New Insights into Fully Homomorphic Encryption Libraries via Standardized Benchmarks

Charles Gouert*, Dimitris Mouris*, Nektarios Georgios Tsoutsos
{cgouert, jimouris, tsoutsos}@udel.edu
University of Delaware

Abstract

Fully homomorphic encryption (FHE) enables arbitrary computation on encrypted data, allowing customers to upload ciphertexts to third-party cloud servers for meaningful computation while mitigating security and privacy risks. Many cryptographic schemes fall under the umbrella of FHE, and each scheme has several open-source implementations with its own strengths and weaknesses. Nevertheless, developers have no straightforward way to choose which FHE scheme and implementation is best suited for their application needs, especially considering that each scheme offers different security, performance, and usability guarantees.

To address this open question and allow programmers to effectively utilize the power of FHE, we employ a series of benchmarks collectively called the *Terminator 2 Benchmark Suite* and present new insights gained from running these algorithms with a variety of FHE back-ends. Contrary to generic benchmarks that do not take into consideration the inherent challenges of encrypted computation, our methodology is tailored to the security primitives of each target FHE implementation. To ensure fair comparisons, we developed a versatile compiler (called *T2*) that converts arbitrary benchmarks written in a domain-specific language into identical encrypted programs running on different popular FHE libraries as a backend. Our analysis exposes for the first time the advantages and disadvantages of each FHE library as well as the types of applications most suited for each computational domain (i.e., binary, integer, and floating-point).

Index Terms

Benchmarking, data privacy, encrypted computation, fully homomorphic encryption, performance evaluation.

*The first two authors have equal contribution and appear in alphabetical order.

I. Introduction

Cloud service providers, such as Amazon Web Services (AWS), Microsoft Azure, and Google Cloud, offer on-demand computing platforms through a pay-as-you-go model leveraging the vast computing resources of their data centers. The minimal capital expenditures and low yearly costs of this model has compelled many companies to shift from maintaining their own private data centers and computing hardware to adopting the cloud computing-as-a-service for their operations [1]. This paradigm offers far more than simply cloud storage: complex calculations such as data classification and analytics can be performed on outsourced data without any involvement from the customer.

While cloud computing offers great flexibility for many industries, various concerns are raised regarding the privacy of outsourced data. Since the data resides on servers controlled by the cloud service provider, there is nothing stopping a curious cloud from observing the data. For instance, if a hospital opts to store medical records on a cloud server, the curious service provider could plausibly gather information about individuals such as their pre-existing conditions and disclose this information to advertising partners. Likewise, as the popularity of cloud computing grows and data from numerous customers reside on shared servers, cloud hardware and software have become increasingly targeted by sophisticated attackers. Side-channel attacks [2], [3], cross-VM attacks [4], [5], speculative execution attacks [6]–[8], and hardware Trojans in the supply chain [9]–[11] have demonstrated that it is possible to leak *data-in-use* from remote servers.

A de facto method to protect data confidentiality for data-in-transit and data-at-rest is the use of cryptography [12], [13]. While standard encryption algorithms such as AES can effectively mitigate a variety of eavesdropping attacks by preventing curious cloud providers from accessing plaintext data, such schemes are mostly limited to data storage and network transmissions [14], [15]. To perform any computation on outsourced data, clients must download the encrypted data, decrypt it, perform the intended operation, re-encrypt, and then re-upload the ciphertexts to the cloud. Obviously, this defeats most benefits of adopting cloud computing in the first place.

Candidate cryptographic protocols for privacy-preserving computation include multi-party computation, and zero-knowledge proofs. The former allows a server and client to jointly compute an algorithm over private data [16], [17], while the latter allows one party to obliviously prove to another party that a statement is true, without revealing any additional information [18]–[20]. While the two aforementioned protocols have niche applications, the most promising cryptographic primitive for arbitrary computation on encrypted data is fully homomorphic encryption (FHE), which is capable of protecting *data-in-use*. This effectively allows the cloud to apply arbitrary algorithms on encrypted data and perform meaningful

computations without ever exposing the plaintext data. However, adopting FHE is easier said than done; many considerations need to be made in order to effectively leverage its power for a given application.

There are several major FHE schemes that can be used to compose any arbitrary algorithm as a netlist of arithmetic operations in the encrypted domain [21], and each scheme has multiple open-source implementations using state-of-the-art cryptographic libraries. Notably, each FHE scheme has unique defining characteristics, including how data is encoded and the types of operations available to users. For instance, schemes such as Brakerski-Gentry-Vaikuntanathan (BGV) [22] and Brakerski/Fan-Vercauteren (BFV) [23] encrypt integers, while others encrypt floating-point numbers or even individual bits. Such encoding determines the nature of the operations exposed to programmers, since ciphertexts encoding floating-point and integer values allow addition and multiplication, while ciphertexts encoding bits allow Boolean logic operations. Therefore, expressing an algorithm in the encrypted domain requires different techniques and programming paradigms depending on the underlying plaintext data types.

To complicate things further for developers, FHE schemes require users to track the (unavoidable) noise growth in homomorphic ciphertexts. In particular, the presence of noise is necessary for security purposes and a small amount is injected in each ciphertext during initial encryption. Then, evaluating consecutive operations on the encrypted data causes this noise to grow and once a certain threshold is reached, the ciphertext can no longer be decrypted and is rendered useless. Modern FHE schemes offer one or more mechanisms to manage this noise, with each technique having different capabilities in terms of noise reduction as well as computational overhead.

As a further point of difficulty, encrypted computation comes with certain inherent limitations, one of which is the inability to make runtime decisions when the control values are encrypted. For example, if a loop termination condition remains encrypted, a server executing an encrypted program may not be able to decide *if* or *when* the execution ends (i.e., there exists a "termination problem" [24]–[28]). To account for this, FHE multiplexing can be employed to re-compose conditional code segments, which essentially evaluates alternative branches at runtime and returns the correct result over the encrypted data.

Our main goal in this work is to provide new and intuitive insights on choosing the correct homomorphic scheme for a given application, and analyze the explicit advantages and disadvantages of different popular open-source implementations. More specifically, this paper aims to answer the following questions related to FHE applications:

- What scheme is most appropriate for a given algorithm?
- What techniques and methodologies can be utilized to maximize efficiency in each scheme?
- How do the existing libraries compare to each other in terms of performance?

At first glance, one could attempt to gather insights about the various FHE implementations using

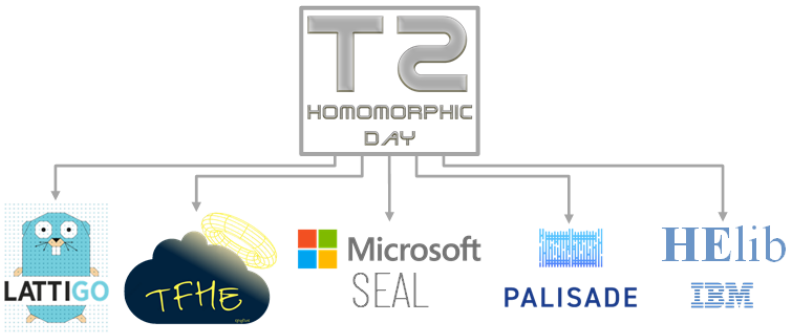


Fig. 1. T2 compiler to FHE backends.

popular general computation benchmark suites, such as the SPEC benchmark suite [29]. However, existing benchmarks like `bzip2` and `mcf` intended solely for plaintext computation do not consider the existence of FHE constructions and do not address FHE termination problems in the algorithm design. These benchmarks essentially make control flow decisions that depend on the value of program data; when implemented in the encrypted domain, it is impossible for the machine executing the program to make a decision based on values encoded inside ciphertexts as it does not have access to the decryption key. As such, a direct translation of these pre-existing benchmarks to the encrypted domain is not meaningful.

To this end, we have employ our custom benchmark suite, dubbed *Terminator 2*, which mitigates termination problems and is tailored for FHE-friendly computation. A key observation in the design of our benchmarks is that we can speculatively evaluate all alternative execution paths before judiciously combining the results in the encrypted domain. Terminator 2 includes 12 real-life applications suitable for secure cloud computing, including private machine learning classification and private information retrieval. This benchmark suite allows us to explore and expose the differences and the benefits between various FHE schemes and their implementations, while enabling us to rigorously test all aspects of encrypted computation. Towards that end, we employ benchmarks that invoke core arithmetic operations between ciphertexts, mixed operations between ciphertexts and constants, relational operations, as well as essential mechanisms for noise mitigation (like *modulus switching* and *bootstrapping*).

A key consideration in our analysis is the *uniformity of benchmark implementations* to inform fair comparisons between libraries. While hand-optimized implementations of encrypted programs for a certain FHE library can help tune runtime performance, this does not translate across different FHE schemes (often with incompatible configurations). Further, subjective factors, such as programmers' experience on one library can further skew the results. Therefore, our methodology relies on *automated compilation* of each benchmarking algorithm across multiple FHE libraries.

To achieve such uniformity, our Terminator 2 benchmarks are implemented in a domain-specific language (DSL), dubbed *T2*, which includes specialized types for encrypted integers, bits, and floating-point numbers. T2 programs are compiled using multiple FHE backends, encompassing five state-of-the-art libraries, to generate corresponding encrypted programs (Fig. 1). Our compiler generates highly optimized functional units that implement FHE circuits for common operations such as comparisons and multiplexing between encrypted values, while minimizing noise growth and computational complexity for efficient evaluation. To the best of our knowledge, this is the first effort to measure and understand the differences among popular FHE libraries in a systematic way.

II. Preliminaries

A. Homomorphic Encryption (HE)

HE is a unique form of encryption that allows for arithmetic operations over ciphertexts where the result is a valid encryption of the expected answer. More formally, the computational capability of HE and the equivalency between encrypted computation and computation over plaintext data is shown as: $y = f(x) \Leftrightarrow Enc(y) = g(Enc(x))$. In this case, $f(x)$ is any arithmetic function in the plaintext domain, $Enc(\cdot)$ refers to an encryption of a plaintext value, and $g(Enc(x))$ parallels $f(x)$ in the encrypted domain. This enables users to encrypt data and outsource computations like $f(x)$ to the cloud without sacrificing privacy, which in turn will execute $g(Enc(x))$ and return $Enc(y)$ to the user, who can decrypt the result with her private key. Any algorithm that can be expressed as an arithmetic or Boolean *circuit* (i.e. a series of additions and multiplications or a netlist of logic gates) can be readily executed with FHE.

In general, there are three overarching types of HE defined principally by their compute capabilities on encrypted data: partial (PHE), leveled (LHE), and fully (FHE). In this work, we only consider the latter two classes as PHE is not functionally complete and therefore not suited for general computation.

1) Ciphertext Encoding: Both LHE and FHE encrypt data as a tuple of polynomials modulo an irreducible cyclotomic polynomial, where the N^{th} cyclotomic polynomial has roots corresponding to the N^{th} primitive roots of unity [30]. In practice, the cyclotomic order (and hence the degree of ciphertext polynomials) is typically chosen to be between 2^{10} to 2^{15} with coefficients modulo a product of primes referred to as q, which can be several hundreds bits in length. FHE schemes inject noise into ciphertexts in accordance with the Learning With Errors (LWE) paradigm [31] to make them resilient to cryptanalyses. The hardness of the LWE problem and its variant Ring LWE (RLWE) [32] guarantee the security of all FHE schemes. The induced noise needs to be kept below a threshold to ensure successful decryption.

Noise management is a crucial component of HE schemes as the noise in ciphertexts dynamically grows with every computation over the encrypted values. More specifically, encrypted addition increases

the noise linearly, however, encrypted multiplication causes exponential noise growth. Therefore, there is only a fixed number of encrypted operations that can be carried out on a ciphertext without exceeding the "noise budget" while still preserving the underlying plaintext data. Interestingly, the choice of security parameters affects how the noise grows and the total noise budget, but many realistic applications are not feasible with secure parameters. To account for that, FHE schemes introduce crafty mechanisms to reduce the noise in ciphertexts and allow for more encrypted operations without exceeding the noise budget. Once the new noise budget has been reached, a noise reduction operation such as *modulus switching* or *bootstrapping* will reduce it and allow for additional encrypted operations. We delve into modulus switching and bootstrapping in the following section.

