HEProfiler: An In-Depth Profiler of Approximate Homomorphic Encryption Libraries

Jonathan Takeshita†, Nirajan Koirala†, Colin McKechney and Taeho Jung*

Department of Computer Science and Engineering, University of Notre Dame, Notre Dame, 46556, Indiana, USA.

*Corresponding author(s). E-mail(s): tjung@nd.edu;
Contributing authors: jtakeshi@nd.edu; nkoirala@nd.edu; cmckechn@alumni.nd.edu;
†These authors contributed equally to this work.

Abstract

Fully Homomorphic Encryption (FHE) allows computation on encrypted data. Various software libraries have implemented the approximate-arithmetic FHE scheme CKKS [1], which is highly useful for applications in machine learning and data analytics; each of these libraries have differing performance and features. It is useful for developers and researchers to learn details about these libraries' performance and their differences. Some previous work has profiled FHE and CKKS implementations for this purpose, but these comparisons are limited in their fairness and completeness. In this article, we compare four major libraries supporting the CKKS scheme. Working with the maintainers of each of the PALISADE, Microsoft SEAL, HElib, and HEAAN libraries, we devise methods for fair comparisons of these libraries, even with their widely varied development strategies and library architectures. To show the practical performance of these libraries, we present HEProfiler, a simple and extensible framework for profiling C++ FHE libraries. Our experimental evaluation is complete in both the scope of tasks tested and metrics evaluated, allowing us to draw conclusions about the behaviors of different libraries under a wide range of real-world workloads. This is the first work giving experimental comparisons of different bootstrapping-capable CKKS libraries.

Keywords: CKKS scheme, Fully Homomorphic Encryption, Profiler

1 Introduction

Fully Homomorphic Encryption (FHE) schemes allow computations to take place on encrypted data. This allows the separation of computation and knowledge, allowing privacy-preserving outsourcing of calculations on user data to external parties. However, FHE is not always feasible to deploy in practice, mainly due to the huge computational overhead incurred with homomorphic operations [2]. Since the first theoretical realization of FHE in 2009 [3], much work has been been done to make FHE more practically usable. One line of work has been in the development of the approximate homomorphic encryption scheme, which performs approximate arithmetic computation and make FHE useful for practical applications in machine learning and data analytics (e.g., the CKKS scheme [1]).

There exist several FHE software libraries implementing CKKS (among other functionality). With the gradual inclusion of many different schemes, capabilities, optimizations, and engineering features, the major libraries' maturity has advanced. This work is motivated by a desire to understand the relative and absolute performance of commonly used CKKS libraries, in order to enable users to decide which libraries to use in varying circumstances.

Many different research works have used these libraries to implement and learn about the real-world performance of their work using CKKS (and other schemes). However, such data is scattered across different papers with differing libraries, hardware, and end applications. It is thus informative to have a unified and fair comparison of different libraries, so that researchers and developers can make informed choices about the advantages and tradeoffs of CKKS implementations. This helps users not only evaluate the relative performance of libraries, but also gives runtime measurements to compare CKKS-based private computation with other methods for secure computation, including Trusted Execution Environments [4, 5], Secure Multiparty Computation [6], or application-specific cryptographic protocols [7].

In this work, we carefully analyze the features of some of the most prominent C++ FHE libraries implementing CKKS, and we present our software framework for profiling them. In particular, we examine the PALISADE, Microsoft SEAL, HElib, and HEAAN libraries. Our open-source software framework HEProfiler is extensible to other libraries, and allows for reproducibility of results. Our comparison is more in-depth than other works in comparing library performance over a wide range of parameters, as well as parameter selection and other features relevant to developers (e.g., licenses, programming features). Finally, we present a comprehensive performance evaluation of these libraries' performance with approximate homomorphic encryption, examining all of latency, throughput, memory consumption, and computation error.

Many other works similarly profile FHE performance [8–14]. In our work, in order to give a fair comparison, we faced the challenge of finding equitable ways to compare libraries while allowing each library the freedom to exercise its parameter selection strategies. This was a difficult task: each library has its own different method of parameter selection, and they are not directly

compatible. (For example: HEAAN allows the user to choose from a small set of precomputed parameter settings based on the computation's multiplicative depth, while HElib asks the user to directly choose many parameters of the underlying ring polynomials used in CKKS.) In contrast, many other works simply mandate parameters for all profiled libraries [8, 9], depriving the libraries of the opportunity to use their own unique strategies for selecting the best parameters for good performance. However, this strategy will lose a great amount of information about how these libraries work in real-world use cases: most users are not FHE experts, and will not carefully tweak parameter settings, but will instead rely on a software library to yield a set of suitable parameters for their application. Therefore, in our survey we aim to take into account the automatic parameter selection provided by a library, as that is a feature that will likely be heavily used and relied upon by most end users. Our review and profiling was advised by lead contributors of each of these libraries, allowing us to very thoroughly perform a fair comparison without disadvantaging any library.

There is a body of work in comparing the performance of different FHE libraries [8, 10, 12, 15]. In contrast to these works, we focus on approximate FHE (CKKS) only, and we take a particular focus on exploring performance at a large range of parameter selections. It has already been demonstrated that different FHE schemes are better suited to different tasks; for example, it is easy to guess and show that comparing the performance of Boolean and approximate-arithmetic FHE schemes on a workload of logical circuit evaluation will show that the Boolean FHE scheme is better-suited to the task [8]. We thus compare FHE libraries on more fair footing by testing their implementations of the same scheme and thoroughly comparing their performance on different low-level and high-level benchmarks at a wide range of parameter settings. We choose the CKKS approximate FHE scheme [1] for our evaluation, due to the many uses of approximate FHE for applications such as analytics, machine learning, and statistical regression. Also, unlike other types of FHE, the CKKS scheme is the dominant scheme in the domain of approximate FHE, and it is implemented in almost all FHE libraries that have approximate FHE. Therefore, HEProfiler has a broader impact even though it focuses on the CKKS scheme.

We focus more on benchmarking the core operations of CKKS than on extensive end-to-end testing, as differences between developers' implementations of high-level tasks within each library can unfairly disadvantage one library or another, even with automated methods of writing a task as an FHE computation [8, 16]. We also perform some end-to-end benchmarking on simple applications, to show how our observations on low-level operations extend to end-to-end computation. Our findings about performance differences can be applied to future work in compilers/transpilers to help select the best FHE backend and parameters for a task and inform future development of compilers/transpilers. Notably, we are able to make comparisons not only for homomorphic operations, but also for bootstrapping, making our work the first to compare libraries' CKKS bootstrapping capabilities. This lets us draw conclusions about the

libraries' relative performance for applications of high multiplicative depth, such as deep learning.

1.1 Contributions

1. We analyze four prominent CKKS [1] libraries, and collaborate with the maintainers of each library in order to fairly evaluate these libraries even with their differing strategies in parameter selection. We overcome the challenge of giving a fair comparison of libraries with different parameter selection by allowing each library to choose its own parameters as much as possible to satisfy the constraints of the task at hand. This lets the libraries compete on the merits of their parameter selection, which is more fair and realistic than previous works that fixed identical parameter settings across different libraries.
2. We wrote an open-source software framework HEProfiler for profiling different CKKS implementations. This allows for reproducibility and further evaluation of future libraries and workloads. HEProfiler is thus valuable for many future explorations of the performance of homomorphic encryption libraries, even beyond what we present in this paper. Our framework is extensible to any C++ library and can implement a wide range of tests (low-level or high-level) in addition to what we implemented. HEProfiler is also particularly useful for examining library usability, as it reports not only latency, but other metrics not reported in previous work, including throughput, memory consumption, and computation error.
3. We present a thorough experimental evaluation of the four CKKS libraries and use these results to draw conclusions and make recommendations about usage and development with these libraries. Our results encompass both core operations of CKKS and high-level applications for a wide range of parameters, yielding complete results useful for informing a wide range of user applications. Furthermore, we compare bootstrapping-capable libraries' performance for bootstrapping and other operations, the results from which can better inform our conclusions about the relative strengths and weakensses of each library.

2 Related Work

2.1 Works Profiling CKKS

Fawaz et al. [13] profile BGV and CKKS in Microsoft SEAL for a limited set of parameters. They profile homomorphic addition, multiplication, and squaring in end-to-end scenarios, including timing for encryption/decryption and encoding/decoding.

FHEBench, by Jiang and Ju [10], compares different FHE schemes and libraries. They profile runtimes for arithmetic FHE in B/FV, BGV, and CKKS using SEAL, PALISADE, and HElib. They also profile logical operations for TFHE and FHEW. They found that PALISADE is the best library in terms of

performance for arithmetic FHE in their experiments. FHEBench is only a set of benchmarks and data, and does not share any test harness code, or claim such as a contribution. In the paper's present state, the graphs are confusing and difficult to read, though their text explains their results and conclusions well.

Dordevic et al. [15] profile BGV in Microsoft SEAL, B/FV in PALISADE, and CKKS in HElib. They analyze the runtimes of encryption, decryption, homomorphic addition, and homomorphic multiplication. Most relevant to our work, they conclude that CKKS homomorphic multiplication is best implemented by PALISADE at smaller parameters, but that SEAL performs better at larger parameters.

Doan et al. [9] survey many different fully and partially homomorphic encryption schemes. Their survey covers many aspects, such as theoretical capabilities and limits, security differences, and performance comparison. Similarly to our work, they test implementations of FHE libraries, also testing PALISADE, SEAL, HElib, and (an older version of) HEAAN. They focus exclusively on lower-level primitive operations (key generation, encryption/decryption, addition, and multiplication), and use only a few parameter settings.

Gouert et al. [8] perform a thorough investigation of the relative performance of FHE libraries and schemes for different tasks. They profile PALISADE, SEAL, HElib, Lattigo, and TFHE for a wide variety of tasks. They accomplish this through the use of a generic compiler that translates a given task into FHE code. This work further confirmed the observation that different FHE schemes are better suited for different tasks, prompting our investigation to focus less on different schemes and more on differences between libraries using the same scheme. While thorough in terms of tasks, the parameter investigation of this work was limited and did not exhaustively compare different libraries at different parameter settings.

HEBench is a benchmarking framework for homomorphic encryption to compare implementations of various workloads on the hardware and software level of various HE libraries [17]. HEBench is developed by Intel Corp., and contributors include Duality, IBM Research, Microsoft, and KU Leuven. HEBench currently has support for five workloads at the time of writing this paper: dot product, element-wise addition and multiplication, logistic regression, and matrix multiplication. Currently, SEAL, PALISADE and HElib are the only libraries that can be benchmarked using HEBench, and they note to add support for additional libraries in the future. HEBench implements separate modules for the front and back end of the HEBench framework. These modules communicate with each other using a middle module named API-Bridge. The modular design of HEBench makes it more applicable to add extensions in the future. They also test mostly end-to-end applications and achieve the best performance for each library for various workloads by fine-tuning the parameters. Unlike HEBench, our framework does not have a design with separate backend and frontend modules, which makes it less extensible. However, the absence of individual modules adds simplicity to our framework and easy for developers to add more customizations. We also focus more on the fairness of the tests by

fine-tuning parameters for each library instead of each individual test. Our work provides a more complete comparison compared to HEBench, which focuses on the runtime of core FHE operations. Fundamental CKKS operations, along with rotation, relinearization, and ciphertext-plaintext operations are tested in our framework; we also test some end-to-end applications. Our framework is simpler compared to HEBench and, at its current stage, offers support for the comparison of one additional CKKS library, HEAAN, with the capability to add support for other libraries.

Zhu et al. [18] compared the performance of OpenFHE, Microsoft SEAL, and HElib for the task of simple convolutional neural networks of low multiplicative depth not requiring bootstrapping. They concluded that SEAL shows the best performance on these tasks, and also note that OpenFHE and HElib are useful for their wide selection of algorithms and historical value, respectively.

2.2 Works Profiling or Analyzing Other Homomorphic Encryption

Sathya et al. [14] compared the Microsoft SEAL, HElib, TFHE, Paillier, ELGamal, and RSA libraries. This review analyzed the features of these libraries for partial and somewhat homomorphic encryption for the various schemes they implement. This analysis did not consider runtime, and mostly compared the features and supported operations of the libraries.

The study of Melchor et al. [12] compares the HElib, FV-NFLlib, and SEAL libraries, focusing on the cases of large plaintext moduli. Their comparison compares overall library performance without considering that the different libraries implement different schemes (HElib implements BGV, while FV-NFLib and SEAL implement B/FV for finite-field homomorphic encryption). They conclude that BGV outperforms B/FV for larger plaintext spaces, and B/FV performs better for smaller plaintext spaces with less depth afforded. They note that SEAL is the generally most preferable choice, due to its user-friendliness and more active development.

Varia et al. [11] presented HEtest to evaluate FHE on Boolean circuits. Their test harness measures several metrics including key and circuit generation time, evaluation, encryption, decryption time, and ciphertext expansion. Their tests, run at 80 bits of security, showed performance results for HElib and a plain evaluation baseline. HEtest is thorough and extensible, though its focus on binary circuits shows a limited and sometimes unwieldy application of homomorphic encryption.

3 Background

Homomorphic encryption has existed in some form since the development of RSA in 1978 [19]. In the 1980s and 1990s, many other attempts like [20], [21], [22], [23], [24], [25], and [26] were made towards practical homomorphic encryption schemes but these schemes allowed either only one type of operation or a limited number of computationally heavy operations on the encrypted data.

For instance, [25] allowed only additive homomorphism and [21] allowed only multiplicative homomorphism in their schemes. These schemes are classifed as Partially Homomorphic Encryption schemes, as they only allow one type of operation with an unlimited number. Another type of homomorphic encryption called Somewhat Homomorphic Encryption (SHE) allows homomorphic evaluation of some functions, but is limited in what it can compute; SHE is frequently limited primarily by the multiplicative depth of computations. [27] proposed the first scheme of this type which is capable of performing two operations: an arbitrary number of additions and one multiplication.

Fully Homomorphic Encryption (FHE) schemes are able to perform arbitrary calculations on encrypted data. Gentry's seminal Ph.D. thesis [28] proposed the first FHE scheme which supports the evaluation of arbitrary circuits using a technique called bootstrapping. However, the bootstrapping technique introduced by Gentry for refreshing ciphertext noise was too costly in terms of computation and made this scheme very impractical in real-world use cases. Following Gentry's work, many new schemes and optimizations have been proposed. Newer FHE schemes include the BGV [29], B/FV [30], THFE [31], FHEW [32] and CKKS [1] schemes. Implementations of these schemes may include only the SHE variant, the full FHE scheme, or both.

