E^3: A Framework for Compiling C++ Programs with Encrypted Operands

Eduardo Chielle, Oleg Mazonka, Nektarios Georgios Tsoutsos and Michail Maniatakos

Abstract

The dramatic increase of data breaches in modern computing platforms has emphasized that access control is not sufficient to protect sensitive user data. Even in the case of honest parties, unknown software/hardware vulnerabilities and side-channels can enable data leakage, leading to the conclusion that as long as data exists decrypted, it can be leaked. Fortunately, recent advances on cryptographic schemes allow end-to-end processing of encrypted data, without any need for decryption. However, besides the reported impractical overheads, such schemes are particularly hard to use by non-crypto-savvy users, which further inhibits their applicability. In this work, we propose the first usability-oriented framework that enables programmers to incorporate comprehensive privacy protections in their programs, by automatically protecting user-annotated variables using encryption. As a proof of concept and without loss of generality, our E^3 framework incorporates three state-of-the-art FHE libraries. In our evaluation, we validate the usability of E^3 by employing various benchmarks written in C++, and directly compare the overhead of the core FHE libraries in terms of runtime performance, as well as memory and storage requirements. While FHE is used as a base study, E^3 can be used as the base for performance comparison of any encrypted computation methodology.

Index Terms

Data Privacy, Fully Homomorphic Encryption, General-purpose computation, Privacy-preserving computation

I. Introduction

The recent disclosure of the Meltdown and Spectre side-channel vulnerabilities is yet another painful reminder of the insecurity of modern computing platforms [1], [2]. With the potential to affect billions

E. Chielle, O. Mazonka and M. Maniatakos are with the Center for Cyber Security, New York University Abu Dhabi, UAE. E-mail: {eduardo.chielle, om22, michail.maniatakos}@nyu.edu

N. G. Tsoutsos is with the Department of Electrical and Computer Engineering, University of Delaware, Newark, DE. E-mail: tsoutsos@udel.edu

of devices, from smartphones to cloud servers, these critical flaws could be exploited remotely (e.g., via JavaScript) and leak sensitive information from privileged memory locations by judiciously probing the cache memory. Equally, the numerous compromises on cloud platforms such as LastPass, Amazon EC2/S3 and Dropbox [3], as well as subtle attacks described is the academic literature (e.g., [4]–[7]), contribute to the lack of trust on behalf of end users, and highlight the shortcomings of contemporary access control mechanisms, even if the information custodians are honest parties.

At the same time, the assumption that the hardware is the ultimate root-of-trust for the software stack continues to be questioned, as attack trends move closer to the silicon itself. For example, stealthy modifications on hardware designs (i.e., hardware Trojans [8]–[11]), can cause substantial security risks, including privilege escalation, denial of service, information leakage or downgraded performance [12], [13]. Likewise, a hardware Trojan at the sub-transistor level, which is capable of extracting information directly from the CPU pipeline, has also been reported in the literature [14]. Evidently, the lack of hardware-level trust goes beyond academic threat models: There exist folklore reports about a hardware Trojan kill-switch in military air defenses, in the context of electronic warfare [15], [16].

In many contexts, the need for data privacy can be addressed using encryption, which provides implicit access controls to authorized entities with knowledge of a private key. Nevertheless, in light of the security threats mentioned in the previous paragraphs, without end-to-end encryption the privacy risks are merely reduced, but not eliminated. For example, Intel's software guard extensions (SGX) [17] allow programmers to protect sensitive data within enclaves by encrypting all memory traffic in and out of the processor package; however, since the data are eventually decrypted before entering the processor pipeline, leakage is still possible, as recently reported [18]. Thus, even though encryption can protect data at rest and in transit, protecting *data in use* remains a challenge.

All is not lost, however: Gentry's remarkable discovery of the first fully homomorphic encryption (FHE) scheme in 2009 allows evaluation of arbitrary-degree polynomials over encrypted variables [19], which is rightly dubbed as cryptography's holy grail [20]. Specifically, given that combinational Boolean circuits can be directly mapped into polynomials, and because Boolean circuits are fundamentally equivalent to Turing Machines [21], FHE has the potential to support computation over encrypted values. The latter could be a candidate solution to protecting the privacy of data in use, by mitigating information leakage risks in light of system vulnerabilities and side-channel attacks.

Problem Statement. Despite the countless applications and the versatility of modern FHE schemes, an important drawback remains their high complexity and low efficiency [22]. Even though recent advances in FHE research have improved the ciphertext sizes and function evaluation efficiency, the *practicality of state-of-the-art schemes remains limited*: For example, the FHE evaluation of an AES-128 operation requires about four minutes [23], while ciphertext sizes can be several hundred kilobytes [24]. Moreover, the complexity of expressing real-life applications as FHE-compatible functions remains another important factor that limits the usability for wide deployments. In fact, *converting a high-level program into an FHE-compatible form is laborious*, as it entails implementing a functionally equivalent circuit. Thus, writing real-life programs that operate on FHE ciphertexts sacrifices the familiar experience of writing program statements in a standard language (e.g., C/C++), which could be increasingly challenging for non-crypto-savvy (but otherwise skillful) programmers.

Our Solution. In this work, we address the usability gap of FHE schemes by allowing programmers to effortlessly incorporate end-to-end privacy guarantees in their programs. Towards that end, we present Encrypt-Everything-Everywhere (E^3): a user-friendly framework that simplifies and accelerates the conversion of standard C++ programs into secure versions that operate over ciphertexts. In our approach, end users annotate their sensitive variables using our novel data types, and our framework automatically compiles each program assignment statement into the corresponding FHE-compatible circuit. Our easily-extensible framework currently offers a selection of three state-of-the-art FHE libraries, namely TFHE [25], FHEW [26], and HElib [27], which allow seamless evaluation of the FHE circuits corresponding to each C++ statement. To the best of our knowledge, E^3 is the first comprehensive framework integrating the manipulation of FHE ciphertexts into standard C++ programs, which can significantly increase the usability of FHE without any expectation regarding cryptography knowledge on behalf of the programmers. In addition to protecting the privacy of actual C++ applications with FHE-based end-to-end encryption, E^3 enables direct comparisons of existing FHE libraries in terms of performance, memory and storage overheads, using the same C++ benchmarks. Furthermore, E^3 can be used with any other methodology for manipulating encrypting data, such as partially homomorphic encryption.

Contributions. In this work, we claim the following contributions:

1) A novel framework for seamlessly incorporating FHE evaluation in C++ programs, in order to provide end-to-end privacy to the users. The framework is open-sourced to also serve as the basis for objective comparisons of existing and new encrypted computation libraries.
2) A performance comparison and efficiency analysis of three state-of-the-art FHE libraries, namely TFHE, FHEW, and HElib, using C++ benchmarks compiled with E^3.

While the presented framework is developed for C++, the methodology is applicable to any imperative programming language. Also, open sourcing the framework enables repeatability and verification of the presented experiments.

Paper Roadmap. The rest of the paper is organized as follows: In Section II we provide a preliminary discussion on Learning with Errors (LWE) and the three FHE libraries incorporated in our framework, while Section III discusses the usability of our secure data types for FHE ciphertexts from the perspective of the C++ programmer. In Section IV we expand on the internals of our E^3 framework and provide an overview of how regular programs can be compiled after FHE operations and variables are instantiated, while in Section V we elaborate on the critical step of integrating our framework with existing FHE libraries. Our evaluation and FHE library comparisons using oblivious benchmarks are presented in Section VI. Section VII summarizes our analysis findings, a brief discussion on related work is provided in Section VIII, and our concluding remarks are presented in Section IX.

II. Preliminaries

The goal of this work is to provide an accurate and fair comparison of the different FHE libraries. Therefore, the first step is to understand the underlying cryptographic schemes employed by the various FHE libraries, in order to use them appropriately. Moreover, a fair comparison requires further understanding of the security levels offered by the FHE libraries under different configurations, allowing us to draw performance comparisons for comparable security levels.

A. Learning with Errors

The Learning with Errors (LWE) problem, which was formally defined in Regev's seminal paper [28], has been the centerpiece in many lattice-based constructions in modern cryptography. As a generalization to the learning with parity problem [29], LWE can be proven to be as hard as the worst lattice problems [28], which offers extraordinary versatility for the design of cryptographic schemes. Moreover, given that the shortest vector problem in lattices (for the average case) is hard [30], and since there are no efficient quantum algorithms to solve lattice problems, LWE-based cryptography does not suffer from post-quantum threats, as is the case with factoring and the discrete logarithm problems.

The LWE problem can be defined in two basic variants: as a *decision* and a *search* problem [31]. Specifically, if n and p are positive integers, $\mathbf{s} \in \mathbb{Z}_p^n$ is a secret vector, and $\chi : \mathbb{Z}_p \to \mathbb{R}^+$ is some probability distribution on $\mathbb{Z}_p$, we can define a new probability distribution $A_{\mathbf{s},\chi}$ on $\mathbb{Z}_p^n \times \mathbb{Z}_p$ as follows: we choose a vector $\mathbf{a} \in \mathbb{Z}_p^n$ uniformly at random and $e \in \mathbb{Z}_p$ according to χ, and output $(\mathbf{a}, b) = (\mathbf{a}, \langle \mathbf{a}, \mathbf{s} \rangle + e) \in \mathbb{Z}_p^n \times \mathbb{Z}_p$, (where $\langle \cdot, \cdot \rangle$ denotes a dot product of two vectors). The decision problem entails deciding whether the pairs $(\mathbf{a}, b)$ are sampled according to $A_{\mathbf{s},\chi}$ or according to the uniform distribution on $\mathbb{Z}_p^n \times \mathbb{Z}_p$. On the other hand, the search problem entails recovering $\mathbf{s}$ from $(\mathbf{a}, b)$ pairs sampled according to $A_{\mathbf{s},\chi}$.

Likewise, if $\Phi_m(x) = x^n + 1 \in \mathbb{Z}[x]$ is an irreducible (*cyclotomic*) polynomial of degree $n = \phi(m) = m/2$, where $m = 2^k$ for a positive integer k, then $R = \mathbb{Z}[z]/\langle \Phi_m(x) \rangle$ is the ring of integer polynomials modulo $\Phi_m(x)$ [32]. Moreover, if n is a power of 2 and $p = 1 \bmod m$ is a prime modulus, then $R_p = R/\langle p\rangle = \mathbb{Z}_p[x]/\langle x^n+1\rangle$ is the ring of integer polynomials modulo both $\Phi_m(x)$ and p. In this case, if $s \in R_p$ is secret, $a \in R_p$ is chosen uniformly at random, χ is a distribution over R_p and e is chosen according to χ, the decision ring-LWE problem entails distinguishing $(a, b) = (a, a \cdot s + e) \in R_p \times R_p$ from uniformly random $(a, b) \in R_p \times R_p$, while the search ring-LWE problem entails finding the secret $s \in R_p$ with high probability [32].

