Received December 10, 2019, accepted January 13, 2020, date of publication January 28, 2020, date of current version February 6, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.2970120

Practical Data-in-Use Protection Using Binary Decision Diagrams

OLEG MAZONKA[1], ESHA SARKAR[2], (Student Member, IEEE), EDUARDO CHIELLE[1], NEKTARIOS GEORGIOS TSOUTSOS[3], (Member, IEEE), AND MICHAIL MANIATAKOS[1], (Senior Member, IEEE)

[1]Center for Cybersecurity, New York University Abu Dhabi, Abu Dhabi 129188, UAE
[2]Department of Electrical and Computer Engineering, Tandon School of Engineering, New York University, Brooklyn, NY 11201, USA
[3]Department of Electrical and Computer Engineering, University of Delaware, Newark, DE 19716, USA

Corresponding author: Esha Sarkar (es4211@nyu.edu)

The work of Esha Sarkar was supported by the NYU Abu Dhabi Global Ph.D. Fellowship.

ABSTRACT Protection of data-in-use, contrary to the protection of data-at-rest or data-in-transit, remains a challenge. Cryptography advances such as Fully Homomorphic Encryption (FHE) provide theoretical, albeit impractical, solutions to functionally-complete computation over encrypted operands, necessary for general-purpose computation. In this work, we propose a practical data-in-use protection mechanism that, contrary to application-specific homomorphic encryption approaches, focuses on arbitrary computation native to established programming languages, such as C++. Therefore, our work provides a more efficient alternative to FHE schemes that can be used for general-purpose computation. Specifically, we use Binary Decision Diagrams (BDD) to transform high-level programming operations to their equivalents working on protected data. To automate this, we develop a framework that allows automatic conversion of program expressions over encrypted operands into efficient circuits that are reduced using BDDs and can simulate corresponding composed operations. Our experimental results show that our methodology is orders of magnitude faster than state-of-the-art FHE schemes and enables execution of real C++ applications with practical overheads. Our framework is complemented with security analysis proving resistance to different attack methods.

INDEX TERMS Data security, privacy, data privacy, security.

I. INTRODUCTION

As modern computational devices become more ubiquitous hosting an enormous amount of sensitive data of millions of users, there is no shortage of concerns about the data protection guarantees offered by these platforms. At the same time, the proliferation of advanced threats to cloud computing [1], reports on state surveillance and mass data collection [2], as well as the disclosure of processor vulnerabilities that can leak information from billions of devices [3], [4], justify the lack of trust on behalf of the end-users.

Contrary to *data in transit* and *data at rest*, which could be protected using encryption algorithms like AES, processing sensitive information (i.e., *data in use*) remains a single point of failure for modern computing platforms, as contemporary processors operate exclusively on plaintexts. In order to compute on sensitive data, current architectures need to decrypt, operate on the data, and then re-encrypt. Even solutions like Intel SGX or AMD Secure Memory Encryption, which encrypt data residing in the main memory, need to decrypt operands before they enter the processor pipeline. As a result, known/unknown software/hardware vulnerabilities can be exploited to leak sensitive data.

Our thesis is that sensitive data should *never appear in the clear* anywhere in a computational device, and should remain protected during processing by the CPU. Thus, in case of an application vulnerabilities (e.g., Heartbleed [5]), operating system vulnerabilities (e.g., Windows zero-days [6]), or hardware vulnerabilities (e.g., Spectre [3], Meltdown [4]), the data leaked would remain protected.

Related Work: Data-in-use protection can be approached either using hardware-enforced isolation (e.g., Intel SGX) or cryptography (e.g., Homomorphic Encryption). For the former, the variety of data leakage attacks reported in the literature (SGXpectre [7], Foreshadow [8]) serve as a painful reminder that as long as data appears in the clear in the

The associate editor coordinating the review of this manuscript and approving it for publication was Tiago Cruz.

CPU pipeline, it can be extracted. Therefore, our focus is to inherit the properties offered by homomorphic cryptographic schemes, such as manipulation of data without decryption, but with practical overheads suitable for everyday general-purpose computation. Other cryptography mechanisms, such as Multi-Party Computation and Functional Encryption, are not applicable for general-purpose computation,[1] as they are used in scenarios with multiple parties computing a function or they impose constraints on what can be computed.

As mentioned earlier, computing on encrypted data is possible by a special type of encryption, dubbed *homomorphic encryption*, which allows manipulation of values directly in encrypted form. Different attempts to implement secure homomorphic computation resulted in an abundance of mathematical models [9]. Based on their computational capability, these models can be split into three distinct classes, as outlined in the next paragraphs.