2) LHE: LHE is the first functionally complete type of HE that is capable of supporting addition and multiplication on encrypted data. The primary mechanism for reducing noise in LHE is *modulus switching*, which effectively reduces the ciphertext size while also scaling down the noise. More specifically, this procedure removes a prime from q, effectively lowering the bit size of the coefficients of the ciphertext polynomials. However, modulus switching can only be invoked a limited number of times; eventually one will run out of primes in q and the noise can no longer be mitigated. As a result, the number of subsequent operations on ciphertexts is bounded and encryption parameters must be chosen carefully to ensure that the application can be evaluated before the noise becomes unmanageable. The encryption parameters guarantee the ability to evaluate a certain number of *levels* on any given ciphertext object before decryption fails. A level is equivalent to the noise accumulated during a multiplication operation; thus, the number of levels required to execute an LHE circuit is synonymous with the multiplicative depth. BFV [23], BGV [22], and CKKS [33] are typically used as LHE schemes.

3) FHE: FHE is the most powerful form of HE that allows for unbounded homomorphic addition and multiplication operations on encrypted data. This is made possible by introducing a *bootstrapping* mechanism to any LHE scheme, which serves as a powerful form of noise reduction. Gentry's bootstrapping computes the homomorphic decryption circuit in the encrypted domain resulting in a new ciphertext with reduced noise [21]. Unfortunately, bootstrapping is the bottleneck of FHE; one must minimize the number of bootstraps needed in an application to optimize performance. Luckily, bootstrapping with modulus switching is far more computationally efficient as mod switching can be invoked until no further primes can be removed from q. Then bootstrapping can be initiated, which will also regenerate all of the primes in q, allowing modulus switching and bootstrapping to be chained ad infinitum.

For algorithms that can be expressed with a shallow depth (i.e., few subsequent ciphertext multiplications), LHE is typically more efficient than FHE. For applications exhibiting a large depth, FHE is a better option because the LHE parameters (such as polynomial degree) required to support a deep homomorphic

circuit at an acceptable level of security are extremely large and result in poor runtime performance. Since the modulus q is typically several hundred bits long, the underlying arithmetic operations are applied over integers greater than the word sizes naturally supported by CPUs. Some implementations use residue number system (RNS) techniques to allow for operations modulo the smaller prime factors of q (typically between 32 and 64 bits) [34].

B. Homomorphic Encryption Schemes

1) Binary Schemes: These HE schemes encrypt individual bits into a single ciphertext, which opens up new avenues of encrypted computation, as it natively supports a broad range of bitwise manipulations in the encrypted domain. Instead of building programs as a sequence of assembly instructions, these binary schemes support Boolean circuits composed of encrypted logic gate constructions. The Torus FHE (TFHE) [35] library implements a ring variant of GSW [36] called Ring GSW (RGSW) and is the most popular in this category, boasting the fastest bootstrapping speeds of any current FHE library. In fact, TFHE strictly supports FHE (but not LHE) because the bootstrapping procedure is instrumental in the correct evaluation of homomorphic logic gates. This requirement informs the need to bootstrap during every logic gate evaluation in the circuit (except for the trivial `NOT` gate).

The ability to compose algorithms as Boolean circuits is a distinct advantage for binary schemes as developers can bring to bear decades of digital circuit design research to help them construct optimal implementations in the encrypted domain. Also, bitwise operations such as shifting are far more efficient compared to the integer domain. For instance, a right shift by k in the binary domain is simply a matter of removing k elements of an array of binary ciphertexts (which has very low cost). Conversely, integer schemes (discussed next) would require division, which is not supported directly in FHE: one would need to compute the modular multiplicative inverse of the divisor and multiply it with the dividend. The latter is far more expensive since it results in substantial noise growth compared to the binary scheme version. Finally, comparisons (i.e., equality, less-than) between two numbers are significantly cheaper in binary schemes vs. non-binary schemes.

2) Integer Schemes: The two most important HE schemes that encrypt integers modulo a user-determined modulus p are BGV [22] and BFV [23]. Here, HE addition and multiplication correspond to modular addition and multiplication over the plaintext values. Consequently, one needs to choose p such that no undesired wrapping will occur. Unlike RGSW, both BGV and BFV have considerably slow bootstrapping procedures; depending on parameter choices (such as the plaintext modulus and cyclotomic order), a single bootstrap can take anywhere from several minutes to several hours [37], which is impractical for any realistic HE computation. Thus, most implementations of BGV/BFV do

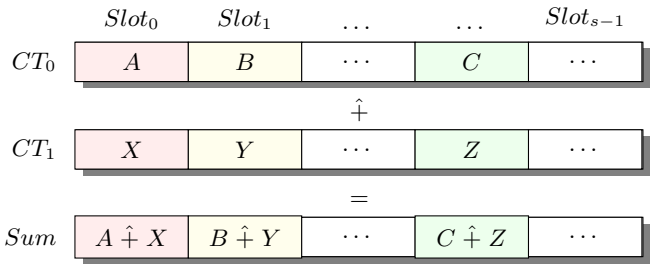


Fig. 2. **Batching Overview.** Each of the ciphertexts CT_1 and CT_2 encrypt multiple integers (i.e., A, B, C, ... and X, Y, Z, ..., respectively), while the result ciphertext Sum contains the slot-wise homomorphic addition ($\hat{+}$) of CT_1 and CT_2. Homomorphic multiplication is similarly supported.

TABLE I

OVERVIEW OF HE LIBRARIES IN TERMS OF THE SUPPORTED SCHEMES, DOMAINS (I.E., INTEGER, BINARY, AND FLOATING-POINT), LEVELS/BOOTSTRAPPING, AND BATCHING OPERATIONS.

Library	Language	Domains	Schemes							
			BGV/BFV			CKKS			(R)GSW	
			Levels†	Bootstrap†	Batch‡	Levels†	Bootstrap†	Batch‡	Levels†	Bootstrap†
Concrete [38]	Rust	Bin, FP	○	○	○	○	○	○	◐	●
FHEW [39]	C, C++	Bin	○	○	○	○	○	○	○	●
HEAAN [33]	C, C++	FP	○	○	○	●	●	●	○	○
HElib [40]	C, C++	Int, Bin, FP	●	●	●	●	○	●	○	○
Lattigo [41]	Go	Int, FP	●	○	◐	●	●	●	○	○
PALISADE [42]	C, C++	Int, Bin, FP	●	○	◐	●	○	●	○	●
SEAL [43]	C, C++, .NET	Int, FP	●	○	◐	●	○	●	○	○
TFHE [35]	C, C++	Bin	○	○	○	○	○	○	○	●

† For Levels (LHE) and Bootstrapping (FHE) we categorize the libraries in three classes: ○ No support, ◐ Partial support (i.e., partially implemented by the library or non-existent API), ● Regular support.

‡ For batching, we classify the libraries as: ○ No support for batching, ◐ Restricted modulus (i.e., in order to enable batching, the plaintext modulus p must satisfy $(p-1) \mid m$, where m is the degree of the cyclotomic polynomial), and ● Flexible batching (i.e., no restrictions on p).

not include bootstrapping and are used exclusively in LHE mode. It is also possible to emulate Boolean circuits in integer schemes by setting $p = 2$, which causes addition to behave as an `XOR` and multiplication to behave as an `AND` gate. The primary difference between this approach and RGSW is that the additions and multiplications do not necessarily require a bootstrap after every operation.

The biggest benefit of using integer schemes over binary is the ability to take advantage of an encoding technique called *batching*, which closely resembles Single Instruction, Multiple Data (SIMD) operations and allows users to encrypt vectors of integers into single ciphertexts [44]. Each individual integer occupies a ciphertext *slot* and the total number of slots available varies depending on parameter choices. Addition and multiplication on "batched" ciphertexts occur slot-wise, as depicted in Fig. 2. An important consideration when using batching is whether or not the individual slots are completely independent or

not. If the slots need to be mixed at some point (such as the need to sum all of the slots together), slot-wise rotations must be executed to align the desired slots, resulting in memory and execution time overheads.

3) Floating-Point Schemes: The final class of HE has an underlying plaintext type of floating-point numbers and the most popular scheme in this category is CKKS [33]. This is especially desirable for certain classes of algorithms, such as machine learning [45], [46]. In practice, the core operations of floating-point schemes neatly parallel integer schemes, with only a few differences. Both integer and floating-point schemes support batching and addition/multiplication operations. Similarly, the floating-point schemes have slow bootstrapping procedures and are predominantly used in LHE mode. The key difference is the need to keep track of the *scale* factor that is multiplied with plaintext values during encoding that determines the bit-precision associated with each ciphertext. The scale doubles when two ciphertexts are multiplied, which may result in overflow as the scale becomes exponentially larger. Thus, a *rescaling* mechanism must be invoked to preserve the original scale after multiplications, which is quite similar to modulus switching and serves a similar role to reduce ciphertext noise.

C. Homomorphic Encryption Libraries

Various open-source HE libraries implement the aforementioned schemes and expose a more or less high-level API with common functionalities like `Encrypt`, `Decrypt`, `KeyGen`, `Add`, and `Multiply`. Also, these libraries internally implement numerous performance optimizations (such as RNS and NTT/FFT for polynomial multiplication [47], [48]) and leverage noise reduction techniques (e.g., modulus switching and bootstrapping) that require deep understanding from the developer in order to be used correctly. Further, most libraries rely on the user to select an appropriate set of encryption parameters for her application. In this paper, we focus on five widely used open-source libraries, described below.

1) HElib: The Homomorphic Encryption Library (HElib) was introduced in 2013 [40] by IBM and supports the BGV scheme (with bootstrapping), as well as CKKS. It is written in C++17 and uses the NTL mathematical library [49].

2) Lattigo: The lattice-based multiparty homomorphic encryption library in Go (Lattigo) is actively developed by the Laboratory for Data Security (LDS) at EPFL [41]. It supports leveled BFV and bootstrapped CKKS with full-RNS optimizations and their respective multiparty versions. Lattigo enables cross-platform builds and offers comparable performance to other state-of-the-art libraries.

3) PALISADE: PALISADE was developed by Duality Technologies, NJIT, MIT, and other organizations [42]. It offers support for leveled BFV, BGV, and CKKS with RNS optimizations as well as RGSW (with bootstrapping).

4) SEAL: The Simple Encrypted Arithmetic Library (SEAL) is developed by Microsoft Research and was first released in 2015 [43]. SEAL supports the BFV and CKKS schemes without bootstrapping.

5) TFHE: The Fast Fully Homomorphic Encryption Library over the Torus (TFHE) was released in 2016 by Chillotti et al. [35] and implements the RGSW cryptosystem. The library exposes homomorphic Boolean gates such as `AND` and `XOR` but does not build complex functional units (e.g., adders, multipliers, and comparators) and leaves that to the developer.

6) Other Libraries: The Concrete [38] library by Zama implements a variant of TFHE that supports floating-point plaintext encodings and bootstrapping that allows evaluation of univariate functions. However, Concrete is not yet mature for general purpose computation as its bootstrapping mechanism necessitates the use of (lossy) low-precision arithmetic. For instance, CKKS bootstrapping allows a precision of up to 40 bits [50], while Concrete is restricted to less than 12 bits of precision [51] (effectively forcing applications to use small plaintext moduli). Additionally, the rounding errors that result from the low-precision will compound over time for deep applications, as is shown in the accuracy loss reported for deep neural networks in [51]. FHEW [39], developed by CWI in Amsterdam, is the predecessor of TFHE; however, its bootstrapping speed is inferior to TFHE and there is no support for homomorphic MUX gates. Lastly, HEAAN [33] implements the CKKS cryptosystem, where both the library and the underlying scheme are created by the Cryptography Lab at Seoul National University. However, HEAAN has not seen a major update since 2018 and has been succeeded by other RNS-based implementations of CKKS in libraries such as SEAL, PALISADE, and Lattigo. The key features of each library are summarized in Table I.