Instances of the LWE problem can be characterized based on the probability distribution χ used for sampling e; if χ is a discrete Gaussian distribution with center 0 and width parameter αp, an instance of LWE could be specified using the dimension n, the modulus p and the scaling parameter α [33].[1] Typically, $\alpha \in (0, 1)$ and $\alpha p > 2\sqrt{n}$ (e.g., $p \approx n^2$), which allows to reduce the decision shortest vector problem (GapSVP) to LWE [31].

B. Estimating the hardness of LWE

Typically, solving the LWE problem entails running a *lattice-basis reduction* algorithm (such as LLL [35], BKZ [36], or BKZ 2.0 [37]) to find a reduced basis of a lattice [33]. If $\mathbf{b}_0$ is the shortest non-zero vector in the reduced basis of lattice Λ, the quality of the reduction can be estimated using the *root-Hermite factor* $\delta_0 \geq 1$; in particular, for an n-dimension lattice Λ it holds that $\|\mathbf{b}_0\| = \delta_0^n \cdot det(\Lambda)^{1/n}$ [38]. Since smaller δ_0 values are better, the runtime of the underlying lattice-basis reduction algorithm is a function of the desired δ_0 (which restricts how small is $\|\mathbf{b}_0\|$).

For example, solving the decision LWE problem requires finding a short vector $\mathbf{v}$ that satisfies $\mathbf{v} \cdot \mathbf{a} = 0 \bmod p$ [38] (this is essentially an instance of the Short Integer Solutions problem [30]). In this case, the distinguishing advantage is approximately $\exp(-\pi(\|\mathbf{v}\| \cdot \alpha)^2)$ (i.e., it is a function of the norm $\|\mathbf{v}\|$) [33]. Thus, to achieve a certain advantage, the lattice-basis reduction algorithm should ensure a minimum δ_0 by running for a sufficiently long time. Likewise, solving the search LWE problem entails solving a Bounded Distance Decoding problem using the Nearest Plane algorithm [38]; in this case, the probability of finding the secret $\mathbf{s}$ is also determined by the reduction quality and δ_0. Hence, the security level of LWE-based cryptosystems depends on the effort required by lattice-basis reduction algorithms to achieve a target δ_0 for a specified advantage.[2]

[1] In case of binary-secret LWE, the secret $\mathbf{s}$ is in $\{0,1\}^{n \log p}$ (increased dimension) [34], and the LWE instance is characterized using an additional parameter ψ corresponding to the distribution of the components of $\mathbf{s}$ [33].

[2] A security-level estimator for LWE instances is available in [39].

C. LWE-based cryptographic schemes

One prominent application of LWE-based cryptography is fully homomorphic encryption (FHE). In fact, modern FHE cryptosystems, such as the BGV scheme [40], which corresponds to second generation FHE, as well as the GSW scheme [41], which corresponds to third generation FHE, are based on (ring-)LWE. These FHE schemes are implemented in state-of-the-art C/C++ libraries, such as TFHE, FHEW and HElib.

TFHE [42] implements a variant of the GSW scheme, as described in [25], [43], and supports the evaluation of any Boolean circuit on encrypted values. Specifically, TFHE supports circuits of unrestricted size that are composed using different binary gates (such as AND, OR, NAND, NOR, XOR, XNOR), as well as the unary NOT gate and a multiplexer (MUX) gate. Unrestricted circuit sizes are possible by incorporating *bootstrapping* [44] in the evaluation of each gate; gate-by-gate bootstrapping reduces the noise of FHE ciphertexts after each operation and allows applying an arbitrary function *after the data are already encrypted* (i.e., it is not necessary to know the target function during encryption). As noted by the TFHE authors, gate-bootstrapping enables the evaluation of manually defined, as well as automatically generated Boolean circuits without size restrictions. Moreover, the default parameters of TFHE provide a security level $\lambda > 100$ bits for intrinsic lattice reduction quality $\delta_0 = 1.0058$ and BKZ 2.0 lattice-basis reductions [25].

FHEW [45] is a predecessor of TFHE and the first FHE library supporting per-gate bootstrapping using a homomorphic accumulator based on a variant of the GSW scheme [26]. The latest version of FHEW implements a functionally-complete set of bitwise operations (i.e., NAND, NOR, AND, OR, NOT) on the encryptions of bits, and each ciphertext output is refreshed using bootstrapping to reduce its noise to similar levels as the corresponding inputs. Thus, FHEW enables the homomorphic evaluation of any Boolean circuit on encrypted bits. Moreover, using the default parameters, FHEW achieves a security level $\lambda > 100$ bits for root-Hermite factor $\delta_0 = 1.0065$, assuming BKZ 2.0 lattice-basis reductions [26].

HElib [46] implements a ring-LWE variant of the BGV scheme (second generation FHE) with the GHS optimizations (detailed in [47]). Implemented in C/C++ with multi-threading support, the library provides bootstrapping (dubbed *recryption*) operations to reduce ciphertext noise [27], as well as *batching* (using the SV technique [48]) to pack multiple plaintexts within the same ciphertext and enable SIMD-style operations. For a given LWE modulus to noise standard deviation ratio (q/σ), HElib's security level λ depends on the cyclotomic ring parameter m that bounds the LWE dimension $n = \phi(m)$: for example, if $n \geq 29.1 \cdot \log{(q/\sigma)}$ then $\lambda > 100$ bits [47, Eq. 8], based on the LWE security estimates from [38].

HElib's programming interface offers routines for homomorphic addition and multiplication of FHE ciphertexts, which are internally represented as vectors over a polynomial ring [47]. For the native plaintext space of binary polynomials R_2, HElib's homomorphic operations can be mapped to addition and multiplication in $GF(2)$ (i.e., bitwise XOR and AND operations). In this case, finite field arithmetic of characteristic 2 is convenient for composing binary circuits of higher-level (compound) functions, such as equality, division, modulo, bitwise manipulations and comparison operations. Nevertheless, HElib also supports a plaintext space of polynomials R_p, where p can be an odd prime, which enables addition and multiplication in $GF(p)$.

Remark: As discussed on Section V-A3, this work evaluates HElib in two different ways: (a) using the native plaintext space of binary polynomials R_2 that enables addition and multiplication in $GF(2)$, we compose higher-level functions on multi-bit inputs as *circuits* (specifically, we implement multi-bit adders and multipliers), and (b) using a plaintext space of ring polynomials R_{127}, we evaluate integer addition and multiplication in $GF(127)$ directly, using multi-bit inputs as coefficients of *polynomials*.

III. Encrypting C++ variables: Programmer's view

A. *A sample program - Assumptions*

In order to put the usability dimension into perspective, we need to outline our assumptions for the programmer skills, as well as our framework design requirements. Specifically, we assume that the programmer:

1) Does not need to understand how cryptography works;
2) Can annotate which variables should be encrypted, understanding that the performance impact is related to the amount of operations performed on encrypted variables;
3) Can optimize performance by setting maximum sizes to variables, understanding that such upper limits can make a significant different in performance;
4) Can develop/adapt the C++ source code to refrain from branching based on encrypted data, understanding that this will leak information and should be avoided.

The last assumption is a direct outcome of the *termination problem* in the program's control flow: If the program is allowed to branch on encrypted values, then information about the encrypted value itself is leaked. Because of the nature of homomorphic operations, even one leakage instance could allow the attacker to decrypt all data in the program. To address this problem, the algorithm needs to execute for a predetermined, upper-bounded amount of iterations (i.e., *obliviously*), which would ensure that a correct result will be reached. An example of this algorithm transformation is presented in Listing 1, presenting a variation of the Fibonacci algorithm. In this example, variables to be protected have been annotated with the `SecureInt` keyword. The main loop calculating the final output iterates `MAX_NUM` times, which is

```
 1 #include <iostream>
 2 #include "secureint.h"
 3 #define MAX_NUM 10
 4
 5 int main()
 6 {
 7     using Secure = SecureInt<8>;
 8     Secure num = _7_E;
 9     Secure f1 = _0_E;
10     Secure f2 = _1_E;
11     Secure fi = _0_E;
12     Secure i = _1_E;
13     Secure res = _0_E;
14
15     for( int cntr=0; cntr<MAX_NUM; cntr++ )
16     {
17         res += (i == num) * fi;
18         fi = f1 + f2;
19         f1 = f2;
20         f2 = fi;
21         i++;
22     }
23     std::cout << "fib: " << res << "\n";
24 }
```

Listing 1. C++ Programming Example.

the upper bound of the computation. In the example of Listing 1, the programmer is requesting the value in location 7 of the Fibonacci sequence defined in Line 8: `num=_7_E`, where the user-defined suffix `_E` indicates an encrypted constant. Therefore, the correct output is selected by the `i==num` comparison in Line 17: When `i` reaches encrypted 7, then `fi` is added to `res`, as the `i==num` expression will return an encrypted 1. In any other case, an encrypted 0 is added to `res`, not affecting the final output. Independent of the input, the above algorithm will always run for 10 iterations (Line 3), therefore not leaking any context about the user input.

In C++ terms, the termination problem means that a `SecureInt` cannot be implicitly cast to a `bool`, and the compiler will return an error. Thus, the expression '`if (x>y) {}`', with `x,y` defined as `SecureInt`, will fail to compile. This affects usability, as existing algorithms may need modifications to be converted to a data-oblivious version. Moreover, it affects practicality too, as a fixed upper bound will cause further performance degradation.

With regards to the framework design, we outline the following requirements:

1) The framework needs to maintain an accurate state after the execution of each statement, which is a requirement of imperative programming languages. In other words, if the program stops at any time and its encrypted variables are to be decrypted, the decryption needs to exactly match the value of the unencrypted version.
2) Everything should compile using a standard conforming C++ compiler (e.g., GCC), and should be

TABLE I

STANDARD C++ OPERATORS AND THEIR USE WITH ENCRYPTED DATA.

Non-applicable	Unchanged	Overloaded		
		Using circuits		Implemented in C++
		Direct	Indirect	
:: a() . -> .* ->* *a	&a sizeof new delete new[] delete[] throw a,b alignof typeid	a++ a-- ++a --a -a !a ~a a*b a/b a%b a+b a-b a>>b a<<b a>b a<b a>=b a<=b a==b a!=b a|b a^b a&b a&&b a||b a?b:c[1]	a*=b a/=b a%=b a+=b a-=b a>>=b a<<=b a|=b a^=b a&=b	(type)[2] +a a=b "a"_b a<<i[3] a>>i a<<=i a>>=i a[][4]

[1] The ternary operator cannot be overloaded in C++, therefore we implement it as a function (MUX).

[2] Explicit conversion between SecureInt and SecureBool, and between SecureInt of different sizes.

[3] Shift by unencrypted number.

