E3: A Framework for Compiling C++ Programs with Encrypted Operands

Eduardo Chielle, Oleg Mazonka, Homer Gamil, Sanja Kastratovic, 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 in a decrypted form, it can be leaked. Recent advances on cryptographic homomorphic schemes allow end-to-end processing of encrypted data without any need for decryption. Such schemes, however, still incur impractical overheads and they are difficult to use by non-crypto-savvy users, inhibiting their applicability.

In this work we propose a framework which allows optimal, in terms of performance, execution of standard C++ code with encrypted variables. The framework automatically generates protected types so the programmer can remain oblivious to the underlying encryption scheme. C++ protected classes redefine operators according to the encryption scheme effectively making the introduction of a new API unnecessary. The performance of encrypted computation is enhanced by our novel methodology, dubbed bridging, that blends faster and restricted modular computation with slower and comprehensive bit-level computation. Experimental results show that bridging computation can lead to a performance improvement of more than two orders of magnitude.

I. Introduction

With the ever increasing rates of data generation, digital information is becoming extremely valuable over time. Consequently, certain types of attacks have emerged, focusing to capitalize on this advancement. Adversaries employ various methods to exploit vulnerabilities appearing on different layers of the data management chain. Attacks have already been deployed against servers of well-established organizations,

E. Chielle, O. Mazonka, H. Gamil, S. Kastratovic and M. Maniatakos are with the Center for Cyber Security, New York University Abu Dhabi, UAE.

E-mail: {eduardo.chielle, om22, homer.g, sk6981, 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

some of them being Equifax, Marriot International, and Scottrade [21]. To overcome the aforementioned threats, the concept of encryption algorithms was proposed as a measure to protect leaked data. Even though accepted standards such as AES have been successful in protecting data-in-transit and data-at-rest, they fail to provide protection towards data-in-use. Data-in-use can be defined as volatile information that is required to undergo certain operations before reaching a state of rest (i.e. data-at-rest). Consequently, encryption algorithms have no effect on data-in-use, as current technology operates exclusively on plaintexts, and therefore requires information to be decrypted before performing any operation on them.

A solution to the problem of protecting data-in-use is Fully Homomorphic Encryption (FHE) as it allows unconstrained computation on ciphertexts. Since its release, numerous FHE schemes have been developed, namely BGV [8], BFV [17], CKKS [12], GSW [18]. While these advancements focus on improving the overall performance of fully homomorphic encryption, they fail to address critical issues that appear in real-life scenarios. Consequently, introducing the concept of fully homomorphic encryption to the software development industry has proven to be a difficult task so far. Huge performance overheads [22] and the complexity of encryption schemes and their implementation are major reasons for the limited deployment of FHE applications.

Contribution: In this work, we present a novel methodology for deploying private computation in C++ programs. Our methodology leverages the properties of both 1) bit-level arithmetic (universal, but slower) and 2) modular arithmetic (faster, but not universal) within the same algorithm, which greatly improves performance of general-purpose programs processing encrypted data. Furthermore, we develop framework E3 (Encrypt-Everything-Everywhere) providing novel protected types using standard C++ without introducing a new library API. While the presented framework is developed for C++, the methodology is applicable to any imperative programming language. E3 is open source [6] and works in Windows and Linux OSes. Finally, another major contribution of this work is the benchmarking of a set of state-of-the-art FHE libraries for different arithmetic operations, in an effort to highlight the sources of performance degradation of FHE deployment.

Unique features of E3:

1) Enables secure general-purpose computation using protected types equivalent to standard C++ integral types. The protected types, and consequently the code using them, do not depend on the underlying encryption scheme, which makes the data encryption independent of the program.
2) Allows using different FHE schemes in the same program as well as the same FHE scheme with different keys.
3) Supports *bridging*, which is a novel technique we introduce in this work for mixing different arithmetic abstractions, enabling a two orders of magnitude performance improvement (discussed in Section V).

In E3, the specifications of the encryption scheme and parameters are detached from the program. The program developer can use predefined encryption configurations or construct new, assuming corresponding expertise. E3 *does not introduce yet another library API*. Instead, it uses standard C++ syntax for operations on protected variables whose types are automatically generated. This provides portability between different FHE schemes; hence the same code can work using different encryption schemes and/or parameters.

II. Preliminaries

A. Homomorphic Encryption

The homomorphic function F over ciphertexts c_x and c_y is defined as follows: Let $f(m_x, m_y)$ denote a two-argument function on plaintexts m_x and m_y; $\mathrm{E}_k(m,r)$ denotes a probabilistic encryption function generated by a random sequence k (key) that maps a plaintext m to a set of ciphertexts c depending on a probabilistic parameter r, and $\mathrm{D}_k(c)$ denote a deterministic decryption function corresponding to E_k that maps ciphertext c to the corresponding plaintext m. Then, F is defined by the homomorphism of surjective D_k:

$$\mathrm{D}_k\big(\mathrm{F}(c_x, c_y)\big) = f\big(\mathrm{D}_k(c_x), \mathrm{D}_k(c_y)\big)$$

which can be converted into the explicit composition over ciphertexts:

$$\mathrm{F}(c_x, c_y) = \mathrm{E}_k\Big(f\big(\mathrm{D}_k(c_x), \mathrm{D}_k(c_y)\big), \mathrm{H}(c_x, c_y)\Big)$$

where H is an arbitrary function generating randomness r from the input ciphertexts.

B. FHE schemes and libraries

Since the appearance of Fully Homomorphic Encryption, various FHE schemes have been released, each focusing on improving different features and functionalities. The BGV (Brakerski-Gentry-Vaikuntanathan) scheme provides a method of generating leveled fully homomorphic encryption schemes with the capability to evaluate arbitrary polynomial-size circuits [8]. This method is based on the Learning with Errors (LWE) problem, and its variant, Ring Learning With Errors (RLWE). The next major scheme that followed after BGV was BFV (Brakerski/Fan-Vercauteren), which introduces two versions of relinearization that generate a smaller relinearization key with faster performance [17]. GSW (Gentry-Sahai-Waters) is yet another FHE scheme based on LWE [18]. This scheme proposes the approximate eigenvector method, making the computational cost of operations much cheaper, as addition and multiplication become simple matrix addition and multiplication, respectively. CKKS provides several new features to fully homomorphic encryption. This scheme utilizes a new rescaling procedure that allows the management of the plaintext's magnitude. Additionally, this technique proposes a batching method for RLWE-based development [12].

Libraries have been developed to support the aforementioned schemes, with the most prominent ones being TFHE [14], FHEW [16], HELib [20], and Microsoft SEAL [26]. While TFHE and FHEW are developed to support GSW, HElib supports BGV, and SEAL supports both BFV and CKKS. As anticipated, different coding knowledge is required to operate each library, thus making the use of different FHE schemes impractical. While a standardization effort is ongoing [9], [11], it still builds on the concept of adding a new API.

A common practice in FHE is to reduce programs to circuits. As we will show later, that is not always convenient, imposes many constrains on the program in question, and also does not always result in the most efficient computation.

C. Data-oblivious programming

Data-oblivious computation is a type of computation where the input data does not influence the behavior of the program. Under "behavior" we imply execution branching or memory access. Operations performed on encrypted data have to be in a data-oblivious form since they are never decrypted, therefore the program is not able to make decisions based on their plaintext values. Also, such operations on data are constant-time. Recent work has highlighted that data-oblivious programs are more resilient to side-channel attacks [28].

Consider a simple program calculating Fibonacci numbers in Listing 1. The iterations are interrupted once the index `i` reaches the input value. Assume that computation is now protected with input and output required to be encrypted. Since variable `input` cannot participate in decision to interrupt iterations, it must be processed in a data-oblivious way. To do that, we replace the interrupt condition with a multiplexer accumulator `r` and introduce a fixed number of iterations `max_iter` not depending on the input, as shown in Listing 2. In this case, the number of iterations should exceed the input index, otherwise the accumulator will not reach the input index and will not get updated. The program shown in Listing 2 works in constant time regardless of the input.

Changing a program to its data-oblivious form is not trivial. This unavoidable requirement reflects the fundamental property of computation where data is never decrypted, which is the case for all FHE computation. This constraint affects usability, as existing algorithms may need modifications to be converted to a data-oblivious version[1]. Moreover, it affects practicality too, as fixed iteration upper bounds, as shown in the example above, cause further performance degradation.