D. Threat Model

To enable meaningful and fair comparisons between FHE libraries, we formalize a security model. As homomorphic encryption is used for privacy-preserving outsourced storage and computation, our threat model is twofold. For one, we assume a *rational, honest-but-curious* cloud service provider that executes the protocol correctly but has incentives to peek at sensitive user data. For another, our model assumes that adversaries may compromise the cloud server and exfiltrate sensitive data that are either in use or at rest. Thus, in our T2 benchmarks, we explicitly define which data are private and should remain encrypted during and after the computation.

E. Related Works

A related work is [52], which compares (mostly obsolete) FHE implementations and is missing modern state-of-the-art libraries. Moreover, when comparing the different schemes, the authors incorporate results

from prior works configured for non-equivalent security levels, parameter sets, and even benchmarks. Lastly, [52] lacks any implementation and fair comparisons across the studied FHE libraries.

A recent SoK [28] focuses on state-of-the-art FHE compilers such as: Cingulata [53], which compiles C++ code to Boolean circuits for TFHE and a BFV variant, CHET [54] which targets CKKS and is geared towards private neural network inference for HEAAN and SEAL, as well as its successor EVA [55] that uses a DSL for vector arithmetic and targets SEAL and CKKS. Likewise, E3 [56] targets SEAL, HElib, FHEW, PALISADE, and TFHE, but has limited batching support, does not support relational operations over integers, and only allows users to select plaintext and poly modulus degrees, but other important parameters such as ciphertext modulus size remain hidden from the user. Marble [57] is a C++ extension that allows users to write code similar to a plaintext implementation and currently employs encrypted binary arithmetic in HElib as an FHE backend. However, [28] does not consider compilers such as Google's Transpiler [58] and ROMEO [59] that convert programs (written in C++ and Verilog, respectively) into optimized netlists through the use of synthesis tools, and then finally into TFHE (and PALISADE's RGSW implementation for the former).

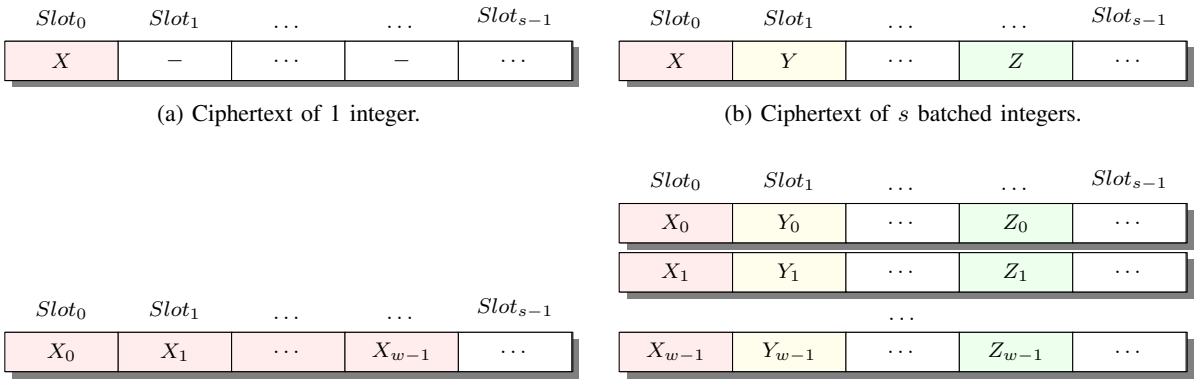
While [28] performs thorough comparisons between many different compilers, it only uses three minimal applications used during experimentation that are limited in scope. Most importantly, [28] does not perform any comparisons between the underlying FHE libraries themselves and different encoding types were not explored thoroughly, which is the scope of our work. Finally, all benchmarks in [28] were manually implemented for each compiler, which results in potentially non-uniform comparisons. Contrary to earlier works, our T2 compiler automatically converts each benchmark into its encrypted implementation for all major FHE libraries.

III. Methodology for Fair Comparisons

A. Notation

We represent the encryption of a variable x as X, while we use *accents* ($\hat{}$) to represent encryptions of numbers, such as $\hat{0}$ and $\hat{1}$. Similarly, $\hat{+}$, $\hat{-}$, $\hat{*}$, and $\hat{\sim}$ represent homomorphic addition, subtraction, multiplication, and negation, respectively. These private arithmetic operations are inherently supported by all the schemes except RGSW, for which we use hand-optimized Boolean circuits implementing arithmetic functional units in the encrypted domain. Specific circuit design details and considerations are discussed in Section III-C. In binary arithmetic we also support homomorphic `XOR` between two ciphertexts (represented as $\hat{\oplus}$), as well as homomorphic `AND`.

Finally, we represent private comparisons as follows: $\hat{=}$ symbolizes private equality, $\hat{<}$ refers to private less than, and $\hat{\leq}$ represents private less than or equal. The comparisons return $\hat{1}$ (i.e., encryption of



(a) Ciphertext of 1 integer.

(b) Ciphertext of s batched integers.

(c) Ciphertext of 1 integer using horizontal binary packing.

(d) Ciphertext of s batched integers using vertical binary packing.

Fig. 3. **Encodings Overview.** Each of the four encodings demonstrates how one or more integers are encrypted utilizing one or more slots. In (a), a single integer x is encrypted utilizing only one slot, whereas in (b) all the slots are being used to encrypt multiple integers (e.g., x, y, z, and so on). We refer to encoding (b) as *integer batching*. The encoding in (c) utilizes multiple slots to encrypt an integer using a bitwise representation (the number of slots used is equal to the word size w of x). We refer to this approach of encoding w binary digits of an integer as *packing*. Finally, (d) utilizes w ciphertexts to encode multiple integers (e.g., x, y, z) by using a slot-wise binary representation. We refer to encoding (d) as *vertical binary packing*.

one) if the corresponding plaintext operation is `True`, or $\hat{0}$ otherwise. We discuss the design of private comparison units in section III-D as most libraries do not natively support them.

B. ENCINT, ENCFP, *and* ENCBIN *Data Encodings*

The four primary types of ciphertext encodings are depicted in Fig. 3. For encoding single integers or floating-point numbers, one can simply encrypt them directly in the first slot of the ciphertext, as shown in Fig. 3 (a). Encoding a single plaintext element per ciphertext is inefficient for schemes that support batching; instead, s independent program inputs can fit inside a single ciphertext as illustrated in Fig. 3 (b). This allows for higher occupancy of the available slots to take advantage of SIMD-style computation. Depending on the underlying data type of the scheme (i.e., integer or floating-point), we refer to encodings (a) and (b) as ENCINT and ENCFP. Alternatively, a single value can be decomposed into w binary digits and each digit is encrypted in a separate slot (Fig. 3 (c)). This row-wise encoding approach is useful for encrypting integers that are greater than the chosen plaintext modulus as well as representing an integer in binary format. Finally, as shown in Fig. 3 (d), we can encrypt each digit into a separate ciphertext and take advantage of the slots of each ciphertext to fit s independent elements (column-wise encoding across multiple ciphertexts). We refer to the encoding in Fig. 3 (d) as ENCBIN. Below, we delve into the benefits and drawbacks of each encoding.

1) Integer and Floating-point Batching: Depending on the choice of parameters for HE schemes that support batching, ciphertexts have a number of available slots, which can range from one to several thousand. If extra slots are available, they can be occupied by additional plaintext values instead of resorting to creating new ciphertexts. This reduces memory consumption and enables SIMD-style computation; the key benefit in terms of the latter approach is that HE arithmetic operations will take the same amount of time regardless of how many slots are occupied. Therefore, this approach, also highlighted in Fig. 3 (b), significantly improves throughput without negatively impacting latency. However, to enable batching with several thousand slots, larger FHE parameters must be chosen. In particular, batching is bounded by the degree of the ciphertext polynomials such that the maximum number of slots available is less than or equal to the polynomial degree. Unfortunately, increasing the polynomial degree negatively impacts the performance of all HE operations, while also increasing the ciphertext sizes.

2) Horizontal & Vertical Binary Packing: The two primary methods for encoding binary arrays into packed ciphertexts involve utilizing the slots of one or more ciphertexts, as shown in Fig. 3 (c) and (d). The row-wise approach introduces a more straightforward encoding that only uses one ciphertext, however, the downside to this approach is that the digits are inherently dependent for most operations. For instance, addition and multiplication can not be done digit-wise as these operations are sequential and the output for the current digit depends on the computation from the previous digit (such as accounting for carries). Therefore, in order to perform an addition using this encoding, each slot needs to be isolated and then fed into an adder circuit, which requires convoluted slot rotations and thus is expensive. In fact, slot extraction is used in the bootstrapping procedure and is one of the core bottlenecks of this operation [60]. While this form of batching is the most intuitive and straightforward, we do not consider this encoding in the design of the T2 benchmarks as it is inappropriate for most types of computation.

The ENCBIN column-wise approach (i.e., Fig. 3 (d)) does not suffer from the same issues as the horizontal binary encoding, yet still benefits from efficient SIMD computation. The key insight is that now all slots across a given ciphertext are completely independent since all digits are separated across w ciphertexts, where w is the plaintext word size. This allows one to directly evaluate binary arithmetic circuits in FHE, such as adders, multipliers and comparators, without having to perform any digit extraction procedures. As such, all of the advantages provided by the standard integer batching technique apply to this case as well. A potential trade-off with this approach is the additional memory overhead of dealing with w ciphertexts instead of one.

C. Arithmetic Operations for HE Benchmarks

1) Arithmetic in ENCINT *and* ENCFP*:* All libraries that implement integer or floating-point schemes naturally support addition and multiplication by encoding the values in separate slots as shown in Fig. 3 (b) (i.e., ENCINT or ENCFP). In addition, these libraries support negation, which further enables converting an addition operation into a subtraction.

2) Arithmetic in ENCBIN*:* For binary schemes like RGSW (or ENCBIN in general), these operations are not naturally supported. Instead, one must construct Boolean circuits composed of primitive logic gates to facilitate these operations. In T2 benchmarks, we have constructed binary arithmetic circuits for every cryptosystem (with the exception of floating-point schemes), customized for efficient HE evaluation. For RGSW, we have minimized the number of logic gates in the circuit and prioritized the nearly free `NOT` gate where possible. This effectively reduces the total number of bootstraps required to evaluate the circuit. Conversely, for LHE schemes, we introduce two circuit variants that deviate depending on the value of the plaintext modulus p. In the ideal binary case, where $p = 2$, addition operations are equivalent to `XOR` gates while multiplications are equivalent to `AND` gates. Since multiplication is far more expensive in terms of execution time and noise growth, we have utilized circuits with as few subsequent `AND` operations as possible and prioritized the nearly free `XOR` operations. However, it is impossible to achieve a high number of slots with $p = 2$ and for this reason all implementations of BFV and BGV do not allow batching at all with this setting, with the exception of HElib.

The emulation of logic gates with higher values of p is more complicated than in the $p = 2$ case. Luckily, as long as the underlying plaintext values are constrained to $[0, 1]$, multiplication is still equivalent to `AND`. Nevertheless, addition is no longer equivalent to `XOR` since, for instance, $1 + 1 = 2 \pmod p$ instead of $0 \pmod p$. Thus, the `XOR` operation can be computed by performing $(X \mathbin{\hat{-}} Y)^2$, which requires a homomorphic subtraction followed by a multiplication (squaring). Obviously, this `XOR` variant is more expensive than both the `XOR` with $p = 2$ and the `AND` operation. As a result, we prioritize `AND` over `XOR`; still, FHE circuits for $p > 2$ are more expensive in terms of noise growth and execution time.

D. Comparison Operations for HE Benchmarks

1) Comparisons in ENCINT *and* ENCFP*:* Comparing two multi-bit values in the encrypted domain remains an active area of research and is generally considered to be quite inefficient. For ENCINT and ENCFP encodings, we employ Fermat's Little Theorem to compute equality, which becomes $(X \mathbin{\hat{-}} Y)^{p-1} \mod p$, where p is a prime [61]. The computational bottleneck involves computing the power $p-1$, which requires approximately $\log_2 p$ multiplications. For applications that require a large plaintext range (or

require a high degree of batching), this method becomes particularly inefficient as it can easily consume dozens of levels.