[4] Access to individually encrypted bits.

loaded and executed in the exact same way as standard executables.

3) As long as the program compiles, it should work as expected. Encrypted processing caveats (such as branching on encrypted data) should be caught and reported during compilation.

These design requirements aim to enhance usability. An alternative approach would be to extract and separate the private computation from the main program and treat it as a separate computational block; While this would potentially improve performance, it severely affects usability as it impacts the familiar software development environment of imperative programming languages. The programmer cannot debug line-by-line, and unexpected control flow variations (low-level interrupts, inter-process communication signals, debugging breakpoints, etc.) would leave the programmer with non-understandable memory contents.

Therefore, in this work, we treat the C++ statements as atomic computational blocks which would yield the same results as their unencrypted counterparts. In essence, this implies that a) lazy evaluation is not allowed, and b) arithmetic needs to strictly be in power-of-2 ring, e.g., 256 (8-bit), 512 (9-bit), etc., otherwise bitwise manipulations and arithmetic overflows would yield unexpected or counter-intuitive (unsound) results. Moreover, we provide the option to explicitly set the exact value of bit size per variable. This slightly deviates from standard C++ and its predefined variable bit sizes `char`, `short`, `int`, `long`.

In modern computers, using a 32-bit `int` instead of a 16-bit `short` does not incur overhead. In encrypted computation, however, doubling the number of bits of the plaintext can lead to a dramatic increase in performance overhead. As experimental results demonstrate, even 1 bit can make a big difference in performance, affecting the practicality dimension. Explicit size declaration can be found in functional programming languages, so the programmer may already be familiar with this practice. In the example of Listing 1, all variables are 8 bits (Line 7).

As discussed in Section II, modern FHE libraries encrypt single bits. Therefore, `SecureInt` is essentially an array of independent encrypted bits. Processing `SecureInts` is similar to processing data using Boolean circuits. For example, in case of addition, the `+` operator maps to an adder of the appropriate bit size. Every gate of the adder, however, is evaluated using the underlying FHE library.

B. Supported C++ operators

The usability requirement implies that all standard operators are available to the programmer when using encrypted variables. Table I summarizes the C++14 operators, classified into 5 groups:

Non-applicable: This groups consists of member and structure reference/dereference operators, as well as function call and scope resolution. These operators are not intended to be defined for `SecureInt`.

Unchanged: These operators retain their native default semantics.

Overloaded: These operators are overloaded for `SecureInt`. The class defines these operators, which in turn call the appropriate functions corresponding to the semantics of unencrypted data. Some class operations do not require manipulation on encrypted data; for example, copy, or expanding/shrinking the number of bits. These operators are implemented purely at a high-level without calling circuit functions, and appear in the 'Implemented in C++' category. All the other overloaded operations (e.g., `a+b`) require calls to *functions implementing Boolean circuits using FHE-evaluated gates* (dubbed "FHE circuits"). These operators can be further classified into two categories: 'Direct', which actually call FHE circuit functions, and 'Indirect', which do not call such functions directly but are expressed using Direct operators. Usually in C++, compound assignment operators (such as `a+=b`) serve as building blocks for their counterpart operators. For example, the operator `a+b` is expressed as `t=a;t+=b`. In other words, the semantics of a binary operator (not bitwise) are defined by the corresponding compound assignment operator. Nevertheless, when processing encrypted variables, we have the opposite case: the compound assignment operators have to be defined via their binary counterparts, since each FHE circuit function defines a *referentially transparent* function with its output being distinct from any of its inputs.

C. Mechanics of the `SecureInt` *class*

Our `SecureInt` class is built on top of a uniform `FHE API` we have developed. This `FHE API` consists of the following components:

- `Bit` - a `class` representing one bit. The class defines constructors, assignment operators (copy and move) along with export and import to and from a string in encrypted form.
- Secret and evaluation keys generation with loading and saving capabilities.
- Functions to encrypt and decrypt one bit using FHE.
- List of logic gates - *referentially transparent* functions taking one or more `Bits` as arguments and returning one `Bit`;
- a `Bit` instance for encrypted bit *zero* and a `Bit` instance for encrypted bit *one*, if the evaluation key is available.

The motivation for our `FHE API` is to abstract the different APIs of existing FHE libraries, so that the C++ source code becomes oblivious to the underlying FHE library. The advantage of this approach is that a new FHE library can be plugged-in without any change to the implementation of the `SecureInt` class, so the programmers simply need to link their compiled binary with the corresponding FHE library and the newly generated 'Framework API .cpp file'. In this work, we have developed adapters to all three FHE libraries in our scope.

The data representation of `SecureInt` is an array of `Bits` sized to the template parameter of the class. For different N, each template specialization `SecureInt<N>` realizes an independent class. Therefore, binary operators, including multiplication, between `SecureInt<`N_1`>` and `SecureInt<`N_2`>` of different sizes are not defined.

However, `SecureInt` can be promoted or downcast using the explicit cast operator to enable incompatible binary operations. For example, in order to convert `SecureInt<8>` to `SecureInt<16>`, the cast operator pads the 8 most significant bits with `Bit` instances of encrypted zero provided by the `FHE API`. Downcasting is done by discarding `Bits` from the array. Every overloaded operator in `SecureInt` is a method of the class that is templated by the size N (i.e., the number of bits). If the operator is from the "Direct" group (Table I), the corresponding FHE circuit function is called; these circuit functions are `static noexcept private` members of `SecureInt`, but are still templates of the parameter N.

In C++, logical operators on `int` result in `bool` type. In case of `SecureInt`, a logical operator must produce an encrypted 0 or 1, so it cannot be of the `bool` type. Therefore, another `SecureInt` should hold the encrypted result. Nevertheless, this approach is suboptimal as `SecureInt` is defined to hold multiple bits. Hence, similar to how C++ produces `bool` type out of logical expressions, we introduce a new class `SecureBool`, which is derived from `SecureInt<1>` and inherits its functionality. Additionally, `SecureBool` defines multiplication and conditional operators between `SecureBool` and `SecureInt<N>`, so, even though multiplication between `SecureInt<1>` and `SecureInt<N>` is forbidden, the latter is allowed between `SecureBool` and `SecureInt<N>`, resulting in `SecureInt<N>` type.

Using the `SecureBool` class provides substantial performance improvements to the selector operations, without any burden to the programmer. Consider the following expression: `(a<b)*c`. If only the `SecureInt` class was available, this expression would invoke an FHE circuit function for comparisons, followed by a circuit function for multiplication. The latter is a complex operation, and in case of multi-bit inputs, it is quite expensive. On the other hand, if `(a<b)` results in `SecureBool`, the multiplication can be defined as an operator between `SecureBool` and `SecureInt` invoking only a Boolean multiplexer FHE circuit function, which is significantly simpler and faster to evaluate. In both cases, the expression evaluates to the same result and has the same `SecureInt` type. We remark that this happens obliviously, without the user being aware that `SecureBool` exists. Still, the programmer can use `SecureBool` class explicitly in the program. In summary, our `SecureInt` class has the following properties:

- Exposes an internal type `Bit` aliased to `FHE API`'s `Bit`, and provides access to individual `Bits` by overloading the `[]` operator.
- Exports and imports its encrypted representation into a string.
- Offers functions for FHE encryption and decryption.[3]
- Defines all "Overloaded" C++ operators of Table I.
- Defines an explicit cast operator between `SecureInt` of different sizes.
- Defines an explicit cast operator to `SecureBool` (which is different from a `SecureInt<1>` cast, as it entails reducing all encrypted bits using an FHE OR circuit, following the C++ convention that any non-zero value is Boolean `true`).

IV. Building FHE-encrypted executable: Framework's view

As discussed in the previous section, the programmer should be completely oblivious to the mechanics of instantiating FHE schemes. This section discusses the process taking place behind the scenes, after the programmer compiles the source code. The process diagram for the compilation and execution of a C++ program using our framework appears in Fig. 1.

[3]These functions work obviously only when a secret key is defined, which is true during pre-processing the user's program, post-processing the results, or during debugging.

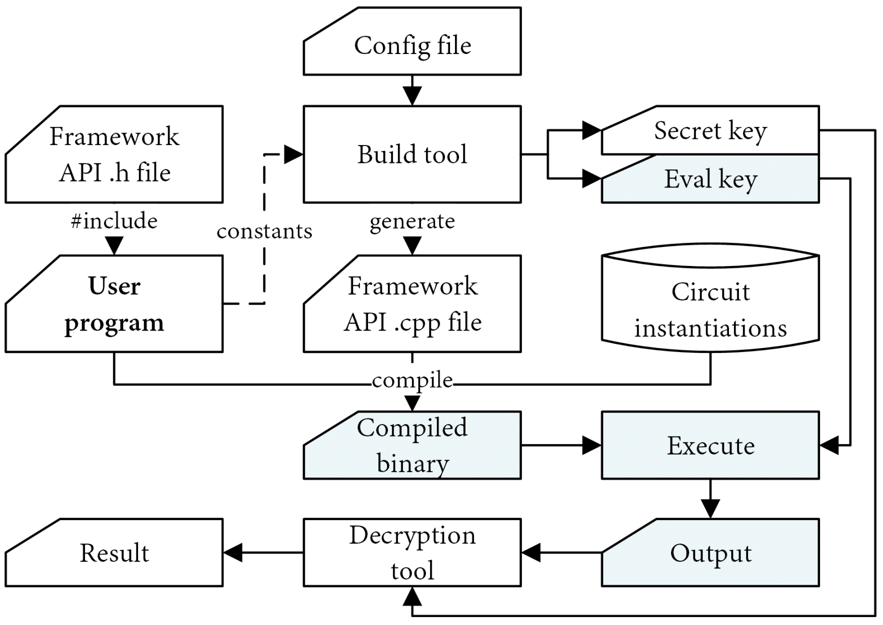


Fig. 1. Process diagram presenting the components required to compile and execute a C++ program operating on FHE encrypted data. The shaded parts can be outsourced to a third party.

The process starts with the 'User program', which is C++ code (for example, the source code in Listing 1). The 'User program' needs to `#include` the 'Framework API .h file' of our framework (i.e., secureint.h), to have access to the new secure data types. Furthermore, encrypted constants have to be appropriately annotated (e.g., `7_E`).

