

# SoK: New Insights into Fully Homomorphic Encryption Libraries via Standardized Benchmarks

Charles Gouert\*, Dimitris Mouris\*, and Nektarios Georgios Tsoutsos

University of Delaware  
{cgouert, jimouris, tsoutsos}@udel.edu

**Abstract.** Fully homomorphic encryption (FHE) enables arbitrary computation on encrypted data, allowing users to upload ciphertexts to cloud servers for computation while mitigating 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 allow programmers to effectively utilize the power of FHE, we employ a series of benchmarks 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 secure computational 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).

**Keywords:** Benchmarking, data privacy, encrypted computation, fully homomorphic encryption, performance evaluation

## 1 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 their vast computing resources. The minimal capital expenditures and low yearly costs of this model have compelled companies to shift from maintaining private data centers and computing hardware to adopting the cloud computing-as-a-service for their operations [36]. 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. 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 [18,51], cross-VM attacks [87,84], speculative execution attacks [57,11,79], and hardware Trojans in the supply chain [53,54,77] have demonstrated that it is possible to leak *data-in-use* from remote servers.

A standard and widely deployed method to protect data confidentiality for data-in-transit and data-at-rest is the use of cryptography [29,76]. 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 [70,68]. To perform any computation on outsourced data, clients must download the encrypted data, decrypt it, perform the intended

---

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

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 if sensitive data is involved.

Candidate cryptographic protocols for privacy-preserving computation include homomorphic encryption, multi-party computation (MPC), and zero-knowledge proofs. MPC allows multiple parties to jointly compute an algorithm over private data [85,86], but has a somewhat weaker threat model as colluding servers may be able to decrypt the users’ data. On the other hand, zero knowledge proofs allow one party to obliviously prove to another party that a statement is true, without revealing any additional information [9,64,83]. Finally, fully homomorphic encryption (FHE) enables *oblivious computation on the cloud* and protects *data-in-use*. In more detail, FHE allows the cloud to apply arbitrary oblivious 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 an arbitrary algorithm as a netlist of arithmetic operations in the encrypted domain [41], and each scheme has multiple open-source implementations using state-of-the-art cryptographic libraries. Notably, each FHE scheme has unique characteristics, including how data is encoded and the types of operations available. For instance, schemes such as Brakerski-Gentry-Vaikuntanathan (BGV) [15] and Brakerski/Fan-Vercauteren (BFV) [37] encrypt integers reduced by a chosen plaintext modulus, while others encrypt floating-point numbers or even individual bits. Such encoding determines the nature of the operations, since ciphertexts encoding floating-point and integer values allow multi-bit addition and multiplication, while ciphertexts encoding bits support Boolean logic operations instead. Therefore, expressing an algorithm in the encrypted domain requires different techniques 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 and runtime decisions are impossible when the control values are encrypted. For the former, 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. For the latter, 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” [17,88,22,65,80]).

Our main goal is to provide new 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 popular general computation benchmark suites, such as the SPEC benchmark suite [50]. 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 introduce a novel 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. Notably, we employ benchmarks that invoke core arithmetic operations between ciphertexts, mixed operations between



Fig. 1. Our T2 compiler for five major FHE backends.

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 across different libraries. While tailored implementations of encrypted programs for a certain FHE library can help fine tune runtime performance, this does not translate among different FHE schemes (often with incompatible configurations). Further, subjective factors, such as programmers’ experience on one library can further skew the comparisons. Therefore, our methodology relies on *automated compilation* of each benchmarking algorithm using 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 can compile arbitrary algorithms into their corresponding encrypted form using five state-of-the-art FHE libraries as backends (Fig. 1). Our compiler maps high-level code to functional blocks that implement identical arithmetic/Boolean circuits for given operations across all backends. We have analyzed all common FHE operations and developed optimized implementations using individual features of each scheme, which the T2 compiler uses to generate encrypted programs. While other works have investigated theoretical differences between FHE schemes [30], our work systematically compares the differences among FHE libraries.

**Overview:** The rest of the paper is organized as follows: Section 2 provides a primer on the homomorphic encryption theory and practice. Section 3 introduces the T2 compiler for fair comparisons of HE backends and Section 4 explains the algorithms included in the T2 benchmarks. Further, Section 5 evaluates all studied backends for both primitive operations and T2 benchmarks, while Section 6 presents insights gleaned from the experiments. Lastly, Appendices A, B, and C include discussions on parameter selection, future directions, and additional T2 benchmarks followed by experimental comparisons with other state-of-the-art HE compilers.

## 2 Preliminaries

### 2.1 Homomorphic Encryption (HE)

HE is a powerful form of encryption that allows for arithmetic operations over ciphertexts where the result is an encryption of the expected answer. More formally, the computational capability of HE and the equivalency between encrypted and plaintext computation is shown as:  $y = f(x) \Leftrightarrow y = Dec(g(Enc(x)))$ . 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  $g(Enc(x))$  which is equivalent to an encryption of  $y$ . Finally, the user can decrypt the result with her private key. All algorithms can be expressed as an arithmetic or Boolean *circuit* (i.e., a series of additions and multiplications or a netlist of logic gates), which can be readily executed with FHE.

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

**Ciphertext Encoding:** FHE schemes inject noise into ciphertexts in accordance with the Learning With Errors (LWE) problem [71,4] to make them resilient to cryptanalyses, and the induced noise must be kept below a threshold to ensure successful decryption. The hardness of LWE and its variant Ring LWE (RLWE) [62] guarantee the security of all known FHE schemes. In more detail, the RLWE problem involves solving for  $s \in R_q$  in the equation  $b = a \cdot s + e$  where  $b, a \in R_q$  are known and  $e$  is an error term sampled from a distribution over  $R_q$ .  $R_q$  is defined as the ring  $\mathbb{Z}_q[x]/\langle x^n + 1 \rangle$ , where  $n$  is a power of 2 and  $q$  is a prime number that satisfies  $q \equiv 1 \pmod{2n}$ . Both LHE and FHE schemes that are based on the RLWE problem encrypt data as a tuple of polynomials modulo an irreducible cyclotomic polynomial, where the  $N^{\text{th}}$  cyclotomic polynomial has roots corresponding to the  $N^{\text{th}}$  primitive roots of unity [6]. 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. Operations over ciphertexts take the form of polynomial addition and multiplication as a result of this encoding.

Because ciphertext multiplication is akin to multiplying tuples of polynomials, the product of ciphertexts becomes a set of three polynomials modulo the irreducible cyclotomic polynomial. This means that multiplying two ciphertexts results in *ciphertext expansion* and causes the total size of the ciphertext to increase each time a product is computed. Operations on expanded ciphertexts become significantly more expensive and the memory requirements skyrocket as computation progresses. Yet another consequence of this multiplication is that the output product is encrypted under a different key (essentially the square of the secret key). To mitigate these problems, LHE and FHE schemes incorporate *key switching*, or relinearization, that maps the output ciphertext to an encryption under the original secret key and at the same time reduces the dimensionality of the product ciphertext back to a tuple of polynomials [37]. This operation is also noisy and exacerbates the noise accumulation issue when performing computation with encrypted data in LHE and FHE schemes.

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 and encrypted multiplication causes exponential noise growth [41]. Therefore, there is only a fixed number of encrypted operations that can be executed on a ciphertext without exceeding the “noise budget” while still preserving the underlying plaintext data. Interestingly, the choice of security parameters affects noise growth and the total noise budget, but many realistic applications are not feasible with secure parameters. For instance, the SHE framework [61] used for LHE encrypted neural network inference takes approximately one day to classify an image with AlexNet. 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* [16] or *bootstrapping* [41] will reduce it and allow for additional operations. We delve into modulus switching and bootstrapping in the following paragraphs.

**LHE:** LHE is the first functionally complete type of HE that is capable of supporting addition and multiplication on encrypted data. Its primary mechanism for reducing noise is *modulus switching*, which effectively reduces the ciphertext size while also scaling down the noise. The coefficients of ciphertext polynomials are represented as integers modulo the product of primes  $q$ , where each prime is typically between 30-60 bits in length [72]; modulus switching scales  $q$  to a smaller value in order to reduce the magnitude of the noise. More specifically, this procedure removes a prime from  $q$ , effectively lowering the coefficient bit size. 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 *multiplicative levels* on any given ciphertext object before decryption fails [41]. BFV [37], BGV [15], and CKKS [23] are typically used as LHE schemes.

**FHE:** FHE is the most powerful form of HE that allows for unbounded addition and multiplication on encrypted data. This is made possible by introducing a *bootstrapping* mechanism to any LHE scheme, which is a powerful form of noise reduction. Gentry’s bootstrapping computes the decryption circuit in the encrypted domain resulting in a new ciphertext with reduced noise [41]. Unfortunately, bootstrapping is the bottleneck

of FHE; one must minimize the number of bootstraps in an application to optimize performance. Luckily, bootstrapping with modulus switching is 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 required to support a deep homomorphic circuit at an acceptable level of security are extremely large and result in poor runtime performance. Since the ciphertext 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  $q$  [47]. Using the Chinese remainder theorem, it is possible to do operations over these RNS polynomials (essentially one polynomial for each prime in  $q$ ) and then recombine the results to achieve the final polynomial with coefficients modulo  $q$ . Besides enabling arithmetic over smaller integers, this technique also enhances parallelism as all RNS polynomials can be operated on concurrently.

**Threat Model:** To enable 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. First, we assume an *honest-but-curious* cloud service provider that executes the protocol correctly but has incentives to peek at sensitive user data. Moreover, 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.

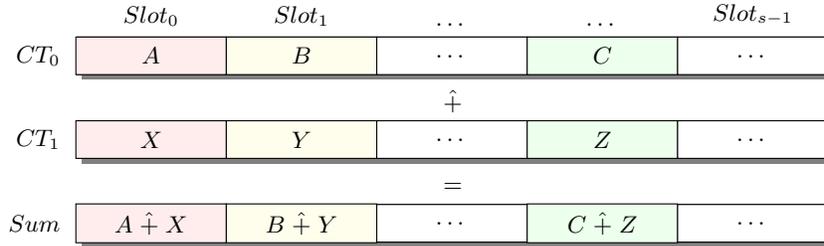
**Notation:** We represent the encryption of a variable  $x$  as  $X$ , while we use hat ( $\hat{\cdot}$ ) to represent encryptions of numbers, such as  $\hat{0}$  and  $\hat{1}$ . Similarly,  $\hat{+}$ ,  $\hat{-}$ ,  $\hat{*}$ , and  $\hat{\sim}$  represent homomorphic addition, subtraction, multiplication, and negation, respectively. These private arithmetic operations are inherently supported by all the schemes except CGGI, for which we use optimized Boolean circuits implementing encrypted arithmetic functional units. Specific circuit design details and considerations are discussed in Section 3.2. In binary arithmetic we also support homomorphic XOR between ciphertexts (represented as  $\hat{\oplus}$ ), as well as homomorphic AND.

Finally, we represent private comparisons as follows:  $\hat{=}$  denotes 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 one) if the corresponding plaintext operation is `True`, or  $\hat{0}$  otherwise. We discuss the design of private comparison units in Section 3.3 as most libraries do not support them natively.