3.1 Approximate Homomorphic Encryption

The CKKS scheme [1] is similar to the BGV [29] and B/FV [30] schemes, as its core operations are on ring polynomials over finite fields. BGV and B/FV use differing methods to manage noise and ensure exactly correct decryption on finite-field plaintexts. In contrast, CKKS uses a *complex canonical embedding* to encode and operate with fixed-point numbers. CKKS does not try to completely remove noise from the decrypted result as BGV and B/FV do; instead, the noise is part of the error inherent in limited-precision approximate arithmetic. This difference does lead to a subtle security issue [33], which has been mitigated in some libraries via the addition of extra noise [34].

In this work, we focus on the CKKS scheme for our performance evaluation of FHE libraries. CKKS is applicable to a wide variety of tasks involving numerical workloads where some error is tolerable, most notably for deep learning or linear regression.

CKKS operates upon ring polynomials in $R_{p \cdot q} = \frac{\mathbb{Z}_{p \cdot q}[X]}{X^N+1}$ for powers of two N. Usually, the polynomial modulus degree N ranges from 2^{10} to 2^{17} and the ciphertext modulus $p \cdot q$ is approximately 50 to 800 bits wide for 128-bit classical security. These values parameterize CKKS ciphertexts, determining their size. Further, in CKKS $\frac{N}{2}$ operands can be packed into a single ciphertext, so N affects throughput as well.

3.2 AVX512 and HEXL

FHE schemes based upon the Ring Learning With Errors problem operate on polynomial rings. The polynomial elements of these rings typically have

thousands of coefficients that in $\mathbb{Z}_{p \cdot q}$, where $p \cdot q$ may be hundreds of bits wide. For arithmetic on these rings, the operations of polynomial-polynomial multiplication, polynomial-scalar multiplication, and polynomial-polynomial coefficientwise multiplication are significant performance bottlenecks.

In order to reduce the overhead of these operations, the use of new CPU instructions has been explored. The Advanced Vector Extensions extend the x86 instruction set with Single Instruction Multiple Data operations, allowing arithmetic computations to take place on multiple operands simultaneously. Intel's HEXL library [35] utilizes the AVX512 instructions (operating on 512-bit operands) to accelerate polynomial arithmetic. HEXL can be interposed into libraries to replace their original polynomial arithmetic and has been demonstrated to bring speedups of up to 6.26× to PALISADE and SEAL. All of the libraries we profiled optionally use AVX512 via HEXL to improve their performance.

4 Libraries Profiled

In this section, we briefly describe the libraries we studied in this work. We give a shorter high-level overview of each library, and more detailed analyses are provided in Section 5. Our framework can be extended to include any other FHE library for evaluation, though ensuring a fair comparison across different programming languages may pose challenges. Our selection criteria for the libraries we evaluated prioritized factors such as relevance, prominence, community interest (both historical and current), ease of development and integration with user codebases, and potential for future hardware optimization (e.g., multithreading or GPU utilization). We did not include other libraries such as TFHE, Lattigo, or Concrete [31, 36, 37]. For those interested in the evaluation of libraries such as Lattigo, one can refer to existing works [8, 37, 38].

During the writing of this paper, the OpenFHE library [39] was released. OpenFHE is the next-generation successor to PALISADE and retains almost complete API continuity. The code for OpenFHE has been taken directly from PALISADE. Thus, our conclusions about PALISADE in this work also generally apply to OpenFHE. The only major changes at the time of writing between PALISADE and OpenFHE are in the API for parameter creation, which is much easier to use in OpenFHE.

4.1 Microsoft SEAL

Microsoft SEAL is an open-source C++ library implementing the B/FV [30] and CKKS [1] FHE schemes. SEAL is not multithreaded, though it is generally thread-safe. SEAL does not have required external dependencies, but by default uses optional dependencies for testing, serialization, and AVX512 support. SEAL is actively maintained by Microsoft Research. In general, SEAL is easy to use and understand, but is lacking the finer control over parameters and wide selection of algorithms included by other libraries such as PALISADE and HElib.

Some work [40] uses versions of SEAL with bootstrapping available, but this functionality is not publicly available. Despite this, we still refer to SEAL (and HElib) as a FHE library for simplicity. Additionally, some useful functionality (e.g., the Simulator class) available in previous versions is no longer available in the current version.

When using SEAL, the `-pedantic` flag cannot be used during compilation, due to the use of native 128-bit integers.

4.2 PALISADE

PALISADE is an open-source C++ library implementing several FHE schemes, including B/FV [30], BGV [41], CKKS [1], and TFHE [31] FHE schemes. PALISADE is very fully-featured and includes additional functionality such as multi-party FHE utilities and signatures. PALISADE does not have required external dependencies but can include dependencies for multi-precision arithmetic, memory management, and AVX512 support. PALISADE uses OpenMP for multithreading. PALISADE is actively maintained by Duality Technologies, though their support efforts have recently shifted to its direct successor OpenFHE. OpenFHE's code was directly forked from PALISADE, and its API is almost identical.

PALISADE is the most customizable library we reviewed; nearly every parameter for CKKS can be tweaked. This may actually be a disadvantage for non-expert users, especially as there is little in the way of warnings in the case of incorrectly set parameters prior to a fatal error. PALISADE's highly modular and generic design allows a great deal of flexibility in the choice of backend components but makes reading PALISADE code more difficult.

While PALISADE's release and development versions did not have bootstrapping for finite-field or approximate FHE at the time of writing, Duality Technologies provided us with an internal version that implements CKKS bootstrapping.

4.3 HElib

HElib is an open-source C++ library implementing the BGV [41] and CKKS [1] schemes. HElib has GMP and NTL as dependencies, which can be either installed with HElib as a package, or one can link HElib against preexisting installations. HElib can also be linked against separately built Intel HEXL for AVX512 support. HElib uses NTL's threading macros, which themselves use OpenMP. HElib is maintained by various parties.

HElib is generally easy to use, and allows more precise control of some parameters (e.g., the number of columns in key-switching matrices or the Hensel lifting factor) which are not easily configured in other libraries. HElib initially implemented BGV; CKKS support was added at a later date to the library. Some features such as bootstrapping and polynomial evaluation methods are thus only implemented for BGV. HElib's documentation focuses more on the theoretical foundations of the BGV and CKKS schemes and how they

are implemented in the HElib library [42]. It also includes many algorithms specifically designed for homomorphic encryption that can be implemented in other libraries using low-level scheme operations. HElib is targeted primarily at researchers and developers who want to use experimental features and provides documentation that is both thorough and theoretical. HElib is not as actively maintained as the other libraries we profile.

4.4 HEAAN

HEAAN is a proprietary C++ library implementing only the CKKS scheme [1]. HEAAN does not have external dependencies and provides versions with and without both AVX512 and GPU support. HEAAN includes multithreading via OpenMP and is under development for later distribution by CryptoLab. Unlike other libraries, the parameter generation in HEAAN is directly based on the required multiplication level and the homomorphic encryption capabilities (SHE or FHE). HEAAN recommends the use of provided parameter presets, though capabilities for non-preset parameter selection are available. This setting might be helpful to developers who strictly want to choose parameters based on the multiplicative level of their HE application. HEAAN's set of parameter presets is more limited than the capabilities offered by PALISADE or HElib; one example of this was that the smallest parameter setting from HEAAN was larger than needed for applications with a multiplicative depth of one. From our advice, the HEAAN maintainers added a parameter preset for use in such applications.

HEAAN has a rich set of features for CKKS, though at the time of writing it is limited in some features (e.g., deferring relinearization indefinitely). HEAAN was not yet publicly available at the time of experimentation, though some older versions of the library are available online. CryptoLab, the current developers of HEAAN, provided us with HEAAN's headers and compiled libraries for use in our tests. No publicly available open-source version of HEAAN includes both RNS arithmetic [43, 44] and bootstrapping. At the time of writing, HEAAN is proprietary and closed-source. CryptoLab now provides Docker containers with HEAAN for non-commercial use, allowing for the reproducibility of our results.

5 Comparison of Library Features

5.1 Parameter Selection

Parameter selection for SHE and FHE remains a difficult problem even for experts, and fully automatic parameter selection is still an open problem. All of the libraries we profiled had some method of assisting the user in parameter selection. The most important parameter from a user's perspective is the depth of the computation; in most other cases other parameters will be secondary or dependent upon the depth.

HEAAN provides several parameters presets for SHE, for computation depths of up to 19. HEAAN also gives 3 parameter presets for use in FHE,

using polynomial modulus degrees with $N \in \{15, 16, 17\}$. HEAAN also allows "Custom" parameter selection, similar to other libraries, though CryptoLab recommends the use of presets.

PALISADE provides some parameter selection, though some expertise is required with regard to the many arguments needed. No warnings are generated for incorrect or incompatible arguments, which can lead to confusing runtime errors. PALISADE does allow a depth to be specified. In addition to the normal methods for PALISADE's parameter selection, the PALISADE team was also able to provide us with expertly selected parameters for optimizing runtime for both the SHE and FHE cases. We report the ordinary and optimized PALISADE cases separately (as "PALISADE" and "PALISADE_OPT", respectively).

SEAL needs a small amount of help from the user for its parameter selection, but the general strategy is not difficult. The tradeoff of this is that the user has relatively little ability to customize the parameters and algorithms used.

HElib does not have parameter selection in the same way other libraries do. Other libraries have some way of generating parameters from the user's requested depth. HElib, by contrast, does not provide this, but only gives a list of possible parameters that guarantee 128 bits of classical security. We tested each suggested parameter setting for the range of multiplicative depths we benchmarked and chose the smallest set of parameters that gave the desired depth.

In most cases, parameters for different libraries at the same depth were approximately equal or close to each other (e.g., a difference of at most one in $|N|$). To account for this, we report throughput for our tests to compare practical performance even with variations in parameter selection.

5.2 Multithreading

All of PALISADE, HElib, and HEAAN have multithreading available to use via OpenMP. This can be easily controlled by setting the environment variable `OMP_NUM_THREADS`. Even if this is not specified, PALISADE will automatically use multithreading. It should be noted by developers that utilizing OpenMP threads at the application level will disable their use in library code that is called from multithreaded user code.

At the time of writing, HElib's multithreading is not enabled, due to a function not being thread-safe; it is likely that this will be fixed in a future release.

While SEAL does not use threads, it is generally thread-safe. At the time of writing, its developers recommend using multithreading at the application level.

5.3 IND-CPA^D Security

As described by Li and Micciancio [33], approximate homomorphic encryption can leak information about a user's secret key in some scenarios. Since the noise component is a part of the message in CKKS, it results in the linearity of the decryption function to the secret key revealing the decryption noise, and making

it vulnerable to IND-CPA^D attackers. Hence, the result of a decryption may leak a small amount of information about the user's secret key, so an attacker in a chosen-plaintext setting who can repeatedly query an oracle for decryptions of homomorphically encrypted messages can compromise the secret key used to decrypt that information. To mitigate this, PALISADE, HEAAN, and HElib have some method of adding additional noise during decryption to mask the information leakage [34]. This method of defending against such attacks is often referred to as *noise flooding* or *noise smudging*, where properly calibrated noise is attached to the decryption result at the end of the decryption process [45]. SEAL does not take any mitigations, but recommends that users treat decrypted results as privileged information [46]. As pointed out by Badawi et al. [39], IND-CPA^D security is only required when the ciphertext decryption result is shared with other parties who do not possess the secret key in applications such as private set intersection [47]. Therefore, it is important for the users to identify their security needs under which FHE libraries can operate for the given application-case scenarios. To ensure fairness while running the evaluations, we did not enable IND-CPA^D mitigations for PALISADE, HEAAN, and HElib as it would affect the runtime and accuracy for these libraries during noise flooding which would be absent in SEAL.

5.4 Serialization and Objects

Each library uses a different selection of objects to manage parameters and keys. SEAL separates encryption parameters and an encryption context into different objects, and uses different objects to handle encryption, decryption, evaluation, et cetera. HEAAN also follows this approach, additionally using custom `ParameterPreset` objects for parameter settings (HEAAN has formulatic parameter selection, but the authors discourage its use and recommend the use of presets.) PALISADE and HElib use only a pair of keys and a single context object which is used to perform all operations on ciphertexts and plaintexts.

All libraries implement some form of serialization for ciphertexts, allowing the user to save and load ciphertexts to/from C++ I/O streams. Additionally, PALISADE and HEAAN have functions to directly serialize objects to/from files, saving developers some boilerplate code. (PALISADE relies on the `cereal` library for its serialization). SEAL incorporates compression into its serialization, reducing the size of serialized objects by up to 60%. SEAL further optimizes its serialization by saving only seeds used for pseudorandom number generation when possible.

5.5 Hardware Acceleration Support

HEAAN provides a GPU-accelerated version using CUDA, either with or without AVX512 support. While SEAL does not have an official GPU-capable version, previous works [48, 49] have used GPU to accelerate SEAL, showing runtime improvements of up to 140×. PALISADE does not include a GPU

version, but does have an experimental repository for using GPU to accelerate some core operations. HElib does not include GPU acceleration.

All libraries that we profiled can use HEXL to take advantage of AVX512.

5.6 Licenses

SEAL is licensed under an MIT License. PALISADE is under a BSD-2 license. HElib is under an Apache 2.0 license. HEAAN is proprietary, and is not yet available for general use, so its license details are not known. All of the open-source licenses used are permissive, making them usable for proprietary and/or closed-source projects.