To accomplish the FHE "less-than" function, we adopt the polynomial interpolation over a prime field proposed by Iliashenko and Zucca [61]. Specifically, we evaluate:

$$X \hat{<} Y \coloneqq \sum_{a=-\frac{p-1}{2}}^{-1} \left(1 \mathbin{\hat{-}} (X \mathbin{\hat{-}} Y \mathbin{\hat{-}} a)^{p-1}\right) \mod p, \tag{1}$$

which will output $\hat{1}$ if $X \hat{<} Y$ and $\hat{0}$ otherwise. We remark that this operation is more expensive than the FHE equality due to the large number of addition/subtraction operations; still, the multiplicative depth remains the same. Lastly, by swapping X and Y in Eq. 1 and then negating the result, we can achieve FHE "less-than-or-equal" (i.e., $X \hat{\leq} Y \equiv 1 \mathbin{\hat{-}} (Y \hat{<} X)$) for approximately the same cost as less-than.

2) Comparisons in ENCBIN*:* Similarly to the arithmetic operations in ENCBIN, circuits composed of logic gates must be constructed to accomplish encrypted comparison functions. The natural conclusion to solving this problem is to adapt a full-fledged single-bit FHE comparator circuit that outputs three signals (one for less-than, equality, and greater-than) and then cascade them to achieve multi-bit comparisons. However, this approach is sub-optimal if only one of the signals is needed; instead, we compose separate, more optimal circuits for each functionality. For equality, we perform w `XNOR` operations over each pair of bits of the two ciphertexts and then `AND` the partial results together to get a single bit encrypted output. For less-than, we adopt an optimized bit-level approach designed for use with homomorphic sorting [62]:

$$X \hat{<} Y \coloneqq \sum_{i=1}^{w} \left(lt(X_i, Y_i) \prod_{i<j<w} eq(X_j, Y_j)\right) \mod 2, \tag{2}$$

where $lt(x, y) = (\texttt{NOT}\ x)\ \texttt{AND}\ y$ and $eq(x, y) = \texttt{NOT}\ (x\ \texttt{XOR}\ y)$. Finally, to achieve less-than-or-equal we employ the same technique as mentioned earlier, where we swap X and Y in the less-than function and then invert the result.

E. Avoiding HE Termination Problems with Multiplexing

One of the key concerns in encrypted computation is the inability to branch on encrypted data. Since the cloud has no knowledge of the underlying plaintext value of an encrypted condition, it is impossible to make a runtime decision based on this value. Consequently, all possible branches must be evaluated and the correct result must be selected using an HE multiplexing operation. In ENCINT, this can be achieved using the following equation: $\text{MUX}(S, X, Y) \coloneqq X \mathbin{\hat{*}} S \mathbin{\hat{+}} Y \mathbin{\hat{*}} (1 \mathbin{\hat{-}} S)$, where S is the encrypted selection bit and X and Y are the two outcomes of independent branches. In ENCBIN, these operations can be directly translated to logic gates, requiring two `AND`, one `XOR`, and one `NOT` gate. However, in the case of column-wise binary packing, the procedure needs to be executed w times to cover all the bits

of the inputs. The TFHE library natively supports a `MUX` gate for the cost of two bootstraps, as opposed to the expected three bootstraps (two for the `AND` gates and one for the `XOR`).

IV. Algorithms in the T2 Benchmark Suite

In this section, we introduce our Terminator 2 benchmark suite comprising twelve privacy-preserving benchmarks without early termination conditions and data-dependent branching (such as if/else statements and loops over encrypted data). As defined in [24], in encrypted computation the host should remain oblivious to any termination conditions of the algorithm, introducing the "termination problem". Thus, all FHE computations should be made data-oblivious and any termination condition over encrypted data should be transformed into its privacy-preserving counterpart that avoids such conditions.

We divide our benchmarks into three distinct categories indicating the dominant type of homomorphic operation:

- *Arithmetic* benchmarks contain primarily addition and multiplication operations and operate in the integer and floating-point domains.
- *Bitwise* benchmarks are composed of primarily logic gate operations as well as shifts.
- *Relational* benchmarks are those containing numerous comparisons over encrypted data.

We remark that these benchmarks can run in one or more *computational domains* (i.e., integer, floating-point, and binary) which indicate the type of operations in the T2 DSL; not to be confused with the underlying data encodings used by the FHE backends (i.e., EncInt, EncFP, and EncBin). More specifically, the EncInt and EncBin encodings can be used for integer domain benchmarks, while binary domain benchmarks exclusively use the EncBin encoding, and EncFP is only applicable to floating-point benchmarks. We note that the depth for binary evaluation of certain benchmarks is very high, so evaluating them in the binary domain can be inefficient due to the high noise cost of deep Boolean circuits. For this reason, we only show binary results for all benchmarks using RGSW unless explicitly indicated.

A. T2 Arithmetic Benchmarks

1) Chi-Squared: The χ^2 or chi-squared test is a statistical mechanism that can be used to deduce whether a set of data aligns with an expected model; this statistic is useful in a wide-range of applications, such as genomics [63]. First, the server receives the encrypted genotype counts N_0, N_1, N_2, then computes $A \coloneqq 4(N_0 \mathbin{\hat{*}} N_2 \mathbin{\hat{-}} N_1^2)^2$, $B_1 \coloneqq 2(2N_0 \mathbin{\hat{+}} N_1^2)^2$, $B_2 \coloneqq (2N_0 \mathbin{\hat{+}} N_1) \mathbin{\hat{*}} (2N_2 \mathbin{\hat{+}} N1)$, and $B_3 \coloneqq 2(2N_2 \mathbin{\hat{+}} N_1)^2$ and finally returns the encrypted results to the client. Like [63], T2 does not implement costly homomorphic division and leaves this final step as a cheap post-processing procedure done on the

client-side. More specifically, the client decrypts A, B_1, B_2, B_3 and acquires $\alpha, \beta_1, \beta_2, \beta_3$, respectively, which are then used to compute $X^2 = \frac{\alpha}{2N}(\frac{1}{\beta_1} + \frac{1}{\beta_2}\frac{1}{\beta_3})$, where $N \coloneqq N_0 + N_1 + N_2$. We evaluate the χ^2 benchmark both using the ENCINT and ENCFP encodings.

2) Logistic Regression (LR) Inference: Privacy-preserving machine learning (PPML) is an emergent research area in encrypted computation [51], [64]. Logistic regression is an important PPML technique that is similar in construction to a single layer neural network with a sigmoid activation function. In practice, it is often used to classify data into one of two classes and is applicable in areas such as natural language processing (NLP) [65]. The most expensive part of logistic regression inference in FHE is the evaluation of the non-linear sigmoid function. Since the sigmoid can not be evaluated directly over ciphertexts, a polynomial approximation must be utilized. In T2, we employ a Taylor series expansion of the sigmoid function and evaluate the first three terms (i.e., $\sigma(X) \coloneqq 1/2 + X/4 + (X^3)/48 + \dots$) [66]; using more terms yields higher accuracy at the expense of more noise and slower execution times. This benchmark is executed in ENCINT and ENCFP. Since none of the HE libraries supports homomorphic division, for ENCINT we scale the series up by the largest denominator and evaluate $\sigma(X) \coloneqq 24 \hat{+} 12X \hat{+} X^3 + \dots$; at the end, the client can scale down the final result. For ENCFP, we pre-compute the coefficients (i.e., encryptions of 0.5, 0.25, and 0.02083) and perform homomorphic operations with X.

3) Feedforward Neural Network (NN) Inference: The cloud can perform oblivious neural network inference procedures with its own proprietary model (i.e., weights and biases) on sensitive user data for classification, and finally return a set of encrypted probability scores for each class. In this scenario, the cloud has no knowledge of client inputs or the final classification result. To demonstrate the feasibility of FHE libraries for these types of applications, we utilize a feedforward neural network with two fully connected layers and the squaring (X^2) activation function. It is common in the literature to adopt the square activation function for encrypted inference due to its low multiplicative depth, whereas other non-linear activation functions, like ReLU, are much harder to evaluate in the encrypted domain since they require an expensive, high-depth polynomial approximations [67], [68].

As the weights and biases for each layer are known to the cloud, they do not have to be encrypted. Thus, homomorphic multiplications between ciphertext and plaintext values are performed faster and result in moderate noise accumulation compared to multiplication between two ciphertexts. However, this benchmark remains both addition and multiplication intensive as it also performs operations with ciphertexts (e.g., in the square function). We evaluate this benchmark in both ENCINT and ENCFP with a constant image size for a variety of active neuron sizes to show scalability for wider networks.

4) Matrix Multiplication: Matrix multiplication is an incredibly important computation in a wide variety of fields such as machine learning. This benchmark is multiplication-intensive and poses a good

test of the speed of homomorphic multiplication, relinearization, and modulus switching procedures. In our matrix multiplication benchmark, we consider multiplying two square matrices of varying sizes up to 16x16. We run this benchmark in ENCINT and ENCFP.

5) Batched Private Information Retrieval (BaPIR): Private information retrieval (PIR) allows a client to obliviously download an element from an encrypted database without revealing the query (i.e., which element was downloaded) to the server [69], [70]. PIR has a plethora of applications in private computation, such as ad delivery [71], friend discovery [72], and keyword search [73]. For a single query, the server iterates over all database elements and obliviously select the client's element, if present. We assume a database of size n where the keys are mapped to the index idx of a desired value and the key-index mapping is public knowledge. Using this structure, XPIR [69] introduced an optimized PIR that trades bandwidth for performance as the client sends a size n vector that contains an encryption of 1 at the index of the requested element i and fresh encryptions of 0 elsewhere. More specifically: $Q = (q_1, q_2, \ldots, q_n)$, where $q_i = \hat{1}$ for $i = idx$ and $q_i = \hat{0}$, otherwise. The server performs a pair-wise multiplication between Q and the database elements $V = (v_1, v_2, \ldots, v_n)$ and returns a sum of the products (equivalent to the encryption of the requested element) $\sum_{i=1}^{n} (Q[i] \hat{*} V[i])$.

This benchmark can also exploit batching to drastically reduce both the number of encrypted operations as well as communication bandwidth. Instead of representing Q and V as vectors of ciphertexts, these can each be condensed into single ciphertexts with at least n slots. Now, only one slot-wise multiplication is required between Q and V and the product will be returned directly to the user, which will contain the desired value in slot idx. We evaluate PIR over an encrypted database in both ENCINT and ENCFP.

6) Squared Euclidean Distance: The squared Euclidean distance is a mathematical formula used to compute the distance between two n-dimensional points and is also an important statistical measure in cluster analysis [74]. In our case, we consider two n-dimensional points $V = (v_1, v_2, \ldots, v_n)$ and $U = (u_1, u_2, \ldots, u_n)$ in Euclidean space. The squared Euclidean distance is calculated as the sum of the squared differences between the points as $E^2(V, U) \coloneqq \sum_{i=1}^{n} (V[i] - U[i])^2$. Notably, this procedure is more FHE-friendly than the standard Euclidean distance that requires a square root operation at the end, which is not possible to directly evaluate in the encrypted domain. As a result, the user would need to compute the final square root in the plaintext domain after decrypting the cloud's output. For this benchmark we use ENCINT and ENCFP.

B. Bitwise Benchmarks

1) Cyclic Redundancy Check (CRC): The CRC algorithm is commonly used to detect errors in digital data and can also serve as a non-cryptographic hash. Notably, this algorithm can perform meaningful

work in the encrypted domain as well: for example one could check whether two encrypted images are identical. CRC iterates over each input bit and performs bitwise operations with an accumulate step. Its most expensive operation is this accumulation, which is an n-bit addition where n can be up to 64 bits, depending on the CRC variant used. T2 features the CRC-8 and CRC-32 algorithms using ENCBIN, which represent good tests for bitwise operations such as `XOR`, shifting, and binary addition. We remark that CRC-32 is much deeper as it requires 32-bit additions in FHE.