Partially Homomorphic Encryption schemes, such as Paillier [10], RSA [11], or ElGamal [12] allow a single type of homomorphic operation, which can be performed indefinitely. For example, Paillier enables the addition of plaintexts by modular multiplication of the ciphertexts, and RSA enables plaintext multiplication, correspondingly. While these schemes allow very interesting applications, such as e-voting [13], their single operation support renders them unsuitable for general-purpose computation.

Somewhat Homomorphic Encryption [14], [15], on the other hand, expands the set to two available operations (e.g., addition and multiplication over bits), which allows the evaluation of encrypted combinational circuits and enables more possible applications. On the downside, these schemes support the execution of certain operations for a limited number of times only, before the accumulated noise corrupts the ciphertexts. As a consequence, they can implement programs with known input size and the ability of preliminary global static execution analysis, but cannot be used for arbitrary, indefinite computation on unknown size of encrypted inputs. Indeed, *general-purpose computation* requires that a program is able to calculate μ-recursive functions, or equivalently to emulate a Turing machine. Likewise, it requires that a program, as a normal computer process, can run indefinitely until some external event; for example, in an *interactive* program, the program operator enters new inputs or makes the decision to halt the execution based on the output produced so far.

The invention of the first fully homomorphic encryption (FHE) scheme in 2009 has enabled unconstrained computation on encrypted values [16], which is precisely our goal. Many schemes of improved performance have been proposed since [17]–[19], with the BGV [20] and GSW [19] schemes being two of the most impactful proposals for evaluating arbitrary functions. While employing FHE can solve the data-in-use problem, its practicality remains a huge challenge for general-purpose computation. Programming with FHE requires, to some extent, understanding of theory and application of homomorphic encryption. Moreover, ciphertexts can be in the order of megabytes [21], which can make very simple algorithm take hours to execute in fully homomorphic domain [22].

[1] Apart from some special cases, e.g. Private Information Retrieval.

Binary Decision Diagrams is a mathematical structure commonly used for analysis and processing of digital circuits. They also have been used in secure computation in the past: For example, in multi-party computation as a replacement for explicit gate computation [23], and in countermeasures against side-channel attacks [24] for hiding capacitance and delays in computation. We use this mathematical construction for an entirely different purpose: As a form of canonical representation of combinational circuits.

Our Approach: Motivated by the impractical overheads of applying FHE to general-purpose computation, in this work we propose a new methodology for practical general-purpose data-in-use protection based on reduced ordered Binary Decision Diagrams (called BDDs in this paper for simplicity).

Similar to FHE, however, our methodology transforms a plaintext into an encrypted counterpart (ciphertext) using a probabilistic encryption function. Our proposed approach starts at the bit level, implementing logic gates (such as NOT, AND, etc.) that operate on encrypted Boolean arguments and output encrypted Boolean values instead of plaintext bits. We call such gates *homomorphic gates* (`HG`), which can be used to implement high-level programming operators as circuits; these special operators are then used within standard C++ programs to manipulate encrypted integers. Specifically, our methodology introduces a new C++ type for integers that are encrypted using a probabilistic cipher. During compilation of a C++ program, our framework translates each `HG` to a Boolean circuit, which corresponds to a logic gate in the circuit implementing each programming operator for encrypted integers. Our framework protects computation on encrypted data using the BDD representation of each `HG` circuit [25]; after translation, the code implementing this BDD representation is included in the final executable of the C++ program. Using our developed framework, end-users can incorporate our methodology to high-level C++ programs, by replacing the type of integer variables whose values should be protected.

The security of BDD processing-based protection relies on the *property of the canonical representation of a circuit* [26], and the hardness of reverse engineering a BDD-processed circuit. As a motivating example, let C_h be a circuit that implements the modular multiplication operation (which is the homomorphic operation of the Paillier scheme [10]), and C_s be a circuit that simulates the same function explicitly: Specifically, C_s decrypts its two input ciphertexts using a private key, performs modular addition on the plaintexts, and re-encrypts the plaintext result using a public key and the product of the random nonces of the inputs ciphertexts. In this case, since C_s implements the same function as Paillier's homomorphic operation, the BDD-processed circuit corresponding to C_h *is the same* as the one corresponding to C_s.

Thus, breaking the security of the BDD-processed C_s implies breaking the security of the Paillier cryptosystem, which, however, is considered secure.