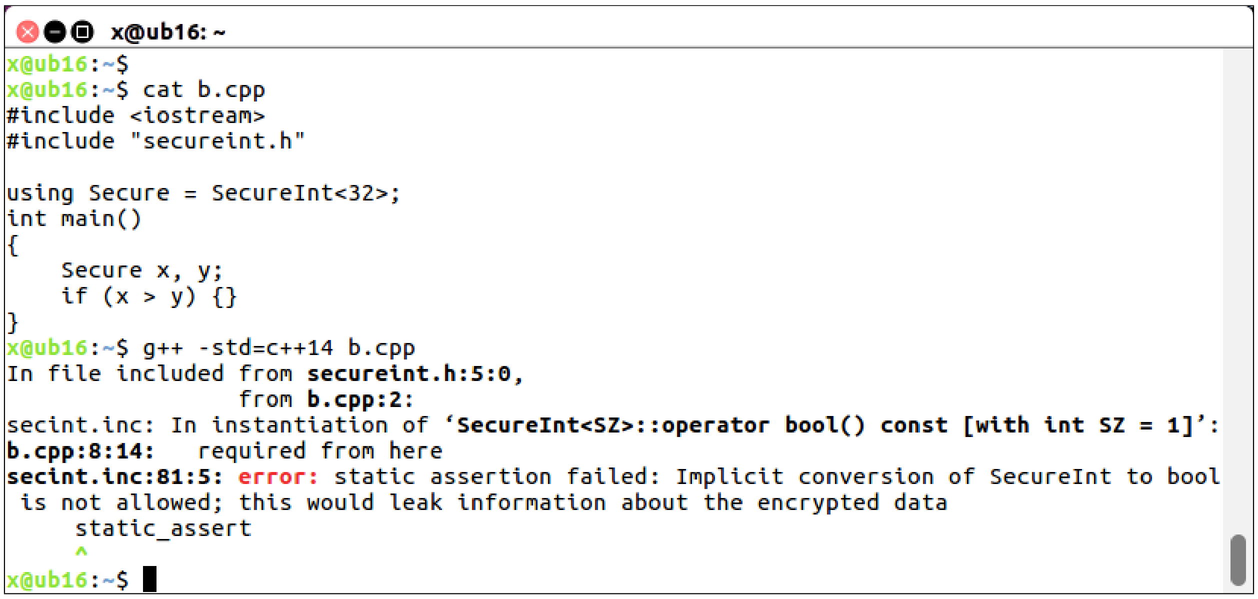
A. Compilation process

Before the programmer compiles for the first time, the framework needs to generate the appropriate set of secret/evaluation keys. This process is required once per program. The programmer has the option to re-use the same keys for different programs by setting the configuration file parameters accordingly, given that the keys are stored in a file. Generating keys can take up to a few minutes, as shown in Table II, therefore the first compilation will take much longer compared to the subsequent ones. Generation of keys is required before the first compilation, since the secret key is needed to encrypt the constants in the source code. Therefore, our developed 'Build tool' is generating the appropriate 'Secret key' and 'Eval key', for a given 'Config file' delineating the FHE scheme to be instantiated, and generates the 'Framework API .cpp file'.

The latter, besides the instantiation of our overloaded operators, also contains encryptions of program constants. In our framework, we use the `_E` suffix to: 1) allow the building tool to extract the list of constants used in the program; 2) encrypt these constants into string literals; and 3) update the string literal operator (`std::string operator ""_E(unsigned long long x)`) defined in the `SecureInt` implementation file with the returns of string literals representing the encrypted constants. Without this automated process, the programmer would have to manually instantiate the FHE cryptosystem, generate the secret key, and use it once per encrypted constant, replacing the user-defined literal (e.g., `7_E`) with the corresponding encrypted value. This value would expand over numerous lines of code, since every ciphertext is in the order of 2KB-10MB, severely affecting usability. Notably, the compilation time is also affected by the introduction of new encrypted constants: Every time the programmer declares a new constant (not currently in the pool of already encrypted ones), the 'Build tool' needs to encrypt it and make it available to the compiler.

One of the requirements outlined in the previous Section is that if the program compiles, it works as expected. Thus, meaningful compiler errors are very important in the process. The most critical one is the implicit conversion of `SecureInt` or `SecureBool` to `bool`: For example, consider the statement `if(x>y)`{}, where `x` and `y` are `SecureInt`. In C++, if the arguments were integer, the > operator returns a `bool`. In our framework, the > operator returns a `SecureBool`, which cannot be used in an `if` statement. Typically, C++ implicitly converts any data type to `bool`, with the common convention that zero values convert to `false` and non-zero values convert to `true`. Nevertheless, this convention would be wrong and misleading here, therefore the compiler needs to stop the compilation and highlight the error. Fig. 2 presents an error message from our framework. In this case, the `SecureBool` class defines the cast-to-`bool` operator, which uses templates –along with `static_assert` C++ mechanisms– to produce a meaningful error message to the user.

The last part of the building process is to instantiate the group of 'Direct' operators which, as discussed in Section III, are directly mapped to FHE circuits. The 'Circuit instantiation' database (input in Fig. 1 and output in Fig. 3) is a collection of *explicit template specializations* of all possible combinations of FHE circuit functions and possible numbers of bits, where the bit size is the template argument. In our work, this collection is generated separately for each FHE library, because the evaluation of the FHE gates has different performance; hence, the optimal function for evaluating each FHE circuit is different as well. The optimizer of a standard conforming C++ compiler must remove the specializations (the FHE circuit functions) that are not used in the program (i.e., those not implicitly *instantiated*). The other groups of the overloaded operators (namely, "Indirect" and "Implemented in C++") are implemented as non-specialized template members without calls to FHE circuit functions.



```
x@ub16: ~
x@ub16:~$
x@ub16:~$ cat b.cpp
#include <iostream>
#include "secureint.h"

using Secure = SecureInt<32>;
int main()
{
    Secure x, y;
    if (x > y) {}
}
x@ub16:~$ g++ -std=c++14 b.cpp
In file included from secureint.h:5:0,
                 from b.cpp:2:
secint.inc: In instantiation of ‘SecureInt<SZ>::operator bool() const [with int SZ = 1]’:
b.cpp:8:14:   required from here
secint.inc:81:5: error: static assertion failed: Implicit conversion of SecureInt to bool
 is not allowed; this would leak information about the encrypted data
     static_assert
     ^
x@ub16:~$ 
```

Fig. 2. Compilation fails due to implicit conversion of `SecureBool` to `bool` (SZ=1 because `SecureBool` is derived from `SecureInt<1>`).

B. *Encrypted binary execution*

In order to execute a compiled binary, the evaluation key needs to be loaded. Table II indicates that the evaluation key size ranges from 78MBs to 2.5GB, impacting the loading and execution overhead of a given binary. As expected, this may cause a significant increase in memory usage during execution, while certain ciphertext properties could further impact the execution time. In HElib, for instance, each bit requires 10MBs of storage. Hence, depending on the FHE ciphertexts incorporated in the binary, the execution times and the memory consumption overheads are impacted accordingly.

In outsourced computation scenarios, the user/programmer would need to transmit the binary and the evaluation key to the third party, and receive the encrypted output. In this case, the third party needs to have the appropriate execution environment, which includes the corresponding FHE library and its prerequisites. We remark that the FHE library can be statically linked into the binary: This would increase the binary size by ≈4MBs for TFHE and FHEW, or by 32MBs for HElib. The computation host also needs to have FFTw3 installed for TFHE and FHEW, while HElib requires GMP and NTL. Apparently, the fundamental bottleneck in outsourced computation scenarios is the transmission of the evaluation key and the FHE ciphertexts. For the Fibonacci code of Listing 1, the final binary+evaluation key size is 82MB, 2.5GB, 1GB for TFHE, FHEW, and HElib respectively.

Finally, as soon as the execution terminates, the end user runs our decryption tool that loads the secret

TABLE II

FHE LIBRARY OVERHEADS (KGT/ENCT/DECT - KEY GENERATION/ENCRYPTION/DECRYPTION TIME (MS), SKSZ/EVSZ/CTXTSZ - SECRET KEY/EVALUATION KEY/CIPHERTEXT SIZE).

Lib	KGT	EncT	DecT	SkSz	EkSz	CtxtSz
TFHE	317	0.024	0.00024	78MB	78MB	2KB
FHEW	11,066	0.004	0.0005	1.13KB	2.5GB	2KB
HElib	53,659	175	78	1GB	1GB	$\approx$10MB

key and decrypts the results. For TFHE and HElib, the secret key has the same size as the evaluation key, while for FHEW the secret key is significantly smaller. The time for decrypting a ciphertext is reported in Table II. We observe that all libraries have practical decryption times.

V. ADDING FHE LIBRARIES: INTEGRATOR'S VIEW

The function body for the overloaded C++ operators is retrieved from a 'Circuit Instantiations' database, which is different for every FHE library and for every `SecureInt` size. In order to generate this database, 'Verilog prototypes', which are Verilog modules with equivalent functionality to the required C++ functions, are synthesized into gate-level netlists using the Synopsys Design Compiler against a 'Standard Cell Library'. The latter is a collection of building block cells (i.e., logic gates) with their corresponding properties (e.g., area, timing, etc.). The process of adding an FHE library to our framework is summarized in Fig. 3.

A. Building a custom Standard Cell Library

Each FHE library exposes different functions to the user. The first step towards building our custom 'Standard Cell Library' is to *execute* these functions and *benchmark* the performance of each FHE gate operation. Afterwards, based on our performance evaluation, we can assign values to the *area property* of each logic gate in the 'Standard Cell Library' that are proportional to the runtime performance of the corresponding FHE gate. This process informs the Synopsys Design Compiler to optimize a gate-level netlist for area, which ultimately allows prioritizing the faster FHE circuit functions in the final 'Circuit Instantiation' file.

1) TFHE: The TFHE API provides the following FHE gates (building blocks): NOT, AND, NAND, OR, NOR, XOR, XNOR, and MUX. This set is functionally complete and allows us to build any compound FHE circuit. In Table III, we summarize the performance evaluation of TFHE's building

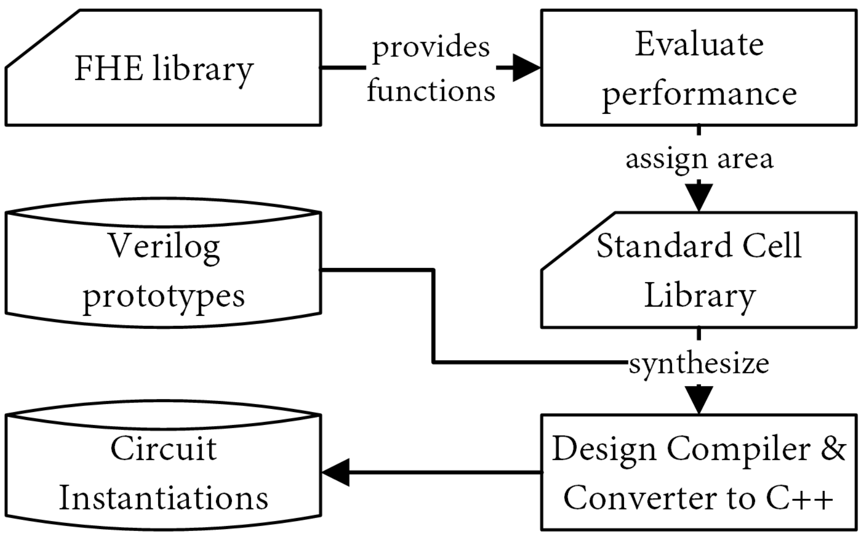


Fig. 3. Process diagram describing the addition of an FHE library to the framework.

blocks; we observe that NOT gate evaluation is practically free, while all the remaining FHE gates have comparable performance (around 13ms). One exception is MUX, which takes twice as long to evaluate.