2) Insertion Sort: Sorting algorithms are an interesting, and mostly unsolved, problem for encrypted computation; in all cases, one will necessarily have to deal with the worst-case complexity $\mathcal{O}(n^2)$ (where n is the size of the array), due to the termination problem. The insertion sort itself makes frequent use of $\hat{<}$ for the conditional swap, making it an intensive comparison benchmark that is impractical in ENCINT for anything other than restrictively small values of p (as in Eq. 1). We also remark that the FHE Boolean circuit for ENCBIN is incredibly deep as each element is updated continuously with `MUX` gates (and each one increases the multiplicative depth). As such, we only consider libraries with bootstrapping in ENCBIN (namely TFHE and HElib) for this benchmark.

3) Binary Manhattan Distance: The Manhattan distance is a measure of the distance between two points $V = (v_1, v_2, \ldots, v_n)$ and $U = (u_1, u_2, \ldots, u_n)$ in an n-dimensional space and is defined as $M(V,U) \coloneqq \sum_{i=1}^{n} (\mid V[i] \hat{-} U[i] \mid)$. The key difference between Manhattan distance and Euclidean distance is the usage of a taxicab geometry as opposed to Euclidean space. This benchmark differs slightly in ENCBIN as the absolute value can be modeled as a multiplexing operation for each iteration between $V[i] \hat{-} U[i]$ and $-(V[i] \hat{-} U[i])$, where the select bit is the encrypted sign bit of the difference.

4) Binary Private Information Retrieval (BinPIR): Similarly to BaPIR from Section IV-A5, we assume a database $V = (v_1, v_2, \ldots, v_n)$ of size n where the keys correspond to the index idx of a desired value. In this binary version of PIR, the client does not perform any pre-processing and solely sends an encrypted index Q to the cloud. From an algorithmic standpoint, the binary approach varies greatly from the ENCINT and ENCFP implementations as each encrypted element in the database is visited and an equality circuit is used to check if the index is equal to Q. Then, the output of the equality is used as the control input of a `MUX` which will select between the desired value or an encryption of zero. Lastly, the results of the `MUX` from each element in the database are `XOR`ed together to compute the final result, which is returned to the user for decryption.

C. Relational Benchmarks

1) Hamming Distance: The Hamming distance between two strings measures the minimum number of substitutions required to change one string into the other, and is widely used for error detection in

TABLE II

NOISE GROWTH AND LATENCY OF CORE OPERATIONS FOR INTEGER, BINARY, AND FLOATING-POINT ENCODINGS. PARAMETER SETS HAVE BEEN JUDICIOUSLY CHOSEN TO BE AS EQUIVALENT AS POSSIBLE ACROSS LIBRARIES.

Encoding	**Binary Encoding (EncBin) with word size $= 8$***						**Integer Encoding (EncInt)***						**Floating-Point Encoding (EncFP)***			
Operation	**Add.**	**Mult.**	**Eq.**	**LT.**	**ROT.**	**Ctxt**	**Add.**	**Mult.**	**Eq.**	**LT.†**	**ROT.**	**Ctxt**	**Add.**	**Mult.**	**ROT.**	**Ctxt**
Noise	‡					**Size¶**						**Size**				**Size**
HElib	§					49.4						6.2				9.5
Lattigo						50.3						6.3				9.4
PALIS.						14.7						1.8				5.8
SEAL						14.6						1.8				4.5
TFHE						0.02	-	-	-	-	-	-	-	-	-	-
Fastest	TFHE	TFHE	TFHE	TFHE	TFHE#	N/A	PALIS.	PALIS.	PALIS.	PALIS.	Lattigo PALIS.	N/A	PALIS.	PALIS.	Lattigo PALIS.	N/A
Slowest	PALIS.	PALIS., SEAL	HElib, PALIS.	PALIS., SEAL	HElib	N/A	SEAL	HElib	HElib, SEAL	HElib, SEAL	HElib	N/A	Lattigo	HElib, Lattigo	HElib	N/A

* For integer and floating-point domain operations, we used a cyclotomic polynomial degree of 16k while binary domain operations were conducted with a degree of 32k (with the exception of TFHE, which uses degree 1024).

† The execution time scales exponentially with increasing plaintext modulus and is infeasible to compute with large plaintext moduli.

‡ Noise accumulation (or depth) relative to other HE operations, where the patterned bar indicates noise growth. For instance, binary addition results in significantly higher noise than integer addition (lower is better).

§ The number of bars indicates latency relative to other HE operations. For instance, integer addition is among the fastest HE operations and is awarded a single bar while integer comparisons are much slower and earn multiple bars (lower is better).

These rotations are performed on the vectors holding the encrypted bits and are independent of the HE library.

¶ Ciphertext size in MBs using the smallest parameters for each library to encrypt an 8-bit number and achieve depth 10 at 128 bits of security.

telecommunications, as well as to determine genetic distance in biology applications. Notably, it has also been used as an FHE application in prior works [57]. The Hamming distance over two vectors can be computed as $H(V,U) := \sum_{i=1}^{n} (V[i] \neq U[i])$ and constitutes a critical benchmark for FHE since its core operation is a comparison. This benchmark is executed using both EncInt and EncBin.

2) Relational Manhattan Distance: Instead of using the sign bit directly as the input to the MUX operation to compute the absolute value of the difference of $V[i] \hat{-} U[i]$ as presented in Section IV-B3, this benchmark takes a different approach to determine the sign, as each individual bit can not be easily extracted in EncInt. We employ the encrypted "less than" operator $\hat{<}$ to determine whether the difference is less than $\hat{0}$ and use the output as the selector of the multiplexer. Here, instead of the MUX and additions being the core bottleneck of the algorithm, the $\hat{<}$ becomes the most expensive operation.

V. Experimental Evaluation & Implementation

We run the T2 benchmarks using three different encodings, i.e., EncInt, EncBin, and EncFP, where applicable. The benchmarks from the Arithmetic and the Relational categories are inherently a better fit

for the ENCINT and ENCFP encodings, whereas the ones from the Bitwise category can only be run in ENCBIN because they require operations (e.g., bitwise right rotation) which are not possible under most circumstances in the two former encodings. Still, to investigate the effect that the target domain makes on the execution time of a benchmark, we additionally run selected benchmarks from the Arithmetic and the Relational categories using the ENCBIN encoding. Finally, we carefully choose FHE parameter sets for each library to achieve security of at least 128 bits, and where applicable, we varied the input length of the benchmark, and the word-size w for ENCBIN. We only consider the timings of FHE operations conducted on the server-side, which consists of all evaluation operations except key generation and encryption/decryption. For the former, this is a one-time cost for the user, while the latter operations are generally much faster than homomorphic evaluation operations.

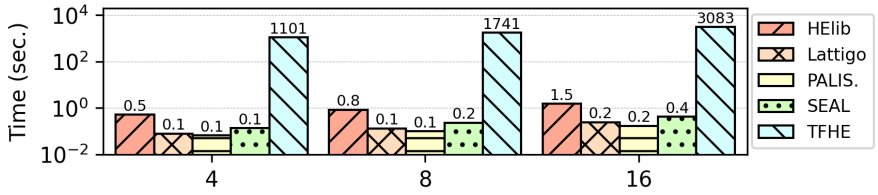
All experiments were performed on an AWS instance (c5.4xlarge) running Ubuntu 20.04, equipped with 8 vCPUs and 32 GBs of memory. Where applicable, we measured the runtime performance of our benchmarks with all encodings, namely, ENCINT, ENCFP, and ENCBIN.

Our T2 compiler targets the latest versions of the FHE backends at the time of writing, specifically: HElib v2.2.1, Lattigo v3.0.2, PALISADE v1.11.6, Microsoft SEAL v4.0, and TFHE v1.0.1. For TFHE, we used the `SPQLIOS-FMA` FFT engine, which is a dedicated fused multiply–add (FMA) assembly version for fast Fourier transform (FFT) computations. Finally, as Lattigo and PALISADE use the RNS optimizations for BFV/BGV they can easily exploit multi-threading. Since these libraries are configured to run on multiple cores by default, we did not restrict them to a single core. Similarly, we did not add any custom form of parallelization to HElib, SEAL, and TFHE as they only utilize one core by default.

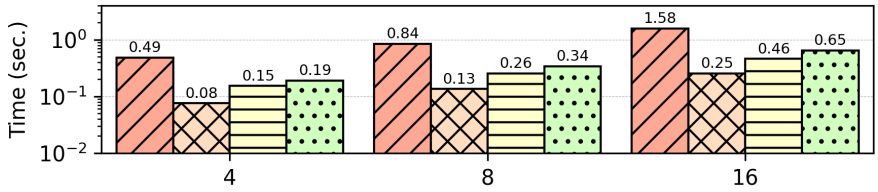
In addition to the three categories of benchmarks analyzed in this section, we also evaluate the timings of the FHE primitive operations across all libraries in all encodings where applicable. Our results are illustrated in Table II and showcase the relative speeds of each library for different operations as well as the ciphertext sizes generated by each library for a fixed depth and security level. We note that TFHE ciphertexts are approximately three orders of magnitude smaller than the other FHE libraries, yet they are non-batchable. The larger ciphertexts, which are several megabytes in size, may support thousands of batching slots. We remark that some libraries are unable to run certain benchmarks successfully because there are no allowable parameters to support the required noise depth (i.e., there is no valid configuration to support the required noise budget while maintaining 128 bits of security). We indicate these cases as "Noisy" with a background color denoting the library (as shown in Fig. 7).

A. Selected Arithmetic Benchmarks[1]

1) Chi-Squared: As this benchmark mainly comprises additions and multiplications with low-depth, the runtime performance for all the libraries except TFHE for both EncInt and EncFP is under half a second. For TFHE, we use a word size of 8 bits and the runtime is 44.3 seconds, a considerable difference which is due to the evaluation of binary adders and multipliers, which require several addition, multiplication, and bootstrapping operations for every circuit.



(a) Integer domain with EncBin for TFHE and EncInt for other libraries.



(b) Floating-point domain with EncFP.

Fig. 4. Measured execution time for the LR benchmark for integer and floating-point domains for an increasing number of attributes. TFHE uses EncBin with a word size $w = 16$.

2) LR Inference: The LR benchmark has competitive timings across all the libraries for both EncInt and EncFP, save for TFHE, which is orders of magnitude slower, as shown in Fig. 4. Moreover, taking into consideration that all the other libraries allow multiple inferences in parallel using batching without incurring any execution time penalty, TFHE is not a good fit for this benchmark. Between the other libraries, HElib is slightly slower, but as the total execution time is around 1 second for every configuration, this difference is negligible.

3) Feedforward NN Inference: Fig. 5 demonstrates the timings for our neural network benchmark for both EncInt and EncFP with an increasing number of hidden neurons. This benchmark includes large numbers of ciphertext additions as well as multiplications between a ciphertext and plaintext value (since we assume that the cloud owns the network weights and can work with them in the clear). For EncInt, we observe HElib and SEAL being significantly faster than Lattigo and PALISADE, whereas in EncFP HElib outperforms all the libraries by more than an order of magnitude. This performance difference in EncFP is attributed to the way HElib automatically invokes the `rescale` operation, whereas in all the

[1]The evaluations of batched PIR and the squared Euclidean distance are shown in Appendix A.