[1]Computer-assisted program transformation to data-oblivious variants is an active topic of research, focusing currently on domain-specific languages and compilers [10], [15].

```
#include <iostream>
#include <utility>

int main()
{
    int i(0), input(7);
    int a(0), b(1);

    while( i++ != input )
    {
        std::swap(a,b);
        a += b;
    }
    std::cout << a << '\n' ;
}
```

Listing 1. Simple Fibonacci program.

```
#include <iostream>
#include <utility>

int main()
{
    int i(0), input(7);
    int a(0), b(1), r(0);

    int max_iter = 10;
    while( max_iter-- )
    {
        r += (i++ == input) * a;
        std::swap(a,b);
        a += b;
    }
    std::cout << r << '\n' ;
}
```

Listing 2. Data-oblivious Fibonacci program.

III. Our approach

A. Philosophy

Any non-standard API is an extension to the language syntax bringing extra burden to the programmer. The basic idea of E3 is to keep the application code as close to standard C++ as possible. To achieve this, E3 retains the syntax of the original code by defining new types and operators for protected data. Using C++ rich machinery for operator redefinition, E3 allows us to hide the FHE library API from the programmer's consideration. In other words, the code remains the same for different underlying libraries,

and does not require a unified common API. With regards to the framework design, we outline the following requirements:

1) Following the imperative programming paradigm, the framework needs to maintain an accurate state after the execution of each statement. 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, and should be 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 during compilation.
4) Neither plaintext nor ciphertext data is required before the program compiles into binaries, as they can be input during runtime.
5) Ciphertext encryption is independent of the program. An input ciphertext is not required to be adjusted or re-encrypted for arbitrary program modifications.

Items 4 and 5 are consequences of the idea of general purpose computation. In the presented example (Listings 1 and 2), variables are declared and initialized inside the code. They could also be provided as an input during runtime. For example, `max_iter` may not be known during compilation and could be given by `std::cin` or read from a file. In the latter scenario, a combinational logic circuit cannot be generated during compile time, as the depth of the circuit is not known. Thus, requirement 5 expands applicability of E3 to programs written to process data encrypted in advance; for example, queries to an encrypted database stored at an untrusted server.

E3 allows the programmer to focus on the algorithm and not on the FHE intricacies. This is achieved by separating encryption specifications into a configuration file read by the framework during compilation. Therefore, the programmer identifies and annotates variables in the program that are to be protected, by replacing unencrypted `int` with an E3 secure type in the source code.

B. E3 building blocks

1) Modular and Bit-level arithmetic: Some FHE schemes define arithmetic addition, subtraction, and multiplication on numeric rings, but not other arithmetic such as division or comparison. While some applications may not require any other operation, a programmer expects to have all programming operations available in their programs. For example, the C++ statement "`if (a<b) c+=a`" and its corresponding data-oblivious form: "`c+=(a<b)*a`", cannot be evaluated with an FHE scheme where the comparison operation is not defined. Addition, subtraction, and multiplication on integers is an incomplete set of arithmetic operations; for example, the comparison operation cannot be reduced to these three operations.

However, when operating on bits, the same set of operations is universal, having addition and subtraction correspond to logical `XOR` and multiplication to logical `AND`.

E3 solves this problem by allowing the programmer to use types constructed out of sequences of encrypted bits. Indeed, as the least possible requirement to the evaluation power, any FHE scheme has to be able to evaluate the logic `NAND` (or `NOR`) gate on ciphertexts, since this elementary function is sufficient for universal computation. In this way, the variables in the expression "`c+=(a<b)*a`" are defined as *integral types* where all three operations (comparison, multiplication, and addition) are performed using bit-level arithmetic circuits. We call this computation *Bit-level arithmetic*, as opposite to the natively provided *Modular arithmetic*, where addition and multiplication are performed directly on ciphertexts.

The transition from Modular to Bit-level arithmetic is straightforward. Since the encrypted bit values are limited to 0 and 1, logic gates, such as `AND`, `XOR`, `NOT`, etc, can be expressed via the following expressions:

$$x \texttt{ AND } y = xy, \qquad x \texttt{ NAND } y = 1 - xy$$

$$x \texttt{ OR } y = x + y - xy, \qquad x \texttt{ NOR } y = 1 - (x \texttt{ OR } y)$$

$$x \texttt{ XOR } y = x + y - 2xy, \qquad x \texttt{ XNOR } y = 1 - (x \texttt{ XOR } y)$$

$$\texttt{NOT } x = 1 - x, \qquad \texttt{MUX}(x,y,z) = x(y-z) + z$$

`MUX` is the multiplexer operation (as in $x?y:z$). The set of values $\{0,1\}$ is closed under the above set of expressions. In this paper, we use the term *homomorphic gates* to describe logic gates operating on encrypted bits. Given these homomorphic gates, higher level operations (comparison, addition, multiplication) can be built using standard combinational arithmetic circuit design, which allows to perform any programming operation, i.e. giving the full spectrum of C++ arithmetic.

Listing 3 demonstrates the transition from plaintext computation to secure computation: Integer variables are declared with a new secure type `SecureInt`; the rest of the code remains the same[2]. Note that the type is parameterized with the size. In encrypted computation, increasing the number of bits can lead to a dramatic increase in performance overhead. As experimental results demonstrate, even 1 bit can make significant 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 3, all variables are 8 bits (lines 7, 8).

Listing 4 outlines the elements of the generated type `SecureInt` and the function providing encrypted values used in variable initialization. Note the postfix `_e` after each constant. The configuration file with

[2]E3 allows the user to name types. In this work, we use the type names as placeholders.

```
#include <iostream>
#include <utility>
#include "framework.h"

int main()
{
    SecureInt<8> i(0_e), input(7_e);
    SecureInt<8> a(0_e), b(1_e), r(0_e);

    int max_iter = 10;
    while( max_iter-- )
    {
        r += (i++ == input) * a;
        std::swap(a,b);
        a += b;
    }
    std::cout << r << '\n' ;
}
```

Listing 3. User type `SecureInt` behaves as native `int`.

```
template <int Size> class SecureInt{ ... };
constexpr std::string operator""_e
   (unsigned long long int x){ ... }
```

Listing 4. An outline example of `SecureInt` definition.

the specifications of the encryption defines the type name and postfix used for constants in the program. E3 automatically encrypts and replaces the plaintext constants with their encrypted representations, so the final binary does not have information about the initial values used in the program.

2) Abstraction layers: Normally, programming operations are expressed in one or more assembly instructions that are computed by the processor. In our framework, however, we generate classes and their operations. As mentioned earlier Modular and Bit-level arithmetic use FHE schemes in different ways. Modular operations are direct definitions of C++ operators in the program. Table I shows the abstraction layers: C++ operator is defined by E3 and its body uses the API provided by FHE scheme.

In Bit-level arithmetic, each variable is represented as a sequence of encrypted bits. E3 generates circuits with a Verilog compiler for standard programming operations. For example, the C++ multiplication operator $\star$ is generated as a circuit by a Verilog compiler using the Verilog expression $\star$. These circuits are translated into C++ functions and use FHE functions operating on encrypted bits instead of ordinary logic gates.[3]

[3]The translation to C++ is automated in E3 (see Section IV-B).

TABLE I

ABSTRACTION LAYERS OF MODULAR ARITHMETIC.

L	Element	Source	Example
1	C++ operator	User code	`SecureInt a,b; a*b;`
2	Class function implementation	Framework	`SecureInt operator* (SecureInt) {return mult();}`
3	Arithmetic operation	FHE API	`mult(){...}`

TABLE II

ABSTRACTION LAYERS OF COMBINATIONAL ARITHMETIC.

L	Element	Source	Example
1	C++ operator	User code	`SecureInt a,b; a*b;`
2	Class function implementation	Framework	`SecureInt operator* (SecureInt) {... mult(); ...}`
3	Verilog primitive	Verilog generated	`mult() {... NAND(); ...}`
4	Logic gate	FHE API	`NAND(){...}`

Table II lists these four layers explicitly. Level 1 is the user code and does not reveal any encryption concept. It solely represents the computational logic of the program. In other words, if protected variables are declared with the corresponding plain integral types, then the program remains valid without any dependencies introduced by the encryption scheme. Level 2 is the implementation of the operators provided by the framework. Its code is written once and can be reused for any homomorphic encryption. Level 3 consists of basic functions of computation, such as addition, division, comparison, and others. These functions are pre-generated by a circuit design compiler from Verilog expressions. The functions are expressed in terms of logic gates to be used in integrated circuit. Instead, we supply software implementations of these logic gates at Level 4.

3) Specialized circuits: Listing 3 shows the complete program operating on protected variables using Bit-level arithmetic for all variables involved in the computation. As mentioned previously, all computation boils down to the logic gate operations operating on encrypted bits. It is possible in E3 however, to

TABLE III

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.

improve performance of the program by utilizing optimization algorithms applied onto combinational circuits, by porting parts of the program to Verilog and compiling the code into gate-level netlist. This feature is simliar to assembly insertions in C/C++. Then, E3 converts the compiled netlist into C++ functions that can be called directly from the program. In our example, the iteration body (lines 13-15), instead of atomic programming operations, could be compiled into a combinational circuit processing five variables: `a`, `b`, `i`, `input`, and `r`.