## 2.2 Homomorphic Encryption Schemes

**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, such as shifting and logic gate evaluations. 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) [25] library implements a ring variant of GSW [42] called CGGI 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 (except for the trivial NOT gate).

The ability to compose algorithms as Boolean circuits is an advantage for binary schemes as developers can utilize decades of digital circuit design research to help build optimal implementations in the encrypted domain. Also, bitwise operations such as shifting are more efficient compared to the integer domain. For instance, a right shift by  $k$  in the binary domain involves 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



**Fig. 2. Batching Overview.** Each of the ciphertexts  $CT_0$  and  $CT_1$  encrypt multiple integers (i.e.,  $A, B, C, \dots$  and  $X, Y, Z, \dots$ , respectively), while the result ciphertext  $Sum$  contains the slot-wise homomorphic addition ( $\hat{+}$ ) of  $CT_0$  and  $CT_1$ . Homomorphic multiplication is similarly supported.

**Table 1.** 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		GSW
			Leveled/Fully <sup>†</sup>	Batching <sup>‡</sup>	Leveled/Fully <sup>†</sup>	Batching <sup>‡</sup>	Leveled/Fully <sup>†</sup>
Concrete [26]	Rust	Bin, FP	○	○	○	○	●
FHEW [35]	C, C++	Bin	○	○	○	○	●
HEAAN [23]	C, C++	FP	○	○	●	●	○
HElib [48]	C, C++	Int, Bin, FP	●	●	●	●	○
Lattigo [78]	Go	Int, FP	●	●	●	●	○
PALISADE [69]	C, C++	Int, Bin, FP	●	●	●	●	●
SEAL [72]	C, C++, .NET	Int, FP	●	●	●	●	○
TFHE [25]	C, C++	Bin	○	○	○	○	●

<sup>†</sup> For Leveled (LHE) and Fully (FHE) we categorize the libraries in four classes: ○ No support for either, ● Support for LHE only, ● Support for FHE only, and ● Support for both LHE and FHE.

<sup>‡</sup> 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  $p$ ).

compared to the binary scheme version. Finally, comparisons (i.e., equality, less-than) between two numbers are significantly cheaper in binary schemes.

**Integer Schemes:** The two most important HE schemes that encrypt integers modulo a user-determined modulus  $p$  are BGV [15] and BFV [14,37]. 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 CGGI, both BGV and BFV have considerably slow bootstrapping procedures; depending on parameter choices (such as the ring dimension), a single bootstrap can take anywhere from several seconds to several hours [39], which is impractical for any realistic HE computation. Thus, most implementations of BGV/BFV do 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 CGGI 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 [75]. 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 high memory and execution time overheads.

**Floating-Point Schemes:** The final class of HE has a plaintext type of floating-point numbers and the most popular scheme in this category is CKKS [23]. This is desirable for certain classes of algorithms, such as machine learning [12,28]. In practice, the core operations of floating-point schemes 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, *rescaling* must be invoked to preserve the original scale after multiplications, which is similar to modulus switching and serves a similar role to reduce ciphertext noise.

## 2.3 Homomorphic Encryption Libraries

Various open-source HE libraries implement the aforementioned schemes and expose a high-level API with functionalities like `Encrypt`, `Decrypt`, `KeyGen`, `Add`, and `Multiply`. Also, these libraries implement numerous performance optimizations (such as RNS and NTT/FFT for polynomial multiplication [82,3]) and leverage noise reduction techniques (e.g., modulus switching and bootstrapping) that are non-trivial to employ. Further, most libraries require the user to select encryption parameters for her application. In this paper, we focus on five widely used libraries, described below.

**HElib:** The Homomorphic Encryption Library (HElib) was introduced in 2013 [48] 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 [74].

**Lattigo:** The lattice-based multiparty homomorphic encryption library in Go (Lattigo) was first developed by the Laboratory for Data Security (LDS) at EPFL and is currently maintained by Tune Insight [78]. 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.

**PALISADE:** PALISADE was developed by Duality, NJIT, MIT, and other organizations [69]. It offers support for leveled BFV, BGV, and CKKS with RNS optimizations as well as CGGI (with bootstrapping).

**SEAL:** The Simple Encrypted Arithmetic Library (SEAL) is developed by Microsoft Research and was first released in 2015 [72]. SEAL supports leveled BFV, BGV, and CKKS.

**TFHE:** The Fast Fully Homomorphic Encryption Library over the Torus (TFHE) was released in 2016 by Chillotti et al. [25] and proposes the CGGI 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.

**Other Libraries:** The Concrete [26] 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 [59], while Concrete is restricted to less than 12 bits of precision [27] (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 [27]. FHEW [35], 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 [23] 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 1.

## 2.4 Related Works

The work in [1] compares (mostly obsolete) FHE implementations and is missing modern state-of-the-art libraries. When comparing the different schemes, the authors incorporate results from prior works configured for non-equivalent security levels, parameter sets, and even benchmarks. Thus, [1] lacks any implementation for uniform comparisons across the studied FHE libraries.

The SHEEP library [7] allows users to write programs in an assembly-like language that targets multiple FHE libraries such as TFHE and HELib. However, this approach requires users to manually design the FHE circuit, which limits usability and mimics the approach required to implement programs directly with the FHE library itself. The T2 compiler, on the other hand, allows users to program in a subset of C that offers a high degree of programmability and enables rapid development, since users can leverage techniques familiar to general-purpose programming.

A recent SoK [80] surveys state-of-the-art FHE compilers such as Cingulata [19], which compiles C++ code to Boolean circuits for TFHE and a BFV variant, CHET [33] which targets CKKS and is geared towards private neural network inference for HEAAN and SEAL, as well as its successor EVA [32] that uses a DSL for vector arithmetic and targets SEAL and CKKS. Moreover, E<sup>3</sup> [24] 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, while Marble [81] 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, [80] does not consider compilers such as Google’s Transpiler [43] and ROMEO [45] 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 CGGI implementation for the former).

While [80] discusses many different compilers, it only employs three minimal applications that are limited in scope. Most importantly, [80] 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. We emphasize that the efficiency of any compiled program will be largely dictated by the efficiency of the crypto backend, so we focus mainly on the core components (instead of the frontends). The goal of our work is to use a fair compiler and standardized benchmarks to facilitate informed comparisons among the studied backends.

Notably, the benchmarks in [80] were manually implemented independently 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 and incorporates a broad range of benchmarks lacking in prior works.

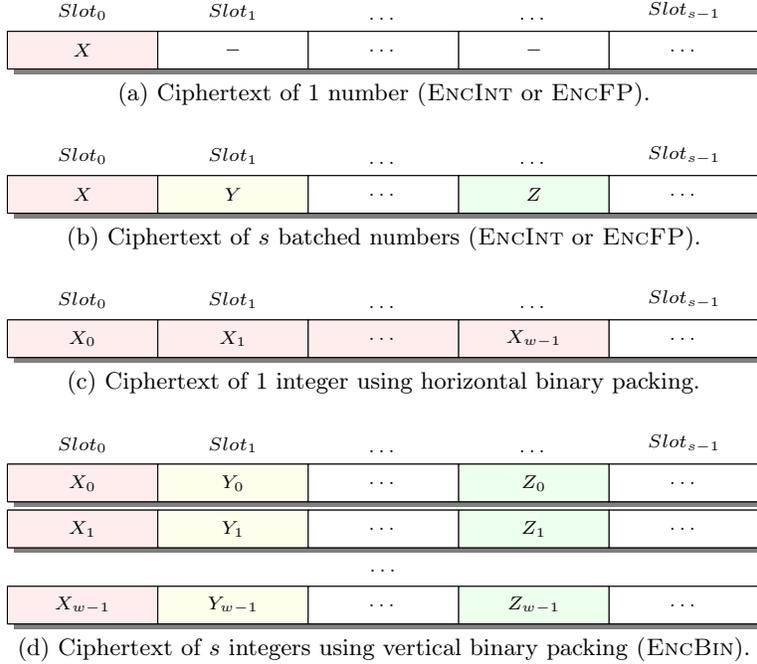
Finally, competitions like iDash [66] pose challengers with a specific problem and require them to develop efficient solutions for private computation, including FHE. While competitors sometimes develop solutions that beat existing implementations solving similar problems, the techniques employed by participants are application-dependent and do not scale to other problem domains.

## 3 T2 Universal Compiler

This section introduces design choices for implementing our T2 universal compiler to map the same high-level code to a variety of HE backends. Notably, our compiler generates optimized code for a broad range of FHE operations and is designed for uniform implementations across HE libraries. Towards that end, we introduce:

- 3.1:* judicious encodings that leverage batching capabilities of the underlying schemes,
- 3.2:* optimized encrypted arithmetic operations for all backends,
- 3.3:* encrypted comparison operations,
- 3.4:* oblivious conditional assignment for encrypted values,
- 3.5:* automatic noise mitigation strategies for FHE,
- 3.6:* our T2DSL high-level language that incorporates all innovations from Sections 3.2–3.6.

While we have integrated backends corresponding to five state-of-the-art libraries, our T2 compiler is future-proof and new backends can be added based on the proposed universal encodings.



**Fig. 3. Encodings Overview.** (a) A single number (floating-point or integer)  $x$  is encrypted using only one slot. (b) *Integer or floating-point batching*: all the slots are being used to encrypt multiple numbers (floating-point or integers) (e.g.,  $x, y, z$ , etc.). (c) *Packing*: utilizes multiple slots to encrypt an integer using a bitwise representation (the number of slots used is equal to the word size  $w$  of  $x$ ). (d) *Vertical binary packing*: utilizes  $w$  ciphertexts to encode multiple integers (e.g.,  $x, y, z$ ) by using a slot-wise binary representation.

### 3.1 ENCINT, ENCFP, and ENCBIN Encodings

We identify four potential types of ciphertext encodings that encompass a range of computational models in FHE, depicted in Fig. 3. We remark that a plethora of other potential encoding strategies exist, such as the encoding used by Gazelle’s privacy-preserving machine learning (PPML) framework [55]. However, these strategies are typically tuned towards niche applications instead of general computation. 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 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, 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 greater than the plaintext modulus and 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.

**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 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.

To maintain consistency and a universal syntax, we simulate batching for schemes that do not naturally support it. We allow the user to supply vectors of plaintext elements to be batched, regardless of the scheme that the compiler targets. For instance, using the `ENCINT` encoding to compile a program for SEAL’s BFV implementation (which supports batching), the T2 compiler utilizes the library’s capabilities to encode the vector into a single ciphertext. However, to target the TFHE library which only supports operations over encrypted bits and does not support batching natively, the T2 compiler automatically translates the batching syntax into a two-dimensional array (i.e., one dimension for each integer and one dimension for each bit of every integer). We note that such translation can be slower than actual batching due to the increased number of FHE operations and larger functional units, but rather reflects the capabilities of the non-batchable schemes and allows for uniform implementation.

**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, to perform an addition using this encoding, each slot needs to be isolated and then fed into an adder circuit, which requires convoluted and expensive slot rotations. In fact, slot extraction is used in the bootstrapping procedure and is one of the core bottlenecks of this operation [49]. 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.

Our `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.

### 3.2 Arithmetic Operations for HE Benchmarks

**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`); These libraries also support negation, which further enables converting an addition operation into a subtraction. As such, the T2 compiler maps these T2DSL operations directly to the underlying library calls (e.g., ciphertext addition/multiplication in SEAL, HELib, etc.).