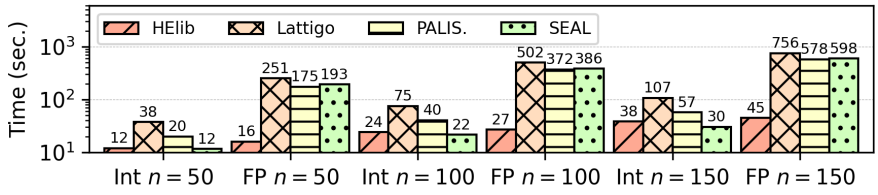
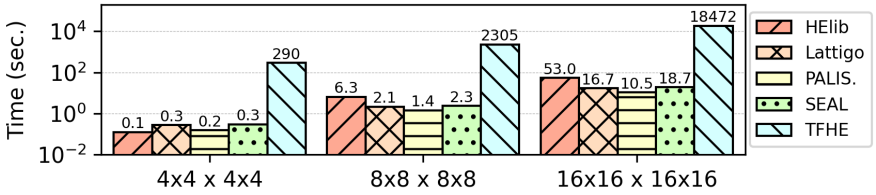
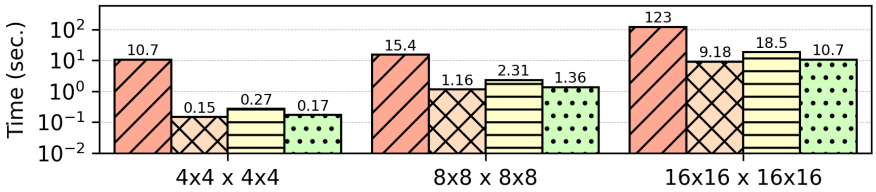


Fig. 5. Measured execution time for the neural network benchmark in both the integer (with EncInt) and floating-point (with EncFP) domains with an increasing number of hidden neurons.

other libraries we invoke these operations manually in the T2 code. We note that for other benchmarks where rescaling may not be strictly necessary, HElib suffers in performance because of this automated procedure. Finally, TFHE could not evaluate this benchmark in less than eight hours and thus is omitted from Fig. 5.



(a) Integer domain with EncBin for TFHE and EncInt for other libraries.



(b) Floating-point domain with EncFP.

Fig. 6. Measured execution time for the Matrix Multiplication benchmark for square matrices ranging from 4×4 to 16×16. TFHE word size $w = 8$.

4) Matrix Multiplication: For this benchmark, we use two square matrices in EncInt and EncFP. Similarly to the χ^2 benchmark, in Fig. 6 (a) we observe that all the libraries are at least three orders of magnitude faster than TFHE since the latter has to evaluate very expensive binary multiplication circuits to compute the product of multi-bit operands. Contrary to Fig. 5, Lattigo and PALISADE are the two fastest frameworks for integers. This showcases that these libraries have efficient multiplication implementations when multiplying two ciphertexts, while HElib and SEAL may be faster at multiplying ciphertexts with constants. Finally, in Fig. 6 (b) we observe that HElib is significantly slower than all the other libraries, which contradicts our results in Fig. 5. However, this performance difference is expected since in the matrix multiplication benchmark there is no need to invoking the `rescale` operation after

every multiplication operation, yet HElib invokes it automatically anyway, and suffers a performance slowdown.

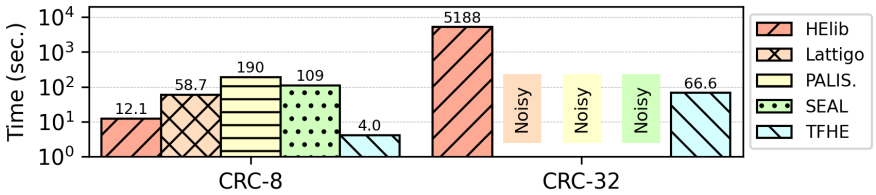


Fig. 7. Measured execution time for the CRC-8 and CRC-32 benchmarks in the binary domain with EncBin for word sizes of 8 and 32 bits, respectively.

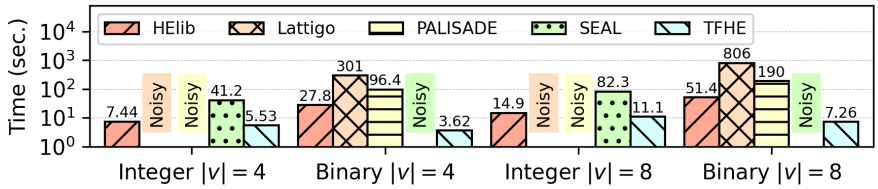


Fig. 8. Measured execution time for the Manhattan distance benchmark for both integer and binary domains for vectors with four and eight elements. The integer domain uses EncInt for every library except TFHE which uses EncBin, while the binary domain uses EncBin for all libraries. For the binary domain we used word size $w = 4$ as well as a binary optimization for the computation of the absolute value function.

B. Selected Bitwise Benchmarks[2]

1) CRC: Fig. 7 presents the timings of CRC-8 and CRC-32 evaluations using EncBin as it is not possible to use the integer or floating-point encodings due to the bitwise shift and XOR operations. All libraries were able to run CRC-8 without issues, yet for CRC-32 only the libraries that support bootstrapping were able to finish, namely HElib and TFHE, due to the deep multiplicative depth needed to support large 32-bit addition circuits. These results emphasize how much more efficient the bootstrapping mechanism is in TFHE versus HElib, and additionally, the restrictions that all LHE schemes have for deep algorithms. While the CRC-32 is theoretically executable with LHE, all LHE libraries do not support parameters large enough to achieve the required depth at 128 bits of security (i.e., depth is traded for security).

2) Binary Manhattan Distance: This benchmark includes a multiplexing operation to calculate the absolute value of the difference (described in section IV-B3) as it is far more efficient to evaluate a multiplexing operation using the sign bit of the difference between $V[i]$ and $U[i]$, than it is to employ

[2]See Appendix A for insertion sort.

an FHE "less than" followed by a MUX (as is done in the integer domain). Fig. 8 shows the timings for EncInt and EncBin. We note that TFHE (which is evaluated using both the integer and binary versions of the benchmark) performs better when using the binary-optimized algorithm while the other libraries perform better using the relational variant (section V-C2).

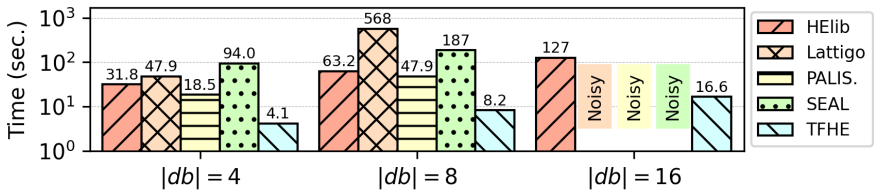


Fig. 9. Measured execution time for the BinPIR benchmark in the binary domain with EncBin and word size $w = 5$ for an increasing database size.

3) BinPIR: The binary PIR benchmark incurs a significant overhead compared to the batched version (BaPIR), as shown in Fig. 9. More specifically, we observe that Lattigo, PALISADE, and SEAL can only run with a database of 8 elements due to significant noise growth, which is unrealistic. HElib and TFHE scale linearly to the number of elements, with the latter being one order of magnitude faster than the former. This benchmark demonstrates the superiority of the BaPIR benchmark for the libraries that support batching.

C. Relational Benchmarks

1) Hamming Distance: For libraries with built-in support for binary arithmetic (i.e., HElib and TFHE), this benchmark is more efficient using EncBin. However, for the rest of the libraries EncInt is faster and exhibits a lower depth than the binary approach. Notably, TFHE drastically outperforms all implementations for both domains, as depicted in Fig. 10.

2) Relational Manhattan Distance: The relational version of Manhattan distance differs from the binary case because direct extraction of the sign of an encrypted integer is expensive. We note that the integer version of the multiplexer follows the same equation as the binary case, with the caveat that the control input must be $\hat{0}$ or $\hat{1}$. As shown in Fig. 8, Lattigo and PALISADE are unable to evaluate this benchmark using EncInt because they require $p - 1$ to be a prime that evenly divides the cyclotomic polynomial degree m. Since the depth of the LT circuit scales linearly with the bit size of p, ideally one needs a small poly degree to allow for a relatively smaller p. However, small poly degrees do not yield enough levels to evaluate the circuit, which requires larger parameters and forces the user to employ a larger p.

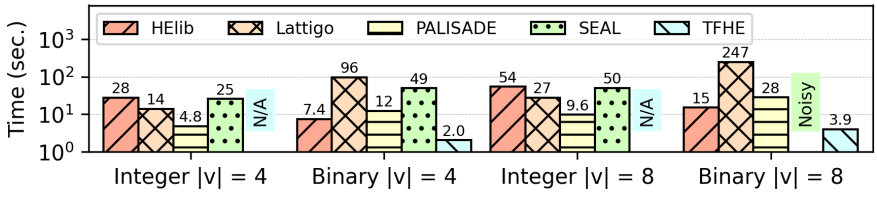


Fig. 10. Measured execution time for the Hamming distance benchmark for both integer and binary domains for vectors with four and eight elements. All libraries use ENCINT for the integer domain except for TFHE, which uses ENCBIN. All libraries in ENCBIN use a word size $w = 4$.

VI. DISCUSSION & LESSONS LEARNED

ENCBIN: From the results presented in Table II and the T2 benchmarks, we can glean several important insights about the performance and viability of different FHE libraries and schemes in a variety of situations. First, it is clear that TFHE is the optimal choice for the binary encoding as it can evaluate both arithmetic and relational Boolean circuits faster than any other FHE library and it boasts the smallest ciphertext size by a wide margin. For deep circuits, such as the CRC-32 benchmark, TFHE performs far better than HElib, which has the only other bootstrappable implementation in the binary encoding under study. This is primarily due to the vastly superior bootstrapping speeds of TFHE compared to the BGV bootstrapping in HElib. There is, however, one case that TFHE might not result in the fastest evaluation: i.e., when the user wants to evaluate multiple instances of the benchmark in parallel. For this case, HElib is the best choice as it allows for efficient binary gate evaluations and has the unique ability to use $p = 2$ in a batching context. Thus, although TFHE has by far the smallest latency for one instance, when the core algorithm has to be repeated x times, for a relatively small x, the latency of TFHE is often larger than the latency of HElib, which can extend to x instances at the same cost as $x = 1$.

ENCINT & ENCFP: From Table II, it is clear that PALISADE exhibits the fastest overall speeds for

TABLE III
FEATURES, PARAMETERIZATION, AND EASE-OF-USE OF FHE LIBRARIES.

Library	Mod Chain Control	Automated Ctxt Maintenance	Configurable Security	Control of Cyclotomic
HElib	○	●	●	●
Lattigo	●	○	●	○
PALISADE	●	●	●	○
SEAL	●	○	○	○
TFHE	○	●	○	○

both the integer and the floating-point encodings, but based on the benchmark results this is not always the case. For algorithms that include relational operators, PALISADE and Lattigo are poor choices for ENCINT as they require large plaintext moduli. Because the depth of the comparison circuits scale with the plaintext modulus size, these operations quickly become prohibitively expensive in terms of noise, as illustrated in the case of the relational Manhattan distance. For these types of applications, HElib is a better option since it enables batching with much smaller choices of p. In the floating-point domain, when rescaling frequently is required, HElib becomes the best option due to its fast, automatic rescaling procedure as is shown in the case of the neural network inference benchmark.

Domain selection: We can deduce that ENCBIN is ill-suited for arithmetic-heavy benchmarks; for instance, a 16x16 $\times$ 16x16 matrix multiplication took less than a minute for most libraries in ENCINT while TFHE was three orders of magnitude slower using ENCBIN. However, algorithms that include right shifts and bitwise rotations are only possible in ENCBIN. In general, we found that most libraries had similar timings for both ENCINT and ENCFP, with the exception of the neural network inference due to rescaling. Some applications, such as full-precision neural networks, are better suited for the ENCFP as inputs and weights do not need to be scaled or quantized.

Parameter selection: While meticulously minimizing the parameters required in each library for our T2 benchmarks, we have found that all libraries are not created equal when it comes to selecting parameter sets. Table III depicts our findings with respect to four important criteria related to parameter selection. The first consideration involves the ciphertext modulus q; some libraries allow users to fine-tune the sizes of individual primes as well as the total bit size of q, which can affect both performance, security, and the number of levels. Next, automated ciphertext maintenance, referring primarily to modulus switching/rescaling and bootstrapping, is vital for reducing the difficulty of developing FHE applications, particularly for non-experts. For instance HElib and TFHE can fully automate these procedures and will invoke them behind the scenes. Conversely, Lattigo and SEAL will require operations like key-switching and rescaling to be invoked manually; while this could be useful in rare cases, it may become easier for a developer to schedule these operations sub-optimally and incur a significant performance hit. Configurable security refers to the ability to select parameter sets at several security levels in order to allow users to balance security versus efficiency for their specific applications. Lastly, control of the cyclotomic refers to the ability to directly manipulate the degree of the ciphertext polynomials; nearly all libraries allow you to specify the ring-dimension N, but we note that this parameter is related, but distinct from the actual cyclotomic order.