4) Bridging: Declaring variables with a protected integral type and using solely Bit-level arithmetic as in Listing 3 has a potential drawback: When an FHE scheme provides fast Modular arithmetic operations, the usage of circuits operating on separate bits is slow. E3 brings the novel idea of *Bridging* - mixing both Modular and Bit-level arithmetic in one program with the ability to convert variables from integral type to modular. Some variables can be declared as of protected type supporting only Modular arithmetic while others with the other type supporting Bit-level arithmetic. In bridging mode, a type of Bit-level arithmetic declares a conversion function into a type of Modular arithmetic. Obviously, the encryption of the two different C++ types must share the same keys.

For performance reasons, it is desirable to perform the entire computation in Modular arithmetic, since it is much faster than Bit-level. If, however, a program requires an unsupported operation in modular arithmetic (e.g. comparison), then –without Bridging– the whole program must perform all computations using Bit-level arithmetic, severely degrading performance. Bridging allows the isolation

```
1  #include <iostream>
2  #include <utility>
3  #include "framework.h"
4
5  int main()
6  {
7      SecureInt<8> i(0_e), input(7_e);
8      SecureMod a(0_m), b(1_m), r(0_m);
9
10     int max_iter = 10;
11     while( max_iter-- )
12     {
13         r += (i++ == input) * a;
14         std::swap(a,b);
15         a += b;
16     }
17     std::cout << r << '\n' ;
18 }
```

Listing 5. Bridging (i.e., mixing `SecureMod` and `SecureInt` types) enables performance improvement.

of the computation requiring Bit-level arithmetic. For example, the expression "`c+=(a<b)*a`" can use Bit-level arithmetic for the comparison only. The variables required by the comparison (`a` and `b`) must be of integral type. Nevertheless, the operands of multiplication can be implicitly cast to the modular type, allowing multiplication and addition to be executed in Modular arithmetic, resulting in a variable `c` of modular type.

Listing 5 demonstrates the code of the same Fibonacci program with *Bridged* arithmetic: only `input` and counter `i` are declared as integral type `SecureInt`, while the others are replaced with the `SecureMod` type. Line 13 does implicit conversion from `SecureInt` to `SecureMod`; therefore Bit-level multiplication (which is expensive) is not executed. Instead the native multiplication of ciphertexts is only required. Comparison between `i` and `input` is the slowest operation in the program. Its result is one encrypted bit which can naturally be casted to `SecureMod`, since 0 and 1 is a subset of the plaintext range. Casting of an integral type (Bit-level) into modular (FHE native) is possible, but not the other way around. Such operation is a summation of the encrypted bits of the integral type. Listing 6 shows an example of how casting from `SecureInt` to `SecureMod` is achieved in E3.

5) Batching: Batching is the ability to pack several plaintexts into a single ciphertext. This feature is supported by some FHE schemes. In practice, it enables parallel processing of plaintexts, since they are all part of the same ciphertext, in a Single Instruction Multiple Data (SIMD) style. This can provide significant performance improvements for algorithms with parallel computation properties.

Each ciphertext variable is effectively a vector of values and any unary or binary operation has the effect of array operations on all elements of the vectors. Integral types in case of batching naturally have arrays of bit values inside each bit of the variable, and gate operations on each separate bit have the effect of the gate operation on all bit values. E3 supports two ways to specify variables with packing: 1) Direct encryption with packing to use as input to the program, and 2) Specific syntax for constants to use inside the program.

IV. E3 Framework Design

In this section we describe the technical details of the framework, focusing on the implementation challenges faced while trying to adhere to the requirements outlined in Section III-A.

A. Mechanics of protected types

1) Naming convention: In this work we use protected class names: `SecureInt`, `SecureMod`, `SecureBool`. As mentioned earlier, these names serve only as placeholders and not the names used in E3. Users can define their own names as well as mix different protected classes in the same program.

2) C++ operators: E3 enables the use of all standard C++ operators with encrypted variables. Table III summarizes C++ operators, classified into the following 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 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 high-level without calling circuit functions, and appear in the 'Implemented in C++' category. All the other overloaded operators (e.g., `a+b`) require calls to functions implementing Boolean circuits using homomorphic gates. These operators can be further classified into two categories: 'Direct', which actually call 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 circuit function defines a *referentially transparent* function with its output being distinct from any of its inputs.

3) The `SecureInt` *class:* Our `SecureInt` class is built on top of an internal uniform API of E3 for each encryption scheme. This API consists of the following components: 1) A class (`Bit`) representing one encrypted bit. The class defines constructors, assignment operators (copy and move) along with export and import to and from a string in encrypted form. 2) Secret and evaluation keys generation with loading and saving capabilities. 3) Functions to encrypt and decrypt one bit. 4) List of logic gates - *referentially transparent* functions taking one or more `Bits` as arguments and returning one `Bit`. 5) A `Bit` instance for encrypted bit *zero* and a `Bit` instance for encrypted bit *one*.

The motivation 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 C++ file generated by E3 with class definitions and functions.

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 an unsigned `SecureInt<8>` to `SecureInt<16>`, the cast operator pads the 8 most significant bits with `Bit` instances of encrypted zero provided by the API. For a signed `SecureInt`, the cast operator sign-extends the number using its most significant bit. 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 III), the corresponding 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 operation, without any burden to the programmer. Consider the following expression: (a<b)*c. If only the SecureInt class was available, this expression would invoke a 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 is defined as an operator between SecureBool and SecureInt invoking only a Boolean multiplexer 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 the SecureBool class explicitly in the program. In summary, our SecureInt class has the following properties:

- Exposes an internal type Bit and provides access to individual Bits by overloading the [] operator.
- Exports/imports its encrypted representation into a string.
- Offers functions for encryption and decryption.[4]
- Defines all "Overloaded" C++ operators of Table III.
- 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 OR circuit (called OR-reduction), following the C++ convention that any non-zero value is Boolean true.

4) The SecureMod *class:* SecureMod is a class representing ciphertext with native operations. If the encryption scheme supports modular addition, subtraction, and multiplication, then this class defines corresponding operators. Others remain undefined and preclude compilation of the program had the programmer used them. A significant difference compared to C++ standard types is modular arithmetic on bases that can be different from a power of two. If this is the case, the results of operations with overflow can be confusing to a programmer without knowledge of modular arithmetic.

For an encryption scheme supporting modular arithmetic, the internal API, described above in Section IV-A3, extends to encryption and decryption of integers and the underlying arithmetic operation functions. SecureMod type has limited number of operations but these operations are faster comparing to operations on SecureInt class variables. An important feature of E3 is the ability to mix these

[4]Decryption only works when a secret key is defined, which is true during pre-processing the user's program, post-processing the results, or during debugging.

```
template <int Size>
SecureMod to_SecureMod(SecureInt<Size> v)
{
    auto i = Size;
    SecureMod n = v[--i];
    while ( i-- ) n += n + v[i];
    return n;
}
```

Listing 6. Casting from `SecureInt` to `SecureMod`. Since `SecureInt` is a set of ciphertexts representing encrypted bits, it is possible to access each bit individually.

classes taking advantage of both worlds: Speed and universality. When both classes are present in the program, variables of `SecureInt` can be casted to `SecureMod`. Listing 6 shows the algorithm of conversion from `SecureInt` into the `SecureMod` type.

B. Compilation

1) User's Perspective: Our framework works with a C++ program possibly spanning across separate files. The encryption parameters are specified in the configuration file of the framework. Our framework generates classes for protected variables in the form of C++ functions which are automatically embedded in the compiled binary. In this scenario, the user (programmer)

1) generates the protected version of the program along with the secret key;
2) provides the program, either as source code[5] or binary, and the evaluation key, to run at an untrusted party;
3) obtains the output result from the program; and finally
4) decrypts the result using the secret key.

Only the user can decrypt the output, which differs from applications that require multiple parties to be able to compute different functions on the encrypted data, where Functional Encryption schemes [25] can be used. Our framework can be naturally applied anywhere Fully Homomorphic Encryption can be applied, such as using the cloud for faster processing or permanent data storage. From the user (programmer) perspective, the program has the same functional structure in both encrypted and unencrypted forms.

2) Structure: As discussed in Section IV-A, the programmer is oblivious to the mechanics of instantiating FHE schemes. The process diagram for the behind the scenes compilation and execution of a C++ program using our framework appears in Fig. 1. The 'User's code' (for example, the source code in Listing 5) needs

[5]In that case, source code must be pre-processed so user-defined protected constants are replaced with encrypted ones.