Table 1: Feature List of FHE Libraries

Feature	SEAL (-)	PALISADE (-)	HElib (-)	HEAAN (-)
Open-Source	✓	✓	✓	×
License	MIT	BSD-2	Apache	TBA
Multithreading	×	✓	✓	✓
Serialization	✓	✓	✓	✓
AVX512 Support	✓	✓	✓	✓
GPU Support	×	×	×	✓
IND $-$ **CPA**$^{\mathbf{D}}$ mitigations [33]	×	✓	✓	✓

6 Profiling

Our testing was run on a computer with an Intel Xeon Gold 6226 CPU @ 2.70GHz, 192GB memory, and a NVIDIA RTX6000 GPU. We chose a test computer to be representative of a fairly powerful server, as this best fits the use case of outsourced private computation. All the tests were run for 50 iterations; we use the average latency and error for each test. We set inputs for all tests to be equal to 1 in all slots. This allowed us to calculate error relative to the computation without divide-by-zero issues. The raw data we collected can be found at https://drive.google.com/file/d/1ALa39CS2l7P2w9KqRfXRARo_XUJk-nIQ/view?usp=sharing.

Throughput and error were computed using the following formulae in all of our tests:

$$\text{Throughput (operations/second)} = \frac{\text{Batch Size(operations/batch)}}{\text{Runtime (second/batch)}}$$

$$\text{Error\%} = \left| \frac{\text{Expected Value} - \text{Actual Value}}{\text{Actual Value}} \right| \cdot 100\%$$

6.1 Design of HEProfiler

HEProfiler is implemented in C++17, and is available at https://gitlab.com/jtakeshi/homenc-profile. Our design goals were ease of use, extensibility, and

allowing each library's benchmark implementations the freedom to select parameters and perform operations with only minimal specifications. As such, HEProfiler is implemented using a base class `HEProfiler` and an inheriting class for each library we profiled. Each class must implement a constructor that takes in basic parameters, most importantly the desired depth of the computation. Further, each class implements functions to run both basic (i.e., primitive) operations and simple end-to-end computations, as well as functions that give other information (e.g., ciphertext size, key sizes). Each of these functions reports back runtime or other relevant metrics, e.g., size in bytes. Functions also report computation error, which is an important metric when considering the usability of HE libraries. Extension of HEProfiler to include other libraries is simple; the user must only write their implementation, define appropriate preprocessor macros (strongly recommended, to allow compilation with/without a library), and add the necessary headers and libraries to the build system (see below).

We were constrained by the need to include HEAAN, whose compiled libraries were targeted for specific platforms. As a result of this, we also wrote a custom build system for HEProfiler. The driver program is built and run in a Singularity container [50] that is compatible with the HEAAN libraries, and has had the other (open-source) libraries installed. Our system also allows optional selection of AVX512 optimizations (all libraries) and GPU acceleration (HEAAN only). The need for either requesting binaries ad-hoc or using containers when working with closed-source software illustrates a tradeoff that developers should consider when selecting a library to use.

6.2 Parameter Selection

In our evaluation, we primarily examined the impact of increasing parameter size on library performance. After much consultation with library developers and CKKS experts, we chose to use each library's methods of parameter selection to choose settings based on the computation depth. This allows each library to use its own parameter selection methods to choose parameters efficiently by whatever methods the library's developers chose, allowing us to indirectly test the quality of each library's parameter selection. We provide assistance and basic requirements (e.g., computation depth) to allow each library to select its own parameters for a computation, avoiding the manual selection of a single unified set of parameters for all profiled libraries. Our approach follows the guidelines set forth by each library and is thus representative of the methods that developers will use. (It may be possible to tweak each library's parameter selection to obtain even better parameters for efficiency or error, but doing so is extremely tedious even for experts.) For completeness, we further consider handpicked parameters from the developers of PALISADE, to show the differences between library-chosen and human-selected parameters. This approach to parameter selection allows us to fairly evaluate library performance in a wide range of circumstances without the tedious manual selection of parameters. Other works [8, 9] assign CKKS parameters (e.g., N, $|q|$) directly

and force the libraries to adhere to these; we instead allow libraries to select their own parameters. We give the parameters for leveled CKKS that each library chooses in Table 2. All parameter sets used in this evaluation yielded at least 128 bits of classical security.

For parameters allowing bootstrapping, HEAAN allows the user to choose from presets giving $|N| \in \{2^{15}, 2^{16}, 2^{17}\}$. PALISADE's parameter selection for bootstrapping is much more configurable (though it did not allow). For fairness, we chose parameters for PALISADE yielding equivalent N and allowed depth between bootstraps.

Table 2: Leveled CKKS parameters

Depth	SEAL		PALISADE		HElib		HEAAN	
	$log_2(N)$	$log_2(p \cdot q)$	$log_2(N)$	$log_2(p \cdot q)$	$log_2(N)$	$log_2(p \cdot q)$	$log_2(N)$	$log_2(p \cdot q)$
1	13	160	13	101	13	268	13	217
2	13	200	14	140	13	322	13	217
3	14	240	14	181	14	429	13	217
4	14	280	14	221	14	453	15	662
5	14	320	14	261	14	483	15	662
6	14	360	14	301	14	593	15	662
7	14	400	15	341	14	637	15	662
8	15	440	15	381	15	667	15	652
9	15	480	15	421	15	755	15	866
10	15	520	15	461	15	811	15	866
11	15	560	15	501	15	854	15	866
12	15	600	15	541	15	934	15	866
13	15	640	15	581	15	1002	15	866
14	15	680	15	621	15	1024	15	866
15	15	720	16	661	15	1118	15	860
16	15	760	16	701	15	1178	15	860
17	15	800	16	741	15	1216	15	860
18	15	840	16	781	15	1309	15	860
19	15	880	16	821	15	1355	15	860

6.3 Core Operation Performance with Increasing Parameters

For testing SHE computations, our profiler takes the strategy of accepting a desired multiplicative depth of the computation from the user and then allowing each library to select a set of parameters to satisfy the depth and other constraints. This strategy is closest to what a non-expert end user would use. For HEAAN, we noted a relative disadvantage for single-depth computation due to the smallest depth preset being for a depth of 3. The CryptoLab team was able to provide us with custom parameters for single-depth SHE, which we utilized. HEAAN's parameter presets allowed a depth of up to 19 for SHE, which is large enough for most practical applications; we thus tested all libraries up to that depth.

6.3.1 Homomorphic Multiplication

We first discuss the relative performance of each library for homomorphic multiplication. Figure 1 show the relative runtimes of each library for homomorphic multiplications. We see that for smaller depths, there is little difference in performance. At higher depths, differences in the libraries' parameter choices and performance start to make themselves more apparent. Interestingly, HElib suddenly shows much higher latency than other libraries for depths of 17 to 19, despite having a comparable performance at lower depths. Similarly, other sudden jumps in performance are apparent, indicating that differences in parameter selection are highly influential in the libraries' performance at a given depth. As the user-specified depth increases, different libraries' choices of $|N|$ at a certain depth may differ.

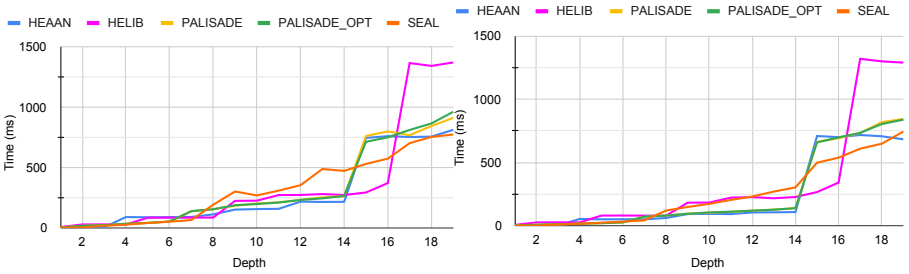
Besides only runtime, other metrics are important in determining the performance of a library. Figure 2 shows the error (relative to the inputs) incurred in homomorphic multiplication. We see that error is generally tolerable for a single multiplication. Looking at Figure 3b, we see that the error remains under at most 5% of the original value for up to 18 consecutive multiplications for all libraries, though the error does begin to increase at a much higher rate for all of HEAAN, HElib, and PALISADE at about 14 consecutive multiplications.

We recall that different libraries may have different methods of parameter selection that lead to different choices and sizes of parameters and operands (see Section 5.1. In particular, choices of the polynomial modulus degree may differ, affecting the amount of operands that can be packed into a single CKKS ciphertext. This can lead to latency only not giving a complete picture of the practical performance of libraries. For this reason, we also investigate the throughput of FHE operations, in order to fairly evaluate performance with differing parameter selections. Figures 4a and 4b show the throughput in operations/second of the libraries we profiled. We see that HEAAN and SEAL show better throughput at very low depths, but that the throughput of all libraries quickly degrades, with only minor differences. We conclude that for most practical purposes, at higher depths the choice of the library does not affect computational performance much and that other factors such as error, ciphertext size, and key size should be more important considerations.

6.3.2 Other Primitive Operations

We also evaluated the runtime of other CKKS operations besides homomorphic multiplication. Though the runtime of homomorphic multiplication dominates that of other arithmetic operations, it is still informative to compare library performance on less-intensive operations.

Runtimes for homomorphic addition are shown in Figures 5a and 5b. We see that as parameter sizes increase, HEAAN generally has the best performance for homomorphic addition, followed closely by SEAL. At lower-depth parameter settings, all libraries have comparable performance. Throughput for homomorphic addition is shown in Figures 5b and 6a. As parameter sizes



(a) AVX2 (b) AVX512

Fig. 1: Homomorphic Multiplication Runtimes

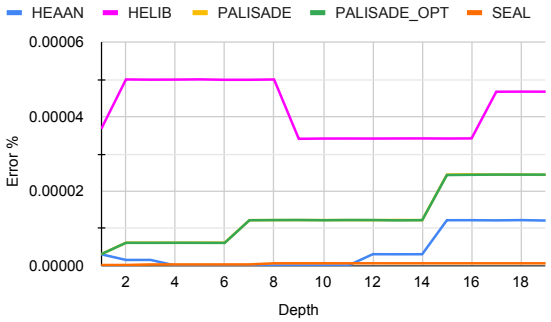
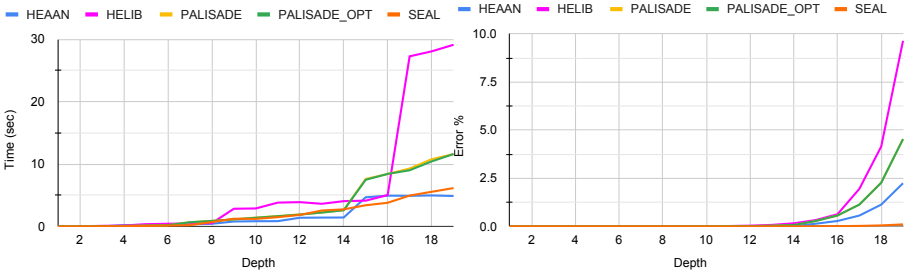


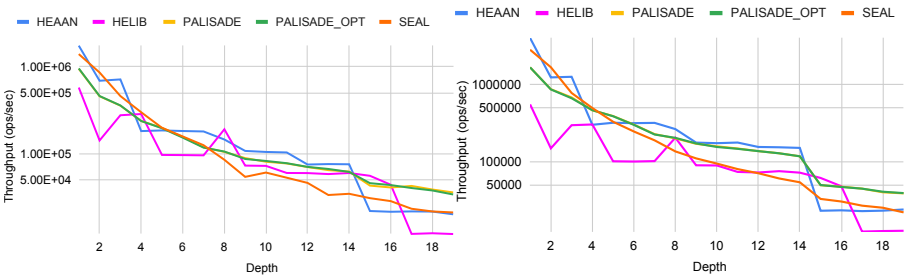
Fig. 2: Homomorphic Multiplication Error



(a) Runtimes (b) Error

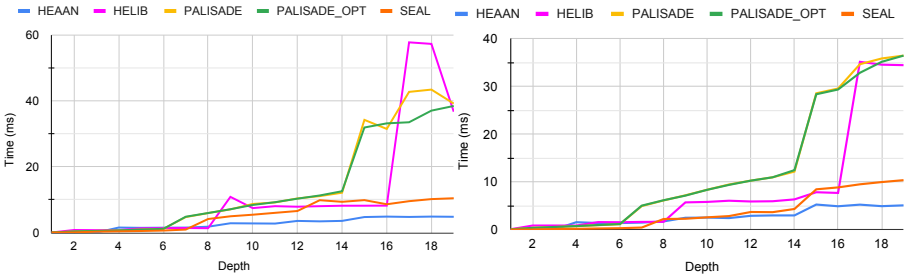
Fig. 3: Homomorphic Multiplication (Chained up to allowable depth)

increase, all libraries have comparable performance. Error in homomorphic addition is shown in Figures 5b and 7a. HEAAN has the best (lowest) error for parameter settings yielding multiplicative depths of 4 to 12, though SEAL shows better error in other cases. For addition-heavy workloads, we recommend using HEAAN, as it performs best at scale and has the best throughput at lower-depth parameters.



(a) AVX2 (b) AVX512

Fig. 4: Homomorphic Multiplication Throughput (log scale)

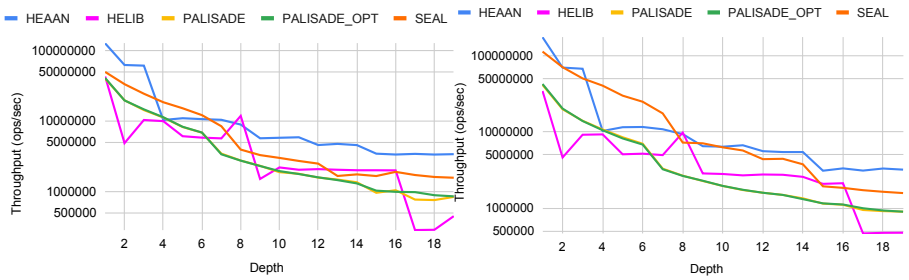


(a) AVX2 (b) AVX512

Fig. 5: Homomorphic Addition Runtimes

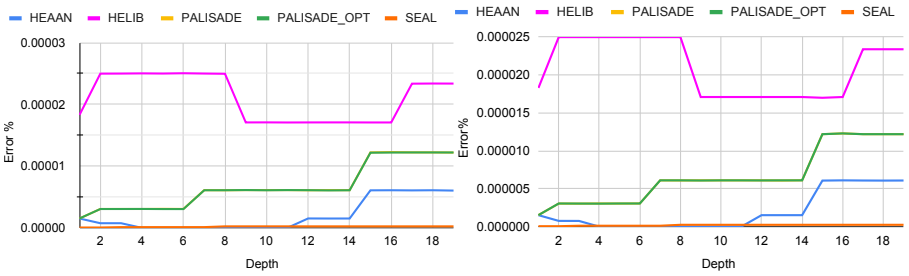
We show runtimes for homomorphic rotation in Figures 10a and 10b, and throughput in Figures 10b and 11a. Error is shown in Figures 12a and 12b. While performance is generally similar among libraries, HEAAN and SEAL again show the lowest error, making the best for rotation-heavy workloads. Runtimes for the relinearization step of homomorphic multiplication are shown in Figure 13. We observe that the runtimes for relinearization are very similar to that of homomorphic multiplication (see Figure 1), as relinearization is a dominating subprocedure of homomorphic multiplication.