There will always be a trade-off between ease of use and freedom when it comes to parameterization. From a usability standpoint, we argue that although Lattigo and SEAL may be beginner-friendly due

to limited parameterization options, HElib offers the most flexibility, while PALISADE is somewhere in between. PALISADE has a flexible API for instantiating FHE parameters, with a number of optional parameters and settings that users can tweak, which benefits both new users and experienced developers. The downside is that it is impossible to directly set the cyclotomic order and non-power of 2 cyclotomic orders are not fully supported. For that matter, TFHE is also easy to configure, but it is not straightforward to use for a non-expert as it does not include any arithmetic functional units (e.g., multi-bit adder/multiplier).

Next steps for FHE libraries: Based on our experience, there is a need for all libraries to introduce standardized binary circuits for common functionalities as the performance of binary FHE is almost entirely dependent on the underlying Boolean circuits. Currently, only HElib natively supports EncBin operations, rendering it the most versatile library. Unfortunately, although TFHE is meant for EncBin, it does not include any elaborate circuit, only two-input logic gates. As EncBin enables benchmarks not possible in other domains, we expect more libraries to follow HElib's footsteps in the future and include Boolean addition and comparison circuits. Additionally, it is crucial to come up with better comparisons for the integer and floating-point encodings, as currently comparisons are not viable for real-world applications. Latest trends in encrypted computation circumvent this by avoiding comparisons (such as PPML applications), but faster comparisons will enable several new HE applications.

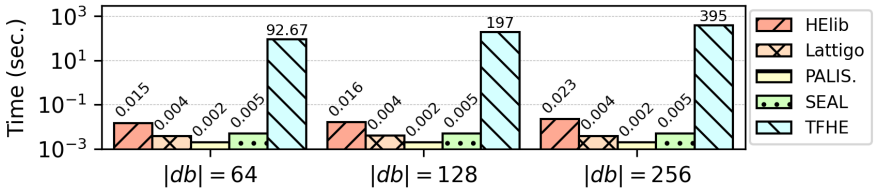
Finally, while TFHE exhibits fast bootstrapping for non-batched binary ciphertexts, batchable schemes such as BFV and CKKS exhibit incredibly slow speeds for bootstrapping. Therefore, a future research direction in HE should be to improve bootstrapping speeds while maintaining a high degree of batching, either by proposing new bootstrapping mechanisms or by using hardware acceleration. While custom hardware for FHE is actively being developed, current hardware platforms such as GPUs show great promise for accelerating FHE operations [64], [75]. Going forward, FHE libraries should continue to incorporate support for these devices as well as other emergent dedicated hardware units.

Appendix A
Additional Experimental Results

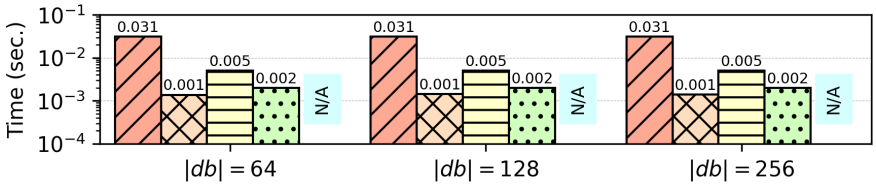
For completeness, we report additional performance results for the batched PIR, the squared Euclidean distance, and the binary insertion sort benchmarks.

A. Arithmetic Benchmarks

BaPIR: Fig. 11 shows the execution time for the batched PIR benchmark for databases with 64, 128, and 256 entries. Notably, save for TFHE, all frameworks have approximately constant time for the different



(a) Integer domain with EncBin for TFHE and EncInt for other libraries.



(b) Floating-point domain with EncFP.

Fig. 11. Measured execution time for the PIR benchmark for integer and floating-point domains for an increasing database size. TFHE uses EncBin with a word size of 5 in the integer domain.

database sizes since they all fit in the slots of one ciphertext and therefore the number of operations is identical across all variants. Since TFHE does not support batching, it incurs a significant performance overhead, shown with the blue bar in Fig. 11 (a). This benchmark highlights the two core advantages of all the arithmetic schemes compared to RGSW (i.e., the underlying scheme in TFHE). First, the schemes with batching capability perform only a single multiplication compared to RGSW, which needs to perform as many multiplications as the database size. Secondly, as EncBin has to perform binary multiplications, it incurs additional overhead compared to the fast multiplications in EncInt and EncFP for all the other libraries.

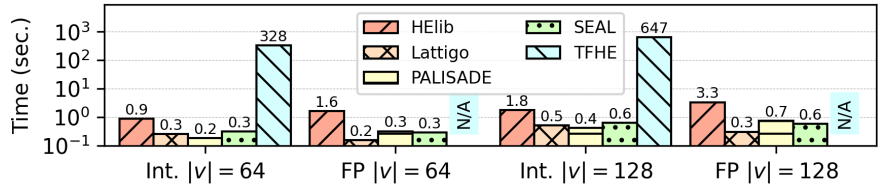


Fig. 12. Measured execution time for the Euclidean distance benchmark for both integer and floating-point domains for vectors with 64, 128, and 256 elements. The integer domain uses EncInt for the backends, except for TFHE which uses EncBin with word size $w = 8$. The floating-point domain uses EncFP for the relevant backends.

Squared Euclidean Distance: This benchmark comprises one subtraction, one multiplication, and one addition for every two elements of the input vectors, rendering it a representative benchmark for core arithmetic operations. Fig. 12 shows the runtime performance for both EncInt and EncFP with vector sizes of 64 and 128 elements. Lattigo, PALISADE, and SEAL are the three fastest libraries, with HElib

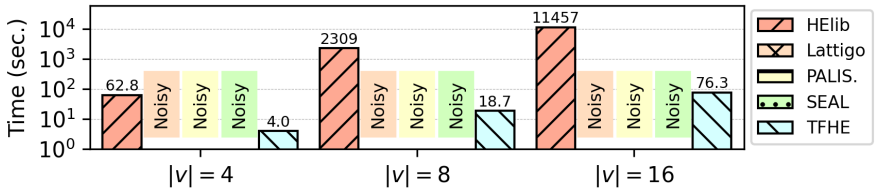


Fig. 13. Measured execution time for the insertion sort benchmark for different input array sizes with word size $w = 4$ in the binary domain using EncBin for all libraries.

having competitive performance, and TFHE being orders of magnitude slower because of the large binary circuits needed. Additionally, this benchmark can take advantage of batching and compare multiple sets of vectors, further accelerating the amortized performance of the baseline algorithm for all libraries except TFHE, which will incur a linear overhead that scales with the total number of elements.

B. Bitwise Benchmarks

Insertion Sort: As in CRC-32 shown in Fig. 7, insertion sort is a very deep benchmark in terms of multiplicative depth, particularly for the first elements in the array as it iterates through each element and performs multiplexing procedures with all prior elements during each iteration. Recall from Section III-E that a multiplexing operation has depth marginally over one, (since two depth-1 ciphertexts are added together, resulting in slightly more noise accumulation). Thus, as shown in Fig. 13, only HElib and TFHE were able to finish this benchmark as they are the only two libraries supporting bootstrapping for EncBin. TFHE is significantly faster than HElib due to its fast bootstrapping speeds and the much smaller ciphertext polynomial degrees employed by the library. Although this benchmark incurs high overheads even for an array of 16 elements, we emphasize the need of faster bootstrapping and enhanced comparison units for all the libraries.

Appendix B
Future Directions of Standardized FHE Benchmarks

The T2 benchmark suite comprises a plethora of representative applications of FHE at the current time of writing. However, FHE is still in its infancy and continues to expand and become more efficient in terms of memory and computational overhead each year. As new FHE schemes and accelerators are developed, the use-cases for this powerful technology will inevitably continue to grow and branch into more industries. As such, we have built the T2 compiler *with extensibility in mind to incorporate new FHE schemes as they arise* and we plan to expand T2 in the future to reflect new trends in FHE. Our goal with T2 is to provide a benchmark suite that enables easy comparisons between FHE libraries

and we foresee future research to significantly benefit from our standardized benchmarks and their easy deployment for a variety of FHE back-ends.

We also aim to draw attention to open problems in FHE using T2, an including automated parameterization for general purpose computation. Finding the smallest parameter set that works for a given application and results in the fastest execution time is a time-consuming process that requires a lot of trial-and-error to get right. A mechanism to automate this process without actually running anything in the encrypted domain would go a long way to streamline the incorporation of new benchmarks and make comparisons between schemes and libraries more fair.

REFERENCES

[1] C. Duffy, "One of big tech's top moneymakers is getting a pandemic boost," CNN Business, 2020, https://www.cnn.com/2020/10/05/tech/cloud-growth-coronavirus/index.html.

[2] E. Brier and M. Joye, "Weierstraß elliptic curves and side-channel attacks," in *International workshop on public key cryptography*. Springer, 2002, pp. 335–345.

[3] H. Hlavacs, T. Treutner, J.-P. Gelas, L. Lefevre, and A.-C. Orgerie, "Energy consumption side-channel attack at virtual machines in a cloud," in *2011 IEEE Ninth International Conference on Dependable, Autonomic and Secure Computing*. IEEE, 2011, pp. 605–612.

[4] Y. Zhang *et al.*, "Cross-VM Side Channels and Their Use to Extract Private Keys," in *Computer and Communications Security*. ACM, 2012, pp. 305–316.

[5] Y. Xiao, X. Zhang, Y. Zhang, and R. Teodorescu, "One bit flips, one cloud flops: Cross-vm row hammer attacks and privilege escalation," in *USENIX Security Symposium*, 2016, pp. 19–35.

[6] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher *et al.*, "Spectre attacks: Exploiting speculative execution," in *Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 1–19.

[7] A. Bhattacharyya, A. Sandulescu, M. Neugschwandtner, A. Sorniotti, B. Falsafi, M. Payer, and A. Kurmus, "Smotherspectre: exploiting speculative execution through port contention," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 785–800.

[8] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx, "Foreshadow: Extracting the keys to the intel SGX kingdom with transient {Out-of-Order} execution," in *USENIX Security Symposium*, 2018, pp. 991–1008.

[9] Y. Jin, N. Kupp, and Y. Makris, "Experiences in hardware trojan design and implementation," in *International Workshop on Hardware-Oriented Security and Trust*. IEEE, 2009, pp. 50–57.

[10] Y. Jin, M. Maniatakos, and Y. Makris, "Exposing vulnerabilities of untrusted computing platforms," in *2012 IEEE 30th International Conference on Computer Design (ICCD)*. IEEE, 2012, pp. 131–134.

[11] N. G. Tsoutsos, C. Konstantinou, and M. Maniatakos, "Advanced Techniques for Designing Stealthy Hardware Trojans," in *Design Automation Conference*. ACM/EDAC/IEEE, 2014, pp. 1–4.

[12] R. Chow *et al.*, "Controlling Data in the Cloud: Outsourcing Computation without Outsourcing Control," in *Workshop on Cloud Computing Security*. ACM, 2009, pp. 85–90.

[13] H. Takabi, J. B. Joshi, and G.-J. Ahn, "Security and Privacy Challenges in Cloud Computing Environments," *IEEE Security & Privacy*, vol. 8, no. 6, pp. 24–31, 2010.

[14] R. A. Popa, C. M. Redfield, N. Zeldovich, and H. Balakrishnan, "Cryptdb: protecting confidentiality with encrypted query processing," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, 2011, pp. 85–100.

[15] R. Poddar, T. Boelter, and R. A. Popa, "Arx: A strongly encrypted database system." *IACR Cryptol. ePrint Arch.*, vol. 2016, p. 591, 2016.

[16] A. C. Yao, "Protocols for secure computations," in *Symposium on Foundations of Computer Science (SFCS)*. IEEE, 1982, pp. 160–164.

[17] A. C.-C. Yao, "How to generate and exchange secrets," in *Symposium on Foundations of Computer Science (SFCS)*. IEEE, 1986, pp. 162–167.