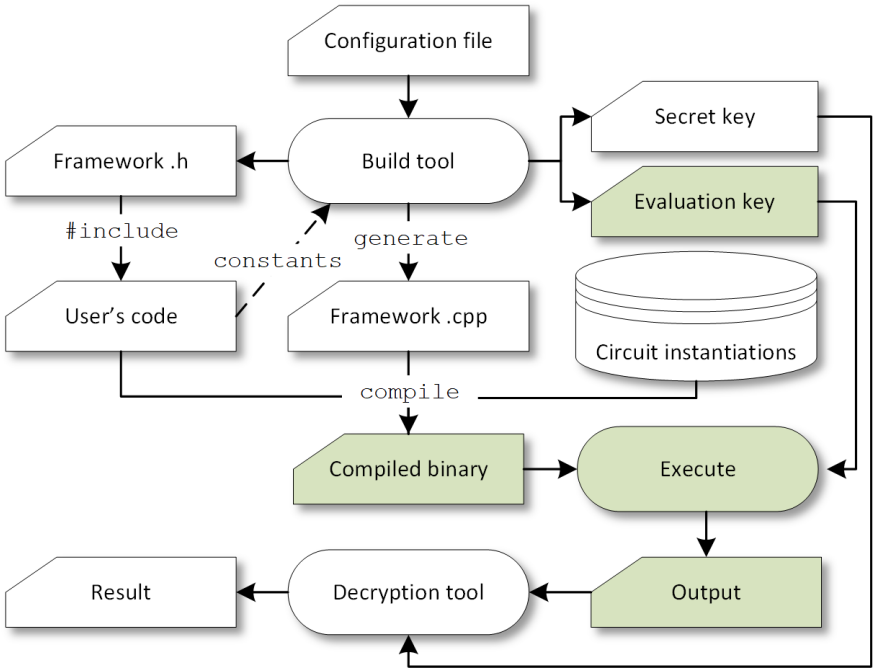

Fig. 1. Process diagram presenting the components required to compile and execute a C++ program operating on FHE encrypted data. The shaded parts can be outsourced to a third party.

to include the header of our framework, in order to have access to the new secure data types. Furthermore, encrypted constants have to be appropriately annotated (e.g., `7_e`). Our developed 'Build tool' generates the appropriate 'Secret key' and 'Evaluation key', for a given 'Configuration file' delineating the FHE scheme to be instantiated, and generates the implementation file of the protected classes. The latter, besides the instantiation of the overloaded operators, also contains encryptions of program constants. In our framework, we use the suffix (e.g. `_e`) to: allow the building tool to extract the list of constants used in the program; encrypt these constants into string literals; and update the string literal operator defined in the secure type classes implementation file. Without this automated process, the programmer would have to manually instantiate the FHE cryptosystem, generate keys, and encrypt each constant, replacing the user-defined literal with the corresponding encrypted value.

3) Catching errors during compilation: One of the requirements outlined above is that if the program compiles, it works as expected. A critical compiler error is the implicit conversion of `SecureInt` or `SecureBool` to `bool`. 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`. Without decryption this is impossible, therefore the compiler must stop the compilation highlighting the error. In this case, the `SecureBool` class defines the cast-to-`bool` operator, which uses templates – along with the `static_assert` C++ mechanisms – to produce a meaningful error message to the user.

4) Bit-level arithmetic circuits: The last part of the building process is to instantiate the group of 'Direct' operators which are directly mapped to Bit-level arithmetic circuits. 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.

The 'Circuit instantiations' database (input in Fig. 1) is a collection of *explicit template specializations* of all possible combinations of 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 using parametrizable RTL (Register Transfer Level) designs. The RTL designs are compiled for different bit sizes using design compiler and a customized standard cell library, which is optimized for each FHE library according to the speed of each gate – the evaluation of the gates has different performance; hence, the optimal function for evaluating each circuit is different as well. The design compiler generates gate-level netlist which E3 converts into C++ template functions.

The libraries exposing homomorphic gates can directly be linked to E3 circuits. When a particular gate is not available in a specific library, we construct it on top of native gates. For modular arithmetic based schemes, we build gates on top of native arithmetic operations. In such schemes, additions and subtractions are much faster than multiplication. Therefore, gates should be implemented minimizing the number of multiplications. In addition, when the plaintext modulus is 2, the gate equations (as in Section III-B1) can be simplified: All subtractions are replaced with additions and the XOR gate does not require multiplication.

V. Experimental Results

A. Overview and setup

The purpose of the experiments described in this section is to demonstrate the benefits of our methodology while proving its efficacy in practice. We compare performance of three different modes: Bit-level arithmetic, bridge mode, and specialized circuits using several C++ programs and FHE libraries. We use four libraries, namely FHEW, HElib, SEAL, and TFHE, and select their appropriate combination and encryption parameters for our benchmarks. HELib and SEAL expose modular operations, while FHEW and TFHE expose gates, therefore they can only be used in Bit-level arithmetic.

We first perform experiments with the modular operations of HElib and SEAL to figure out the best parameters for usage in Bit-level arithmetic and bridging. Then, we analyze the performance for Bit-level arithmetic using all four libraries. After, we evaluate the performance of Bit-level arithmetic, bridge mode, and specialized circuits using several C++ programs and a selected set of FHE libraries. Lastly, we show the performance improvements provided by batching.

All results in this section were collected using a computer equipped with an Intel i7-4790 CPU @ 3.60GHz $\times$ 8 and 16 GB of RAM, 64 GB of swap area, running Ubuntu 16.04.3, with GCC 7.4.0, GMP 6.1.2, NTL 11.0.0, and FFTW 3.3.4. We use FHEW #f53cd4b, HElib #65ef24c, SEAL 3.3.2, and TFHE #3319e2c. All library parameters are selected to provide at least 100 bits of security.

B. Modular arithmetic

We evaluate two libraries exposing modular arithmetic, HElib and SEAL, in terms of speed and noise budget (how many operations one can perform until a ciphertext is corrupted). Since the noise is mainly affected by multiplication, we test it by checking how many square operations corrupt a ciphertext. Regarding speed, we check the average speed of addition and multiplication. All experiments are repeated 20 times with different random seeds. Execution times show negligible standard variation.

HElib: For HElib, we are interested in parameters that enable bootstrapping. For this reason, we use and test all the parameters HElib's authors define in [1]. Of those, we select the smallest and the largest plaintext moduli available. The smallest plaintext modulus ($p = 2$) is the most efficient for bit-level arithmetic, while the largest ($p = 127$) is suitable for modular arithmetic.

Fig. 2 compares the execution time of addition and multiplication without bootstrapping, and the security level, of different parameters in HElib for plaintext modulus 2 and 127. We can see that the execution time of both addition and multiplication, as well as the security level grows with the increase of the cyclotomic field parameter m [19], while the plaintext modulus does not affect performance.

Since we require at least 100 bits of security, we select the fastest parameters for plaintext moduli 2 and 127 that meet the criteria. Then, we evaluate them in terms of the average execution time of the square operation, bootstrapping when needed. We can see in Fig. 3 that when bootstrapping becomes necessary, the performance degradation is around 3 orders of magnitude. In addition, we notice that smaller plaintext moduli create less noise per operation, which effectively translates in better performance when compared to larger plaintext moduli.

SEAL: Although BFV supports bootstrapping, SEAL does not implement it. Therefore, SEAL is treated as Somewhat Homomorphic Encryption. First, we check how many square operations are necessary to corrupt a ciphertext. The number of operations that can be performed on a ciphertext depends on the polynomial modulus degree and the plaintext modulus. Fig. 4 shows the number of square operations until a ciphertext is corrupted. Note how larger polynomial modulus degrees and smaller plaintext moduli increase the noise budget, and consequently, the number of operations on a ciphertext. In Fig. 5, we compare the speed of addition and multiplication in SEAL. The polynomial modulus degree is the only

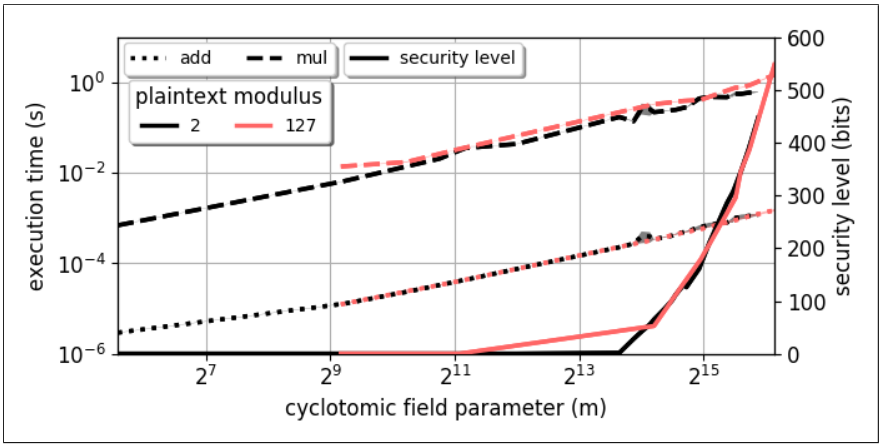

Fig. 2. Security level, and speed of addition and multiplication in HElib for plaintext modulus 2 and 127.