Conceptually, our methodology works as FHE on the bit level, which is necessary for general-purpose computation. Different homomorphic encryption schemes rely on different hardness assumptions: For example, the Paillier cryptosystem is based on the decisional composite residuosity assumption. Popular FHE schemes base their security in the Learning With Errors (LWE) mathematical problem which is conjectured to be hard. The security of our encryption methodology, on the other hand, is based on the innumerability of Boolean circuits with a large number of inputs/outputs, similar to logic encryption [27] and IC camouflaging [28] security guarantees. The difficulty of reverse engineering BDD-processed Boolean circuits can also be considered a Boolean Satisfiability problem (SAT), for which no general/efficient algorithm is known. Thus, we use SAT attacks to assess security. As illustrated in our experiments, the complexity of breaking the security of our proposed scheme is exponential to the bit size of the random nonce added to ciphertexts.

Our Contribution: In this work, we develop the methodology, the framework, and the corresponding full-support software stack to enable a programmer/user to create a C++ program that computes on encrypted data without decryption. The user can control the balance between security and performance of the program by adjusting the security parameter λ, for which we give recommendations and estimates. Specifically:

1) We propose a novel methodology for processing private data based on BDD-processed circuits. We also define a novel algorithm for the design of probabilistic encryption and decryption modules.
2) We develop `Circle`, a tool that automates the development of `HG` Boolean circuits utilizing a BDD engine. The tool translates circuits into C/C++ functions, and optionally to Verilog modules for FPGA acceleration. Also our framework automatically generates C++ classes for private integers.
3) We perform performance evaluation as well as security analysis using different types of attacks, and demonstrate the strength of our data protection mechanism. Our results show thousands of times improvement in the performance compared to the best equivalent FHE scheme (TFHE [29]): Faster computational speed ($\times 10^3 - 10^4$) and smaller memory sizes for encrypted bits ($\times 2000$).[2]

Our software is written in standard C++ and tested on Linux and Windows and is open-sourced.

II. PRELIMINARIES

BDD: In this work, we use *reduced ordered Binary Decision Diagrams* (or simply BDD), which is a special case of general binary decision diagrams. These diagrams are specific data structures representing Boolean functions in the form of a graph. Any Boolean function f can be represented by Shannon expansion over its argument x_m, with other arguments $x_1, x_2, \ldots, x_n$ as:

$$f(x_1, .., x_m, \ldots, x_n) = x_m f(x_1, .., 1, .., x_n) + \bar{x}_m f(x_1, \ldots, 0, \ldots, x_n)$$

where the 'bar' symbol is negation, the 'plus' symbol is OR, and 'multiplication' is AND. Such an expansion applied to all the function arguments increases the number of terms exponentially. Binary decision diagrams alleviate such growth by finding and merging common parts of functions $f(x_1, \ldots, 1, \ldots, x_n)$ and $f(x_1, \ldots, 0, \ldots x_n)$. In BDD, a function is represented as a sequence of layers of nodes. Each layer corresponds to a function argument (variable) and each node is connected to subsequent layer nodes by exactly two connectors: One corresponding to variable equal to 1, and the other corresponding to 0. The final terminal nodes of the structure specify the final result: either 1 or 0. BDD has the property of representing the function in a unique (canonical) way for the predefined order of its variables, if all redundant substructures can be removed, which has been shown in [26].

Homomorphic Gate: Formally, we define a *homomorphic gate* (`HG`) over ciphertexts c_x and c_y as follows: Let $f(m_x, m_y)$ denote a two-argument Boolean 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, `HG` is defined by the homomorphism of surjective D_k:

$$\mathrm{D}_k\left(\mathtt{HG}(c_x, c_y)\right) = f\left(\mathrm{D}_k(c_x), \mathrm{D}_k(c_y)\right)$$

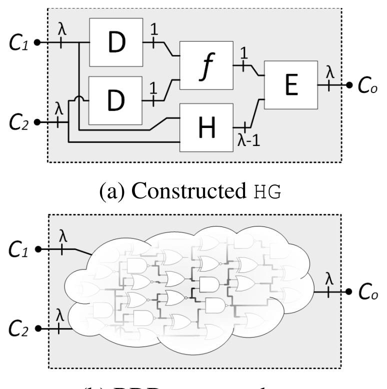
which can be converted into the explicit composition over ciphertexts:

$$\mathtt{HG}(c_x, c_y) = \mathrm{E}_k\Big(f\left(\mathrm{D}_k(c_x), \mathrm{D}_k(c_y)\right), \mathrm{H}(c_x, c_y)\Big) \qquad (1)$$

where H is a digest (hash) function generating randomness r from the input ciphertexts. The definition of Eq. 1 is shown as a diagram in Fig. 1(a).

