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

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

Abstract

In this technical report we describe E3 (Encrypt-Everything-Everywhere), a framework which enables execution of standard C++ code with homomorphically 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. At its current version, E3 supports a variety of homomorphic encryption libraries, batching, mixing different encryption schemes in the same program, as well as the ability to combine modular computation and bit-level computation.

I. Introduction

In this document, we describe the E3 (Encrypt-Everything-Everywhere) framework for deploying private computation in C++ programs. Our framework combines the properties of both bit-level arithmetic and modular arithmetic within the same algorithm. The framework provides novel protected types using standard C++ without introducing a new library API. E3 is developed in C++, is open source [2], and works in Windows and Linux OSes.

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.

E. Chielle, O. Mazonka, H. Gamil, and M. Maniatakos are with the Center for Cyber Security, New York University Abu Dhabi, UAE.
E-mail: {eduardo.chielle, om22, homer.g, 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

3) Supports *bridging*, a technique mixing different arithmetic abstractions, in some instances enabling orders of magnitude performance improvement.

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. 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 [3]. 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 [9]. GSW (Gentry-Sahai-Waters) is yet another FHE scheme based on LWE [10]. 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 [6].

Libraries have been developed to support the aforementioned schemes, with the most prominent ones being TFHE [7], FHEW [8], HELib [11], Microsoft SEAL [15], and PALISADE [1]. While TFHE and FHEW are developed to support GSW, HElib supports BGV, and SEAL supports both BFV and CKKS. PALISADE supports BGV, BFV, CKKS, FHEW, and TFHE. As anticipated, the programmer has to learn a different API for each library or encryption scheme, thus making the use of different FHE schemes difficult. While a standardization effort is ongoing [4], [5], it still builds on the concept of adding a new API.