In many applications such as machine learning (which we explore further in Section 6.8), performing ciphertext-plaintext addition or multiplication may be useful, though they generally contribute less noise and runtime to homomorphic computations than homomorphic (ciphertext-ciphertext) operations. We show runtimes for ciphertext-plaintext addition and multiplication in Figures 8 and 9. Without AVX512, SEAL and HEAAN show the best performance for addition, though PALISADE is generally comparable, and HElib is just as fast at lower-depth parameters. For multiplication, HEAAN's performance is poorer at lower depths; the other libraries generally have better runtimes. With AVX512, addition runtimes are similar to the case without AVX512, but SEAL now more clearly has the best latency for depths of 10 and higher.



(a) AVX2 (b) AVX512

Fig. 6: Homomorphic Addition Throughput (log scale)

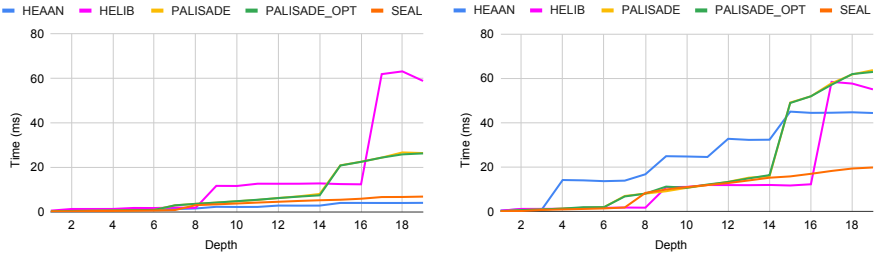


(a) AVX2 (b) AVX512

Fig. 7: Homomorphic Addition Error

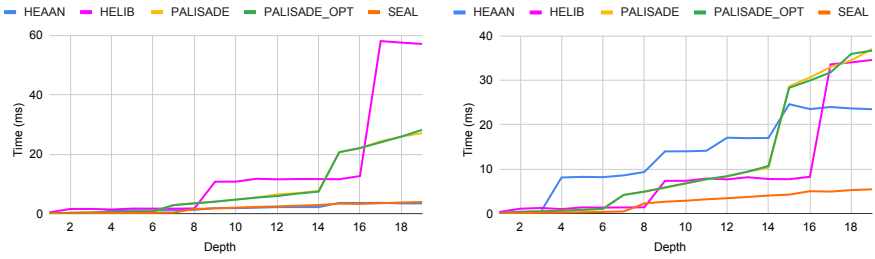
6.3.3 Parameter Sizes

Besides the computational factors of runtime, error, and throughput, the communication overhead and disk/memory footprint of FHE libraries is an important metric to manage. Figure 14 shows the ciphertext sizes of different libraries at differing parameter settings, indicating communication overhead. (We omitted the optimized PALISADE parameter selections for this test since the optimized parameters did not affect the sizes of PALISADE objects.) The FHE ciphertexts' sizes (measured in MB even at smaller depths) are generally similar for different libraries, though at higher depths differences become more starkly apparent. SEAL generally has the most consistent ciphertext growth, and has the smallest ciphertexts at higher depths. Figures 15a and 15b show how public and secret key sizes increase as depth increases. The public keys exhibit relatively consistent growth across libraries, with an exception being in the case of HElib at depths ≥ 16. For secret keys, HElib's keys are much larger than those of the other libraries, to the point where a log scale was necessary to represent them on the same graph. This is most likely due to HElib not including any kind of compression with their serialization or other strategies to reduce the size of serialized objects. PALISADE and SEAL perform key



(a) Addition (b) Multiplication

Fig. 8: Ciphertext-Plaintext homomorphic operations (AVX2)



(a) Addition (b) Multiplication

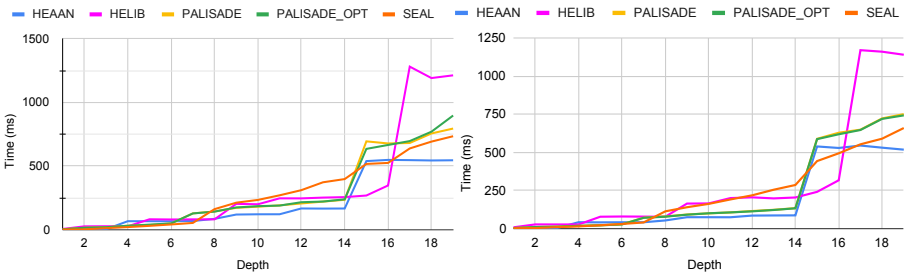
Fig. 9: Ciphertext-Plaintext homomorphic operations using SHE presets (AVX512)

generation from a single seed, so that the seed can be serialized instead of the entire polynomial that it generates.

While these keys are not frequently sent between parties in outsourced computations in the same way homomorphic ciphertexts would be, the overhead they incur may be of importance in bandwidth-limited scenarios. If communication overhead from keys is a concern (e.g., in a server that must read in many public keys for many clients' homomorphic computations), then HElib should be avoided, and SEAL or PALISADE should be preferred.

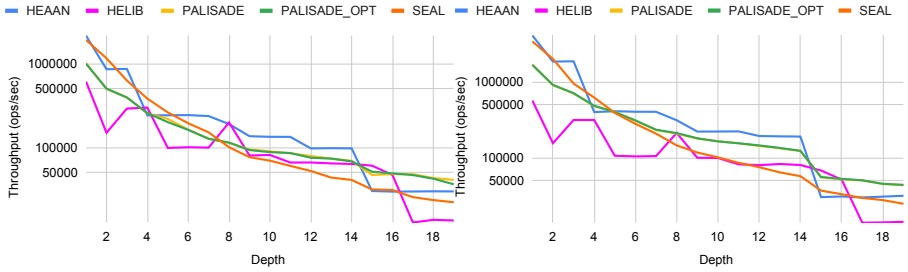
6.4 End-to-End Task Performance with Increasing Parameters

We examined the performance of FHE libraries on some simple end-to-end tests. To simulate the application-level tasks of HE, we implemented three end-to-end tests for each library: dot product, linear transformation, and polynomial evaluation. These three tests are some of the fundamental operations used by data-driven AI models used in privacy-preserving Machine Learning [51, 52].



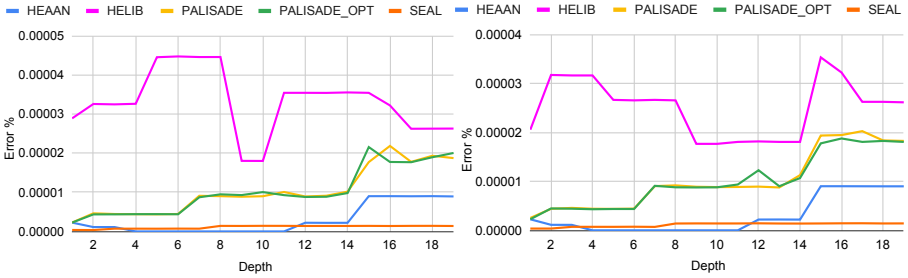
(a) AVX2 (b) AVX512

Fig. 10: Homomorphic Rotation Runtimes



(a) AVX2 (b) AVX512

Fig. 11: Homomorphic Rotation Throughput (log scale)

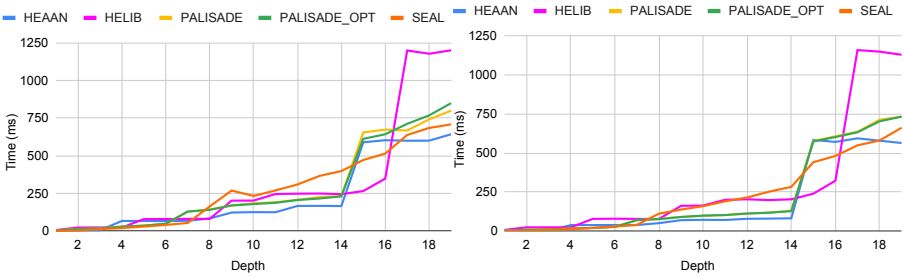


(a) AVX2 (b) AVX512

Fig. 12: Homomorphic Rotation Error

Other work [53] shows that using polynomial approximations of the activation functions in neural networks can achieve very high accuracy and performance metrics.

Some libraries have existing built-in methods for performing these operations. For instance, PALISADE has built-in methods for the polynomial evaluation and the dot product. HElib has its own implementation of linear transformation



(a) AVX2 (b) AVX512

Fig. 13: Homomorphic Relinearization Runtimes

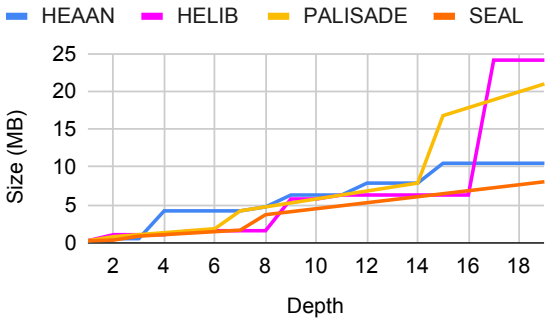
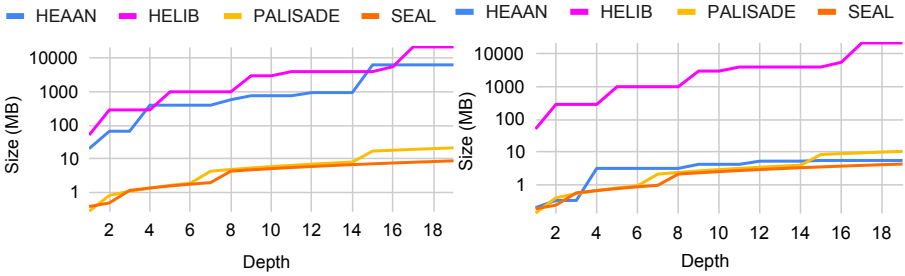


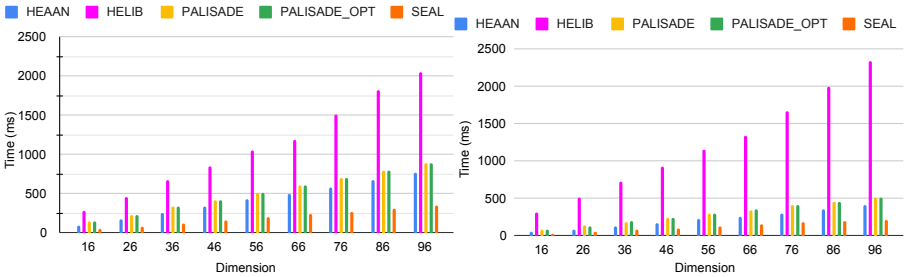
Fig. 14: Ciphertext Size with Increasing Depth



(a) Public Key (b) Secret Key

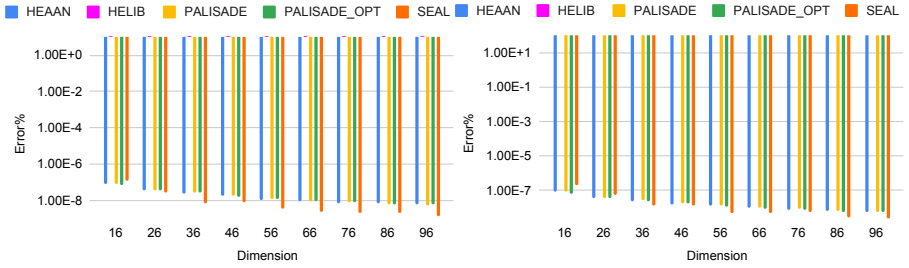
Fig. 15: Key Size with Increasing Depth (log scale, AVX2)

[44] and some replication-based algorithms described in [42]. SEAL and HEAAN do not include any built-in methods for matrix arithmetic operations. For a fair comparison, we implemented the end-to-end tasks for each library using generic algorithms found in the literature. We implement the same specific algorithm for each library for the three end-to-end tests using low-level scheme operations exposed by each library, such as multiplication, addition, and rotation.