[18] O. Goldreich and Y. Oren, "Definitions and properties of zero-knowledge proof systems," *Journal of Cryptology*, vol. 7, no. 1, pp. 1–32, 1994.

[19] E. Ben-Sasson, A. Chiesa, E. Tromer, and M. Virza, "Succinct {Non-Interactive} zero knowledge for a von neumann architecture," in *USENIX Security Symposium*, 2014, pp. 781–796.

[20] D. Mouris and N. G. Tsoutsos, "Zilch: A Framework for Deploying Transparent Zero-Knowledge Proofs," *IEEE Transactions on Information Forensics and Security*, vol. 16, pp. 3269–3284, 2021.

[21] C. Gentry, "A Fully Homomorphic Encryption Scheme," Ph.D. dissertation, Stanford University, 2009.

[22] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, "(Leveled) fully homomorphic encryption without bootstrapping," in *Innovations in Theoretical Computer Science Conference*, 2012, pp. 309–325.

[23] J. Fan and F. Vercauteren, "Somewhat practical fully homomorphic encryption." Cryptology ePrint Archive, Report 2012/144, 2012, http://eprint.iacr.org/2012/144.

[24] M. Brenner *et al.*, "Secret Program Execution in the Cloud Applying Homomorphic Encryption," in *Digital Ecosystems and Technologies Conference*. IEEE, 2011, pp. 114–119.

[25] D. Zhuravlev *et al.*, "Encrypted program execution," in *2014 IEEE 13th International Conference on Trust, Security and Privacy in Computing and Communications*, 2014, pp. 817–822.

[26] A. Chatterjee and I. Sengupta, "Translating algorithms to handle fully homomorphic encrypted data on the cloud," *IEEE Transactions on Cloud Computing*, vol. 6, no. 1, pp. 287–300, 2015.

[27] D. Mouris, N. G. Tsoutsos, and M. Maniatakos, "Terminator suite: Benchmarking privacy-preserving architectures," *IEEE Computer Architecture Letters*, vol. 17, no. 2, pp. 122–125, 2018.

[28] A. Viand, P. Jattke, and A. Hithnawi, "SoK: Fully homomorphic encryption compilers," in *Symposium on Security and Privacy (SP)*. IEEE, 2021, pp. 1092–1108.

[29] J. L. Henning, "SPEC CPU2006 Benchmark Descriptions," *ACM SIGARCH Computer Architecture News*, vol. 34, no. 4, pp. 1–17, 2006.

[30] A. Arnold and M. Monagan, "Calculating cyclotomic polynomials," *Mathematics of Computation*, vol. 80, no. 276, pp. 2359–2379, 2011.

[31] O. Regev, "On lattices, learning with errors, random linear codes, and cryptography," *Journal of the ACM (JACM)*, vol. 56, no. 6, pp. 1–40, 2009.

[32] V. Lyubashevsky, C. Peikert, and O. Regev, "On ideal lattices and learning with errors over rings," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2010, pp. 1–23.

[33] J. H. Cheon, A. Kim, M. Kim, and Y. Song, "Homomorphic encryption for arithmetic of approximate numbers," in *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 2017, pp. 409–437.

[34] S. Halevi, Y. Polyakov, and V. Shoup, "An improved rns variant of the bfv homomorphic encryption scheme," in *Topics in Cryptology – CT-RSA 2019*, M. Matsui, Ed. Cham: Springer International Publishing, 2019, pp. 83–105.

[35] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène, "TFHE: fast fully homomorphic encryption over the torus," *Journal of Cryptology*, vol. 33, no. 1, pp. 34–91, 2020.

[36] C. Gentry, A. Sahai, and B. Waters, "Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based," in *Annual Cryptology Conference*. Springer, 2013, pp. 75–92.

[37] R. Geelen, "Bootstrapping Algorithms for BGV and FV," Ph.D. dissertation, KU Leuven, 2021.

[38] I. Chillotti, M. Joye, and P. Paillier, "Programmable bootstrapping enables efficient homomorphic inference of deep neural networks," in *Cyber Security Cryptography and Machine Learning*, S. Dolev, O. Margalit, B. Pinkas, and A. Schwarzmann, Eds. Cham: Springer International Publishing, 2021, pp. 1–19.

[39] L. Ducas and D. Micciancio, "FHEW: bootstrapping homomorphic encryption in less than a second," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2015, pp. 617–640.

[40] S. Halevi and V. Shoup, "Algorithms in HElib," in *Annual Cryptology Conference*. Springer, 2014, pp. 554–571.

[41] "Lattigo v3.0.2," Online: https://github.com/tuneinsight/lattigo, Feb. 2022, ePFL-LDS, Tune Insight SA.

[42] Y. Polyakov, K. Rohloff, and G. W. Ryan, "Palisade lattice cryptography library user manual," *Cybersecurity Research Center, New Jersey Institute of Technology (NJIT), Tech. Rep*, vol. 15, 2017.

[43] "Microsoft SEAL (release 4.0)," https://github.com/Microsoft/SEAL, Mar. 2022, microsoft Research, Redmond, WA.

[44] N. P. Smart and F. Vercauteren, "Fully homomorphic simd operations," *Designs, codes and cryptography*, vol. 71, no. 1, pp. 57–81, 2014.

[45] F. Boemer, R. Cammarota, D. Demmler, T. Schneider, and H. Yalame, "MP2ML: a mixed-protocol machine learning framework for private inference," in *Proceedings of the 15th International Conference on Availability, Reliability and Security*, 2020, pp. 1–10.

[46] E. J. Chou, A. Gururajan, K. Laine, N. K. Goel, A. Bertiger, and J. W. Stokes, "Privacy-preserving phishing web page classification via fully homomorphic encryption," in *ICASSP 2020-2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 2020, pp. 2792–2796.

[47] W. Wang, Y. Hu, L. Chen, X. Huang, and B. Sunar, "Exploring the feasibility of fully homomorphic encryption," *IEEE Transactions on Computers*, vol. 64, no. 3, pp. 698–706, 2013.

[48] C. Aguilar-Melchor, J. Barrier, S. Guelton, A. Guinet, M.-O. Killijian, and T. Lepoint, "NFLlib: NTT-based fast lattice library," in *Cryptographers' Track at the RSA Conference*. Springer, 2016, pp. 341–356.

[49] V. Shoup *et al.*, "NTL: A library for doing number theory," 2001.

[50] J.-W. Lee, E. Lee, Y. Lee, Y.-S. Kim, and J.-S. No, "High-precision bootstrapping of RNS-CKKS homomorphic encryption using optimal minimax polynomial approximation and inverse sine function," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2021, pp. 618–647.

[51] I. Chillotti, M. Joye, and P. Paillier, "Programmable bootstrapping enables efficient homomorphic inference of deep neural networks," in *International Symposium on Cyber Security Cryptography and Machine Learning*. Springer, 2021, pp. 1–19.

[52] A. Acar, H. Aksu, A. S. Uluagac, and M. Conti, "A survey on homomorphic encryption schemes: Theory and implementation," *ACM Computing Surveys (CSUR)*, vol. 51, no. 4, pp. 1–35, 2018.

[53] S. Carpov, P. Dubrulle, and R. Sirdey, "Armadillo: a compilation chain for privacy preserving applications," in *Proceedings of the 3rd International Workshop on Security in Cloud Computing*, 2015, pp. 13–19.

[54] R. Dathathri, O. Saarikivi, H. Chen, K. Laine, K. Lauter, S. Maleki, M. Musuvathi, and T. Mytkowicz, "CHET: an optimizing

compiler for fully-homomorphic neural-network inferencing," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2019, pp. 142–156.

[55] R. Dathathri, B. Kostova, O. Saarikivi, W. Dai, K. Laine, and M. Musuvathi, "Eva: An encrypted vector arithmetic language and compiler for efficient homomorphic computation," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2020, pp. 546–561.

[56] E. Chielle, O. Mazonka, H. Gamil, N. G. Tsoutsos, and M. Maniatakos, "E3: A framework for compiling C++ programs with encrypted operands," Cryptology ePrint Archive, Report 2018/1013, 2018, http://eprint.iacr.org/2018/1013.

[57] A. Viand and H. Shafagh, "Marble: Making fully homomorphic encryption accessible to all," in *Proceedings of the 6th Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, 2018, pp. 49–60.

[58] S. Gorantala *et al.*, "A general purpose transpiler for fully homomorphic encryption," arXiv preprint, arXiv:2106.07893, 2021, https://arxiv.org/abs/2106.07893.

[59] C. Gouert and N. G. Tsoutsos, "Romeo: conversion and evaluation of hdl designs in the encrypted domain," in *2020 57th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2020, pp. 1–6.

[60] S. Halevi and V. Shoup, "Bootstrapping for HElib," in *Advances in Cryptology – EUROCRYPT 2015*, E. Oswald and M. Fischlin, Eds. Springer Berlin Heidelberg, 2015, pp. 641–670.

[61] I. Iliashenko and V. Zucca, "Faster homomorphic comparison operations for BGV and BFV." *Proc. Priv. Enhancing Technol.*, vol. 2021, no. 3, pp. 246–264, 2021.

[62] G. S. Çetin, Y. Doröz, B. Sunar, and E. Savas, "Low depth circuits for efficient homomorphic sorting." Cryptology ePrint Archive, Report 2015/274, 2015, http://eprint.iacr.org/2015/274.

[63] K. Lauter, A. López-Alt, and M. Naehrig, "Private computation on encrypted genomic data," in *International Conference on Cryptology and Information Security in Latin America*. Springer, 2014, pp. 3–27.

[64] L. Folkerts, C. Gouert, and N. G. Tsoutsos, "REDsec: Running encrypted DNNs in seconds," Cryptology ePrint Archive, Report 2021/1100, 2021, https://ia.cr/2021/1100.

[65] A. Genkin, D. D. Lewis, and D. Madigan, "Large-scale bayesian logistic regression for text categorization," *technometrics*, vol. 49, no. 3, pp. 291–304, 2007.

[66] M. Kim, Y. Song, S. Wang, Y. Xia, X. Jiang *et al.*, "Secure logistic regression based on homomorphic encryption: Design and evaluation," *JMIR medical informatics*, vol. 6, no. 2, p. e8805, 2018.

[67] J. Liu, M. Juuti, Y. Lu, and N. Asokan, "Oblivious neural network predictions via minionn transformations," in *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*, 2017, pp. 619–631.

[68] R. Gilad-Bachrach, N. Dowlin, K. Laine, K. Lauter, M. Naehrig, and J. Wernsing, "Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy," in *International conference on machine learning*. PMLR, 2016, pp. 201–210.

[69] C. Aguilar-Melchor, J. Barrier, L. Fousse, and M.-O. Killijian, "XPIR: Private information retrieval for everyone," *Proceedings on Privacy Enhancing Technologies*, vol. 2, no. 2016, pp. 155–174, 2016.

[70] S. Angel, H. Chen, K. Laine, and S. Setty, "PIR with compressed queries and amortized query processing," in *Symposium on Security and Privacy (SP)*. IEEE, 2018, pp. 962–979.

[71] M. Green, W. Ladd, and I. Miers, "A protocol for privately reporting ad impressions at scale," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016, pp. 1591–1601.

[72] N. Borisov, G. Danezis, and I. Goldberg, "DP5: A Private Presence Service." *Proc. Priv. Enhancing Technol.*, vol. 2015, no. 2, pp. 4–24, 2015.

[73] R. Ostrovsky and W. E. Skeith, "Private searching on streaming data," in *Annual International Cryptology Conference*. Springer, 2005, pp. 223–240.

[74] R. L. Carter, R. Morris, and R. K. Blashfield, "On the partitioning of squared euclidean distance and its applications in cluster analysis," *Psychometrika*, vol. 54, no. 1, pp. 9–23, 1989.

[75] N. Samardzic, A. Feldmann, A. Krastev, S. Devadas, R. Dreslinski, C. Peikert, and D. Sanchez, "F1: A fast and programmable accelerator for fully homomorphic encryption," in *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2021, pp. 238–252.