**Arithmetic in `ENCBIN`:** For binary encoding, these operations are not naturally supported by most libraries. Instead, we construct optimized Boolean circuits composed of primitive logic gates to facilitate these operations for every cryptosystem (with the exception of floating-point schemes). For CGI, we minimize 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 utilize circuits with as few subsequent AND operations as possible and prioritize the nearly free XOR operations in the T2 compiler.

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. To enable higher degrees of batching with `ENCBIN`, we include a second set of functional units that provides a trade-off between available slots and more expensive operations. Even under these circumstances, the user must

choose between batching capabilities and more expensive operations when selecting a plaintext modulus for ENCBIN. These operations for  $p > 2$  are costly because emulating logic gates is far more complicated than in the  $p = 2$  case. As long as the underlying plaintext values are constrained to  $[0, 1]$ , multiplication is still equivalent to AND, while addition is no longer equivalent to XOR since, for instance,  $1 + 1 = 2 \pmod{p}$  instead of  $0 \pmod{p}$ . Thus, we compute the XOR operation by performing  $(X \hat{-} Y)^2$ , which requires a homomorphic subtraction followed by a multiplication (squaring). This XOR variant is more expensive than both the XOR with  $p = 2$  and the AND operation. As a result, the T2 arithmetic functional units for  $p > 2$  prioritize AND over XOR, while the functional units for  $p = 2$  prioritize XOR and NOT.

### 3.3 Comparison Operations for HE Benchmarks

**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  $(X \hat{-} Y)^{p-1} \pmod{p}$ , where  $p$  is a prime [52]. The computational bottleneck involves computing the power  $p - 1$ , which requires approximately  $\log_2 p$  multiplications. Nevertheless, in applications that require a large plaintext range (or require a high degree of batching), this method may be inefficient as it can consume multiple multiplicative levels.

To accomplish a homomorphic “less-than” in T2, we adopt the polynomial interpolation over a prime field proposed by Iliashenko and Zucca [52]. Specifically, we evaluate:

$$X \hat{<} Y := \sum_{a=-\frac{p-1}{2}}^{-1} (1 \hat{-} (X \hat{-} Y \hat{-} a)^{p-1}) \pmod{p}, \quad (1)$$

which outputs  $\hat{1}$  if  $X \hat{<} Y$  and  $\hat{0}$  otherwise. This operation is more expensive than the FHE equality due to the larger 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 achieve “less-than-or-equal” [i.e.,  $X \hat{\leq} Y \Leftrightarrow 1 \hat{-} (Y \hat{<} X)$ ] for approximately the same cost as less-than.

**Comparisons in ENCBIN:** Similarly to the arithmetic operations in ENCBIN, we construct Boolean circuits 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, we observe that 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 [21]:

$$X \hat{<} Y := \sum_{i=1}^w (lt(X_i, Y_i) \prod_{i < j < w} eq(X_j, Y_j)) \pmod{2}, \quad (2)$$

where  $lt(x, y) := (\text{NOT } x) \text{ AND } y$  and  $eq(x, y) := \text{NOT } (x \text{ XOR } y)$ . 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.

### 3.4 Avoiding Termination Problems with MUX

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 the T2 ENCINT encoding, we achieve this using the following equation:  $\text{MUX}(S, X, Y) := X \hat{*} S \hat{+} Y \hat{*} (1 \hat{-} S)$ , where  $S$  is the encrypted selection bit and  $X$  and  $Y$  are the two outcomes of independent branches. In ENCBIN, we directly translate these operations 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. For the TFHE library we use the built-in multiplexing gate, which 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).

### 3.5 Automated Noise Maintenance for FHE

Our compiler removes *all noise maintenance complexities* by judiciously inserting noise checks after multiplications and ensures that all ciphertexts do not exceed their noise budgets. Users can declare a bootstrapped context for libraries that support bootstrapping just by passing a compiler flag – without changing the high-level code. The T2 compiler uses the underlying backends to estimate the variance of the noise in the ciphertext to determine when bootstrapping is necessary and automatically invoke it if needed. The bootstrapping procedure itself consumes a number of levels before refreshing, which affects the noise threshold; the multiplicative depth of the procedure is dependent on the parameters, making it non-trivial for users to determine. Notably, T2 leverages optimized thresholds for predefined parameter sets and uses existing HE library features to select appropriate thresholds for custom parameters where applicable.

### 3.6 T2DSL Programming Language

In order to support our proposed encodings (depicted in Section 3.1), we introduce the T2DSL language, which is a strongly-typed C-like language that supports two encrypted data types: `EncInt` and `EncDouble`. Together, these two data types can represent our three ciphertext types (i.e., `ENCINT`, `ENCBIN`, and `ENCFP`). We note that `ENCBIN` shares the same data-type as `ENCINT`, but the compiler treats the former as ciphertext arrays of encrypted bits and the latter as individual ciphertexts encrypting modular integers. The actual encoding can be selected with a flag at compilation time without requiring any changes to the T2DSL program, making it very easy for users to experiment with both encodings across the different libraries.

In terms of operations, the T2DSL supports addition and multiplication for all encodings, multiplexing and comparisons for `ENCINT` and `ENCBIN`, bitwise shifts, as well as standard logic gate operations (i.e., NOT, XOR, etc.) for `ENCBIN`. For plaintext values, all arithmetic operations supported in C are also supported in T2DSL; additionally, we support *mixed operations between plaintexts and ciphertexts*. This is achieved by first encoding the plaintext as a “constant encryption”. Constant (or *noiseless*) encryptions can be generated without the user’s private key and operations between constant encryptions and secure ciphertexts are much faster than ciphertext to ciphertext operations. These operations allow T2 to efficiently execute algorithms where some inputs are considered public and do not need to be securely encrypted, like privacy-preserving machine learning constructions. When the compiler processes an operation between encrypted and plaintext operands, it automatically generates a mixed operation without any guidance required from the user. Lastly, we remark that certain language constructs such as referencing/dereferencing are not supported in T2DSL, which is also the case with other compilers such as E<sup>3</sup> [24] and EVA [32].

We have included two T2DSL examples in Appendix D that showcase the code used to generate two benchmarks used in our evaluations. The first is a squared Euclidean distance in T2DSL, which is suitable for all encodings, while the second is the CRC-32 benchmark (which can only be compatible with `ENCBIN` contexts).

## 4 Algorithms in T2 Benchmark Suite

In this section, we introduce our T2 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 [17], 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 runtime decisions.

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

**Arithmetic** benchmarks contain additions and multiplication and operate in the integer and floating-point domains.

**Bitwise** benchmarks are composed primarily of 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 remark that the binary domain evaluation of certain benchmarks can be high due to the noise cost of deep Boolean circuits, so the integer or floating-point domain may be preferable. For this reason, we only show binary results for all benchmarks using CGGI unless explicitly indicated. Moreover, while many benchmarks utilize similar primitive instructions, they differ in both their complexity and the size of the encrypted operands as well as the depth required to evaluate them. Lastly, most benchmarks are run with different encodings which significantly impact the type and number of operations required to evaluate the algorithm.

#### 4.1 T2 Arithmetic Benchmarks

**Chi-Squared:** The chi-squared ( $\chi^2$ ) 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 [58]. First, the server receives the encrypted genotype counts  $N_0, N_1, N_2$ , then computes  $A := 4(N_0 \hat{*} N_2 \hat{-} N_1^2)^2$ ,  $B_1 := 2(2N_0 \hat{+} N_1^2)^2$ ,  $B_2 := (2N_0 \hat{+} N_1) \hat{*} (2N_2 \hat{+} N_1)$ , and  $B_3 := 2(2N_2 \hat{+} N_1)^2$  and finally returns the encrypted results to the client. Like [58], T2 does not implement costly homomorphic division and leaves this final step as a cheap post-processing procedure done on the client-side. 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  $N := N_0 + N_1 + N_2$ . We evaluate the  $\chi^2$  benchmark both using the ENCINT and ENCFP encodings.

**Machine Learning Inference:** PPML is an emergent research area in encrypted computation [38,27]. The cloud can perform oblivious neural network inference procedures with its own proprietary model 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. For neural network inference, it is common in the literature to adopt the square activation function due to its low multiplicative depth, whereas other non-linear activations, like ReLU, are much harder to evaluate in the encrypted domain since they require expensive polynomial approximations [60,34]. 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, operations between ciphertexts are still needed (e.g., in the square function). We evaluate this benchmark with a constant image size for a variety of active neuron sizes to show scalability for wider networks.

**Logistic Regression (LR):** This 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) [40]. In T2, we employ a Taylor series expansion of the sigmoid function and evaluate the first three terms (i.e.,  $\sigma(X) := 1/2 + X/4 + (X^3)/48 + \dots$ ) [56]; using more terms yields higher accuracy at the expense of more noise and slower execution times. Since none of the HE libraries supports homomorphic division, for ENCINT we scale the series up by the largest denominator and evaluate  $\sigma(X) := 24 \hat{+} 12X \hat{+} X^3 + \dots$ ; 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  $X$ . For both PPML benchmarks, we use both ENCINT and ENCFP.

**Matrix Multiplication:** Matrix multiplication is a crucial 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.

**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 [2,5]. PIR has a plethora of applications in private computation, such as ad delivery [46], friend discovery [13], and keyword search [67]. For a single query, the server iterates over all database elements and obliviously selects 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 [2] 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:  $Q = (q_1, q_2, \dots, q_n)$ , 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  $V = (v_1, v_2, \dots, v_n)$  and returns a sum of the products  $\sum_{i=1}^n (Q[i] \hat{*} V[i])$ , which is equivalent to the encryption of the requested element.

This benchmark can exploit batching to reduce the number of encrypted operations and communication bandwidth. Instead of representing  $Q$  and  $V$  as vectors of ciphertexts, they can be condensed into single ciphertexts with at least  $n$  slots. Now, one multiplication is required between  $Q$  and  $V$  and the product can be returned to the user, which contains the desired value in slot  $idx$ . We evaluate PIR over a database in both ENCINT and ENCFP.

**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 [20] and facial recognition [63]. In our case, we consider two  $n$ -dimensional points  $V = (v_1, v_2, \dots, v_n)$  and  $U = (u_1, u_2, \dots, u_n)$  in Euclidean space. The squared Euclidean distance is calculated as the sum of the squared differences between the points as  $E^2(V, U) := \sum_{i=1}^n (V[i] - U[i])^2$ . Notably, this procedure is more FHE-friendly than the standard Euclidean distance that requires a square root, which is not possible to directly evaluate in the encrypted domain. As a result, the user needs to compute the final square root in the plaintext domain after decrypting the cloud’s output. For this benchmark we use ENCINT and ENCFP.

## 4.2 Bitwise Benchmarks

**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.

**Oblivious Sorting:** Sorting algorithms are an interesting and open problem for encrypted computation; in all cases, one has to deal with the worst-case complexity. For insertion sort, the worst-case complexity is  $\mathcal{O}(n^2)$  (where  $n$  is the size of the array), due to the termination problem. Ideally, one would use an asymptotically better sorting algorithm such as merge sort, but this is data dependent. However, the Batcher odd-even merge sort [8] is not data dependent and has worst-case complexity  $\mathcal{O}(\log^2 n)$ . We incorporate both insertion sort and the Batcher sorting network into our benchmark suite. All sorting algorithms make frequent use of  $\hat{<}$  for the conditional swap, making them intensive comparison benchmarks that are 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.