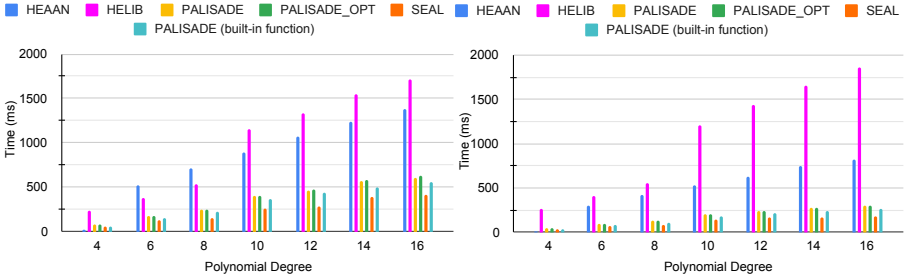
(a) AVX2 (b) AVX512

Fig. 16: Homomorphic Linear Transformation Runtimes



(a) AVX2 (b) AVX512

Fig. 17: Homomorphic Linear Transformation Error

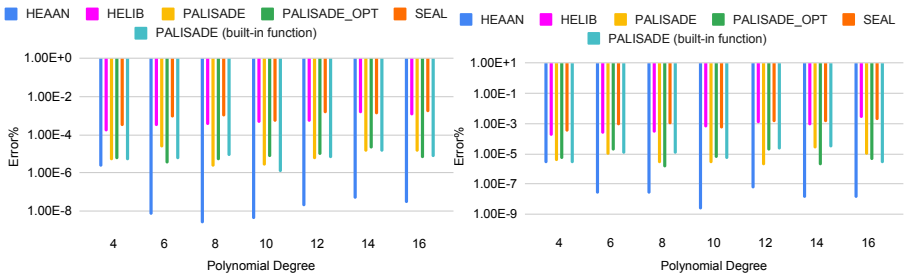


(a) AVX2 (b) AVX512

Fig. 18: Homomorphic Polynomial Evaluation Runtimes

6.4.1 Linear Transformation

For linear transformation, we used a method described in [42], where we pre-process the matrix into a diagonal order before multiplication. For simplicity, we only consider cases of square matrices of size $n \times n$, where n is an integer which is a power of 2. This method of linear transformation uses a depth of



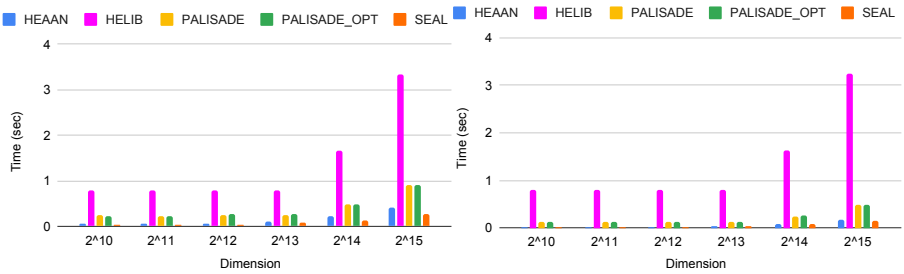
(a) AVX2 (b) AVX512

Fig. 19: Homomorphic Polynomial Evaluation Error

one multiplication and n rotations, multiplications, and additions. Runtimes for homomorphic linear transformation are shown in Figures 16a and 16b. We can observe that as the parameter sizes increase, SEAL generally has better performance for linear transformation. HEAAN and PALISADE show similar performance to each other, and at lower-depth parameter settings, this difference is even smaller. HElib has almost the same runtime on AVX2 and AVX512 but other libraries have better runtime on AVX512. The error in homomorphic linear transformation is shown in Figures 17a and 17b. Almost all libraries have similar errors for linear transformation for vectors up to a dimension of 96, except HElib, which is found to accumulate high errors during the computation. This is due to its higher amount of error incurred in homomorphic multiplication, homomorphic addition, and rotation (see Figures 2, 7a and 12a), which is accumulated during end-to-end tests. The strong performance of SEAL is most likely due to its good performance in both homomorphic addition and homomorphic multiplication (see Figures 1a and 5a).

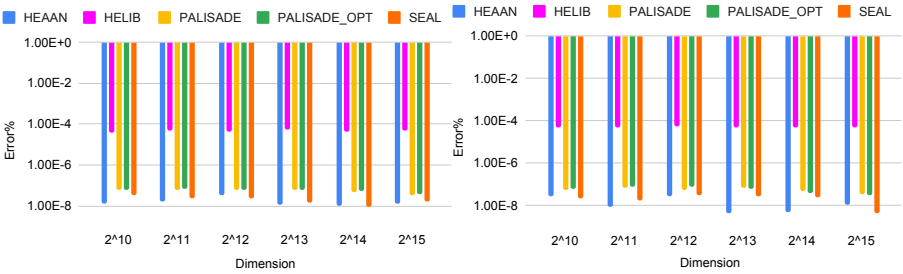
6.4.2 Dot Product

In the dot product end-to-end test, we computed a sum of products of an arbitrary number of inputs. The dot product operation over the ciphertexts is performed by performing homomorphic multiplications on matching pairs of ciphertexts (using batching), then rotating the slots of ciphertexts and summing up the slots repeatedly. Using this method, the number of rotations required is the logarithm of the dimension of the input vector. Runtimes for homomorphic dot product are shown in Figures 20a and 20b. HEAAN and SEAL are found to have almost identical runtimes and PALISADE follows up closely. The difference in the runtime of HElib and other libraries is found to be higher when we increase the dimension of the input vector. This difference in runtime for dot product between HElib and other libraries is similar for the AVX512 architecture as well. As this computation features relatively less homomorphic multiplication and more rotation (for which HEAAN and SEAL show the best performance at low depth), this results in HEAAN and SEAL showing the best performance. Errors in homomorphic dot product are shown



(a) AVX2 (b) AVX512

Fig. 20: Homomorphic Dot Product Runtimes



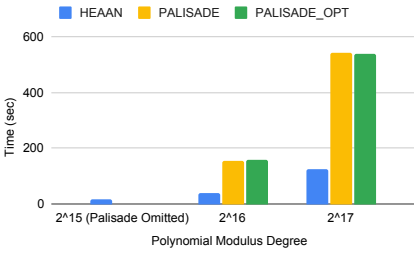
(a) AVX2 (b) AVX512

Fig. 21: Homomorphic Dot Product Error (log scale)

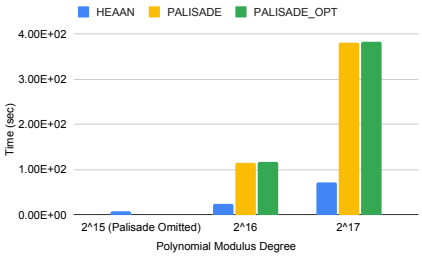
in Figures 21a and 21b. HEAAN, SEAL, and PALISADE all show low error, while HElib's is an order of magnitude higher.

6.4.3 Polynomial Evaluation

When testing homomorphic polynomial evaluation, we used the logistic function approximation found in [54] as a test function. We used the tree method of evaluating polynomials [55], that uses $log(D)$ multiplications and additions where D is the degree of the polynomial being evaluated. Figures 18a and 18b show the polynomial evaluation runtime for AVX2 and AVX512 respectively. SEAL and PALISADE have the best performance for this experiment, followed by HEAAN and HElib. Similar to other tests, we can see that all libraries except HElib perform better on the AVX512 architecture. Error in the homomorphic polynomial evaluation is shown in Figures 19a and 19b. HEAAN has the lowest error among the libraries in both the architectures, followed by PALISADE, HElib and SEAL. PALISADE and SEAL perform well due to their good performance in homomorphic multiplication and addition.

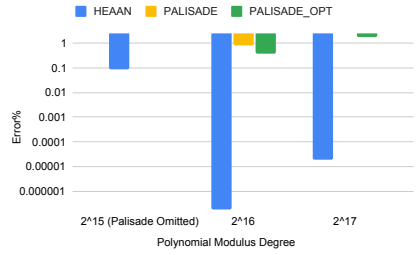


(a) AVX2

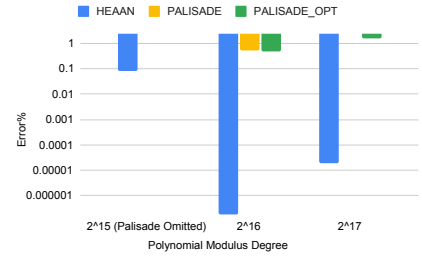


(b) AVX512

Fig. 22: Bootstrapping Latency (only HEAAN and PALISADE included for bootstrapping tests)



(a) AVX2

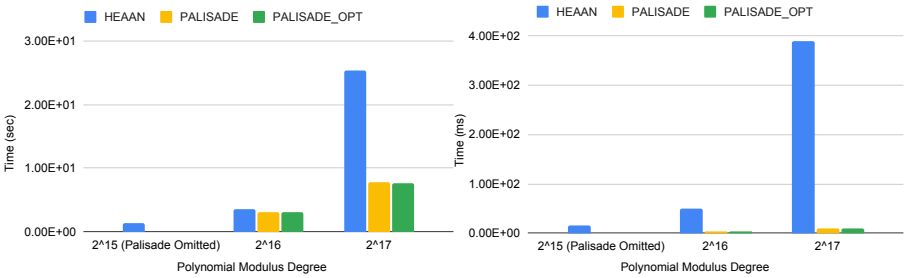


(b) AVX512

Fig. 23: Bootstrapping Error (log scale)

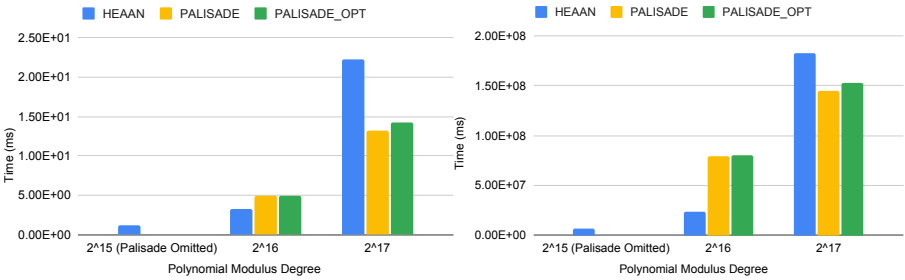
6.5 Bootstrapping

We tested the performance of the libraries using parameters conducive to bootstrapping. To the best of our knowledge, this is the first work that compares CKKS libraries on their bootstrapping capabilities, mostly due to the fact that most CKKS bootstrapping functionality is very newly released or not public. Figures 22a and 22b show the latency of bootstrapping for HEAAN and PALISADE (compared using the same polynomial modulus degree, so that latency directly indicates throughput). PALISADE did not accept $N < 2^{16}$ for bootstrapping parameters due to security reasons, so no measurements for that case are shown. We see that HEAAN has a much better performance than PALISADE for bootstrapping. Further, Figures 23a and 23b show that HEAAN's bootstrapping also has much lower error. This is likely due to the inclusion of advanced optimizations for bootstrapping noise in HEAAN [56]. We thus conclude that for applications that rely heavily on bootstrapping, HEAAN is a strong choice. This is especially true when recalling that HEAAN and PALISADE show similar performance for homomorphic multiplication (see Figures 1a, 1b, 4a and 4b). As these two operations are likely to dominate



(a) Addition (b) Multiplication

Fig. 24: Ciphertext-Plaintext homomorphic operations using FHE presets (AVX2)



(a) Addition (b) Multiplication

Fig. 25: Ciphertext-Plaintext homomorphic operations using FHE presets (AVX512)

runtime for most computations, using HEAAN for faster bootstrapping will reduce computation time. We show ciphertext modulus sizes for bootstrapping-capable parameter settings in Table 3[1]. In this, we see that HEAAN chooses ciphertext moduli conservatively, while PALISADE chooses its moduli aggressively to be smaller; this can reduce the memory footprint for PALISADE's ciphertexts. We also show latency for ciphertext-plaintext operations using bootstrapping-capable parameters in Figures 24 and 25. Again, PALISADE generally outperforms HEAAN for ciphertext-plaintext operations, though when AVX512 is enabled, HEAAN's latency is lower in some cases.

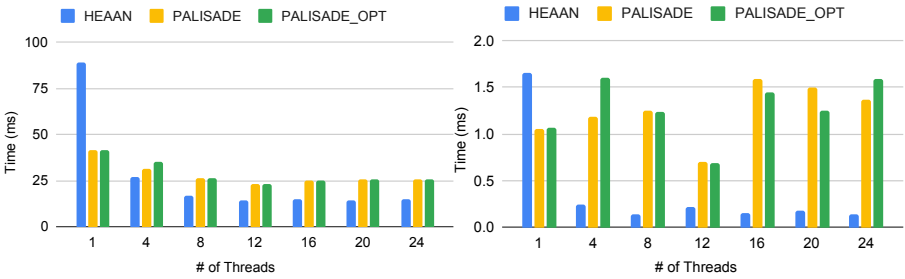
6.6 Impact of Multithreading

We examined the impact of CPU multithreading for the libraries we profile (without using other acceleration, e.g., AVX512 or GPU acceleration). SEAL and HElib were not profiled for these tests, as neither (currently) utilizes library-level multithreading (see Sections 4.1 and 4.3). Figure 26a shows how

[1]Parameters differed slightly but not significantly for AVX512.

Table 3: Bootstrapping CKKS parameters (AVX2)

Depth	PALISADE		HEAAN	
	$log_2(N)$	$log_2(p \cdot q)$	$log_2(N)$	$log_2(p \cdot q)$
5	-	-	15	777
6	16	301	16	1555
13	17	581	17	2070



(a) Homomorphic Multiplication (b) Homomorphic Addition

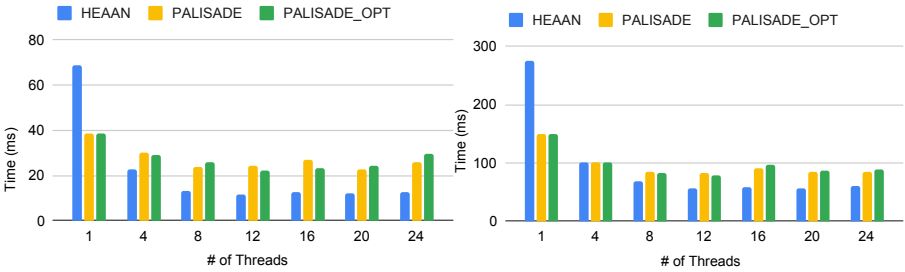
Fig. 26: Effect of Multithreading (AVX2)

increasing the number of threads can help reduce the runtime of homomorphic multiplication. HEAAN is able to most effectively utilize multiple threads. PALISADE is able to gain some modest improvements from multithreading, but does not match the performance of HEAAN. We see that PALISADE performs best with 12 threads used, and HEAAN's performance does not significantly improve when using more than 12 threads.

We also explore the effect of multithreading on other FHE operations. For homomorphic addition (shown in Figure 26b), we note that PALISADE again shows its best performance at 12 threads, but that using other numbers of threads may actually harm PALISADE's performance. HEAAN again shows excellent improvements from multithreading. For homomorphic rotation (shown in Figure 27a), HEAAN benefits the most and shows the best performance with more threads.