2) FHEW: The FHEW API provides the NOT, AND, NAND, OR, and NOR gates. Similar to TFHE, the supported set of FHE gates is universal, NOT is practically free, and all other gates exhibit similar performance (although slower compared to TFHE). Since XOR, XNOR, and MUX are not directly provided, we compose them using the other supported gates. An important concern with FHEW is that *it does not support gate operations on dependent ciphertexts* – evaluation halts with the message "ERROR: Please only use independant ciphertexts as inputs." (sic). Unfortunately, there is no workaround to this, since the library does not provide a function that can check for such dependencies. Therefore, the usability of FHEW is impacted due to this limitation, as the library may not be able to support all possible user programs in its current version. For example, a `SecureInt` expression (`a+a`) generates the error mentioned above.

3) HElib: HElib is fundamentally different, compared to TFHE and FHEW, as it exposes arithmetic operators instead of Boolean gates. Therefore, we need to compose other FHE gates using the available `+`, `*` operators, which correspond to XOR and AND in $GF(2)$:

$$\text{AND}(a, b) = a \cdot b \qquad \mod 2$$

$$\text{OR}(a, b) = a + b + (a \cdot b) \qquad \mod 2$$

$$\text{NOT}(a) = 1 + a \qquad \mod 2$$

$$\begin{aligned}
&\text{NAND}(a, b) = 1 + (a \cdot b) && \mod 2\\
&\text{NOR}(a, b) = (1 + a) \cdot (1 + b) && \mod 2\\
&\text{XNOR}(a, b) = 1 + a + b && \mod 2\\
&\text{XOR}(a, b) = a + b && \mod 2\\
&\text{MUX}(a, b, c) = a \cdot (b + c) + c && \mod 2
\end{aligned}$$

Based on these composed Boolean gates, in Table III we compare the overhead of HElib versus TFHE and FHEW. In all benchmarks, we configure each FHE library with a matching security level $\lambda \geq 100$ bits. As illustrated in Table III, HElib is several orders of magnitude slower compared to TFHE and FHEW, which limits its practicality.

Unlike TFHE and FHEW, bootstrapping in HElib has to be explicitly invoked to decrease noise. In fact, noise accumulation is the reason why the NAND gate operation is significantly slower compared to a sequence of AND and NOT operations with independent operands. At the end of each FHE gate operation, we check the noise of the ciphertext before returning: If the noise passes the defined noise threshold, we invoke the bootstrapping (*recryption*) process. In this work, the noise threshold was determined heuristically to optimize for performance, given hundreds of evaluations: Higher noise thresholds ensure that bootstrapping is invoked less frequently, which improves the overall runtime performance. In all cases, for all the presented results in this work, we always verify the correctness of HElib calculations at the end of every program execution to validate that the selected noise threshold is appropriate.

Remark 1: The results in Table III for HElib are generated using dependent ciphertexts, which accumulate noise faster so that more frequent bootstrapping is necessary. This allows us to determine *worst-case situations*, given that real-life programs exhibit multiple dependencies among variables. As soon as the bootstrapping noise thresholds across different gates are estimated, we are able to determine the area size parameter of each FHE gate in our Standard Cell Library accordingly.

Fig. 4 presents the time required for FHE evaluation of the various gates as ciphertext noise accumulates over time, for HElib. Since bootstrapping needs to be invoked explicitly, it is necessary to perform enough experiments to determine the asymptotic performance of the gate over time. Fig. 4 indicates that the average time needed to evaluate the various gates converges after approximately 100 iterations.[4]

Remark 2: The selection of parameter $\mathtt{p}$ in HElib (i.e., the characteristic prime) limits the different values in each R_p polynomial coefficient. Using the native plaintext space where $\mathtt{p}$=2, each coefficient can be either 0 or 1, so all the above arithmetic operations are modulo 2. Evidently, simulating bit-level

[4]For TFHE and FHEW, since bootstrapping occurs during each gate evaluation, the convergence is almost instant.

TABLE III

TIME (MS) TO EVALUATE A GATE (NON-NATIVE GATES ARE MARKED WITH ITALIC).

Gate	TFHE	FHEW	HElib
NOT	$5.97{\cdot}10^{-4}$	$3.69{\cdot}10^{-4}$	*44.15*
AND	13.0	73.7	53,328
NAND	12.8	73.8	*73,250*
OR	13.3	72.9	*59,838*
NOR	15.7	74.0	*76,520*
XOR	12.7	*224*	3.45
XNOR	13.0	*221*	*37.57*
MUX	24.4	*217*	*104,396*

gate operations by allocating distinct ciphertexts for each individual plaintext bit would be inefficient and such gate constructions would be affecting performance, since HElib is designed to naturally support vectors over large plaintext spaces in each ciphertext.

To illustrate the performance overhead when HElib is used to simulate Boolean circuit operations (so that each individual bit is encrypted as a separate ciphertext), we benchmark 7-bit addition and 7-bit multiplication under two use scenarios: In the first scenario (dubbed "Circuit" for convenience) we instantiate a 7-bit adder and a 7-bit multiplier using the simulated Boolean gates presented above, so that each input bit is encoded in a different ciphertext and `p=2`. In the second scenario (dubbed "Polynomial" for convenience), we use `p=127` so that HElib can add and multiply polynomial coefficients directly (i.e., without simulating adder and multiplier circuits). Table IV summarizes our comparison results: we observe that "Circuit" addition is approximately 153 times slower than "Polynomial" addition, while multiplication is about 15 times slower. For completeness, the Table also reports performance results for TFHE and FHEW (corresponding only to the "Circuit" scenario). We observe that "Polynomial" addition in HElib is faster than "Circuit" addition in FHEW, while TFHE is the fastest.

B. Generating Circuit Instantiations

Following the process outlined above, we create a Standard Cell Library for each FHE cryptosystem in scope. Unlike actual hardware circuits that evaluate their inputs in parallel, software evaluation of circuit gates is not inherently parallelized, since the host processor evaluates each gate sequentially. Therefore, by optimizing our automatically generated circuits for area, we achieve better performance during evaluation. In this work, the area of each gate is set according to our benchmark results reported in Table III.

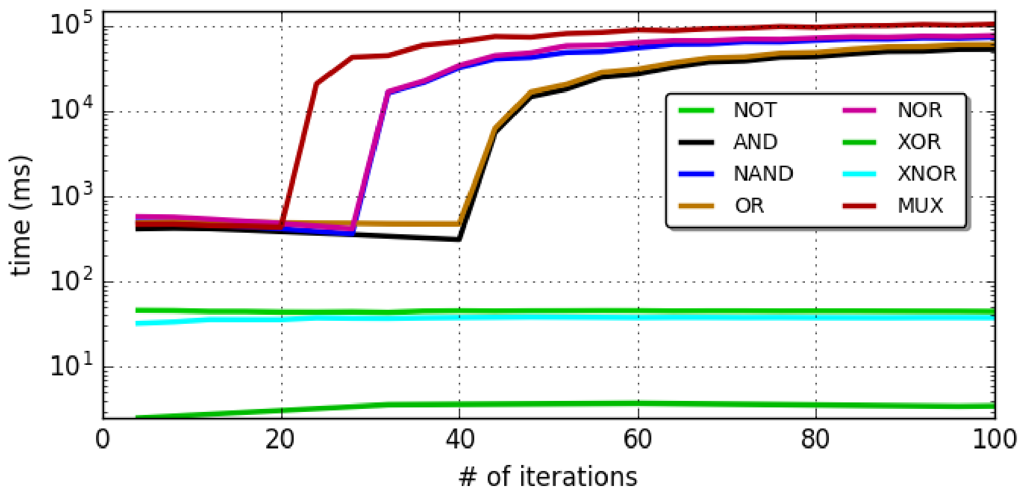


Fig. 4. Average time of one evaluation of HElib gates over sequential evaluations accumulating noise.

TABLE IV

OVERHEAD OF 7-BIT ADDITION AND MULTIPLICATION IN HELIB ("CIRCUIT" AND "POLYNOMIALS"), TFHE AND FHEW (IN MS).

Op	HElib (Poly)	HElib (Circ)	TFHE	FHEW
+	2,430	372,432	357	3,480
*	51,770	767,998	1,257	10,416

Equipped with our Standard Cell Libraries, we generate FHE circuit instances for each overloaded C++ operator and all supported sizes, for all three FHE libraries in scope. Our approach is to describe each Boolean circuit as a register-transfer-level (RTL) abstraction using the Verilog hardware description language. In this work, we have developed a comprehensive database of RTL Verilog implementations for all circuits corresponding to all possible overloaded C++ operators, where the size of each `SecureInt` operand is parameterizable. Notably, most of these C++ operators have an exact syntactic and semantic match in RTL Verilog; only the `++`, `--` operators are not supported, and they are synthesized with appropriate Verilog expressions. Another exception is that the ternary operator in Verilog expects a bit value as the control signal, therefore a `SecureInt` control value needs to be OR-reducted in Verilog.

Our framework offers an automated process that constructs all RTL Verilog files for all desired operators and sizes. These operator descriptions are sent to the Synopsys Design Compiler, which processes Verilog sources and optimizes circuit designs, together with a corresponding Standard Cell Library that informs the "cost" of each Boolean gate. The Design Compiler translates the RTL Verilog input into structural

Verilog (i.e., a *netlist* of logic gates) and optimizes the resulting Boolean circuit for each Standard Cell Library according to the area cost of each gate. Finally, we developed a custom parser which converts each circuit description from the structural Verilog netlist of logic gates into C++ specialization template functions that evaluate a corresponding FHE gate. In our database, these functions are sorted/indexed based on the underlying FHE cryptosystem and bit size.

VI. Evaluation results

A. Experimental setup

All results in this section were collected using a computer equipped with an Intel i7 processor and 8 GBs of RAM, running Ubuntu 16.04.1 and GCC version 5.4.0. For our experiments, we used TFHE version 1.0, FHEW version 2.0b, and HElib commit ID `65ef24c` on GitHub. With regards to dependencies, we used GMP version 2.6.1, NTL version 11.0.0, and FFTw3 version 3.3.4.