**Binary Manhattan Distance:** The Manhattan distance is a measure of the distance between two points  $V = (v_1, v_2, \dots, v_n)$  and  $U = (u_1, u_2, \dots, u_n)$  in an  $n$ -dimensional space and is defined as  $M(V, U) := \sum_{i=1}^n (|V[i] \hat{-} U[i]|)$ . The key difference between Manhattan 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.

**Binary Private Information Retrieval (BinPIR):** Similarly to BaPIR, we assume a database  $V = (v_1, v_2, \dots, v_n)$  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. This benchmark trades efficiency for bandwidth: while more HE operations are required to complete the query, the client requests are constant in size and the client does not need any details about the structure of the key/value storage.

### 4.3 Relational Benchmarks

**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 telecommunications, as well as to determine genetic distance in biology applications. It has also been used as an FHE application in prior works [81]. 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.

**Relational Manhattan Distance:** Instead of using the sign bit directly for the MUX input to compute the absolute value of the difference of  $V[i] \hat{-} U[i]$  (as presented in the binary Manhattan benchmark), 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, the  $\hat{<}$  is the most expensive operation.

## 5 Experimental Evaluations

We run the T2 benchmarks using three different encodings, i.e., ENCINT, ENCBIN, and ENCFP, where applicable. For ENCINT and ENCBIN, we employ BFV where possible as this is widely used in the research community [34,24,80] and BFV outperforms BGV for smaller plaintext moduli up to between 32 and 64 bits [31], which is the range used for the T2 benchmarks. The two exceptions to this are TFHE (which only implements CGGI) and HELib (where the only option for integers is BGV). Lastly, we utilize CKKS for ENCFP as it naturally supports computation on encrypted real numbers. The benchmarks from the Arithmetic and Relational categories are 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. To investigate the effect that the target domain makes on the latency of a benchmark, we run selected benchmarks from the Arithmetic and the Relational categories using the ENCBIN encoding.

Finally, we carefully choose minimal FHE parameter sets for each library to achieve at least 128 bits of security (using the “BKZ-beta” classical cost model provided by the *LWE estimator* [4]) to successfully evaluate each benchmark, and where applicable, we varied the input length of the benchmark, and the word-size  $w$  for ENCBIN. We only consider the timings of server-side FHE operations, which consist of all evaluation operations except key generation and encryption/decryption (which are one-time costs for the

**Table 2.** Noise growth and latency of core operations for integer, binary, and floating-point encodings. Parameter sets have been judiciously selected to be as equivalent as possible across libraries.

Encoding	Binary Encoding (ENC <sub>BIN</sub> ) with word size $s^*$						Integer Encoding (ENC <sub>INT</sub> ) <sup>*</sup>						Floating-Point Encoding (ENC <sub>FP</sub> ) <sup>*</sup>			
Operation	Add.	Mult.	Eq.	LT.	ROT.	Ctxt Size <sup>¶</sup>	Add.	Mult.	Eq.	LT. <sup>†</sup>	ROT.	Ctxt Size	Add.	Mult.	ROT.	Ctxt Size
Noise																
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 <sup>#</sup>	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

<sup>\*</sup> 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).

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

<sup>‡</sup> 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).

<sup>§</sup> The number of bars indicates latency relative to other HE operations on a logarithmic scale starting from 100 ms. For instance, any operation < 100 ms is awarded one bar, 2 bars are < 1 second, and so on.

<sup>#</sup> These rotations are performed on the vectors holding the encrypted bits and are independent of the HE library.

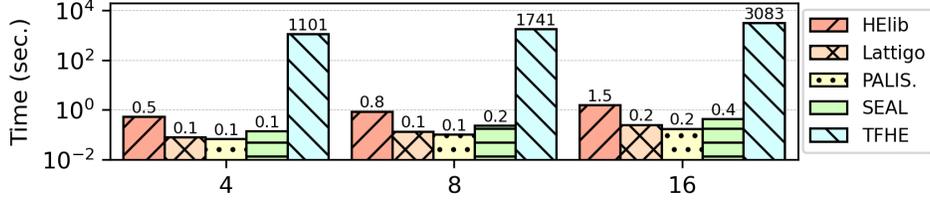
<sup>¶</sup> Ciphertext size in MBs using the smallest parameters for each library to encrypt an 8-bit number and achieve depth 10 at 128 bits of security.

user). We have evaluated the T2 translation methodology and found that it is not only competitive with the state-of-the-art compilers, but it also outperforms them in certain cases. While optimizing compilers usually target a single library, T2 offers support for the largest number of backends, schemes, and encodings. A quantitative evaluation with other compilers can be found in Appendix C.3.

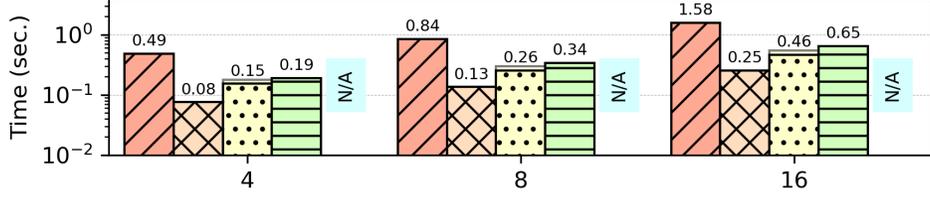
All experiments were performed on an AWS instance (c5.4xlarge) running Ubuntu 20.04, equipped with 16 vCPUs and 32 GBs of memory (however, our benchmarks do not require more than 8GBs). Where applicable, we measured the runtime performance of our benchmarks with all encodings. Our T2 compiler targets the latest versions of the FHE backends at the time of writing: 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. Finally, PALISADE is configured to exploit multi-threading by default. Thus, we allow PALISADE to utilize all the available machine cores since it is not optimized for sequential evaluation. However, for completeness, we additionally conducted single core experiments with PALISADE (depicted as PAL. 1C in the figures), which, to our surprise, had the same execution time in some cases with the multi-threaded implementation, while in other cases the single core version was 2-3x slower. For the benchmarks where the single-core implementation is slower, we overlay the PALISADE bar with a non-patterned bar, as in Fig. 6. This provides a clear visual of how multi-threading impacts the performance of PALISADE. Our evaluation does not add any custom form of parallelization to HElib, Lattigo, SEAL, and TFHE as they only utilize one core by default.

In addition to the three categories of benchmarks analyzed in this section,<sup>1</sup> we also evaluate the timings of FHE primitive operations across all libraries in all encodings where applicable. Our results are summarized in Table 2 and show 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 remark that similar trends shown in this table in terms of ciphertext size also apply to the benchmarks; while the total size of all ciphertexts will change, the ratio between the sizes shown in this baseline is also observed in our benchmarks. 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, however, may support thousands of batching slots. Some libraries are unable to run certain benchmarks because *there aren't any allowable parameters to support the required noise depth* (i.e., there is no valid configuration to support the noise budget while maintaining 128

<sup>1</sup> The evaluations of matrix multiplication, batched and binary PIR, oblivious sorting, and the squared Euclidean distance are included in Appendix C.

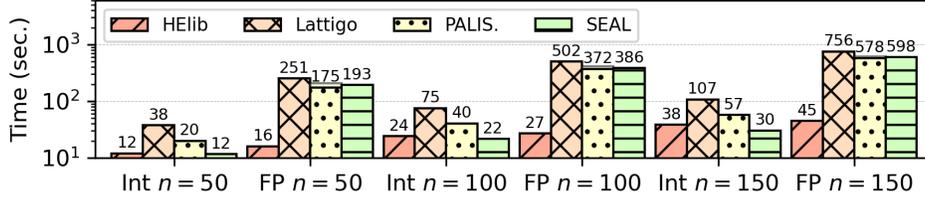


(a) Integer domain with ENCBIN for TFHE and ENCINT for other libraries.



(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  $w = 16$ .



**Fig. 5.** Measured latency for the neural network benchmark in the integer (ENCINT) and floating-point (ENCFP) domains with increasing numbers of hidden neurons.

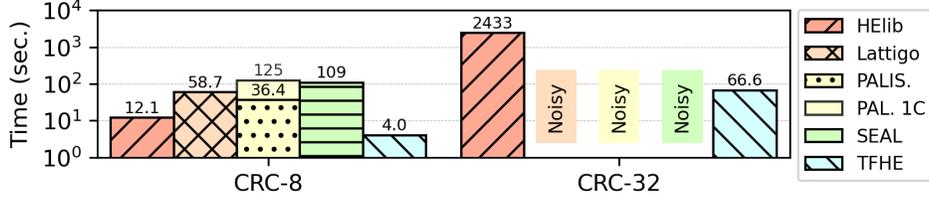
bits of security). We indicate these cases as “Noisy” with a background color denoting the library (as shown in Fig. 6).

### 5.1 Selected Arithmetic Benchmarks

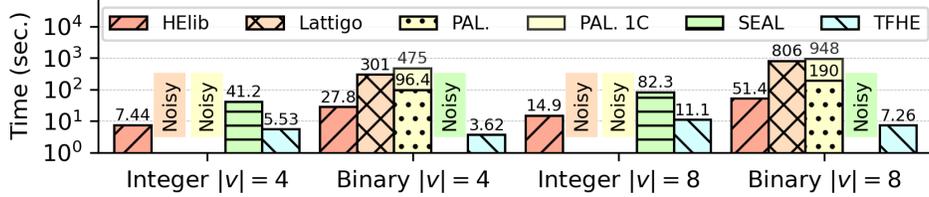
**Chi-Squared:** As this benchmark mainly comprises additions and multiplications with low-depth, the latency for all the libraries except TFHE for both ENCINT and ENCFP is under 0.5 seconds. For TFHE, we use a word size of 8 bits and the runtime is 44.3 seconds, which is due to the evaluation of binary adders and multipliers, which require several addition, multiplication, and bootstrapping operations for every circuit.

**LR Inference:** The LR benchmark has competitive timings across all the libraries for both ENCINT and ENCFP, save for TFHE, which is much 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.

**Feedforward NN Inference:** Fig. 5 demonstrates the timings for our neural network benchmark for both ENCINT and ENCFP for varying numbers of active neurons. This benchmark includes large numbers of ciphertext additions and 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 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 always applies the `rescale` operation in the low-level instructions, whereas we can finely control these operations in the T2 operational units using the other four back-ends. Note that TFHE could not evaluate this benchmark in less than eight hours and thus is omitted from Fig. 5. These results are contrary to our



**Fig. 6.** 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. 7.** 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 ENCBIN 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.

expectations and to the trends depicted in Table 2, which indicate that PALISADE should outperform the other libraries for ENCFP and ENCINT. This emphasizes the importance of considering the performance of actual applications instead of solely primitive operations.