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 focus on general purpose computation. In the presented example in Listing 1, 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 the 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: E3 supports two distinct types of computation: 1) Modular arithmetic, which supports arithmetic operations on numeric rings; and 2) Bit-level arithmetic, where Boolean operations are performed on encrypted bits. In Modular arithmetic, it is possible to perform

```cpp
#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 1. User type `SecureInt` behaves as native `int`.

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

Listing 2. An outline example of `SecureInt` definition.

additions, subtractions, and multiplications, but not other operations, since these are the one natively supported by most of the FHE schemes. Meanwhile, Bit-level arithmetic can represent any Boolean circuit; thus, it supports all C++ operations. Bit-level arithmetic is natively supported by GSW, but other encryption schemes can emulate Boolean operations. For example, the NAND gate can be expressed as $\texttt{NAND}(x,y) = 1 - xy$, where x and y are two ciphertexts encryption either zero or one, and 1 represents the encryption of one.

Modular arithmetic is faster than Bit-level arithmetic since the latter is usually emulated using the former. However, its set of operators is very limited and cannot support any application. For that, Bit-level arithmetic must be utilized. Listing 1 shows an example of a program that requires comparison; therefore, it must employ the Bit-level arithmetic. Listing 2 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.

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

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(){...}`

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 `*` is generated as a circuit by a Verilog compiler using the Verilog expression `*`. These circuits are translated into C++ functions and use FHE functions operating on encrypted bits instead of ordinary logic gates.[1]

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 1 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 improve performance of the program by utilizing optimization algorithms applied onto combinational

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

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 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.

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: E3 also supports *Bridging*, which is the ability to combine the comprehensive but slow Bit-level arithmetic (`SecureInt`) with the fast but limited Modular arithmetic (`SecureMod`) in the same application. Non-native homomorphic operations, such as comparisons, require the use of the `SecureInt` type, while additions, subtractions, and multiplications can be done using `SecureMod`. Bridging defines a conversion from `SecureInt` to `SecureMod`, providing significant performance improvements since only the non-native operations need to use `SecureInt`, while native operations can use the type.

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, since `SecureInt` is a regular C++ class object.

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, e.g. logical AND (`a&&b`) would first convert `SecureInt` into `SecureBool`, then make the logic operation, and return `SecureBool`. 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 the 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.[2]
- Defines all "Overloaded" C++ operators of Table III.

[2]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.

- 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 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`.

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[3] 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 [14] can be used. Our framework can be naturally applied anywhere Fully Homomorphic Encryption can be applied,

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

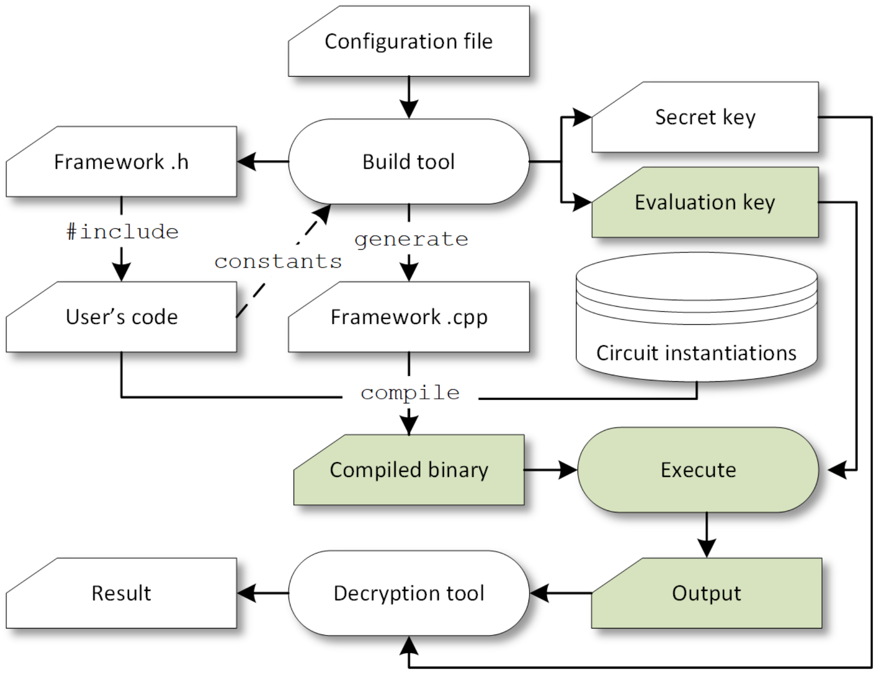

Fig. 1. Process diagram presenting the components required to compile and execute a C++ program operating on FHE encrypted data. The shaded parts can be outsourced to a third party.

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 1) needs 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 the 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 a 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. E3 Features

In this section we present the features provided by E3. In general terms, the provided features can be divided into three categories; Programming, Technical, and Accessibility.

A. Programming Features

E3 uses the C++ programming language. It supports all the C++ operators (addition, subtraction, multiplication, division, comparison, bitwise operations), including division, a feature many frameworks currently do not support. E3 also provides flexibility and allows code reusability, as it enables users to

TABLE IV

LIBRARIES, SCHEMES, AND MODES SUPPORTED BY E3

Library	BFV					BGV					CKKS					GSW		Paillier
	mod	bit	bat	brid	boot	mod	bit	bat	brid	boot	mod	bit	bat	brid	boot	bit	boot	mod
FHEW	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	●	●	×
HELib	×	×	×	×	×	❍	●	●	❍	●	×	×	×	×	×	×	×	×
Palisade	●	❍	●	❍	×	❍	❍	❍	❍	×	●	❍	❍	❍	×	❍	❍	×
SEAL	●	●	●	●	×	×	×	×	×	×	●	❍	❍	❍	×	×	×	×
TFHE	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	●	●	×
Native*																		●

mod: modular arithmetic, bit: bit-level arithmetic, bat: batching, brid: bridging, boot: bootstrapping

* Natively implemented in E3

TABLE V

SCHEMES AND OPERATORS SUPPORTED BY E3

Scheme	Modular Arithmetic				Bit-level Arithmetic							
	add	mul	rot	sub	add	bit	cmp	div	log	mul	rot	sub
BFV	●	●	●	●	●	●	●	●	●	●	●	●
BGV	❍	❍	❍	❍	●	●	●	●	●	●	❍	●
CKKS	●	●	●	●	❍	❍	❍	❍	❍	❍	❍	❍
GSW	×	×	×	×	●	●	●	●	●	●	×	●
Paillier	●	×	×	●	×	×	×	×	×	×	×	×

× Not supported by underlying library or scheme

❍ Implementation is possible, but not currently implemented in E3 version #9fb718f

● Supported by E3

operate with the same code on different FHE libraries or encryption schemes. In addition, E3 allows the integration with other root-of-trusts that emulate homomorphism [12], [13].

B. Technical Features

E3 supports a big variety of Homomorphic Encryption libraries. This includes FHEW, HELib, SEAL, TFHE, as well as Paillier. Consequently, the framework also supports a number of schemes including BFV, BGV, CKKS, and GSW. A unique feature of E3 is bridging. This novel technique mixes different arithmetic abstractions, or, in other words mixes both Modular and Bit-level arithmetic in one program. This provides the ability to convert variables from integral type to modular, which eventually results in performance improvements of several orders of magnitude in certain cases. E3 also fully supports batching capabilities as it allows for the parallel processing of plaintexts in a Single Instruction Multiple Data fashion, along with rotation operations. Regarding parameter selection methods, the user is required to manually select the parameters that will be used according to the application at hand. In addition, E3 automatically handles ciphertext relinearization and rescaling. Lastly, regarding the plaintext space, E3 is capable of operating on both modular arithmetic and binary circuits.

C. Accessibility Features

We define accessibility features as any feature that improves the ease of use of the proposed framework and allows better interaction between E3 and its users. This category includes the availability of the implementation code, examples facilitating the familiarization with the framework, and documentation that provides instructions regarding its operation. E3 makes available all these resources. The code, examples and documentation for the proposed framework are open sourced, and can be accessed at the GitHub repository of E3 [2].

Acknowledgement

We would like to thank Sanja Kastratovic, Yeojin Jung, Reem Hazim, and Maya Fayed for their contributions along the work on E3.

References

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

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

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

[4] 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.

[5] 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.

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

[7] 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.

[8] 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.

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

[10] 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.

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

[12] Oleg Mazonka, Nektarios Georgios Tsoutsos, and Michail Maniatakos. Cryptoleq: A heterogeneous abstract machine for encrypted and unencrypted computation. *IEEE Transactions on Information Forensics and Security*, 11(9):2123–2138, 2016.

[13] Mohammed Nabeel, Mohammed Ashraf, Eduardo Chielle, Nektarios G Tsoutsos, and Michail Maniatakos. Cophee: Co-processor for partially homomorphic encrypted execution. In *2019 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 131–140. IEEE, 2019.

[14] 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.

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