As discussed in Section V-A3, instantiating HElib with a security level $\lambda \geq 100$ bits offers limited practicality. Therefore, in order to understand how HElib performance scales when introducing different bit sizes and benchmarks, we include a configuration offering 0 bits of security (based on the formula in [47, Eq. 8]), also used by the authors of HElib in [49]. The version offering 0 bits of security is dubbed 'HElib-0', while the version of HElib offering a security level comparable to TFHE and FHEW is dubbed 'HElib-1'.

B. Performance of C++ operators

In our experiments, we focus on the evaluation of the overloaded 'Direct' operators category of Table I, since the overhead of the 'Implemented using C++' category is negligible, and the 'Indirect' category operator performance reduces to the corresponding 'Direct' operator. To avoid unnecessary clutter in the results, we first group operators by comparable performance, as shown in Table V. This grouping is natural, as operators within each group are performing similar operations. It should be noted that this grouping was performed with respect to performance consistency across all three FHE libraries; the grouping would be simpler if we consider only one library. For example, groups {eq} and {and}, as well as groups {bit} and {xor} could be merged as one for the TFHE library, since the results are consistent across different bit sizes. We note that this set of experiments presents 4-bit plaintexts, as runtime becomes prohibitive for larger bitsizes for HElib-1.

Fig. 5 shows the performance across our operator groups. Given the results, we can make the following observations:

TABLE V

GROUPING OF OPERATORS. VARIABLES `a`, `b`, `c` ARE `SecureInt`; `z` IS `SecureBool`.

Name	Operators
{inc}	`a++  a--  ++a  --a  -a`
{mul}	`a*b`
{add}	`a+b  a-b`
{gt}	`a>b  a<b  a>=b  a<=b`
{and}	`a&&b  a||b`
{xor}	`a^b`
{bmx}	`z?a:b`
{neg}	`~a`
{not}	`!a  (SecureBool)a`
{div}	`a/b  a%b`
{sh}	`a>>b  a<<b`
{eq}	`a==b  a!=b`
{bit}	`a|b  a&b`
{mux}	`z*a`
{smx}	`a?b:c`

- The bitwise negation operator {neg} is fast across all libraries. The execution time for FHEW and TFHE are fractions of a millisecond, and HElib time is fractions of a second.
- Division is the most costly operation across all the libraries, followed by multiplication. HElib-1 division ({div}) has the worst performance among all operations, with a 20 minute overhead. Division support for encrypted variables is a serious impediment to achieving practicality.
- An optimization offered by our framework is the presence of the `SecureBool` class, which can multiplex expressions much faster compared to the multiplication of `SecureInts` (such as Line 17 of Listing 1). The results corroborate that using the Boolean multiplexer {mux} (`z*a`) instead of multiplication ({mul}) leads to an efficiency improvement of 5x for TFHE, 10x for FHEW, 20x for HElib-0, and 130x for HElib-1.
- The binary bitwise XOR operator ({xor}) is fast in HElib. Helib-0 XOR is 100x faster than TFHE XOR, and 40x faster than HElib-1 XOR. TFHE XOR is 15x faster than FHEW XOR.
- With regards to results consistency across operator performances, indicated by the average and the variance of the results, we identify that TFHE exhibits the most consistent behavior, followed by

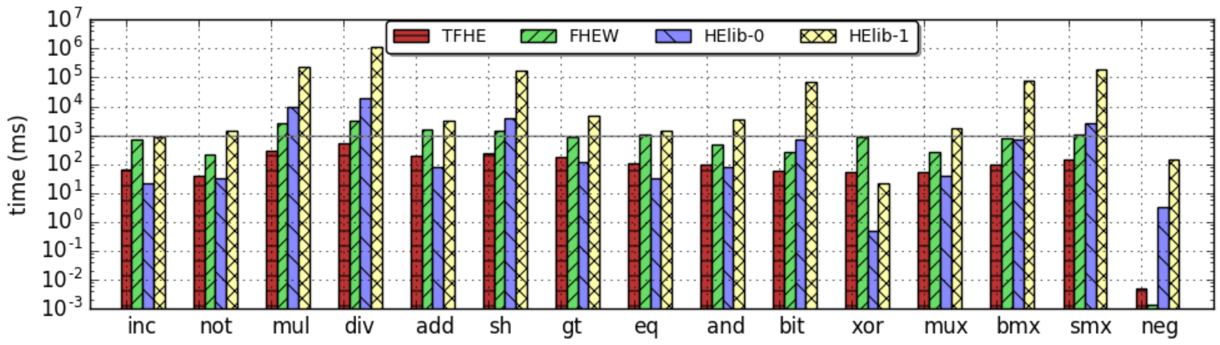


Fig. 5. Time required to execute the operators for `SecureInt<4>`. The gray horizontal line corresponds to 1 second.

FHEW, HElib-0, and HElib-1, in order. Average values and variance of logarithms for TFHE, FHEW, HElib-0, HElib-1 are {150, 1100, 2700, 130000} and {0.57, 0.67, 8.2, 8.8}, respectively.

The second set of results, presented in Fig. 6, provides insight on how performance degrades when increasing the effective plaintext size. We explore 2-, 4-, 8-, 16-, and 32-bit plaintexts. We only present results for the most general groups of programming operations: two arithmetic groups: {add} and {mul}, comparison {gt}, and the {not} group, which consists of the cast to `SecureBool` and the logical `!` operators. As mentioned above, HElib-1 does not scale above 4-bit plaintexts, so results are not shown. Similarly, only TFHE was able to provide results for 32-bit plaintexts in reasonable time. We further observe that:

- TFHE and FHEW demonstrate roughly linear performance degradation as the effective plaintext size increases. The results are consistent across different operators. HElib-0, on the other hand, degrades much faster for {add} and {gt} as the bitsize increases from 4 to 8 bits, due to extensive bootstrapping.
- TFHE is faster compared to the other two libraries. Moreover, TFHE on 32-bit plaitexts consistently outperforms FHEW on 16-bit plaintexts. Still, a 32-bit multiplication using TFHE takes 29.4 seconds, limitig its practicality in real programs operating on standard 32-bit integers.

C. *Comparisons using benchmarks*

In order to evaluate real programs, we use eight data-oblivious benchmarks from the TERMinator Suite [50]. These benchmarks manipulate sets of encrypted variables (i.e., the equivalent of `SecureInts`) using predetermined execution paths, i.e., without branches, to prevent leaking information about the ciphertexts. The provided algorithms are classified into three categories: 1) encoder benchmarks, such

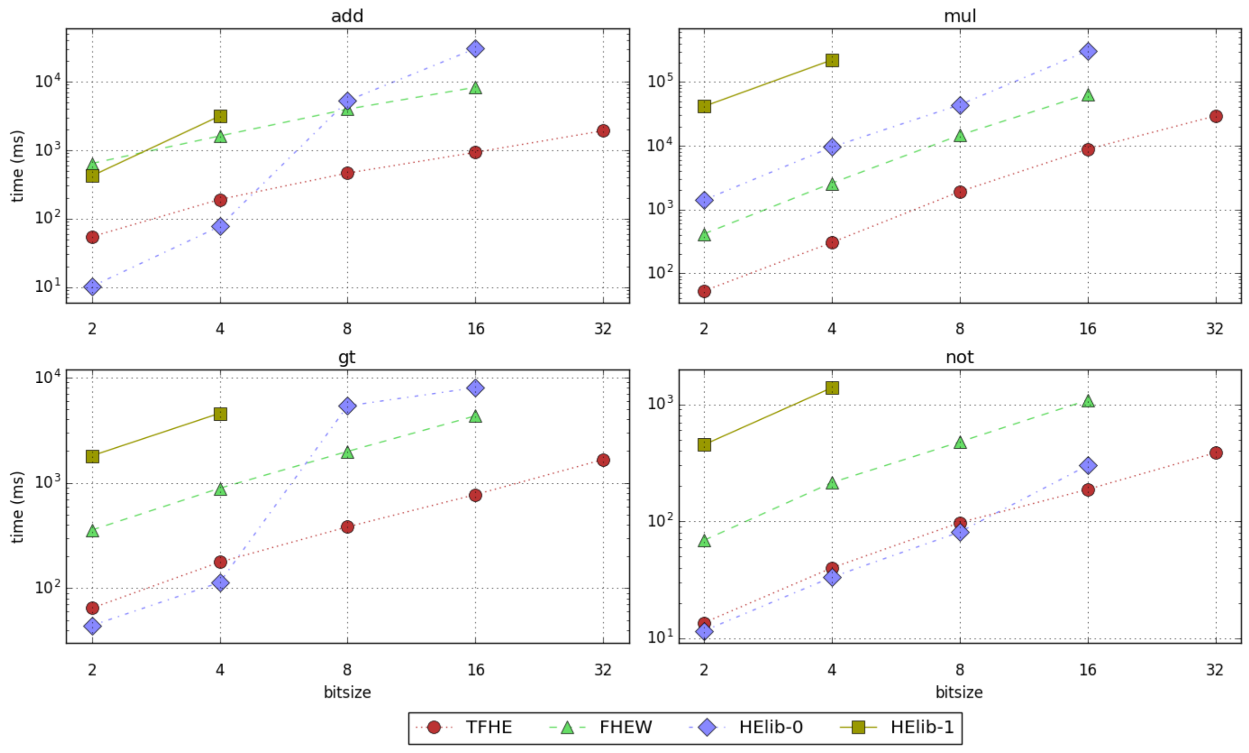


Fig. 6. Time required to evaluate selected groups of operators: {add}, {mul}, {gt}, and {not}, for different effective plaintext sizes and FHE libraries.

as Jenkins (JEN) and the Speck Cipher (SC), which implement real-life bitwise-intensive cryptographic and hash applications, 2) kernel benchmarks, such as Bubble Sort (BS), Matrix Multiplication (MM), Insertion Sort (IS), and Sieve of Eratosthenes (SoE), which stress arithmetic and logical operators, and 3) microbenchmarks, such as the multiplication-intensive Factorial (FAC) and the addition-intensive Fibonacci (FIB).

For this set of results, Fig. 7 presents the performance overhead of encrypted variables using 16-bit and 32-bit (only for TFHE) effective plaintext sizes, as our usability objective requires having sufficient variable bitsizes (we remark that only TFHE can practically scale to 32-bit plaintexts). Also, since HElib-1 does not practically scale over 4 bits, we only report results for HElib-0. The performance degradation is measured as the ratio of the execution time using `SecureInts` over the time of using unencrypted `int` types. We immediately observe that, following our discussion in Section VI-B, TFHE is the fastest library, followed by FHEW and HElib.