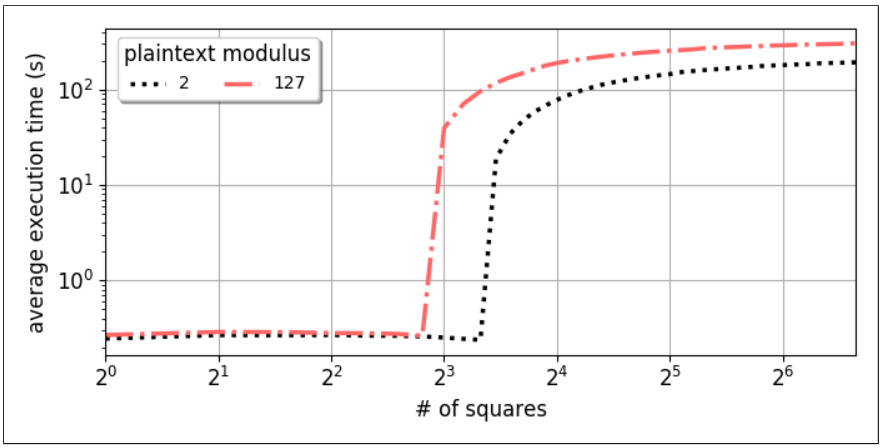

Fig. 3. Average speed of multiplication for deeper circuits in HElib.

factor that affects speed. The larger it is, the slower the operation. In SEAL, the polynomial modulus degree *poly* must be a power of two such that $2^{12} \leq poly \leq 2^{15}$ in order for relinearization to work.

Insights: From this set of experiments we conclude that HElib is more suitable for Bit-level arithmetic, since it supports bootstrapping, which is handful in iterative programs with deep circuits, while SEAL is better used in modular arithmetic due to its faster speed for larger plaintext moduli.

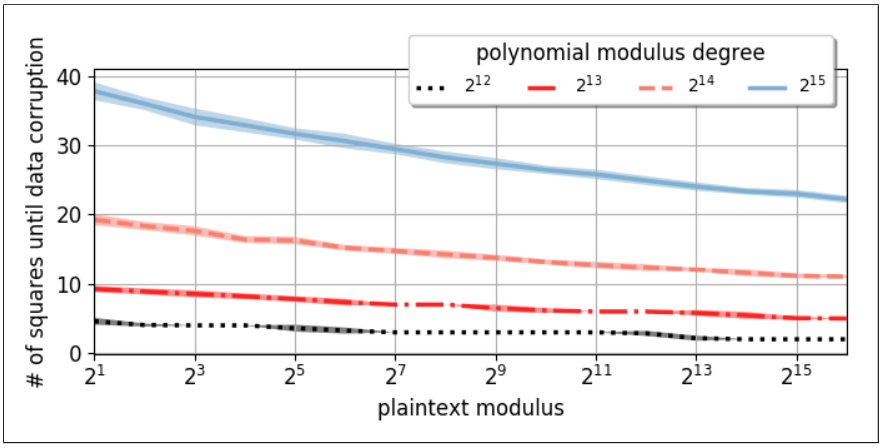

Fig. 4. Number of square operations in SEAL until the data is corrupted. Shaded area indicates standard deviation.

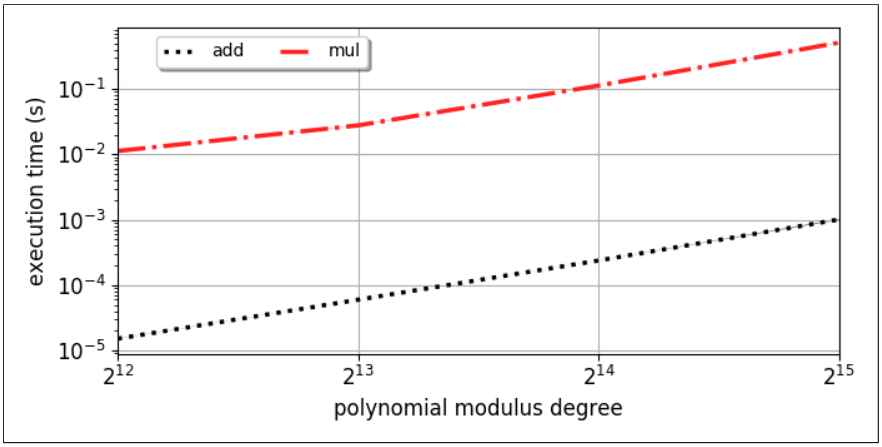

Fig. 5. Speed of addition and multiplication in SEAL. The standard deviation is negligible.

C. *Bit-level arithmetic*

For Bit-level arithmetic, we compare four libraries: HElib ($p = 2$), SEAL ($poly = 2^{15}; p = 65537$),[6] FHEW, and TFHE. The last two are FHE libraries based on the GSW scheme which exposes homomorphic gates.

Gates: First we evaluate the performance of gate operations following the implementation defined in Section IV-B. Our test consist of four independent ciphertexts containing the encryption of zero or one, which we use as parameters for the gate operations in a cyclic way. Fig. 6 shows the average execution time for gates in HElib. Gates `NOT`, `XOR`, and `XNOR` are implemented using only additions, therefore, they

[6]Modulus $p = 65537$ is selected specifically to support batching.

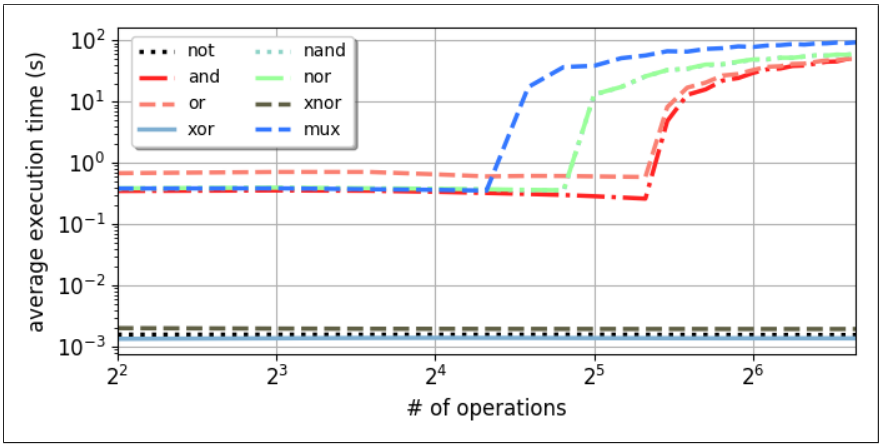

Fig. 6. Average gate speed for deeper circuits in HElib.

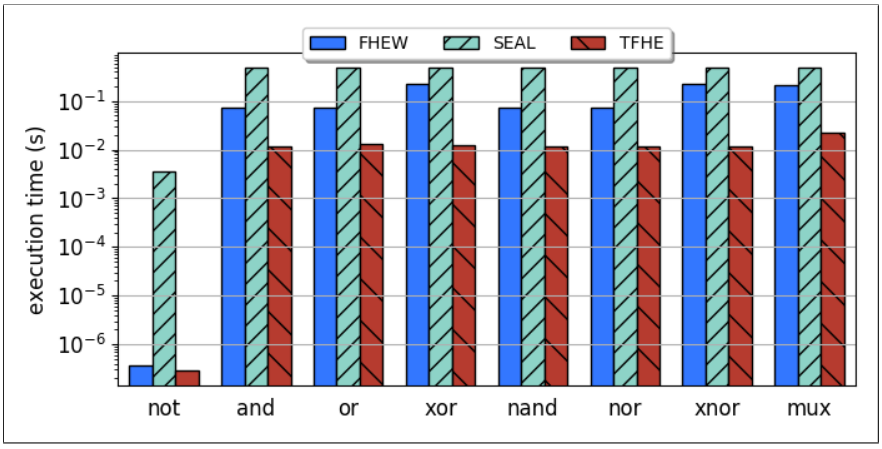

Fig. 7. Gate speed for FHEW, SEAL, and TFHE.

are efficient and produce little noise. Meanwhile, all other gates require at least one multiplication, which is slower and noisier, and sooner leads to bootstrapping. Here bootstrapping starts later when compared to squaring in Fig. 2. This is due to the use of four independent ciphertext instead of only one. The gate `AND` can be used as a guide, since it is implemented using a single multiplication. In Fig. 7, we compare the gate speed of FHEW, SEAL, and TFHE. TFHE outperforms FHEW and SEAL in all gate operations, however, gates `XOR` and `XNOR` would be faster in SEAL if $p = 2$, in a similar way to HElib, since they would not require multiplication.