## 5.2 Selected Bitwise Benchmarks

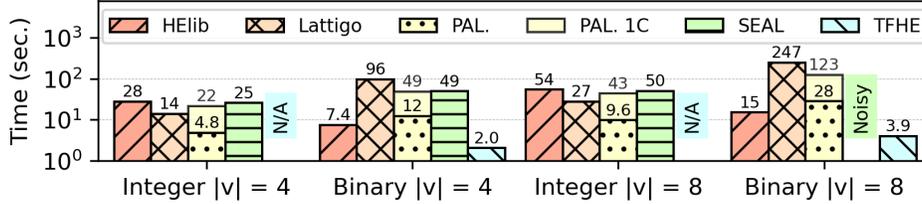
**CRC:** Fig. 6 presents the timings of CRC-8 and CRC-32 evaluations using ENCBIN as it is not possible to use the integer or floating-point encodings due to the bitwise shift and XOR operations. All libraries were able to run CRC-8 without issues, yet for CRC-32 only the libraries that support bootstrapping 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 did not support parameters large enough to achieve the required depth at 128 bits of security (i.e., depth is traded for security).

**Binary Manhattan Distance:** This benchmark includes a multiplexing operation to calculate the absolute value as it is more efficient to evaluate a MUX using the sign bit of the  $V[i] \hat{=} U[i]$  than it is to employ an  $\hat{<}$  followed by a MUX (as in the integer domain). Fig. 7 shows the timings for ENCINT and ENCBIN. We note that TFHE performs better when using the binary-optimized algorithm while the other libraries perform better using the relational variant.

## 5.3 Relational Benchmarks

**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 other libraries, ENCINT is faster and has a lower depth. Notably, TFHE drastically outperforms all implementations for both domains, as depicted in Fig. 8.

**Relational Manhattan Distance:** This differs from the binary case because sign extraction with encrypted integers is expensive. The integer multiplexer follows the same equation, however the control input must be  $\hat{0}$  or  $\hat{1}$ . As shown in Fig. 7, 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



**Fig. 8.** 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  $w = 4$ .

circuit depth of the LT operation scales linearly with the size of  $p$ , one needs a small poly degree to allow for a smaller  $p$ . However, small poly degrees do not yield enough multiplicative depth to evaluate the circuit, which requires larger parameters and forces the user to employ a larger  $p$ .

## 6 Discussion & Lessons Learned

**ENCBIN:** From the results presented in Table 2 and the T2 benchmarks, we can glean several important insights about the performance 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 `HELlib`, 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 `HELlib`. 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, `HELlib` 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  $n$  times, for a relatively small  $n$ , the latency of `TFHE` is often larger than the latency of `HELlib`, which can extend to  $n$  slots at the same cost as  $n = 1$ .

**ENCINT & ENCFP:** From Table 2, it is clear that `PALISADE` in its default, multi-core setting exhibits the fastest overall speeds for 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, such as in the relational Manhattan distance. For these types of applications, `HELlib` is a better option since it enables batching with much smaller choices of  $p$ . In the floating-point domain, when rescaling frequently is required, `HELlib` 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  $16 \times 16 \times 16 \times 16$  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 NN inference due to rescaling. Some applications, such as full-precision neural networks, are better suited for the `ENCFP` as weights do not need to be quantized.

## 7 Future directions of HE libraries

Based on our findings in this SoK, all libraries should introduce standardized binary circuits for atomic operations that form the building blocks of most applications (such as multi-bit arithmetic circuits) as the

performance of binary FHE is dependent on the underlying Boolean circuits. Currently, only HELib natively supports ENCBIN operations, rendering it the most versatile library in this regard. Unfortunately, although TFHE is meant for ENCBIN, it does not include any circuit designs, only two-input logic gates, and relies on the user to construct such circuits. As ENCBIN enables benchmarks not possible in other domains, more libraries should follow HELib’s footsteps in the future and include Boolean addition, multiplication, and comparison circuits. Libraries can derive optimal implementations by leveraging established techniques in hardware design, such as logic optimization, in order to provide arithmetic functionality for Boolean ciphertexts. In fact, some compilers such as ROMEO and the Google Transpiler have started exploring these techniques but they are only limited to certain backends. While these works have already adopted generic synthesis flows, future work can further explore how logic optimizations common to electronic design automation (EDA) could be optimally adopted for FHE. Additionally, it is crucial to come up with better relational operations for the integer and floating-point encodings, as currently comparisons in these domains are not viable for real-world applications. Latest trends in encrypted computation circumvent this by avoiding relational operations (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 [38,73]. Going forward, FHE libraries should continue to incorporate support for these devices as well as other emergent dedicated hardware units.

## 8 Concluding Remarks

Our goal with this systematization of knowledge is to highlight the weaknesses of HE libraries and draw more researchers to the problem of making HE practical for general computation through standardized benchmarks. First, we establish the universal T2 compiler and accompanying T2DSL that can map to any arbitrary HE library and encoding. We show that our compiler is fair across the board by expounding upon the design of the optimized functional units that are employed uniformly in all supported backends. Using the compiler as a platform for meaningful comparisons, we employ a custom suite of benchmarks that represents real use-cases of HE and covers a broad range of computational patterns to analyze the performance of HE libraries. We remark that any runtime differences between the libraries is due to the underlying cryptographic implementations themselves and the optimizations they exploit, rather than the compilation process (which is identical across the board). Our goal for the T2 compiler and benchmarks is to become instrumental in evaluating and improving new implementations of both current and future HE schemes.

## Resources

Our T2 compiler and benchmarks are available online as open-source software [44].

## Acknowledgments

This work was partially supported by the University of Delaware Research Foundation Grant 21A01012 and the Electrical and Computer Engineering department at the University of Delaware.

## References

1. Abbas Acar, Hidayet Aksu, A Selcuk Uluagac, and Mauro Conti. A survey on homomorphic encryption schemes: Theory and implementation. *ACM Computing Surveys (CSUR)*, 51(4):1–35, 2018.

2. Carlos Aguilar-Melchor, Joris Barrier, Laurent Fousse, and Marc-Olivier Killijian. XPIR: Private information retrieval for everyone. *Proceedings on Privacy Enhancing Technologies*, 2(2016):155–174, 2016.
3. Carlos Aguilar-Melchor, Joris Barrier, Serge Guelton, Adrien Guinet, Marc-Olivier Killijian, and Tancrede Lepoint. Nflib: Ntt-based fast lattice library. In Kazue Sako, editor, *Topics in Cryptology - CT-RSA 2016*, pages 341–356, Cham, 2016. Springer International Publishing.
4. Martin R Albrecht, Rachel Player, and Sam Scott. On the concrete hardness of learning with errors. *Journal of Mathematical Cryptology*, 9(3):169–203, 2015.
5. Sebastian Angel, Hao Chen, Kim Laine, and Srinath Setty. PIR with compressed queries and amortized query processing. In *Symposium on Security and Privacy (SP)*, pages 962–979, Piscataway, New Jersey, 2018. IEEE, Institute of Electrical and Electronics Engineers.
6. Andrew Arnold and Michael Monagan. Calculating cyclotomic polynomials. *Mathematics of Computation*, 80(276):2359–2379, 2011.
7. Nick Barlow, Tomas Lazauskas, Oliver Strickson, and Adria Gascon. SHEEP: A homomorphic encryption evaluation platform. Online, 2019. <https://github.com/alan-turing-institute/SHEEP>.
8. Kenneth E Batcher. Sorting networks and their applications. In *Proceedings of the April 30–May 2, 1968, spring joint computer conference*, pages 307–314, New York, NY, USA, 1968. Association for Computing Machinery.
9. Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Scalable zero knowledge with no trusted setup. In Alexandra Boldyreva and Daniele Micciancio, editors, *Advances in Cryptology – CRYPTO 2019*, pages 701–732, Cham, 2019. Springer International Publishing.
10. Ayoub Benaissa, Bilal Retiat, Bogdan Cebere, and Alaa Eddine Belfedhal. Tenseal: A library for encrypted tensor operations using homomorphic encryption. *CoRR*, abs/2104.03152:1–12, 2021.
11. Atri Bhattacharyya, Alexandra Sandulescu, Matthias Neugschwandtner, Alessandro Sorniotti, Babak Falsafi, Mathias Payer, and Anil Kurmus. Smotherspectre: Exploiting speculative execution through port contention. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS ’19*, page 785–800, New York, NY, USA, 2019. Association for Computing Machinery.
12. Fabian Boemer, Rosario Cammarota, Daniel Demmler, Thomas Schneider, and Hossein Yalame. Mp2ml: A mixed-protocol machine learning framework for private inference. In *Proceedings of the 2020 Workshop on Privacy-Preserving Machine Learning in Practice, PPMLP’20*, page 43–45, New York, NY, USA, 2020. Association for Computing Machinery.
13. Nikita Borisov, George Danezis, and Ian Goldberg. DP5: A Private Presence Service. *Proc. Priv. Enhancing Technol.*, 2015(2):4–24, 2015.
14. Zvika Brakerski. Fully homomorphic encryption without modulus switching from classical gapsvp. In Reihaneh Safavi-Naini and Ran Canetti, editors, *Advances in Cryptology – CRYPTO 2012*, pages 868–886, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.
15. Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (leveled) fully homomorphic encryption without bootstrapping. In *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference, ITCS ’12*, page 309–325, New York, NY, USA, 2012. Association for Computing Machinery.
16. Zvika Brakerski and Vinod Vaikuntanathan. Efficient fully homomorphic encryption from (standard) lwe. *SIAM Journal on computing*, 43(2):831–871, 2014.
17. Michael Brenner et al. Secret Program Execution in the Cloud Applying Homomorphic Encryption. In *Digital Ecosystems and Technologies Conference*, pages 114–119, New York, NY, USA, 2011. IEEE, Institute of Electrical and Electronic Engineers.
18. Éric Brier and Marc Joye. Weierstraß elliptic curves and side-channel attacks. In David Naccache and Pascal Paillier, editors, *Public Key Cryptography*, pages 335–345, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.
19. Sergiu Carpov, Paul Dubrulle, and Renaud Sirdey. Armadillo: A compilation chain for privacy preserving applications. In *Proceedings of the 3rd International Workshop on Security in Cloud Computing, SCC ’15*, page 13–19, New York, NY, USA, 2015. Association for Computing Machinery.
20. Randy L Carter, Robin Morris, and Roger K Blashfield. On the partitioning of squared euclidean distance and its applications in cluster analysis. *Psychometrika*, 54(1):9–23, 1989.
21. Gizem S Çetin, Yarkin Doröz, Berk Sunar, and Erkay Savas. Low depth circuits for efficient homomorphic sorting. Cryptology ePrint Archive, Report 2015/274, 2015. <http://eprint.iacr.org/2015/274>.
22. Ayantika Chatterjee and Indranil Sengupta. Translating algorithms to handle fully homomorphic encrypted data on the cloud. *IEEE Transactions on Cloud Computing*, 6(1):287–300, 2015.
23. Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. Homomorphic encryption for arithmetic of approximate numbers. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 409–437, Cham, Switzerland, 2017. Springer, Springer, Cham.