With regards to the benchmarks performance, Jenkins exhibits the worst degradation, due to its rich set of operation, including additions, XORs, and bit shifts. Conversely, the Sieve of Erathosthenes instantiates

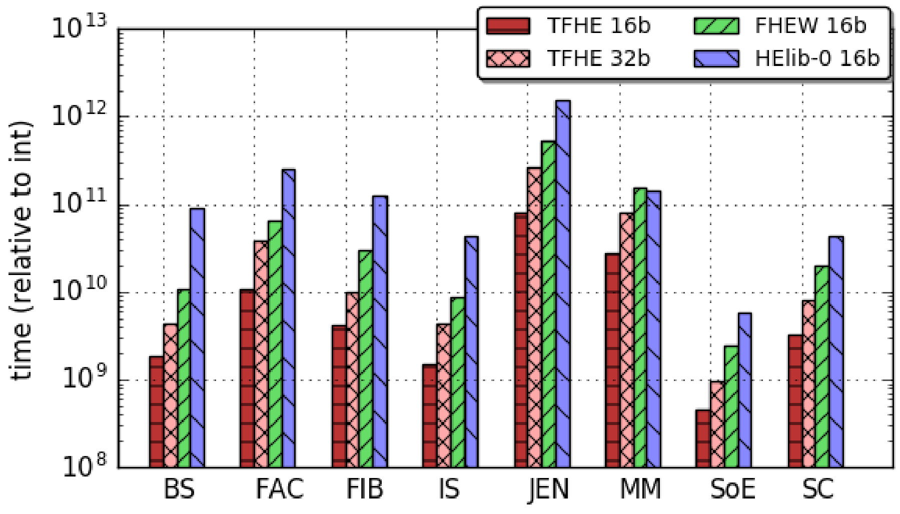


Fig. 7. Impact of using state-of-the-art FHE libraries to encrypt the variables in real applications, for effective plaintext sizes of 16-bits (all FHE libraries) and 32-bits (TFHE only).

TABLE VI

MEMORY CONSUMPTION (IN MB) WHILE EXECUTING FIBONACCI, FOR DIFFERENT EFFECTIVE PLAINTEXT SIZES.

Lib	Gate	2-bit	4-bit	8-bit	16-bit	32-bit
TFHE	375	375	375	375	375	375
FHEW	2500	2500	2500	2500	2500	-
HElib-0	38	38	44	51	64	-
HElib-1	1100	1200	1600	-	-	-

only copying and Boolean multiplexer operators, which are significantly faster compared to other operator classes.

We note that all the eight benchmarks are returning the expected and correct output, suggesting that the design requirements of our framework are met, and that encrypted processing works correctly and obliviously to the programmer. Moreover, the benchmark implementations did not trigger the aforementioned exception in FHEW regarding operations on dependent variables (Section V-A2).

The final set of results discusses the memory consumption of our benchmarks. Since the memory usage is consistent across benchmarks, Table VI presents the result only for the Fibonacci benchmark. In the first column of the table we also report the overhead of each library while evaluating a single FHE gate. Our results indicate that both TFHE and FHEW have constant memory overheads, independent of the effective plaintext size. Conversely, in HElib the amount of required memory grows as the effective plaintext size increases.

VII. Analysis Findings

Integrating the three FHE libraries under a unified framework was an extremely challenging task, due to vast differences in approaches, implementation, and documentation.

TFHE, besides being the fastest library, it provides a sufficiently simple interface. The user of the library does not need to worry about incorrect data due to noise since it is handled automatically by the library. TFHE offers easy functions to load/save keys and ciphertexts. Furthermore, TFHE provides a toy example demonstrating how to use the library, implementing a real scenario. Memory management of ciphertexts, however, is problematic as they do not exist as classes.

FHEW is also simple to use, albeit it is slower and provides fewer FHE gates. The major problem of FHEW is that is cannot operate on dependent variables, as discussed earlier, affecting usability. Another major technical issue is the lack of export/import keys and ciphertexts functions. The lack of such functions prevents FHEW from being used in a privacy-outsourcing scenario, since users cannot deploy their program along with the evaluation key while excluding the secret key. In the context of this work, we analyzed the implementation of the library at the source level, understanding the internal structure of the keys in order to develop custom functions for storing and loading the keys and ciphertexts to/from a file.

HElib has proven to be the most challenging to integrate. It appears to be more of a collection of functions to be used by cryptographers. It has a very complex process to generate keys, requiring several parameters which are subject to careful adjustment, otherwise security may collapse. The library tests provide representative security parameters for various security levels. A major open problem with HElib is estimating the need for bootstrapping, as this has to be handled manually by the user. Also, HElib does not provide a sufficiently documented API, and understanding the library functions reduces to source code analysis.

Another technical issue we encountered is that initializing HElib outside `main` and subsequently using it after `main` starts executing causes a segmentation fault. This scenario can happen when a C++ class is using an FHE library and a variable of this class is declared globally and used inside any function. The latter may trigger initialization of the FHE library before `main` starts. The practice of complex initialization in global scope is not recommended by the programming community; still, the programmer expects the program to work properly. A probable cause of this bug is that HElib or one of its dependent libraries initializes static data and the order of initialization causes inconsistent usage of such data. Both TFHE and FHEW, however, do not have this problem. Given that a natural C++ requirement is the ability to declare encrypted variables in global scope, this problem appears to be a usability limitation, yet it is an engineering problem that could be solved through proper debugging.

Our developed framework provides a standardized and simple interface for new FHE libraries working at the bit level. It can be used as a baseline for performance comparisons across newer versions of existing or new libraries.

VIII. Related Work

The potential of homomorphic manipulation of encrypted values without decrypting them was first observed by Rivest, Adleman and Dertouzos in their 1978 work on *privacy homomorphisms* [51]. Before 2009, partially homomorphic encryption (PHE) remained the only viable option of data manipulation in the encrypted domain, although it enabled only limited (i.e., functionally incomplete) manipulation of encrypted values. Notable PHE schemes include Paillier [52], Goldwasser–Micali [53], and Damgård-Jurik [54] that are additive homomorphic, El Gamal [55] and RSA [56] that are multiplicative homomorphic, as well as Boneh-Goh-Nissim [57] that supports the evaluation of quadratic (2-DNF) formulas with arbitrary monomials. In this context, one popular PHE application is secure electronic voting (e.g., [58]).

Since the discovery of FHE by Gentry [44], constant improvements have been reported in the literature. In addition to the BGV [40] and GSW [41] schemes mentioned earlier, notable examples also include the DGHV [59], FV [60], LTV [61], YASHE [62], and AGCD [63] fully homomorphic schemes. In an effort to assess the behavior of FHE in practice, previous work has focused on evaluating and comparing contemporary schemes. For example, Lepoint and Naehrig compare implementations of the FV and YASHE schemes [64], while Varia, Yakoubov and Yang develop an specialized testing framework for FHE and evaluate the performance of HElib under different configurations [24].

The prohibitive overhead of FHE has been a concern from its early days [22], which has motivated the design of special hardware accelerators for large FHE operands. The authors of [65] present an architecture for multiplying million–bit integers in milliseconds using the Schönhage-Strassen algorithm, while [66] presents an FPGA-based design of a 768K-bit multiplier using an FFT core. Since arbitrary computation on encrypted data remains a highly-sought application of FHE, earlier work has also focused on developing FHE processors for cloud computing. The authors of [67] present a methodology for private execution of a program on an untrusted host, and discuss the leakage problem due to *early termination*, which motivates us to prohibit branch decisions over encrypted data in this work.

Recent work has also focused on using FHE in machine learning applications over encrypted values [68], as well as in constructions for reusable garbled circuits [69] and indistinguishability obfuscation [70]. As both the academic community and the industry have embraced FHE, many open-source implementations have been reported in the literature. The authors of cuHE [71] accelerate FHE operations using

GPGPUs, while HEANN [72] provides support for fixed-point approximate FHE arithmetic. Likewise, $\Lambda o \lambda$ [73] provides FHE support in Haskell, Palisade [74] implements lattice-based homomorphic schemes, and SEAL [75] is Microsoft's implementation of the FV scheme.

IX. Conclusion

In this work, we developed E^3, which is a comprehensive framework that enables data processing directly in the encrypted domain using Fully Homomorphic Encryption. The primary objective of E^3 is to assist non-crypto-savvy programmers to incorporate privacy in their developed programs. One major highlight is the automatic compilation of real C++ programs using fully homomorphic variables, which guarantee end-to-end protection for sensitive data. The development of our framework is the first effort, to the best of our knowledge, to automate the use of fully homomorphic data in real programs. The framework can be used as a baseline for optimizations at different computation abstraction layers, including, but not limited to, hardware accelerators, compiler optimizations, algorithmic optimizations, etc.

Resources

The E^3 framework can be downloaded from https://github.com/momalab/e3.

References

[1] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown," *arXiv preprint arXiv:1801.01207*, 2018.

[2] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," *arXiv preprint arXiv:1801.01203*, 2018.

[3] C. Barron, H. Yu, and J. Zhan, "Cloud computing security case studies and research," in *World Congress on Engineering*, 2013, pp. 1287–1291.

[4] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage, "Hey, you, get off of my cloud: Exploring information leakage in third-party compute clouds," in *Computer and Communications Security (CCS)*. ACM, 2009, pp. 199–212.

[5] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Cross-VM side channels and their use to extract private keys," in *Computer and Communications Security (CCS)*. ACM, 2012, pp. 305–316.

[6] S. J. Stolfo, M. B. Salem, and A. D. Keromytis, "Fog computing: Mitigating insider data theft attacks in the cloud," in *IEEE Symposium on Security & Privacy Workshops (SPW)*. IEEE, 2012, pp. 125–128.

[7] F. Zhou, M. Goel, P. Desnoyers, and R. Sundaram, "Scheduler vulnerabilities and coordinated attacks in cloud computing," *Journal of Computer Security*, vol. 21, no. 4, pp. 533–559, 2013.

[8] K. Yang, M. Hicks, Q. Dong, T. Austin, and D. Sylvester, "A2: Analog malicious hardware," in *IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2016.

[9] D. Hély, M. Augagneur, Y. Clauzel, and J. Dubeuf, "Malicious key emission via hardware trojan against encryption system," in *International Conference on Computer Design (ICCD)*. IEEE, 2012, pp. 127–130.

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

[11] K. Xiao, D. Forte, Y. Jin, R. Karri, S. Bhunia, and M. Tehranipoor, “Hardware trojans: Lessons learned after one decade of research,” *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 22, no. 1, p. 6, 2016.

[12] R. Karri, J. Rajendran, K. Rosenfeld, and M. Tehranipoor, “Trustworthy hardware: Identifying and classifying hardware trojans,” *Computer*, vol. 43, no. 10, pp. 39–46, 2010.

[13] N. G. Tsoutsos and M. Maniatakos, “Fabrication attacks: Zero-overhead malicious modifications enabling modern microprocessor privilege escalation,” *IEEE Transactions on Emerging Topics in Computing*, vol. 2, no. 1, pp. 81–93, 2014.