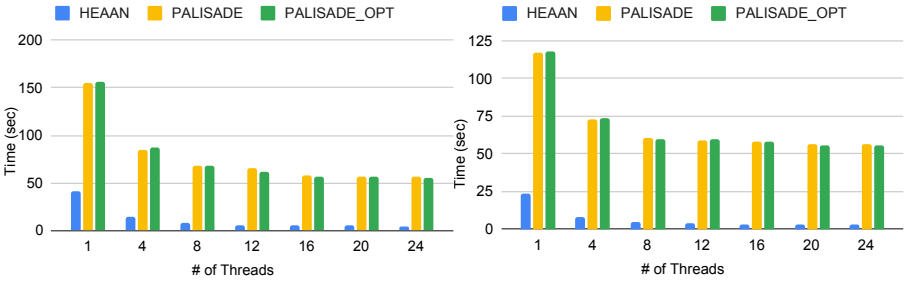
6.6.1 End-to-End Functions

Figures 29a, 29b and 30 show the performance difference in the end-to-end tests when the number of threads is increased. SEAL and PALISADE do not gain any significant level of performance increase while increasing the number of threads from 1 to 24 in all three tests. HEAAN gains a significant performance increase (decrease in runtime) while increasing the number of threads from 1 to 4. We see a further increase in performance when the number of threads is



(a) Homomorphic Rotation (b) Chained Homomorphic Multiplication

Fig. 27: Effect of Multithreading (AVX2)



(a) AVX2 (b) AVX512

Fig. 28: Multithreading impact on bootstrapping

increased to 8 but adding any additional threads after 8 do not significantly affect the performance in HEAAN.

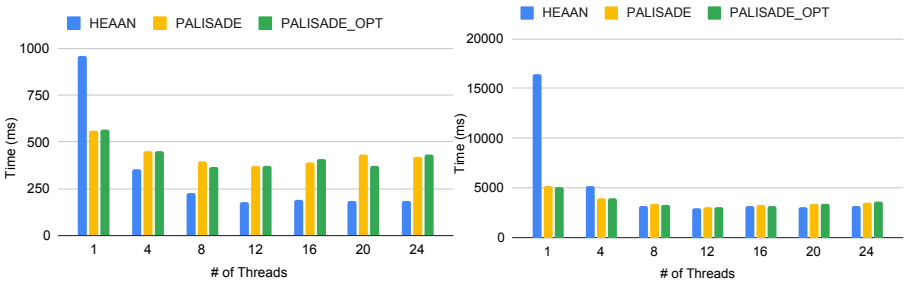
6.6.2 Bootstrapping

We also examine the impact of multithreading on bootstrapping, using parameter settings with $N = 2^{16}$. For a full discussion on bootstrapping performance, see Section 6.5. Regarding multithreading for bootstrapping, we see that HEAAN's relative speedup from an increased number of threads is even greater than that of PALISADE's, on top of the better base performance of HEAAN.

6.7 Impact of GPU Acceleration

Many other works have explored the use of GPU or other hardware (e.g., ASIC, FPGA, etc.) for accelerating homomorphic encryption [57–63]. GPUs are ubiquitously available in cloud computing clusters. As using GPUs on services such as AWS comes with additional cost, it is informative for users to understand the performance improvements gained from the use of a GPU.

HEAAN is the only library in this investigation that comes with GPU support. There have been various research works implementing GPU acceleration



(a) Homomorphic Dot Product (b) Homomorphic Linear Transformation

Fig. 29: Effect of Multithreading (AVX2)

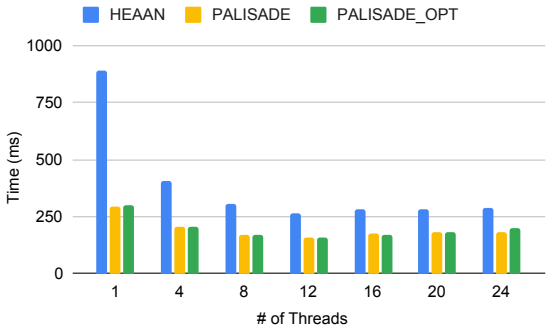
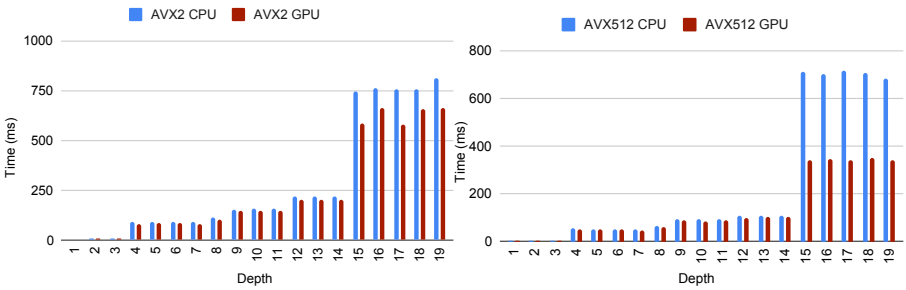


Fig. 30: Multithreading Homomorphic Polynomial Evaluation (AVX2)

for other libraries, but these changes are not usable in available versions of the libraries [49, 64] The most intensive and dominant operations in CKKS are homomorphic multiplication and bootstrapping; we profile these operations on GPU to see how GPU acceleration affects the overall performance of computations.

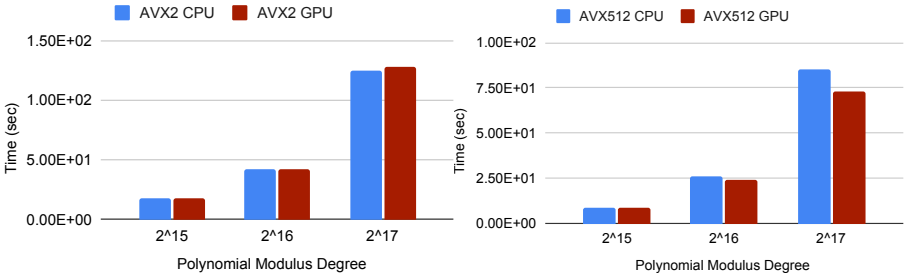
Figures 31a and 31b show the improvements for homomorphic multiplication brought by GPU acceleration. At smaller parameter sizes (up to a depth of 14), there is only a slight improvement. For larger depths, there is a more notable improvement for the AVX2 case (approximately up to 30%). In the case of depth ≥ 14 when using AVX512, the improvement is much larger - an improvement of about 2x is shown.

Figures 32a and 32b show the impact from GPU acceleration for bootstrapping. When using only AVX2, GPU acceleration does not bring significant benefits and is even slightly slower in the case of $N = 2^{17}$. The most likely reason for this is that the additional latency incurred when transferring data between the GPU and the CPU's memory hierarchy outweighs any performance improvements from GPU acceleration in this case. However, when AVX512 is used, a modest improvement in latency is seen for larger parameter settings.



(a) AVX2 (b) AVX512

Fig. 31: Homomorphic Multiplication Latency Comparison for HEAAN



(a) AVX2 (b) AVX512

Fig. 32: Bootstrapping Latency Comparison for HEAAN

From these tests, we can conclude that for multiplication-heavy workloads requiring larger parameter settings, GPU acceleration can improve runtimes by up to 2x. However, using GPUs may come with a significant increase in monetary cost. For example, GPU-capable AWS instances can be up to 3.16x more expensive than CPU-only counterparts [65, 66], forcing users to make a tradeoff between latency and cost. We also note that other research shows larger improvements from GPU (e.g., up to 60x faster for homomorphic multiplication [63]). This is most likely due to these papers reporting the timing of the actual computation, without accounting for the overhead of data transfer to/from the GPU.

6.8 Projections in Complex Applications

FHE can be used in applications including finance [67, 68], healthcare [69, 70], blockchain [71–73], cyber-physical systems [74–76], and education [77]. Machine learning, which is widely utilized in almost all of the aforementioned sectors, is also a highly useful application of FHE [77, 78]. FHE can be used to aid the fundamental operations occurring inside an ML pipeline, e.g., matrix multiplications, and in computing activation functions including the sigmoid

and ReLU functions to preserve the privacy of the data being used by the ML model within the pipeline. Logistic regression is a common ML technique used to make accurate predictions. However, with encrypted inputs or models, it is very difficult to achieve the same level of speed of prediction which can be achieved using unencrypted input or models [79, 80]. Masters et al. [67] explored the task of running the prediction of a generated logistic regression model using CKKS in HElib and they found that the prediction computation consisted of an inner product followed by an application of an approximated sigmoid function. Since an inner product of a vector is basically a dot product, using Figures 18a, 18b, 20a and 20b, we can conclude that SEAL is the most suitable library for this application given its low runtime on both dot-product and polynomial evaluation tasks. SEAL's low runtime in the linear transformation test (see Figures 16a and 16b) also makes it suitable for use in scenarios similar to neural networks, which contain a high number of matrix and vector multiplications within its layers during the training phase.

For applications such as genome sequencing and similar healthcare related applications, runtime may not be the most important factor to be considered when judging which library is best. In such cases, a higher precision (or lower error rate) provided by the library along with modest runtime would be of higher significance than higher runtime alone. Crawford et al. [69] observed that when approximating the parameters of a logistic regression model using HE, maintaining good accuracy in their approximation can be particularly challenging. They found that their approximation formula for performing logistic regression only yields valid results in settings where the number of records greatly exceeds the number of attributes. They later used a solution based on table lookup to implement a low-precision approximation of the functions. In similar applications, libraries such as HEAAN might be more suitable due to its low runtime and very low polynomial evaluation error in the polynomial approximation task (see Figures 19a and 19b).

Jang et al. [81] introduced an extended CKKS scheme called MatHEAAN that provides efficient matrix representations and operations and improved noise control. The scheme is specifically designed to work with recurrent neural networks. They designed a custom gated recurrent unit (GRU) [82] using MatHEAAN. Within the GRU, they perform various operations, including trainable function approximation, element-wise dot product, and matrix-vector multiplication and addition. Runtime and error for HEAAN and PALISADE are the best for all of these operations when performed homomorphically in our tests (see Figures 16a, 16b, 18a, 18b, 20a and 20b).

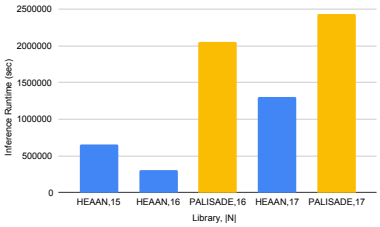
In this work, we primarily focused on low-level benchmarks of primitive CKKS operations and simple applications. To give a more complete view of the libraries for real-world use, we also studied the relative performance of bootstrapping-capable libraries for high-depth machine learning tasks. Specifically, we estimated the runtime of HEAAN and PALISADE for inference with the ResNet50, VGG16, and MobileNet convolutional neural networks [83–85]. Because the publicly available versions of SEAL and HElib did not include

Table 4: Bootstraps in deep CNNs

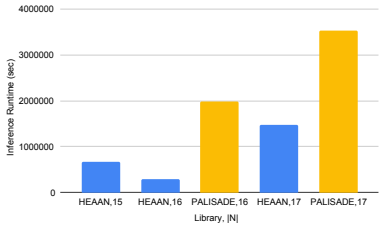
Network	Library	$\|N\|$	Depth	#. of Bootstraps
ResNet50	HEAAN	15	5	14976
	HEAAN & PALISADE	16	6	12992
	HEAAN & PALISADE	17	13	4416
VGG16	HEAAN	15	5	2816
	HEAAN & PALISADE	16	6	2048
	HEAAN & PALISADE	17	13	1280
MobileNet	HEAAN	15	5	15168
	HEAAN & PALISADE	16	6	12544
	HEAAN & PALISADE	17	13	6464

CKKS bootstrapping, we omit them from this portion of the study, and also note that those libraries may thus not be suitable for high-depth computations such as CNN inference. In this, we replace the max-pooling layers with average-pooling layers. Average-pooling is much easier to compute homomorphically than max-pooling; using average-pooling for FHE-based deep learning is common and has minimal loss of accuracy [86–91]. We used a degree-6 polynomial approximation of a ReLU activation function. While low-degree approximations (as low as degree-2) are possible and more efficient, using these may lead to high computation error [92, 93]. Our estimate assumes that bootstrapping is done when necessary and not before. We used the same parameter settings as our tests in Section 6.5, giving polynomial modulus degrees and multiplicative depth of $(N, d) \in \{(2^{15}, 5), (2^{16}, 6), (2^{17}, 13)\}$. A listing of the number of bootstraps used in each CNN is given in Table 4.

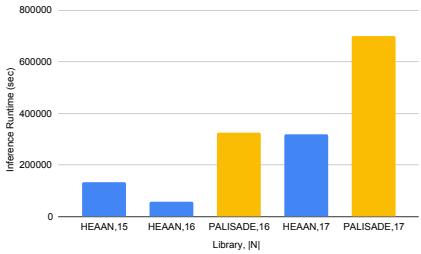
Our estimated results are shown in Figure 33. We note that this is only a very simple runtime performance estimate, which does not take into account many real-world factors such as accuracy, memory consumption, applying parallelism, or programming optimizations. (There does exist other work that considers the noise incurred when performing deep learning homomorphically [80, 94]). Interestingly, we noted that doubling N from 2^{15} to 2^{16} improved performance. This is most likely due to fewer bootstraps needed because of the increase in multiplicative depth from 5 to 6 allowed by the larger parameters. However, this trend did not continue when $N = 2^{17}$; at this point the larger polynomial modulus degree incurs much more overhead in polynomial multiplication to the point that the reduction in bootstraps does not reduce runtime. HEAAN's performance is better than PALISADE's in all cases. We note that when accounting for batched throughput (assuming fully-packed ciphertexts), HEAAN at $|N| = 16$ still shows the best performance, and both HEAAN and PALISADE show better performance at $|N| = 17$ than PALISADE does at $|N| = 16$ for ResNet50 and MobileNet. This is shown in Figure 34.