24. Eduardo Chielle, Oleg Mazonka, Homer Gamil, Nektarios Georgios Tsoutsos, and Michail Maniatakos. E3: A framework for compiling C++ programs with encrypted operands. Cryptology ePrint Archive, Report 2018/1013, 2018. <http://eprint.iacr.org/2018/1013>.
25. Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. TFHE: fast fully homomorphic encryption over the torus. *Journal of Cryptology*, 33(1):34–91, 2020.
26. Ilaria Chillotti, Marc Joye, and Pascal Paillier. Programmable bootstrapping enables efficient homomorphic inference of deep neural networks. In Shlomi Dolev, Oded Margalit, Benny Pinkas, and Alexander Schwarzmann, editors, *Cyber Security Cryptography and Machine Learning*, pages 1–19, Cham, 2021. Springer International Publishing.
27. Ilaria Chillotti, Marc Joye, and Pascal Paillier. Programmable bootstrapping enables efficient homomorphic inference of deep neural networks. In Shlomi Dolev, Oded Margalit, Benny Pinkas, and Alexander Schwarzmann, editors, *Cyber Security Cryptography and Machine Learning*, pages 1–19, Cham, 2021. Springer International Publishing.
28. Edward J Chou, Arun Gururajan, Kim Laine, Nitin Kumar Goel, Anna Bertiger, and Jack 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)*, pages 2792–2796, New York, NY, USA, 2020. IEEE, Institute of Electrical and Electronic Engineers.
29. Richard Chow, Philippe Golle, Markus Jakobsson, Elaine Shi, Jessica Staddon, Ryusuke Masuoka, and Jesus Molina. Controlling data in the cloud: Outsourcing computation without outsourcing control. In *Proceedings of the 2009 ACM Workshop on Cloud Computing Security, CCSW '09*, page 85–90, New York, NY, USA, 2009. Association for Computing Machinery.
30. Ana Costache and Nigel P. Smart. Which ring based somewhat homomorphic encryption scheme is best? In Kazue Sako, editor, *Topics in Cryptology - CT-RSA 2016*, pages 325–340, Cham, 2016. Springer International Publishing.
31. Anamaria Costache, Kim Laine, and Rachel Player. Evaluating the effectiveness of heuristic worst-case noise analysis in fhe. In *Computer Security – ESORICS 2020: 25th European Symposium on Research in Computer Security, ESORICS 2020, Guildford, UK, September 14–18, 2020, Proceedings, Part II*, page 546–565, Berlin, Heidelberg, 2020. Springer-Verlag.
32. Roshan Dathathri, Blagovesta Kostova, Olli Saarikivi, Wei Dai, Kim Laine, and Madan 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, PLDI 2020*, page 546–561, New York, NY, USA, 2020. Association for Computing Machinery.
33. Roshan Dathathri, Olli Saarikivi, Hao Chen, Kim Laine, Kristin Lauter, Saeed Maleki, Madanlal Musuvathi, and Todd 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, PLDI 2019*, page 142–156, New York, NY, USA, 2019. Association for Computing Machinery.
34. Nathan Dowlin, Ran Gilad-Bachrach, Kim Laine, Kristin Lauter, Michael Naehrig, and John Wernsing. Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy. In *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48, ICML'16*, page 201–210, New York, NY, USA, 2016. JMLR.org.
35. Léo Ducas and Daniele Micciancio. Fhew: Bootstrapping homomorphic encryption in less than a second. In Elisabeth Oswald and Marc Fischlin, editors, *Advances in Cryptology – EUROCRYPT 2015*, pages 617–640, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.
36. Clare 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>.
37. Junfeng Fan and Frederik Vercauteren. Somewhat practical fully homomorphic encryption. Cryptology ePrint Archive, Report 2012/144, 2012. <http://eprint.iacr.org/2012/144>.
38. Lars Folkerts, Charles Gouert, and Nektarios Georgios Tsoutsos. REDsec: Running encrypted DNNs in seconds. Cryptology ePrint Archive, Report 2021/1100, 2021. <https://ia.cr/2021/1100>.
39. Robin Geelen. *Bootstrapping Algorithms for BGV and FV*. PhD thesis, KU Leuven, 2021.
40. Alexander Genkin, David D Lewis, and David Madigan. Large-scale bayesian logistic regression for text categorization. *technometrics*, 49(3):291–304, 2007.
41. Craig Gentry. *A Fully Homomorphic Encryption Scheme*. PhD thesis, Stanford University, 2009.
42. Craig Gentry, Amit Sahai, and Brent Waters. Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based. In *Annual Cryptology Conference*, pages 75–92, Berlin, Germany, 2013. Springer, Springer Berlin, Heidelberg.

43. Shruthi Gorantala et al. A general purpose transpiler for fully homomorphic encryption. arXiv preprint, arXiv:2106.07893, 2021. <https://arxiv.org/abs/2106.07893>.
44. Charles Gouert, Dimitris Mouris, and Nektarios Georgios Tsoutsos. Terminator Suite 2: Data-Oblivious Benchmarks for Encrypted Data Computation. <https://github.com/TrustworthyComputing/T2-FHE-Compiler-and-Benchmarks>, 2023.
45. Charles Gouert and Nektarios Georgios Tsoutsos. Romeo: conversion and evaluation of hdl designs in the encrypted domain. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6, New York, NY, USA, 2020. IEEE, Institute of Electrical and Electronic Engineers.
46. Matthew Green, Watson Ladd, and Ian Miers. A protocol for privately reporting ad impressions at scale. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1591–1601, New York, NY, USA, 2016. Association for Computing Machinery.
47. Shai Halevi, Yuriy Polyakov, and Victor Shoup. An improved rns variant of the bfv homomorphic encryption scheme. In Mitsuru Matsui, editor, *Topics in Cryptology – CT-RSA 2019*, pages 83–105, Cham, 2019. Springer International Publishing.
48. Shai Halevi and Victor Shoup. Algorithms in HELib. In *Annual Cryptology Conference*, pages 554–571, Berlin, Germany, 2014. Springer, Springer, Berlin, Heidelberg.
49. Shai Halevi and Victor Shoup. Bootstrapping for HELib. In Elisabeth Oswald and Marc Fischlin, editors, *Advances in Cryptology – EUROCRYPT 2015*, pages 641–670, Berlin, Germany, 2015. Springer Berlin Heidelberg.
50. John L Henning. SPEC CPU2006 Benchmark Descriptions. *ACM SIGARCH Computer Architecture News*, 34(4):1–17, 2006.
51. Helmut Hlavacs, Thomas Treutner, Jean-Patrick Gelas, Laurent Lefevre, and Anne-Cecile Orgerie. Energy consumption side-channel attack at virtual machines in a cloud. In *2011 IEEE Ninth International Conference on Dependable, Autonomic and Secure Computing*, pages 605–612, New York, NY, USA, 2011. IEEE, Institute of Electrical and Electronic Engineers.
52. Iliia Iliashenko and Vincent Zucca. Faster homomorphic comparison operations for BGV and BFV. *Proc. Priv. Enhancing Technol.*, 2021(3):246–264, 2021.
53. Yier Jin, Nathan Kupp, and Yiorgos Makris. Experiences in hardware trojan design and implementation. In *International Workshop on Hardware-Oriented Security and Trust*, pages 50–57, New York, NY, USA, 2009. IEEE, Institute of Electrical and Electronic Engineers.
54. Yier Jin, Michail Maniatakos, and Yiorgos Makris. Exposing vulnerabilities of untrusted computing platforms. In *2012 IEEE 30th International Conference on Computer Design (ICCD)*, pages 131–134, New York, NY, USA, 2012. IEEE, Institute of Electrical and Electronic Engineers.
55. Chiraag Juvekar, Vinod Vaikuntanathan, and Anantha Chandrakasan. {GAZELLE}: A low latency framework for secure neural network inference. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 1651–1669, USA, 2018. USENIX Association.
56. Miran Kim, Yongsoo Song, Shuang Wang, Yuhou Xia, Xiaoqian Jiang, et al. Secure logistic regression based on homomorphic encryption: Design and evaluation. *JMIR medical informatics*, 6(2):e8805, 2018.
57. Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, et al. Spectre attacks: Exploiting speculative execution. In *Symposium on Security and Privacy (SP)*, pages 1–19, New York, NY, USA, 2019. IEEE, Institute of Electrical and Electronic Engineers.
58. Kristin Lauter, Adriana López-Alt, and Michael Naehrig. Private computation on encrypted genomic data. In *International Conference on Cryptology and Information Security in Latin America*, pages 3–27, Cham, Switzerland, 2014. Springer, Springer, Cham.
59. Joon-Woo Lee, Eunsang Lee, Yongwoo Lee, Young-Sik Kim, and Jong-Seon 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*, pages 618–647, Cham, Switzerland, 2021. Springer, Springer, Cham.
60. Jian Liu, Mika Juuti, Yao Lu, and Nadarajah Asokan. Oblivious neural network predictions via minionn transformations. In *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*, pages 619–631, New York, NY, USA, 2017. Association for Computing Machinery.
61. Qian Lou and Lei Jiang. SHE: A Fast and Accurate Deep Neural Network for Encrypted Data. *Advances in Neural Information Processing Systems*, 32:1–9, 2019.
62. Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 1–23, Berlin, Germany, 2010. Springer, Springer Berlin, Heidelberg.