[14] G. T. Becker, F. Regazzoni, C. Paar, and W. P. Burleson, “Stealthy dopant–level hardware trojans,” in *Cryptographic Hardware and Embedded Systems Workshop*, 2013, pp. 197–214.

[15] S. Adee, “The hunt for the kill switch,” *IEEE Spectrum*, vol. 45, no. 5, pp. 34–39, 2008.

[16] J. A. Gross, “Ending a decade of silence, israel confirms it blew up assad’s nuclear reactor,” https://www.timesofisrael.com/ending-a-decade-of-silence-israel-reveals-it-blew-up-assads-nuclear-reactor/, 2018.

[17] V. Costan and S. Devadas, “Intel SGX Explained,” *IACR Cryptology ePrint Archive*, vol. 2016, p. 086, 2016.

[18] G. Chen, S. Chen, Y. Xiao, Y. Zhang, Z. Lin, and T. H. Lai, “SGXspectre Attacks: Leaking Enclave Secrets via Speculative Execution,” *arXiv preprint arXiv:1802.09085*, 2018.

[19] C. Gentry, “Computing arbitrary functions of encrypted data,” *Communications of the ACM*, vol. 53, no. 3, pp. 97–105, 2010.

[20] D. Micciancio, “A first glimpse of cryptography’s holy grail,” *Communications of the ACM*, vol. 53, no. 3, pp. 96–96, 2010.

[21] J. E. Savage, *Models of Computation: Exploring the Power of Computing*. Addison-Wesley Longman Publishing Co., Inc., 1997.

[22] M. Naehrig, K. Lauter, and V. Vaikuntanathan, “Can Homomorphic Encryption Be Practical?” in *Cloud Computing Security Workshop (CCSW)*. ACM, 2011, pp. 113–124.

[23] C. Gentry, S. Halevi, and N. P. Smart, “Homomorphic Evaluation of the AES Circuit (Updated Implementation),” *IACR Cryptology ePrint Archive*, vol. 2012, p. 099, 2015.

[24] M. Varia, S. Yakoubov, and Y. Yang, “HEtest: A Homomorphic Encryption Testing Framework,” in *Financial Cryptography and Data Security*. Springer, 2015, pp. 213–230.

[25] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachene, “Faster fully homomorphic encryption: Bootstrapping in less than 0.1 seconds,” in *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 2016, pp. 3–33.

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

[27] S. Halevi and V. Shoup, “Bootstrapping for HElib,” in *Advances in Cryptology–EUROCRYPT 2015*. Springer, 2015, pp. 641–670.

[28] O. Regev, “On lattices, learning with errors, random linear codes, and cryptography,” in *Proceedings of the thirty-seventh annual ACM symposium on Theory of computing*. ACM, 2005, pp. 84–93.

[29] M. Kearns, Y. Mansour, D. Ron, R. Rubinfeld, R. E. Schapire, and L. Sellie, “On the learnability of discrete distributions,” in *Proceedings of the twenty-sixth annual ACM symposium on Theory of computing*. ACM, 1994, pp. 273–282.

[30] M. Ajtai, "Generating hard instances of lattice problems," in *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*. ACM, 1996, pp. 99–108.

[31] O. Regev, "On lattices, learning with errors, random linear codes, and cryptography," *Journal of the ACM (JACM)*, vol. 56, no. 6, p. 34, 2009.

[32] V. Lyubashevsky, C. Peikert, and O. Regev, "On ideal lattices and learning with errors over rings," *Journal of the ACM (JACM)*, vol. 60, no. 6, p. 43, 2013.

[33] M. R. Albrecht, R. Player, and S. Scott, "On the concrete hardness of learning with errors," *Journal of Mathematical Cryptology*, vol. 9, no. 3, pp. 169–203, 2015.

[34] Z. Brakerski, A. Langlois, C. Peikert, O. Regev, and D. Stehlé, "Classical hardness of learning with errors," in *Proceedings of the forty-fifth annual ACM symposium on Theory of computing*. ACM, 2013, pp. 575–584.

[35] A. K. Lenstra, H. W. Lenstra, and L. Lovász, "Factoring polynomials with rational coefficients," *Mathematische Annalen*, vol. 261, no. 4, pp. 515–534, 1982.

[36] C.-P. Schnorr and M. Euchner, "Lattice basis reduction: Improved practical algorithms and solving subset sum problems," *Mathematical programming*, vol. 66, no. 1-3, pp. 181–199, 1994.

[37] Y. Chen and P. Q. Nguyen, "BKZ 2.0: Better lattice security estimates," in *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 2011, pp. 1–20.

[38] R. Lindner and C. Peikert, "Better key sizes (and attacks) for LWE-based encryption," in *Cryptographers' Track at the RSA Conference*. Springer, 2011, pp. 319–339.

[39] M. R. Albrecht *et al.*, "Security Estimates for the Learning with Errors Problem," [Online]. Available: https://bitbucket.org/malb/lwe-estimator, 2018.

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

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

[42] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachene, "TFHE: Fast Fully Homomorphic Encryption Library over the Torus," [Online]. Available: https://github.com/tfhe/tfhe, 2017.

[43] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène, "Faster Packed Homomorphic Operations and Efficient Circuit Bootstrapping for TFHE," in *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 2017, pp. 377–408.

[44] C. Gentry, "Fully homomorphic encryption using ideal lattices," in *ACM Symposium on Theory of Computing*, 2009, pp. 169–178.

[45] L. Ducas and D. Micciancio, "FHEW: A Fully Homomorphic Encryption library," [Online]. Available: https://github.com/lducas/FHEW, 2017.

[46] S. Halevi and V. Shoup, "Design and Implementation of a Homomorphic Encryption Library," [Online]. Available: https://github.com/shaih/HElib, 2018.

[47] C. Gentry, S. Halevi, and N. P. Smart, "Fully homomorphic encryption with polylog overhead," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2012, pp. 465–482.

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

[49] S. Halevi and V. Shoup, "HElib Test Bootstrapping," [Online]. Available: https://github.com/shaih/HElib/blob/master/src/Test_bootstrapping.cpp, 2018.

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

[51] R. L. Rivest, L. Adleman, and M. L. Dertouzos, "On data banks and privacy homomorphisms," *Foundations of secure computation*, vol. 4, no. 11, pp. 169–180, 1978.

[52] P. Paillier, "Public-key cryptosystems based on composite degree residuosity classes," in *Advances in cryptology–EUROCRYPT'99*. Springer, 1999, pp. 223–238.

[53] S. Goldwasser and S. Micali, "Probabilistic encryption & how to play mental poker keeping secret all partial information," in *ACM Symposium on Theory of Computing*. ACM, 1982, pp. 365–377.

[54] I. Damgård, M. Jurik, and J. B. Nielsen, "A generalization of paillier's public-key system with applications to electronic voting," *International Journal of Information Security*, vol. 9, no. 6, pp. 371–385, 2010.

[55] T. ElGamal, "A public key cryptosystem and a signature scheme based on discrete logarithms," *IEEE Transactions on Information Theory*, vol. 31, no. 4, pp. 469–472, 1985.

[56] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Communications of the ACM*, vol. 21, no. 2, pp. 120–126, 1978.

[57] D. Boneh, E.-J. Goh, and K. Nissim, "Evaluating 2-dnf formulas on ciphertexts," in *Theory of Cryptography Conference*. Springer, 2005, pp. 325–341.

[58] O. Baudron, P.-A. Fouque, D. Pointcheval, J. Stern, and G. Poupard, "Practical multi-candidate election system," in *Proceedings of the twelveth annual ACM symposium on Principles of distributed computing*. ACM, 2001, pp. 274–283.

[59] M. van Dijk, C. Gentry, S. Halevi, and V. Vaikuntanathan, "Fully homomorphic encryption over the integers," in *Advances in Cryptology–EUROCRYPT*. Springer, 2010, pp. 24–43.

[60] J. Fan and F. Vercauteren, "Somewhat practical fully homomorphic encryption," Cryptology ePrint Archive, Report 2012/144, 2012.

[61] A. López-Alt, E. Tromer, and V. Vaikuntanathan, "On-the-fly multiparty computation on the cloud via multikey fully homomorphic encryption," in *ACM Symposium on Theory of Computing*. ACM, 2012, pp. 1219–1234.

[62] J. W. Bos, K. Lauter, J. Loftus, and M. Naehrig, "Improved security for a ring-based fully homomorphic encryption scheme," in *IMA International Conference on Cryptography and Coding*. Springer, 2013, pp. 45–64.

[63] J. H. Cheon and D. Stehlé, "Fully homomophic encryption over the integers revisited," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2015, pp. 513–536.

[64] T. Lepoint and M. Naehrig, "A comparison of the homomorphic encryption schemes fv and yashe," in *International Conference on Cryptology in Africa*. Springer, 2014, pp. 318–335.

[65] Y. Doröz, E. Öztürk, and B. Sunar, "A million-bit multiplier architecture for fully homomorphic encryption," *Microprocessors and Microsystems*, vol. 38, no. 8, pp. 766–775, 2014.

[66] W. Wang and X. Huang, "FPGA implementation of a large-number multiplier for fully homomorphic encryption," in *International Symposium on Circuits and Systems (ISCAS)*. IEEE, 2013, pp. 2589–2592.

[67] M. Brenner, J. Wiebelitz, G. Von Voigt, and M. Smith, "Secret program execution in the cloud applying homomorphic encryption," in *Digital Ecosystems and Technologies Conference (DEST)*, 2011, pp. 114–119.

[68] R. Bost, R. A. Popa, S. Tu, and S. Goldwasser, "Machine learning classification over encrypted data." in *NDSS*, vol. 4324, 2015, p. 4325.

[69] S. Goldwasser, Y. Kalai, R. A. Popa, V. Vaikuntanathan, and N. Zeldovich, "Reusable garbled circuits and succinct functional encryption," in *ACM symposium on Theory of computing*. ACM, 2013, pp. 555–564.

[70] S. Garg, C. Gentry, S. Halevi, M. Raykova, A. Sahai, and B. Waters, "Candidate indistinguishability obfuscation and functional encryption for all circuits," in *Foundations of Computer Science (FOCS)*. IEEE, 2013, pp. 40–49.

[71] W. Dai and B. Sunar, "cuhe: A homomorphic encryption accelerator library," in *International Conference on Cryptography and Information Security in the Balkans*. Springer, 2015, pp. 169–186.

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

[73] E. Crockett and C. Peikert, "$\lambda o \lambda$: Functional lattice cryptography," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2016, pp. 993–1005.

[74] K. Rohloff *et al.*, "The PALISADE Lattice Cryptography Library," [Online]. Available: https://git.njit.edu/palisade/PALISADE, 2018.

[75] H. Chen, K. Laine, and R. Player, "Simple encrypted arithmetic library-seal v2.1," in *International Conference on Financial Cryptography and Data Security*. Springer, 2017, pp. 3–18.