(a) MobileNet

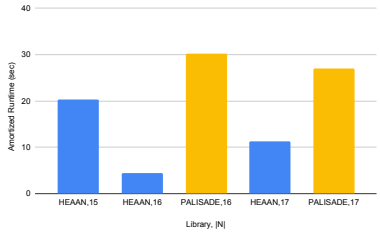


(b) ResNet50

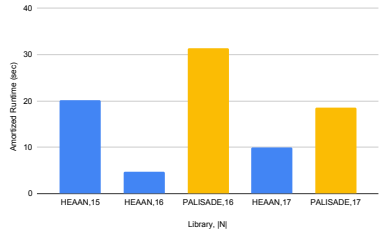


(c) VGG16

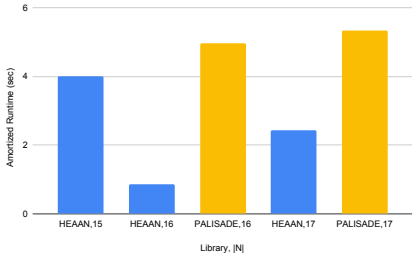
Fig. 33: CNN Inference Runtimes for FHE using PALISADE and HEAAN



(a) MobileNet



(b) ResNet50



(c) VGG16

Fig. 34: CNN Inference Runtimes for FHE using PALISADE and HEAAN, Amortized to Full Batchsize of $N/2$

6.9 Summary: Which Library is Best for Which Tasks?

For workloads of relatively low depth that do not require bootstrapping dominated by homomorphic multiplication, SEAL is the best due to its low latency (though PALISADE and HEAAN also show good runtimes) and smaller error. This holds whether the computation is dominated by homomorphic multiplication, homomorphic addition, or rotation. HEAAN also shows good performance for addition with smaller parameter settings, due in part to the addition of a parameter preset added at our suggestion. HEAAN and PALISADE also perform well for rotation. HEAAN, SEAL, and PALISADE all perform comparably for ciphertext-plaintext operations.

In terms of communication and memory footprint, all four libraries showed similar ciphertext sizes. However, PALISADE and SEAL showed the smallest public key sizes, and HElib's secret key sizes were the largest. In a scenario where a party needs to store a lot of public keys (e.g., oursourced homomorphic evaluation as a service for many different users), PALISADE or SEAL may be preferable to save on the cloud's use of memory and communication bandwidth.

In our simpler end-to-end tests, we found that SEAL again performed well – this is an intuitive conclusion, as SEAL's latency and error were relatively good in tests of primitive operations at low depth. For the application of polynomial evaluation, SEAL and PALISADE showed the best latency; however, HEAAN was slower but yielded much smaller error, making it useful for applications where accuracy is critical. SEAL and HEAAN show the best runtimes for computing dot products. SEAL shows the best runtime and error for our linear transformation tests, though HEAAN and PALISADE are comparable. We thus conclude that for most SHE tasks, SEAL will be the best to use. However, SEAL (like HElib) does not include CKKS bootstrapping in public versions of their library, so it is unsuitable for applications with high multiplicative depth.

For the procedure of bootstrapping, HEAAN's latency and noise were both better than that of PALISADE; this is likely due at least in part to advanced optimizations implemented in HEAAN [56]. However, bootstrappable ciphertexts for HEAAN will be larger than those of PALISADE for the same N and allotted depth; PALISADE chooses its ciphertext moduli more aggressively, reducing the memory footprint of its ciphertexts relative to HEAAN.

HEAAN makes the most relative gains from the use of multiple threads, showing much more significant speedups. HEAAN also showcases how GPU acceleration brings a large improvement for homomorphic multiplication at larger parameter settings when AVX512 is used. However, in many cases GPU acceleration did not show a significant improvement, and even may slow down computation slighly; this is most likely due to the overhead of data transfer. Throughout our testing, we note that using AVX512 can bring modest improvements to runtime.

For the application of deep learning with convolutional neural networks, we note that HEAAN has the best runtime, though we did not consider other aspects such as error in our estimation. This is due to HEAAN's strong performance on the critical operations of bootstrapping and homomorphic

multiplication. Interestingly, in some cases increasing parameter sizes can actually improve performance on CNNs due to the fewer bootstraps required. We also note that neither HElib nor SEAL implements CKKS bootstrapping, making those libraries unsuitable for deep CNNs including any significant number of activation functions to be computed.

For tasks requiring functionality beyond what CKKS alone is well-suited for (e.g., scheme switching CKKS ciphertexts to TFHE ciphertexts [95]), PALISADE is clearly the best choice. This may be important to users who have need of functionality besides only approximate encrypted computations. For example, some protocols may also need functionality such as threshold homomorphic encryption [96].

7 Our Perspectives on Development with CKKS Libraries

In this section, we discuss some of our subjective observations in the process of developing HEProfiler. All of the libraries we profiled featured completeness and fairly good usability, and any of them would probably be suitable for general usage.

7.1 Installation

Installing SEAL and PALISADE was generally easy. Compilation with SEAL was the easiest, as only a single header and static library file are needed. Compilation with PALISADE is slightly more difficult, as many different headers and libraries are available, and it is not always obvious which must be included when compiling. However, this can allow compilation time and binary size to be reduced, as the user can specify only the headers and libraries needed.

HElib offers two installation options: a system-wide installation or a "packed" installation bundling the multiprecision libraries GMP and NTL. HElib also requires manually installing and linking to Intel HEXL for use of AVX512.

HEAAN's headers and shared library binaries were provided to us by CryptoLab. This saved us the trouble of building them ourselves, but introduced additional difficulty because the binaries were built for specific environments. This necessitated the use of containerization via Singularity.

7.2 General Programming

Programming with PALISADE was more difficult at the time of parameter selection, but fairly easy for computations. PALISADE offered the most customization and control of any library, but did not always warn the user of settings or operations that would lead to a failure.

Programming with SEAL was made somewhat tedious by the use of a different object for each set of functionality, i.e., keeping separate objects for encryption, decryption, encoding/decoding, homomorphic operation, each type of key, et cetera. Besides this, using SEAL is generally easy and straightforward,

though some functionality that can be automated and/or done by default is not (e.g., relinearization after multiplication).

HElib is very easy for non-experts to program with, as it overloads arithmetic operators. HElib requires objects for only a context, public key, and secret key, making programming simple. Operations to reduce noise level in ciphertexts like rescaling and relinearlization are automatic in HElib.

Like SEAL, HEAAN uses several different objects for its functionalities. There are some minor confusing points, such as the inclusion of both `Plaintext` and `Message` objects. Besides this, the interface is generally simple and easy to use, though more limited than that of PALISADE.

8 Conclusion

In this work, we present HEProfiler, an extensible profiling framework for CKKS libraries. We collaborated with library maintainers to find the most fair way to choose parameters for each library. We performed an experimental evaluation that is more complete than previous profiling works have considered. Our experiments are thorough both in the scope of the tasks considered, ranging from low-level primitives to high-level end-to-end tasks, and also complete in the metrics measured (latency, throughput, error, memory consumption). Our experiments with the bootstrapping capabilities of HEAAN and PALISADE are the first experiments that comparatively evaluate CKKS bootstrapping. We also provided discussion on the libraries and their features, focusing on the features and differences most relevant to developers and researchers.

We conclude that for most tasks of limited depth, Microsoft SEAL would be preferred. For high-depth computation requiring bootstrapping (e.g., machine learning), HEAAN is the best library. For general functionality, parameter customization, and features and schemes beyond CKKS, PALISADE is clearly the best choice. HElib's performance metrics are not as good as those of the other libraries, but its thorough and highly technical documentation makes it useful for researchers.

Acknowledgements

The authors would like to thank the maintainers of each of PALISADE, Microsoft SEAL, HElib, and HEAAN for their valuable insights. In particular, we thank:

- (Duality Technologies) Yuriy Polyakov, Kurt Rohloff, and David Cousins for their assistance with PALISADE
- (Microsoft) Kim Laine and Wei Dai for their assistance with Microsoft SEAL
- (Intel Corp.) Flavio Bergamaschi for his assistance with HElib
- (CryptoLab) Jung Hee Cheon, Junbum Shin, Taekyung Kim, Sumin Lee, and Minje Park for their assistance with HEAAN

Declarations

The authors did not receive financial support from any organization for this work. The authors have no competing interests to declare that are relevant to the content of this article.

References

[1] Cheon, J.H., Kim, A., Kim, M., Song, Y.: Homomorphic encryption for arithmetic of approximate numbers. In: International Conference on the Theory and Application of Cryptology and Information Security, pp. 409–437 (2017). Springer

[2] Martins, P., Sousa, L., Mariano, A.: A survey on fully homomorphic encryption: An engineering perspective. ACM Computing Surveys (CSUR) **50**(6), 1–33 (2017)

[3] Gentry, C.: Fully homomorphic encryption using ideal lattices. In: Proceedings of the Forty-first Annual ACM Symposium on Theory of Computing, pp. 169–178 (2009)

[4] Zheng, W., Wu, Y., Wu, X., Feng, C., Sui, Y., Luo, X., Zhou, Y.: A survey of intel sgx and its applications. Frontiers of Computer Science **15**(3), 1–15 (2021)

[5] SEV-SNP, A.: Strengthening vm isolation with integrity protection and more. White Paper, January (2020)

[6] Cramer, R., Damgård, I.B., *et al.*: Secure Multiparty Computation. Cambridge University Press, ??? (2015)

[7] Takeshita, J., Karl, R., Gong, T., Jung, T.: Slap: Simple lattice-based private stream aggregation protocol. Cryptology ePrint Archive (2020)

[8] Gouert, C., Mouris, D., Tsoutsos, N.G.: New insights into fully homomorphic encryption libraries via standardized benchmarks. Cryptology ePrint Archive (2022)

[9] Doan, T.V.T., Messai, M.-L., Gavin, G., Darmont, J.: A survey on implementations of homomorphic encryption schemes. The Journal of Supercomputing **79**(13), 15098–15139 (2023)

[10] Jiang, L., Ju, L.: Fhebench: Benchmarking fully homomorphic encryption schemes. arXiv preprint arXiv:2203.00728 (2022)

[11] Varia, M., Yakoubov, S., Yang, Y.: Hetest: A homomorphic encryption testing framework. In: International Conference on Financial Cryptography

and Data Security, pp. 213–230 (2015). Springer

[12] Melchor, C.A., Kilijian, M.-O., Lefebvre, C., Ricosset, T.: A comparison of the homomorphic encryption libraries helib, seal and fv-nfllib. In: International Conference on Security for Information Technology and Communications, pp. 425–442 (2018). Springer

[13] Fawaz, S.M., Belal, N., ElRefaey, A., Fakhr, M.W.: A comparative study of homomorphic encryption schemes using microsoft seal. In: Journal of Physics: Conference Series, vol. 2128, p. 012021 (2021). IOP Publishing

[14] Sathya, S.S., Vepakomma, P., Raskar, R., Ramachandra, R., Bhattacharya, S.: A review of homomorphic encryption libraries for secure computation. arXiv preprint arXiv:1812.02428 (2018)

[15] Dordevic, G., Vuletić, P.V.: Performance comparison of homomorphic encryption scheme implementations

[16] Gorantala, S., Springer, R., Purser-Haskell, S., Lam, W., Wilson, R., Ali, A., Astor, E.P., Zukerman, I., Ruth, S., Dibak, C., et al.: A general purpose transpiler for fully homomorphic encryption. arXiv preprint arXiv:2106.07893 (2021)

[17] Intel: Homomorphic Encryption Benchmarking Framework – HEBench (2021). https://hebench.org/

[18] Zhu, H., Suzuki, T., Yamana, H.: Performance comparison of homomorphic encrypted convolutional neural network inference among helib, microsoft seal and openfhe. In: 2023 IEEE Asia-Pacific Conference on Computer Science and Data Engineering (CSDE), pp. 1–7 (2023). IEEE

[19] Rivest, R.L., Shamir, A., Adleman, L.: A method for obtaining digital signatures and public-key cryptosystems. Communications of the ACM **21**(2), 120–126 (1978)

[20] Goldwasser, S., Micali, S.: Probabilistic encryption & how to play mental poker keeping secret all partial information. In: Providing Sound Foundations for Cryptography: on the Work of Shafi Goldwasser and Silvio Micali, pp. 173–201 (2019)

[21] ElGamal, T.: A public key cryptosystem and a signature scheme based on discrete logarithms. IEEE transactions on information theory **31**(4), 469–472 (1985)

[22] Benaloh, J.: Dense probabilistic encryption. In: Proceedings of the Workshop on Selected Areas of Cryptography, pp. 120–128 (1994)

[23] Okamoto, T., Uchiyama, S.: A new public-key cryptosystem as secure as factoring. In: International Conference on the Theory and Applications of Cryptographic Techniques, pp. 308–318 (1998). Springer

[24] Naccache, D., Stern, J.: A new public key cryptosystem based on higher residues. In: Proceedings of the 5th ACM Conference on Computer and Communications Security, pp. 59–66 (1998)

[25] Paillier, P.: Public-key cryptosystems based on composite degree residuosity classes. In: International Conference on the Theory and Applications of Cryptographic Techniques, pp. 223–238 (1999). Springer

[26] Sander, T., Young, A., Yung, M.: Non-interactive cryptocomputing for nc/sup 1. In: 40th Annual Symposium on Foundations of Computer Science (Cat. No. 99CB37039), pp. 554–566 (1999). IEEE

[27] Boneh, D., Goh, E.-J., Nissim, K.: Evaluating 2-dnf formulas on ciphertexts. In: Theory of Cryptography Conference, pp. 325–341 (2005). Springer

[28] Gentry, C.: Computing arbitrary functions of encrypted data. Communications of the ACM **53**(3), 97–105 (2010)

[29] Brakerski, Z., Gentry, C., Vaikuntanathan, V.: (leveled) fully homomorphic encryption without bootstrapping. ACM Transactions on Computation Theory (TOCT) **6**(3), 1–36 (2014)

[30] Fan, J., Vercauteren, F.: Somewhat practical fully homomorphic encryption. IACR Cryptol. ePrint Arch. **2012**, 144 (2012)

[31] Chillotti, I., Gama, N., Georgieva, M., Izabachène, M.: TFHE: fast fully homomorphic encryption over the torus. Journal of Cryptology **33**(1), 34–91 (2020)

[32] Ducas, L., Micciancio, D.: Fhew: bootstrapping homomorphic encryption in less than a second. In: Annual International Conference on the Theory and Applications of Cryptographic Techniques, pp. 617–640 (2015). Springer

[33] Li, B., Micciancio, D.: On the security of homomorphic encryption on approximate numbers. In: Annual International Conference on the Theory and Applications of Cryptographic Techniques, pp. 648–677 (2021). Springer

[34] Cheon, J.H., Hong, S., Kim, D.: Remark on the security of ckks scheme in practice. Cryptology ePrint Archive (2020)

[35] Boemer, F., Kim, S., Seifu, G., DM de Souza, F., Gopal, V.: Intel hexl:

Accelerating homomorphic encryption with intel avx512-ifma52. In: Proceedings of the 9th on Workshop on Encrypted Computing & Applied Homomorphic Cryptography, pp. 57–62 (2021)