Circuits: We evaluate the execution time of a set of Bit-level arithmetic circuits using HElib, SEAL, and TFHE. We do not take FHEW into account since it implements the same encryption scheme of TFHE and it is slower. We create optimized circuits for each library. In software, gates are executed sequentially, thus, our default optimization is for area. Still, we also create circuits optimized for performance for HElib

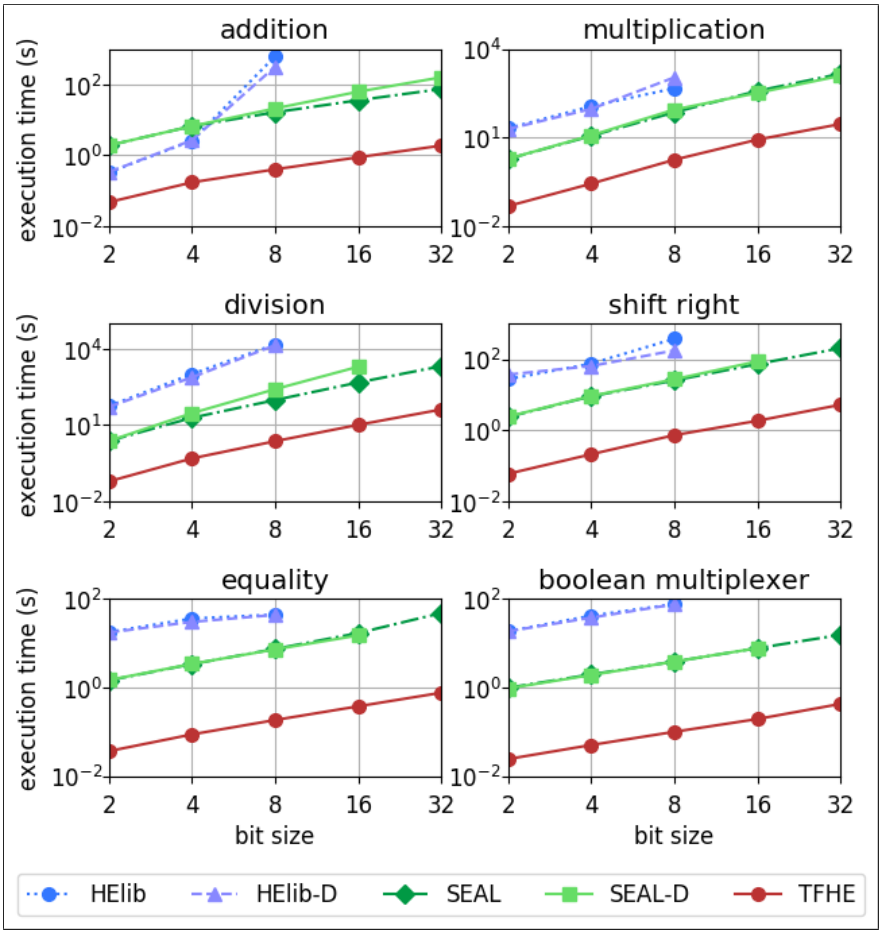

Fig. 8. Execution time of a set of operators in HElib, SEAL, and TFHE. (postfix `-D`: optimized for depth)

and SEAL, since such circuits have smaller depths, which affects noise propagation and, consequently, bootstrapping.

Fig. 8 presents the execution time of addition, multiplication, division, shift right, equality, and boolean multiplexer with bit sizes ranging from 2 to 32 for HElib, SEAL, and TFHE. The postfix `-D` denotes circuit optimization for depth/performance.

Insights: As expected, TFHE outperforms the others libraries as it has the fastest gates. HElib becomes prohibitively slow for large circuit sizes due to bootstrapping, while SEAL is between both. In addition, we can see that optimizing for depth slightly increases performance for smaller circuits while decreases for larger circuits.

D. Benchmarks

In order to evaluate different execution modes (Bit-level arithmetic, bridge, specialized circuits), we use eight data-oblivious benchmarks from the TERMinator Suite [23] that execute for arbitrary maximum

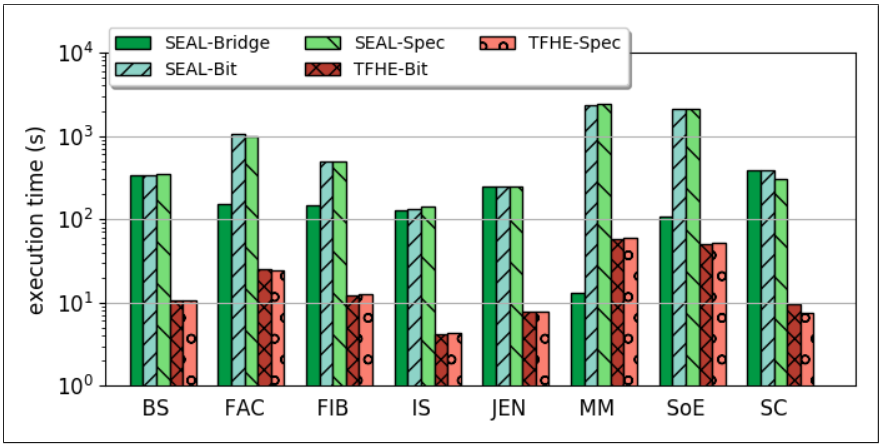

Fig. 9. Benchmark execution time for SEAL and TFHE. 'Bit' is bit-level arithmetic, and 'Spec' is specialized circuit.

number of iterations and input vector sizes. These benchmarks are developed to be data-oblivious and manipulate sets of encrypted variables adjusted to 8 bit size. The algorithms are classified into three categories: 1) encoder benchmarks, such 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).

Fig. 9 shows the execution time of the benchmarks for five different combinations. We select TFHE as it is the fastest Bit-level arithmetic, and SEAL for bridging and batching functionality.

Insights: The results show that Bit-level arithmetic has comparable performance to specialized circuits. This result reveals that the algorithms under consideration do not benefit from writing parts of the code in Verilog, and E3 in this case produces almost optimal executable. Bridging, on the other hand, demonstrates substantial performance improvement (up to 2+ orders of magnitude) for algorithms that allow mixing `SecureInt` and `SecureMod`, while in the worst case, bridge is equivalent to Bit-level arithmetic. Encoder algorithms (JEN, SC) do not benefit from Bridging due to heavy bitwise operations that preclude using `SecureMod` type.

Batching: When the underlying FHE library (e.g. SEAL) supports batching, E3 can utilize this functionality. Benchmark results above show that on average TFHE outperforms SEAL. However, it is worthwhile to check how batching affects performance, since TFHE does not support it but SEAL does. The number of batching slots in SEAL is equal to the polynomial modulus degree (i.e, 2^{15}). The maximum speed up given by batching occurs when all plaintexts packed in a ciphertext produce independent outputs, i.e.

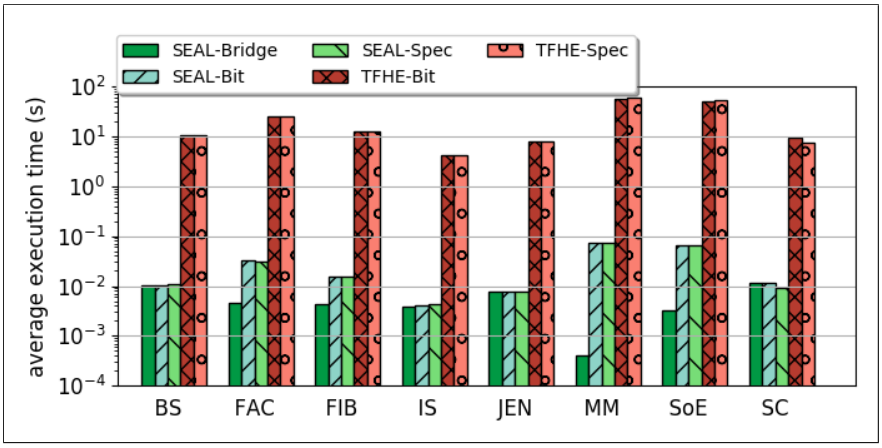

Fig. 10. Benchmark execution time for SEAL (with batching) and TFHE (without batching). 'Bit' is bit-level arithmetic, and 'Spec' is specialized circuit.

each output only depends on a single value from each ciphertext. For example, instead of calculating the Fibonacci value of one plaintext at a time, by packing a number of plaintexts into a single ciphertext, we can calculate the Fibonacci for all inputs at the same time. Fig. 10 shows the benchmarks comparing SEAL and TFHE. We also study the performance of Bridging compared to Bit-level arithmetic for different bit-sizes, as shown in Fig. 11. Apart from Fibonacci, an additive-intensive program, we can see an increase of the speed-up provided by bridging for larger bit-sizes. This is due to the non-linear growth of complex Bit-level arithmetic circuits (e.g. multiplication).