63. M. D. Malkauthekar. Analysis of euclidean distance and manhattan distance measure in face recognition. In *Third International Conference on Computational Intelligence and Information Technology (CIIT 2013)*, pages 503–507, Mumbai, 2013. IET.
64. Dimitris Mouris and Nektarios Georgios Tsoutsos. Zilch: A Framework for Deploying Transparent Zero-Knowledge Proofs. *IEEE Transactions on Information Forensics and Security*, 16:3269–3284, 2021.
65. Dimitris Mouris, Nektarios Georgios Tsoutsos, and Michail Maniatakos. Terminator suite: Benchmarking privacy-preserving architectures. *IEEE Computer Architecture Letters*, 17(2):122–125, 2018.
66. Lucila Ohno-Machado, Vineet Bafna, Aziz A Boxwala, Brian E Chapman, Wendy W Chapman, Kamalika Chaudhuri, Michele E Day, Claudiu Farcas, Nathaniel D Heintzman, Xiaoqian Jiang, et al. idash: integrating data for analysis, anonymization, and sharing. *Journal of the American Medical Informatics Association*, 19(2):196–201, 2012.
67. Rafail Ostrovsky and William E. Skeith. Private searching on streaming data. In Victor Shoup, editor, *Advances in Cryptology – CRYPTO 2005*, pages 223–240, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.
68. Rishabh Poddar, Tobias Boelter, and Raluca Ada Popa. Arx: A strongly encrypted database system. *IACR Cryptol. ePrint Arch.*, 2016:591, 2016.
69. Yuriy Polyakov, Kurt Rohloff, and Gerard W Ryan. Palisade lattice cryptography library user manual. Technical report, New Jersey Institute of Technology, 2017.
70. Raluca Ada Popa, Catherine M. S. Redfield, Nikolai Zeldovich, and Hari Balakrishnan. Cryptdb: Protecting confidentiality with encrypted query processing. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles, SOSP '11*, page 85–100, New York, NY, USA, 2011. Association for Computing Machinery.
71. Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. *Journal of the ACM (JACM)*, 56(6):1–40, 2009.
72. Microsoft Research. Microsoft SEAL (release 4.0). <https://github.com/Microsoft/SEAL>, March 2022. Microsoft Research, Redmond, WA.
73. Nikola Samardzic, Axel Feldmann, Aleksandar Krastev, Srinivas Devadas, Ronald Dreslinski, Christopher Peikert, and Daniel Sanchez. F1: A fast and programmable accelerator for fully homomorphic encryption. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 238–252, New York, NY, USA, 2021. Institute of Electrical and Electronic Engineers.
74. Victor Shoup et al. NTL: A library for doing number theory, 2001.
75. Nigel P Smart and Frederik Vercauteren. Fully homomorphic simd operations. *Designs, codes and cryptography*, 71(1):57–81, 2014.
76. Hassan Takabi, James BD Joshi, and Gail-Joon Ahn. Security and Privacy Challenges in Cloud Computing Environments. *IEEE Security & Privacy*, 8(6):24–31, 2010.
77. Nektarios Georgios Tsoutsos, Charalambos Konstantinou, and Michail Maniatakos. Advanced Techniques for Designing Stealthy Hardware Trojans. In *Design Automation Conference*, pages 1–4, New York, NY, USA, 2014. IEEE.
78. Tune Insight SA. Lattigo v3. Online: <https://github.com/tuneinsight/lattigo>, February 2022.
79. Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the intel SGX kingdom with transient {Out-of-Order} execution. In *USENIX Security Symposium*, pages 991–1008, Berkeley, CA, USA, 2018. USENIX.
80. Alexander Viand, Patrick Jattke, and Anwar Hithnawi. SoK: Fully homomorphic encryption compilers. In *Symposium on Security and Privacy (SP)*, pages 1092–1108, New York, NY, USA, 2021. IEEE, Institute of Electrical and Electronic Engineers.
81. Alexander Viand and Hossein Shafagh. Marble: Making fully homomorphic encryption accessible to all. In *Proceedings of the 6th Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, pages 49–60, New York, NY, USA, 2018. Association for Computing Machinery.
82. Wei Wang, Yin Hu, Lianmu Chen, Xinming Huang, and Berk Sunar. Exploring the feasibility of fully homomorphic encryption. *IEEE Transactions on Computers*, 64(3):698–706, 2013.
83. Chenkai Weng, Kang Yang, Jonathan Katz, and Xiao Wang. Wolverine: Fast, scalable, and communication-efficient zero-knowledge proofs for boolean and arithmetic circuits. In *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021*, pages 1074–1091, New York, NY USA, 2021. Symposium on Security and Privacy.
84. Yuan Xiao, Xiaokuan Zhang, Yinqian Zhang, and Radu Teodorescu. One bit flips, one cloud flops: Cross-vm row hammer attacks and privilege escalation. In *Proceedings of the 25th USENIX Conference on Security Symposium, SEC'16*, page 19–35, USA, 2016. USENIX Association.

85. Andrew C. Yao. Protocols for secure computations. In *Proceedings of the 23rd Annual Symposium on Foundations of Computer Science*, SFCS '82, page 160–164, USA, 1982. IEEE Computer Society.
86. Andrew Chi-Chih Yao. How to generate and exchange secrets. In *27th Annual Symposium on Foundations of Computer Science (sfcs 1986)*, pages 162–167, Toronto, ON, Canada, 1986. IEEE.
87. Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Cross-vm side channels and their use to extract private keys. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, page 305–316, New York, NY, USA, 2012. Association for Computing Machinery.
88. Dmytro Zhuravlev et al. Encrypted program execution. In *2014 IEEE 13th International Conference on Trust, Security and Privacy in Computing and Communications*, pages 817–822, New York, NY, USA, 2014. Institute of Electrical and Electronic Engineers.

## Appendix

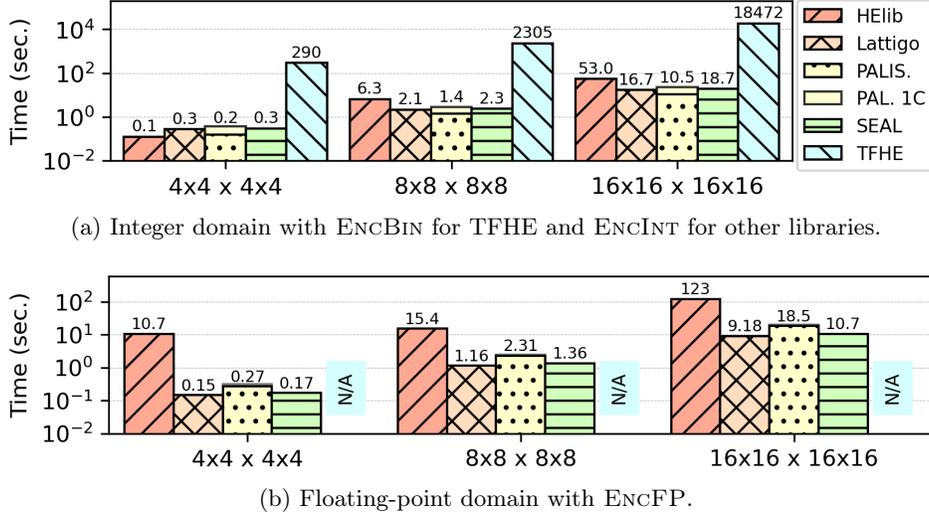
### A FHE Parameter Selection

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 3 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, PALISADE, 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 and PALISADE offer the most flexibility. In the case of PALISADE, it 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).

**Table 3.** Features and ease-of-use of FHE libraries.

Library	Mod Chain Control	Automated Ctxt Maintenance	Configurable Security	Arbitrary Cyclotomics
HELib	○	●	●	●
Lattigo	●	○	●	○
PALISADE	●	●	●	○
SEAL	●	○	○	○
TFHE	○	●	○	○



**Fig. 9.** Measured execution time for the Matrix Multiplication benchmark for square matrices ranging from  $4 \times 4$  to  $16 \times 16$ . TFHE word size  $w = 8$ .

## B Future of Standardized FHE Benchmarks

The T2 benchmark suite comprises a plethora of representative applications of FHE at the time of writing. 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.

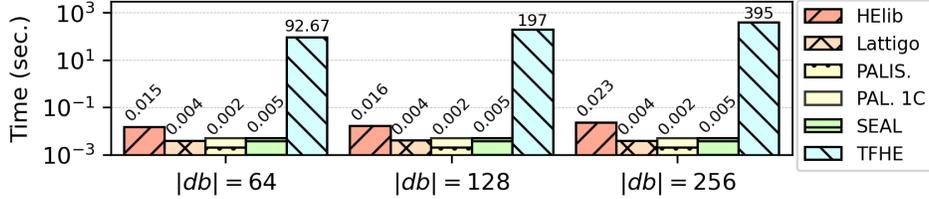
We also aim to draw attention to open problems in FHE using T2, including automated parameterization for general-purpose encrypted computation. Finding the smallest parameter set that works for a given application and results in the fastest execution time is a time-consuming process that requires a lot 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.

## C Extended Experimental Results

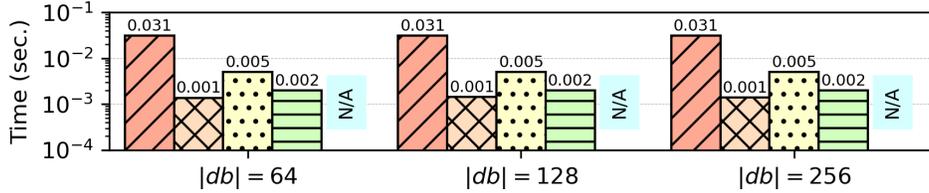
For completeness, we report additional performance results for arithmetic and bitwise benchmarks.

### C.1 Arithmetic Benchmarks

**Matrix Multiplication:** For this benchmark, we use two square matrices in ENCINT and ENCFP. Similarly to the  $\chi^2$  benchmark, in Fig. 9 (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. 9 (b) we observe that HElib is significantly slower than all the other libraries, which contradicts

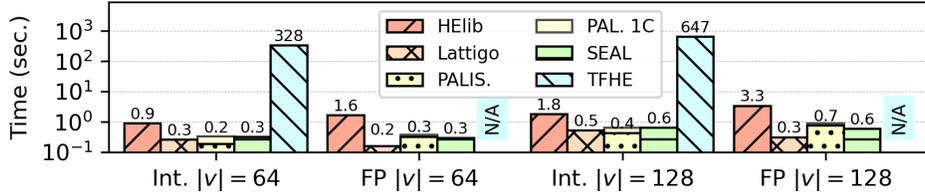


(a) Integer domain with ENCBIN for TFHE and ENCBINT for other libraries.



(b) Floating-point domain with ENCFP.

**Fig. 10.** Measured execution time for the BaPIR 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.

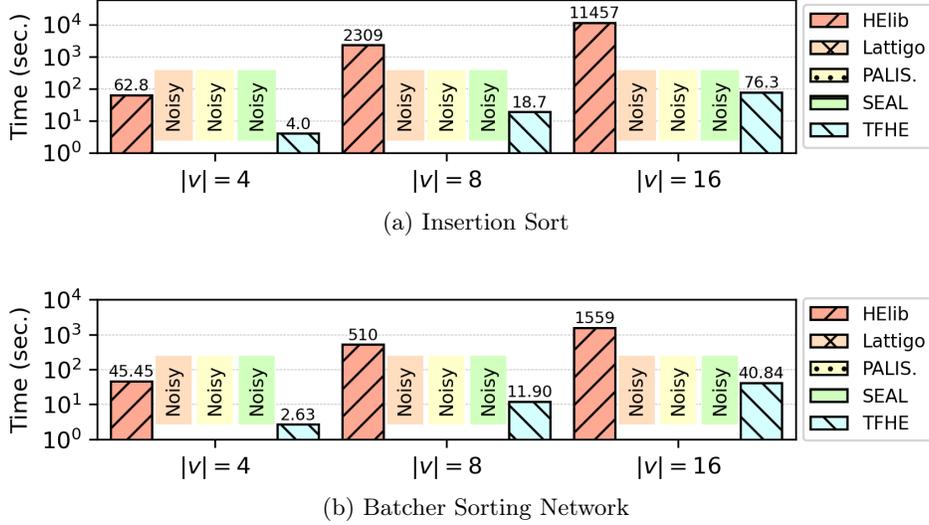


**Fig. 11.** Measured execution time for the Euclidean distance benchmark for both integer and floating-point domains for vectors with 64, 128, and 256 elements. The integer domain uses ENCBINT for the backends, except for TFHE which uses ENCBIN with word size  $w = 8$ . The floating-point domain uses ENCFP for the relevant backends.

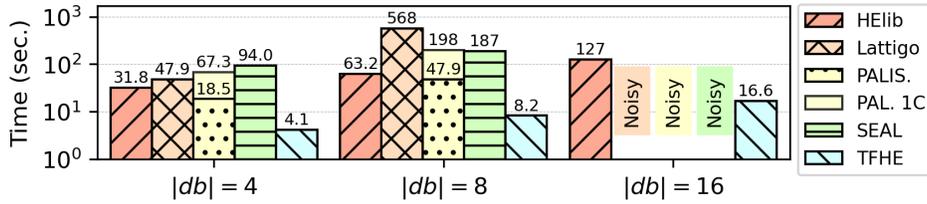
our results in Fig. 5. However, this performance difference is expected since in the matrix multiplication benchmark there is no need to invoke the `rescale` operation after every multiplication, yet HElib invokes it automatically and suffers a performance slowdown.

