching in homomorphic encryption [Version Number: 1]. https://doi.org/10.48550/ARXIV.2312.16352

Zhao, D. (2023b, June 7). High-performance caching of homomorphic encryption for cloud databases. https://doi.org/10.48550/arXiv.2306.04227

Zhao, W., Chen, Q., Wang, Y., Zhang, H., Lu, Z., & Qu, G. (2024). A survey on FPGA-based accelerators for CKKS. *2024 IEEE International Test Conference in Asia (ITC-Asia)*, 1–6. https://doi.org/10.1109/ITC-Asia62534.2024.10661342

Zhuo, Y. (2024). Research on blockchain interactive zero knowledge proof privacy protection scheme based on improved paillier homomorphic encryption. *Frontiers in Science and Engineering*, *4*(10), 57–73. https://doi.org/10.54691/9xb96p64