Insights: SEAL without batching is more than an order of magnitude slower than TFHE on average. When batching is introduced, SEAL outperforms TFHE by 3 orders of magnitude. In addition, Bridging provides further performance improvement for complex circuits with larger bit-sizes. It should be noted that this performance improvement is tightly connected to the parallelization capabilities of the workload. Nevertheless, results show that some benchmarks, when *both* batching and bridging are combined, can be accelerated by more than 7 orders of magnitude, bringing FHE applications much closer to practicality.

VI. Real-world application: URL blacklisting

Private Information Retrieval (PIR) is a major cryptographic building block for privacy-preserving applications. Its main operation is looking up an element from a database, without revealing what the element is. In this scenario, the server hosting the database can be untrusted. PIR can be used for a multitude of applications, including advertisements delivery, media streaming, and URL blacklisting.

Without loss of generality, we focus on URL blacklisting, but the same principles apply to any PIR-based application. URL blacklisting is the process where a URL is being checked against a known database of

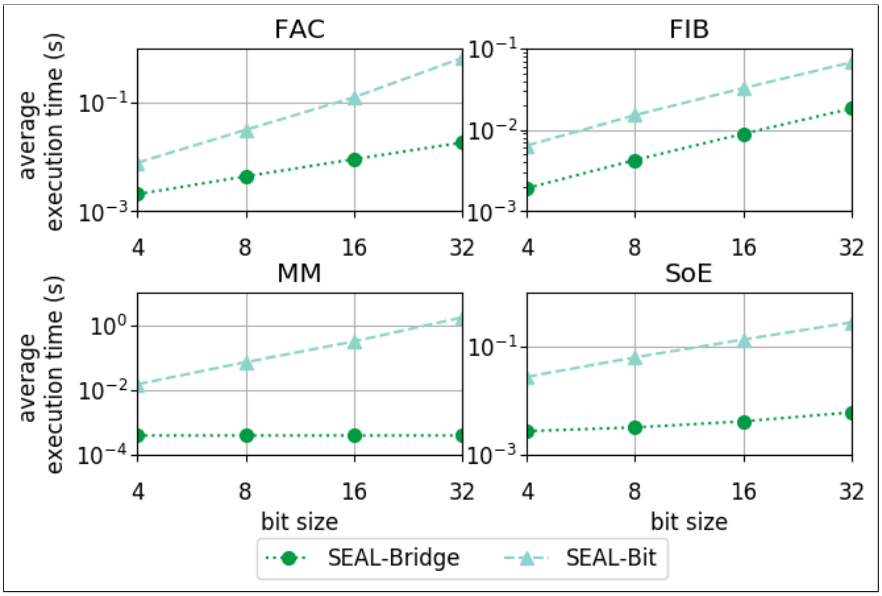

Fig. 11. Execution time of a set of benchmarks in SEAL (with batching). 'Bit' is bit-level arithmetic.

malicious URLs; if there is a match, the access to the website is blocked. URL blacklisting can be used in conjunction with URL rewriting in emails. In other words, when an email comes to the company server, its URLs are being replaced by a redirection to a secure site, which checks whether the link is malicious or not. If it is benign, the user is automatically forwarded to the webpage. While this (supposedly) protects against phishing attacks, it introduces a huge privacy problem, since the company hosting the "secure site" can clearly see which links users are clicking on, and sell the data to 3rd parties.

PIR is computationally expensive, and this performance degradation is fundamental: All database elements have to be looked up in order to provide an answer while maintaining privacy. At the same time, however, PIR is embarassingly parallel, and can be accelerated in direct relation to provided resources. Listing 7 shows the heart of our Private URL Blacklisting application using the E3 framework. The code is very simple, highlighting the strengths of E3.

With regards to performance, since PIR is embarrassingly parallel, we use SEAL to demonstrate the benefits of our bridging methodology. Fig. 12 shows that bridging is around 4 times faster than Bit-level arithmetic and specialized circuits, which corroborates with experiments in Section V. The performance improvement is independent of the bit size, since there is no circuit multiplication. It should be noted that bridging can be combined with any other method of increasing performance, such as oblivious expansion procedures and probabilistic batch codes [7].

```
template <int SZ> SecureMod pir(
    SecureInt<SZ> key, const vector<SecureInt<SZ>> & tb_idx, const vector<SecureMod> &
    tb_val)
{
    SecureMod r = 0_m;
    for (int i=0; i<tb_idx.size(); i++)
        r += (key == tb_idx[i]) * tb_val[i];
    return r;
}
```

Listing 7. Bridging Private Information Retrieval on E3.

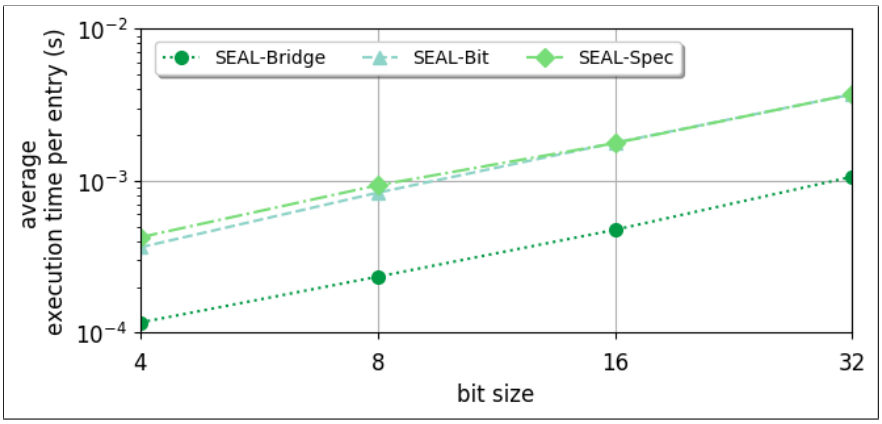

Fig. 12. Private Information Retrieval relative execution time for SEAL (with batching). 'Bit' is bit-level arithmetic, and 'Spec' is specialized circuit.

VII. Related Work

The quality of a homomorphic encryption framework depends on its ability to enable a rich set of operations, such as addition, multiplication, division, and comparison, as well as bit-wise operations. Additionally, some frameworks require declaration of encryption parameters prior to their operation. Parameters may represent degrees of polynomial, ciphertext and plaintext moduli, and can be specified either manually or automatically, depending on the framework. These parameters are capable of affecting the utilized encryption schemes, as well as the performance and security levels of the frameworks. A comparison between the proposed solution and state-of-the-art frameworks is drawn in Table IV.

Cingulata is an open-source compiler toolchain. It transforms C++ code into an equivalent boolean circuit representation, which after being optimized, is executed dynamically using different FHE schemes [2]. Cingulata operates on docker, therefore providing the flexibility to execute on multiple operating

TABLE IV

COMPARISON OF E3 WITH STATE-OF-THE-ART SOLUTIONS

	Type	This work	Cingulata [2]	FHEW [16]	HEAAN [13]	HELib [20]	Lattigo [3]	Marble [27]	Palisade [4]	SEAL [26]	SHEEP [5]	TFHE [14]
Scheme	BFV	●	●	○	○	○	●	○	●	●	●	○
	BGV	●	○	○	○	●	○	●	●	○	●	○
	CKKS	●	○	○	●	○	●	○	●	●	●	○
	GSW	●	●	●	○	○	○	○	●	○	●	●
Operations	Add/Sub	●	●	●	●	●	●	●	●	●	●	●
	Mult	●	●	●	●	●	●	●	●	●	●	●
	Div	●	○	○	○	○	○	○	○	○	○	○
	Compare	●	●	●	○	○	●	●	●	○	●	●
	Bitwise	●	●	●	○	○	○	○	○	○	●	●
Feat.	Bridging	●	○	○	○	○	○	○	○	○	○	○
	Batching	●	○	○	●	●	●	●	●	●	●	○

● Supported ○ Not Supported

systems. It generates the required encryption parameters automatically and supports the wide range of features mentioned above with the notable exception of batching.

HEANN is a library supporting the CKKS scheme that excels in providing a variety of important features to its users [13], including flexibility of the encryption parameters, negative computations, bootstrapping, and ciphertext packaging. The main source code requires the user to set certain parameters such as the ciphertext modulus, the scaling factor, and the number of slots, as well as to call predefined functions for setting the context and generating the required keys. After declaring the plaintexts the users are required to generate their code using pointers. The pointers are later on passed to functions to perform homomorphic operations.