**BaPIR:** Fig. 10 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 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. 10 (a). This benchmark highlights the two core advantages of all the arithmetic schemes compared to CGGI (i.e., the underlying scheme in TFHE). First, the schemes with batching capability perform only a single multiplication compared to CGGI, which needs to perform as many multiplications as the database size. Second, as ENCBIN has to perform binary multiplications, it incurs additional overhead compared to the fast multiplications in ENCBINT and ENCFP for all the other libraries.

**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. 11 shows the runtime performance for both ENCBINT and ENCFP with vector sizes of 64 and 128 elements. Lattigo, PALISADE, and SEAL are the three fastest libraries, with HElib 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,



**Fig. 12.** Measured execution time for the two oblivious sort benchmarks for different input array sizes with word size  $w = 4$  in the binary domain using ENCBIN for all libraries.



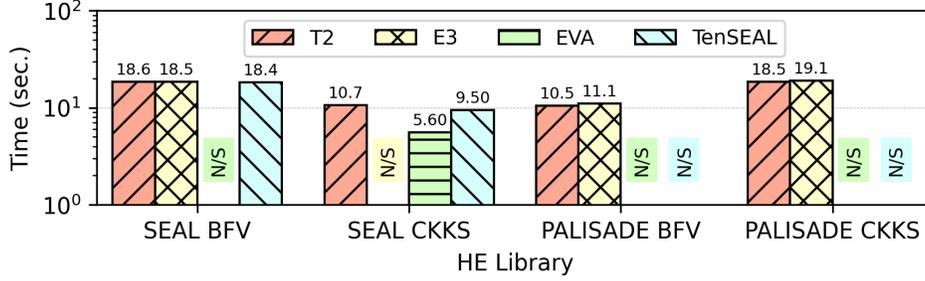
**Fig. 13.** Measured execution time for the BinPIR benchmark in the binary domain with ENCBIN and word size  $w = 5$  for an increasing database size.

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.

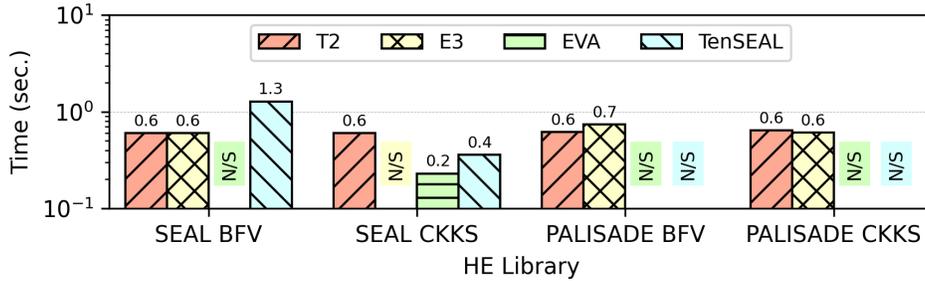
## C.2 Bitwise Benchmarks

**Oblivious Sorting:** Oblivious sorting methods are very deep benchmarks in terms of multiplicative depth. In the case of *insertion sort*, the first elements in the array require a large depth as the algorithm iterates through each element and performs multiplexing procedures with all prior elements during each iteration. Recall from Section 3.4 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. 12 (a), 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. 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.

Similarly to the insertion sort, the *Batcher sorting network* requires a high number of multiplications, and thus, Lattigo, PALISADE, and SEAL were not able to decrypt successfully due to significant noise accumulation. However, in Fig. 12 (b) we observe that both HELib and TFHE are almost one order of magnitude faster for Batchers sorting compared to insertion sort. We remark that the insertion sort performance corresponds to its  $\mathcal{O}(n^2)$  worst-case complexity during data-oblivious execution, whereas the Batchers sorting network has a better worst-case complexity of  $\mathcal{O}(\log^2 n)$ .



**Fig. 14.** Comparisons between T2, E<sup>3</sup>, EVA, and TenSEAL for the matrix multiplication benchmark with  $16 \times 16$  square matrices. The compilers target SEAL BFV and CKKS, as well as PALISADE BFV and CKKS for ENCINT and ENCFP, respectively.



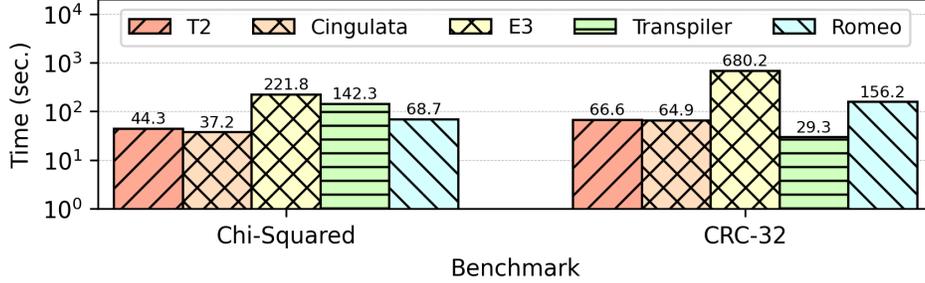
**Fig. 15.** Comparisons between T2, E<sup>3</sup>, EVA, and TenSEAL for the Euclidean distance benchmark with input vectors of 128 elements. The compilers target SEAL BFV and CKKS, as well as PALISADE BFV and CKKS for ENCINT and ENCFP, respectively.

**BinPIR:** The binary PIR benchmark incurs a significant overhead compared to the batched version (BaPIR), as shown in Fig. 13. 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 experiment demonstrates the superiority of the BaPIR benchmark for the libraries that support batching.

### C.3 Comparisons with Optimizing Compilers

To demonstrate how the T2 compiler performs relative to other state-of-the-art works, we perform experiments with four T2 benchmarks across several compiler/backend combinations. Our goal with these comparisons is to show that T2 still performs similarly to state-of-the-art HE compilers even though it is not an optimizing compiler like EVA [32], TenSEAL [10], and Cingulata [19]. To that end, we selected a subset of T2 benchmarks, implemented each algorithm across many popular compilers (EVA, TenSEAL, Cingulata, E<sup>3</sup> [24], Google Transpiler [43], and ROMEO [45]), and allowed the compilers to leverage *all supported optimizations*. The current compilers can be divided into two classes: those that operate in the arithmetic domain (which is synonymous with our ENCINT and ENCFP) and those that operate in the Boolean circuit domain (i.e., ENCBIN). E<sup>3</sup> is the only other framework besides T2 that has support for both computational domains. For fair comparisons, we configured the backends and parameters to be constant across frameworks (at 128 bits of security) and ran all experiments using a single CPU thread.

Figs. 14 and 15 illustrate comparisons with compilers supporting CKKS and BFV using our matrix multiplication and Euclidean distance benchmarks, respectively. For the former, we use square matrices of size 16 while for the latter we use two vectors of 128 elements each. Overall, T2 performs on-par with the state-of-the-art and even outperforms E<sup>3</sup> for PALISADE BFV. TenSEAL and EVA yield the best performance for most of the backends/schemes, which can be attributed to the fact that these compilers were designed



**Fig. 16.** Comparisons between T2, Cingulata, E<sup>3</sup>, Google Transpiler, and Romeo for the  $\chi^2$  and the CRC-32 benchmarks targeting TFHE for ENCBIN.

specifically with encrypted tensors in mind and each will automatically determine the best batching strategy for these tensor operations to minimize latency. The only exception is TenSEAL for BFV in Fig. 15 which performed slower than T2 and E<sup>3</sup>, because it needed a bigger parameter set to avoid noisy results. In general, we see a similar scaling between the two benchmarks (Figs. 14 and 15), which verifies that T2 is competitive with state-of-the-art optimizing compilers. We note that the T2 compiler is the only one of these works that has functional support for both HELib and Lattigo and is the only compiler that has support for SEAL BFV, as well as CKKS and PALISADE BFV (note: lack of support is indicated in Fig. 14 as “N/S” for *not supported*).

Fig. 16 depicts comparisons with multiple compilers that support the TFHE library as a cryptographic backend for both our chi-squared benchmark with an 8-bit word size and CRC-32 with a 32-bit word size. T2 is faster than both E<sup>3</sup> and ROMEO across both benchmarks and is only outperformed by the optimizing compiler Cingulata by a small margin. While ROMEO uses advanced logic optimizations to minimize the FHE boolean circuits, it does not support mixed operations with constants and therefore it needs to encrypt all values used in a program, even those that are considered public. Because both the chi-squared and CRC-32 algorithms incorporate numerous public values, it performs poorly as ciphertext-ciphertext operations are significantly slower than ciphertext-plaintext counterparts. Interestingly, the Google Transpiler outperforms all other compilers for the CRC-32 application, but exhibits poor performance for chi-squared; this suggests that the Transpiler optimizes FHE circuits for applications that exhibit primarily bitwise operations like CRC-32, but produces suboptimal circuits for arithmetic-heavy programs like chi-squared.

**Summary:** Our analysis indicates that T2 generates code with performance analogous to state-of-the-art optimizing compilers. Therefore, enabling comparisons between different libraries with a general-purpose compiler like T2 is of significant importance as the runtime differences between different backends indicate the strength and weaknesses of each underlying library and rather than differences in the T2 compilation for different target libraries. This is also one of the foundations on which the T2 compiler has been built on: using the same functional units and encoding methodologies across all supported libraries. For all alternative encodings, the comparisons with other works demonstrate that the T2 compiler indeed offers competitive performance, and outputs programs of similar caliber to what can be generated by alternative state-of-the-art compilers.

## D Programming in T2DSL

Our custom C-like language supports syntax familiar to developers that program in high-level languages. Figures 17 and 18 show the T2DSL used to generate the homomorphic programs for all supported backends for the squared euclidean distance and CRC-32 benchmarks respectively. In both cases, the depicted code is intuitive and mimics the way that programmers would implement these applications on plaintext data.

```

1  int main(void) {
2      EncInt dist, d;
3      EncInt[] v, u;
4      int i;
5      v = { 10, 15, 3, 9 };
6      u = { 11, 13, 3, 10 };
7      dist = 0;
8      // dist = Sum( (v_i - u_i)^2 )
9      tstart();      // Start timer
10     for (i = 0; i < (v.size); i++) {
11         d = (v[i]) - (u[i]);
12         dist += (d * d);
13     }
14     tstop();        // Print execution time
15     print(dist);   // Decrypt and print
16     return 0;
17 }

```

Fig. 17. Squared Euclidean distance in T2DSL.

```

1  int main(void) {
2      EncInt crc, data, temp;
3      int i, j;
4      crc = 4294967295; // 0xFFFFFFFF
5      data = 1717921090;
6      tstart();      // Start timer
7      for (i = 3; i >= 0; i--) {
8          temp = (data << (i*8));
9          temp >>= 24;
10         crc = crc ^ temp;
11         print(crc);
12         for (j = 0; j < 8; j++) {
13             temp = crc >>> 1;
14             crc = (crc) ? (temp ^ 3988292384) : temp;
15         }
16         print(crc);
17     }
18     crc = ~ crc;
19     tstop();        // Print execution time
20     print(crc);    // Decrypt and print
21     return 0;
22 }

```

Fig. 18. CRC-32 program in T2DSL.