[36] Chillotti, I., Joye, M., Ligier, D., Orfila, J.-B., Tap, S.: Concrete: Concrete operates on ciphertexts rapidly by extending TFHE. In: WAHC 2020–8th Workshop on Encrypted Computing & Applied Homomorphic Cryptography, vol. 15 (2020)

[37] Mouchet, C., Bossuat, J.-P., Troncoso-Pastoriza, J., Hubaux, J.: Lattigo: A multiparty homomorphic encryption library in go. In: WAHC 2020–8th Workshop on Encrypted Computing & Applied Homomorphic Cryptography, vol. 15 (2020)

[38] Pal, S., Swaminathan, K., Aharoni, E., Kushnir, E., Drucker, N., Schaul, H., Buyuktosunoglu, A., Soceanu, O., Bose, P.: Fully homomorphic encryption for computer architects: A fundamental characterization study. In: Annual IEEE/ACM International Symposium on Microarchitecture (2023)

[39] Al Badawi, A., Bates, J., Bergamaschi, F., Cousins, D.B., Erabelli, S., Genise, N., Halevi, S., Hunt, H., Kim, A., Lee, Y., et al.: Openfhe: Open-source fully homomorphic encryption library. Cryptology ePrint Archive (2022)

[40] Wang, W., Jiang, Y., Shen, Q., Huang, W., Chen, H., Wang, S., Wang, X., Tang, H., Chen, K., Lauter, K., et al.: Toward scalable fully homomorphic encryption through light trusted computing assistance. arXiv preprint arXiv:1905.07766 (2019)

[41] Brakerski, Z., Vaikuntanathan, V.: Fully homomorphic encryption from ring-LWE and security for key dependent messages. In: Annual Cryptology Conference, pp. 505–524 (2011). Springer

[42] Halevi, S., Shoup, V.: Algorithms in helib. In: Annual Cryptology Conference, pp. 554–571 (2014). Springer

[43] Bajard, J.-C., Eynard, J., Hasan, M.A., Zucca, V.: A full RNS variant of FV like somewhat homomorphic encryption schemes. In: International Conference on Selected Areas in Cryptography, pp. 423–442 (2016). Springer

[44] Halevi, S., Shoup, V.: Faster homomorphic linear transformations in helib. In: Annual International Cryptology Conference, pp. 93–120 (2018). Springer

[45] Li, B., Micciancio, D., Schultz, M., Sorrell, J.: Securing approximate homomorphic encryption using differential privacy. In: Annual International

Cryptology Conference, pp. 560–589 (2022). Springer

[46] SEAL, M.: Noise flooding - Microsoft SEAL (2019). https://github.com/microsoft/SEAL/issues/89

[47] Chen, H., Laine, K., Rindal, P.: Fast private set intersection from homomorphic encryption. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, pp. 1243–1255 (2017)

[48] Lou, S., Agrawal, G.: A programming api implementation for secure data analytics applications with homomorphic encryption on gpus. In: 2021 IEEE 28th International Conference on High Performance Computing, Data, and Analytics (HiPC), pp. 418–423 (2021). IEEE

[49] Özerk, Ö., Elgezen, C., Mert, A.C., Öztürk, E., Savaş, E.: Efficient number theoretic transform implementation on gpu for homomorphic encryption. The Journal of Supercomputing **78**(2), 2840–2872 (2022)

[50] Kurtzer, G.M., Sochat, V., Bauer, M.W.: Singularity: Scientific containers for mobility of compute. PloS one **12**(5), 0177459 (2017)

[51] Bergamaschi, F., Halevi, S., Halevi, T.T., Hunt, H.: Homomorphic training of 30,000 logistic regression models. In: International Conference on Applied Cryptography and Network Security, pp. 592–611 (2019). Springer

[52] Kim, M., Song, Y., Wang, S., Xia, Y., Jiang, X., *et al.*: Secure logistic regression based on homomorphic encryption: Design and evaluation. JMIR medical informatics **6**(2), 8805 (2018)

[53] Hesamifard, E., Takabi, H., Ghasemi, M., Wright, R.N.: Privacy-preserving machine learning as a service. Proc. Priv. Enhancing Technol. **2018**(3), 123–142 (2018)

[54] Kim, A., Song, Y., Kim, M., Lee, K., Cheon, J.H.: Logistic regression model training based on the approximate homomorphic encryption. BMC medical genomics **11**(4), 23–31 (2018)

[55] Chen, H.: Algorithms in SEAL (2020). https://github.com/haochenuw/algorithms-in-SEAL/

[56] Bae, Y., Cheon, J.H., Cho, W., Kim, J., Kim, T.: Meta-bts: Bootstrapping precision beyond the limit. Cryptology ePrint Archive (2022)

[57] Dai, W., Sunar, B.: cuhe: A homomorphic encryption accelerator library. In: Pasalic, E., Knudsen, L.R. (eds.) Cryptography and Information Security in the Balkans, pp. 169–186. Springer, Cham (2016)

[58] Morshed, T., Al Aziz, M.M., Mohammed, N.: CPU and GPU accelerated

fully homomorphic encryption. In: 2020 IEEE International Symposium on Hardware Oriented Security and Trust (HOST), pp. 142–153 (2020). IEEE

[59] Al Badawi, A., Veeravalli, B., Mun, C.F., Aung, K.M.M.: High-performance fv somewhat homomorphic encryption on gpus: An implementation using cuda. IACR CHES, 70–95 (2018)

[60] Badawi, A.A., Chao, J., Lin, J., Mun, C.F., Jie, S.J., Tan, B.H.M., Nan, X., Aung, K.M.M., Chandrasekhar, V.R.: The alexnet moment for homomorphic encryption: Hcnn, the first homomorphic cnn on encrypted data with gpus. arXiv preprint arXiv:1811.00778 (2018)

[61] Shivdikar, K., Jonatan, G., Mora, E., Livesay, N., Agrawal, R., Joshi, A., Abellan, J., Kim, J., Kaeli, D.: Accelerating polynomial multiplication for homomorphic encryption on gpus. arXiv preprint arXiv:2209.01290 (2022)

[62] Takeshita, J., Reis, D., Gong, T., Niemier, M., Hu, X.S., Jung, T.: Algorithmic acceleration of b/fv-like somewhat homomorphic encryption for compute-enabled ram. In: International Conference on Selected Areas in Cryptography, pp. 66–89 (2020). Springer

[63] Şah Özcan, A., Ayduman, C., Türkoğlu, E.R., Savaş, E.: Homomorphic Encryption on GPU. Cryptology ePrint Archive, Paper 2022/1222. https://eprint.iacr.org/2022/1222 (2022). https://eprint.iacr.org/2022/1222

[64] Diallo, M.H., August, M., Hallman, R., Kline, M., Au, H., Beach, V.: Nomad: A framework for developing mission-critical cloud-based applications. In: 2015 10th International Conference on Availability, Reliability and Security, pp. 660–669 (2015). IEEE

[65] Inc., A.: Amazon EC2 On-Demand Pricing. https://aws.amazon.com/ec2/pricing/on-demand/

[66] Supalov, V.: How Much Will K8s GPU Tinkering Cost? (AWS Edition) (2020). https://deploying.ai/aws-k8s-gpu-tinkering-cost/

[67] Masters, O., Hunt, H., Steffinlongo, E., Crawford, J., Bergamaschi, F., Rosa, M.E.D., Quini, C.C., Alves, C.T., de Souza, F., Ferreira, D.G.: Towards a homomorphic machine learning big data pipeline for the financial services sector. Cryptology ePrint Archive (2019)

[68] Peng, H.-T., Hsu, W.W., Ho, J.-M., Yu, M.-R.: Homomorphic encryption application on financialcloud framework. In: 2016 IEEE Symposium Series on Computational Intelligence (SSCI), pp. 1–5 (2016). IEEE

[69] Crawford, J.L., Gentry, C., Halevi, S., Platt, D., Shoup, V.: Doing real work with fhe: the case of logistic regression. In: Proceedings of the 6th Workshop

on Encrypted Computing & Applied Homomorphic Cryptography, pp. 1–12 (2018)

[70] Kocabas, O., Soyata, T.: Towards privacy-preserving medical cloud computing using homomorphic encryption. In: Virtual and Mobile Healthcare: Breakthroughs in Research and Practice, pp. 93–125. IGI Global, ??? (2020)

[71] Liang, W., Zhang, D., Lei, X., Tang, M., Li, K.-C., Zomaya, A.Y.: Circuit copyright blockchain: blockchain-based homomorphic encryption for ip circuit protection. IEEE Transactions on Emerging Topics in Computing **9**(3), 1410–1420 (2020)

[72] Yan, X., Wu, Q., Sun, Y.: A homomorphic encryption and privacy protection method based on blockchain and edge computing. Wireless Communications and Mobile Computing **2020** (2020)

[73] Loukil, F., Ghedira-Guegan, C., Boukadi, K., Benharkat, A.-N.: Privacy-preserving iot data aggregation based on blockchain and homomorphic encryption. Sensors **21**(7), 2452 (2021)

[74] Li, F., Luo, B., Liu, P.: Secure information aggregation for smart grids using homomorphic encryption. In: 2010 First IEEE International Conference on Smart Grid Communications, pp. 327–332 (2010). IEEE

[75] He, X., Pun, M.-O., Kuo, C.-C.J.: Secure and efficient cryptosystem for smart grid using homomorphic encryption. In: 2012 IEEE PES Innovative Smart Grid Technologies (ISGT), pp. 1–8 (2012). IEEE

[76] Bos, J.W., Castryck, W., Iliashenko, I., Vercauteren, F.: Privacy-friendly forecasting for the smart grid using homomorphic encryption and the group method of data handling. In: International Conference on Cryptology in Africa, pp. 184–201 (2017). Springer

[77] Archer, D., Chen, L., Cheon, J.H., Gilad-Bachrach, R., Hallman, R.A., Huang, Z., Jiang, X., Kumaresan, R., Malin, B.A., Sofia, H., *et al.*: Applications of homomorphic encryption. In: Crypto Standardization Workshop, Microsoft Research, vol. 14 (2017). sn

[78] Drozdowski, P., Buchmann, N., Rathgeb, C., Margraf, M., Busch, C.: On the application of homomorphic encryption to face identification. In: 2019 International Conference of the Biometrics Special Interest Group (biosig), pp. 1–5 (2019). IEEE

[79] Xu, R., Joshi, J.B., Li, C.: Cryptonn: Training neural networks over encrypted data. In: 2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS), pp. 1199–1209 (2019). IEEE

[80] Gilad-Bachrach, R., Dowlin, N., Laine, K., Lauter, K., Naehrig, M., Wernsing, J.: Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy. In: International Conference on Machine Learning, pp. 201–210 (2016). PMLR

[81] Jang, J., Lee, Y., Kim, A., Na, B., Yhee, D., Lee, B., Cheon, J.H., Yoon, S.: Privacy-preserving deep sequential model with matrix homomorphic encryption. In: Proceedings of the 2022 ACM on Asia Conference on Computer and Communications Security, pp. 377–391 (2022)

[82] Cho, K., Van Merriënboer, B., Bahdanau, D., Bengio, Y.: On the properties of neural machine translation: Encoder-decoder approaches. arXiv preprint arXiv:1409.1259 (2014)

[83] He, K., Zhang, X., Ren, S., Sun, J.: Deep residual learning for image recognition. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 770–778 (2016)

[84] Zhang, X., Zou, J., He, K., Sun, J.: Accelerating very deep convolutional networks for classification and detection. IEEE transactions on pattern analysis and machine intelligence **38**(10), 1943–1955 (2015)

[85] Howard, A.G., Zhu, M., Chen, B., Kalenichenko, D., Wang, W., Weyand, T., Andreetto, M., Adam, H.: Mobilenets: Efficient convolutional neural networks for mobile vision applications. arXiv preprint arXiv:1704.04861 (2017)

[86] Brutzkus, A., Gilad-Bachrach, R., Elisha, O.: Low latency privacy preserving inference. In: International Conference on Machine Learning, pp. 812–821 (2019). PMLR

[87] Chabanne, H., De Wargny, A., Milgram, J., Morel, C., Prouff, E.: Privacy-preserving classification on deep neural network. Cryptology ePrint Archive (2017)

[88] Wood, A., Najarian, K., Kahrobaei, D.: Homomorphic encryption for machine learning in medicine and bioinformatics. ACM Computing Surveys (CSUR) **53**(4), 1–35 (2020)

[89] Chen, Z., Hu, G., Zheng, M., Song, X., Chen, L.: Bibliometrics of machine learning research using homomorphic encryption. Mathematics **9**(21), 2792 (2021)

[90] Chou, E., Beal, J., Levy, D., Yeung, S., Haque, A., Fei-Fei, L.: Faster cryptonets: Leveraging sparsity for real-world encrypted inference. arXiv preprint arXiv:1811.09953 (2018)

[91] Lee, J.-W., Kang, H., Lee, Y., Choi, W., Eom, J., Deryabin, M., Lee, E., Lee, J., Yoo, D., Kim, Y.-S., *et al.*: Privacy-preserving machine learning with fully homomorphic encryption for deep neural network. IEEE Access **10**, 30039–30054 (2022)

[92] Garimella, K., Jha, N.K., Reagen, B.: Sisyphus: A cautionary tale of using low-degree polynomial activations in privacy-preserving deep learning. arXiv preprint arXiv:2107.12342 (2021)

[93] Arnold, D., Saniie, J., Heifetz, A.: Homomorphic encryption for machine learning and artificial intelligence applications. Technical report, Argonne National Lab.(ANL), Argonne, IL (United States) (2022)

[94] Boura, C., Gama, N., Georgieva, M., Jetchev, D.: Simulating homomorphic evaluation of deep learning predictions. In: International Symposium on Cyber Security Cryptography and Machine Learning, pp. 212–230 (2019). Springer

[95] Boura, C., Gama, N., Georgieva, M., Jetchev, D.: Chimera: Combining ring-lwe-based fully homomorphic encryption schemes. Journal of Mathematical Cryptology **14**(1), 316–338 (2020)

[96] Zhu, R., Ding, C., Huang, Y.: Practical mpc+ fhe with applications in secure multi-partyneural network evaluation. Cryptology ePrint Archive (2020)