Lattigo is a lattice based cryptographic library written in Go [3]. This framework supports two cryptoschemes, BFV, and CKKS. Some of the features that are planned to be implemented in future versions of Lattigo include bootstraping for CKKS, and network layer implementation of protocols supporting Secure Multiparty Computation.

Marble, is a high-level middleware that provides an HE instruction set for developers to work with [27]. Marble's advantage compared to other tools resides on its ability to support overloaded operations. Furthermore, Marble provides its users with a range of operations, excluding, however, division and

bit-wise operations. Marble claims to support the BFV and BGV schemes. However, as of today, the version available on GitHub only supports the HElib library (BGV scheme), while a stable version of BFV has not yet been released.

Palisade is another open-source library for lattice cryptography [4]. Its strength comes from its ability to support multiple encryption protocols. Specifically, it includes public-key and homomorphic encryption protocols, digital signature schemes, and features such as proxy re-encryption and program obfuscation. Several steps are required when coding using the Palisade library, starting from cryptographic context definitions. A portion of the parameters needs to be declared by the developers while the rest is determined by the library. Afterwards, the developers are required to enable the cryptographic features that will be needed for their application.

Microsoft SEAL is a homomorphic encryption C++ library [26]. Its ability to function without external dependencies allows it to operate on multiple operating systems, as well as the .NET framework. SEAL supports the BFV and CKKS schemes. However, this framework fails to support all operations, such as ciphertext division and comparison. Furthermore, it does not support bootstrapping. Still, it is one of the most popular frameworks due to its performance introduced by batching.

SHEEP is an HE evaluation platform currently in development stage [5]. SHEEP utilizes a library agnostic language such that compatibility is ensured across all possible HE solutions. The feature of flexibility however, comes at the price of ease of use. Library agnostic language exhibits similar structure compared to assembly, often producing big sizes of incomprehensible code. As a result, conversion of real-life applications to a compatible format for SHEEP is complex. Besides, a wrapper file implementing the SHEEP instruction set has to be generated before incorporating any library into the platform, thus creating additional work for developers.

Advantages of E3 over other frameworks: E3 supports 1) All encryption schemes, 2) All C++ operators, including division, and 3) *Bridging*, a technique proposed in this work. In addition, E3 improves code portability/reusability, since the same code works out-of-the-box for different cryptosystems or encryption parameters. Moreover, new libraries can be plugged in E3 by third-parties, which also improves code reusability, since existing programs coded for E3 can be seamlessly compiled to the new underlying library. Finally, as an added feature outside FHE, E3 also includes support for Partially Homomorphic Encryption (specifically the Pallier cryptosystem), and integration with root-of-trusts that emulates homomorphism (e.g. CoPHEE [24]).

VIII. Conclusions

In this work, we presented a new methodology combining universal and fast computation, dubbed *bridging*, in order to accelerate fully homomorphic encryption. Bridging has been incorporated in a newly built framework, E3, integrating state-of-the-art FHE libraries while shielding the programmer from the intricacies of properly instantiating and using FHE schemes. Experiments demonstrate significant performance improvements when using both bridging and batching modes in E3. Bridging by itself can offer more than 2 orders of magnitude performance improvement. New FHE schemes can be easily incorporated in E3, providing a solid foundation for future research and benchmarking of FHE schemes.

References

[1] https://github.com/homenc/HElib/tree/master/src/Test_bootstrapping.cpp (commit #65ef24c).

[2] Cingulata. Online: https://github.com/CEA-LIST/Cingulata, October 2019. CEA-LIST.

[3] Lattigo 1.1.0. Online: http://github.com/ldsec/lattigo, October 2019. EPFL-LDS.

[4] Palisade. Online: https://gitlab.com/palisade/palisade-development, October 2019. palisade.

[5] SHEEP. Online: https://github.com/alan-turing-institute/SHEEP, October 2019. Alan Turing Institute.

[6] E3. Online: https://github.com/momalab/e3, February 2020.

[7] Sebastian Angel, Hao Chen, Kim Laine, and Srinath Setty. Pir with compressed queries and amortized query processing. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 962–979. IEEE, 2018.

[8] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (leveled) fully homomorphic encryption without bootstrapping. *Electronic Colloquium on Computational Complexity (ECCC)*, 18:111, 01 2011.

[9] Michael Brenner, Wei Dai, Shai Halevi, Kyoohyung Han, Amir Jalali, Miran Kim, Kim Laine, Alex Malozemoff, Pascal Paillier, Yuriy Polyakov, Kurt Rohloff, Erkay Savaş, and Berk Sunar. A standard api for rlwe-based homomorphic encryption. Technical report, HomomorphicEncryption.org, Redmond WA, USA, July 2017.

[10] Sunjay Cauligi, Gary Soeller, Brian Johannesmeyer, Fraser Brown, Riad S. Wahby, John Renner, Benjamin Grégoire, Gilles Barthe, Ranjit Jhala, and Deian Stefan. Fact: A dsl for timing-sensitive computation. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, page 174–189, New York, NY, USA, 2019. Association for Computing Machinery.

[11] Melissa Chase, Hao Chen, Jintai Ding, Shafi Goldwasser, Sergey Gorbunov, Jeffrey Hoffstein, Kristin Lauter, Satya Lokam, Dustin Moody, Travis Morrison, Amit Sahai, and Vinod Vaikuntanathan. Security of homomorphic encryption. Technical report, HomomorphicEncryption.org, Redmond WA, USA, July 2017.

[12] Jung Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. Homomorphic encryption for arithmetic of approximate numbers, 11 2017.

[13] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. Homomorphic encryption for arithmetic of approximate numbers. Cryptology ePrint Archive, Report 2016/421, 2016. https://eprint.iacr.org/2016/421.

[14] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika 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*, pages 3–33. Springer, 2016.

[15] Eric Crockett, Chris Peikert, and Chad Sharp. Alchemy: A language and compiler for homomorphic encryption made easy. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 1020–1037, New York, NY, USA, 2018. Association for Computing Machinery.

[16] Léo Ducas and Daniele Micciancio. FHEW: bootstrapping homomorphic encryption in less than a second. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 617–640. Springer, 2015.

[17] Junfeng Fan and Frederik Vercauteren. Somewhat practical fully homomorphic encryption. Cryptology ePrint Archive, Report 2012/144, 2012.

[18] Craig Gentry, Amit Sahai, and Brent Waters. Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based. *Proceedings of Advances in Cryptology-Crypto*, 8042, 08 2013.

[19] Shai Halevi and Victor Shoup. Algorithms in helib. In Juan A. Garay and Rosario Gennaro, editors, *Advances in Cryptology – CRYPTO 2014*, pages 554–571, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[20] Shai Halevi and Victor Shoup. Bootstrapping for HElib. In *Advances in Cryptology–EUROCRYPT 2015*, pages 641–670. Springer, 2015.

[21] Hicham Hammouchi, Othmane Cherqi, Ghita Mezzour, Mounir Ghogho, and Mohammed Koutbi. Digging deeper into data breaches: An exploratory data analysis of hacking breaches over time. *Procedia Computer Science*, 151:1004–1009, 01 2019.

[22] Tancrède Lepoint and Michael Naehrig. A comparison of the homomorphic encryption schemes fv and yashe. In *Progress in Cryptology – AFRICACRYPT 2014*, volume 8469, 05 2014.

[23] Dimitris Mouris, Nektarios Georgios Tsoutsos, and Michail Maniatakos. Terminator suite: Benchmarking privacy-preserving architectures. *IEEE Computer Architecture Letters*, 17(2):122–125, 2018.

[24] M. Nabeel, M. Ashraf, E. Chielle, N. G. Tsoutsos, and M. Maniatakos. Cophee: Co-processor for partially homomorphic encrypted execution. In *2019 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 131–140, May 2019.

[25] Amit Sahai and Brent Waters. Fuzzy identity-based encryption. In *Proceedings of the 24th Annual International Conference on Theory and Applications of Cryptographic Techniques*, EUROCRYPT'05, pages 457–473, Berlin, Heidelberg, 2005. Springer-Verlag.

[26] Microsoft SEAL (release 3.3.2). https://github.com/Microsoft/SEAL, 2019. Microsoft Research, Redmond, WA.

[27] 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*, WAHC '18, pages 49–60, New York, NY, USA, 2018. ACM.

[28] Jiyong Yu, Lucas Hsiung, Mohamad El Hajj, and Christopher W. Fletcher. Data Oblivious ISA Extensions for Side Channel-Resistant and High Performance Computing. Proceedings 2019 Network and Distributed System Security Symposium, 2019.