One important concern with the definition of an `HG` as in Eq. 1 is preventing misuse of D_k to decrypt arbitrary values. This threat can be mitigated by blending the sequence of decryption, plaintext operation, and result re-encryption operations. In this case, the security of the construction relies on the difficulty of recovering the decryption function. As illustrated in the construction of Fig. 1, D_k and E_k can be implemented as distinct sub-circuits that enclose the sub-circuit implementing the operation f. As discussed in Section V-D, processing a Boolean function using BDDs enables the construction of protected Boolean circuits, where functions D_k, f and E_k are *fused together, concealing their distinct sub-circuits.*

[2]The numbers are given for security parameter $\lambda = 80$.



(a) Constructed HG

(b) BDD-processed HG

FIGURE 1. (a) Boolean circuit implementing a HG over ciphertext values, by decrypting the inputs (D), applying an operation *f* on the plaintext values and re-encrypting the result (E). A randomness extraction module H (digest function) enables probabilistic re-encryption. (b) A sketch of the same circuit processed by BDD. Both circuits are functionally equivalent.

It can be argued that BDD processing locates particular types of symmetries in the circuit and collapses the logic by removing those symmetries. For example, given two functions F and F^{-1}, a sequence of $F \circ F^{-1}$ (as well as $F^{-1} \circ F$) implemented as a circuit of two sequential modules and processed by a BDD engine would always collapse into an identity function (zero gate circuit) regardless of BDD implementation or BDD variable ordering.

Terminology: To avoid confusion between different terminologies, we call HG a logic gate working on ciphertexts regardless of whether it has been processed by BDD or not (Fig. 1). BDD-processed gates such as NAND and NOT are called hNand and hNot accordingly. Fig. 1 shows two HGs: one, not BDD-processed (left) and the other, which is BDD-processed (right). If the operation f is NAND, then the HG on the right would be hNand.

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 our own circuits with a Verilog compiler to facilitate standard programming operations. For example, the multiplication C++ operator (i.e., $\star$) is generated as a circuit by a Verilog compiler using the Verilog expression $\star$. These circuits are translated into C/C++ functions and use our HGs instead of ordinary logic gates. HGs are also C/C++ functions which are generated by our software according to the encryption scheme that we define in Section IV.

Table 1 lists these four layers explicitly. Layer 1 is the user code and does not reveal any encryption concepts. 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. Layer 2 is the implementation of the operators and is provided by the framework. Its code is written once and can be reused for any homomorphic encryptions. Layer 3 are basic functions of computation, such as addition, division, comparison, and others. These functions are pre-generated by a circuit design compiler from the basic 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. These are HGs generated by Circle, which have to be initiated by the user since they bind to the secret functions D and E of the encryption instantiation generated.

III. USAGE SCENARIO

A. USER'S PERSPECTIVE

We assume that the user writes a computer program and passes it through our framework. The user specifies the security parameter for the encryption in the configuration file of the software framework. It should be noted that, when comparing to other encryptions such as AES, the performance of the program is much more sensitive to this security parameter. Therefore, the programmer can select the optimal value according to his/her security requirements. The exact secure values of the parameter are discussed in the security analysis section.

Our framework generates BDD-processed HGs (such as hNand) in the form of C++ functions which are automatically embedded in the compiled binary. At the same time, it generates a decryption function corresponding to the encryption of the HG. This decryption function represents the private key and is required to decrypt the output of the program. In this scenario,

1) the user (programmer) generates the protected version of the program along with the decryption key;
2) gives the program, either as source code[3] or binary, to run at an untrusted party;
3) obtains the output result from the program; and finally
4) decrypts the result using the decryption function.

It should be emphasized that *only* the user can decrypt any output. This differs from applications that require multiple parties to be able to compute different functions on the encrypted data, where Functional Encryption schemes [30] can be used. Our framework can be naturally applied to any private outsourcing scenario (i.e., using the cloud for faster processing or permanent data storage), and our goal is to support any arbitrary algorithm on any amount of data without having to re-encrypt the data to support new algorithms.

From the user (programmer) perspective the program has the same functional structure in both encrypted and unencrypted forms. Since the protected data is never decrypted during program execution, a natural restriction arises from the requirement that the program execution cannot depend on protected data. The control flow and memory access are governed only by program data which is not protected. This property is called data-oblivious computation and recent work has

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

TABLE 1. Abstraction layers of computation.

Level	Element	Source	Example
1	C++ operator	User code	`SecureInt a,b; a*b;`
2	Class function implementation	Framework	`SecureInt operator*(Secureint) {... mult(); ...}`
3	Circuit function	Verilog generated	`mult() {... hNand(); ...}`
4	`HG`	`Circle` generated	`hNand() {...}`

```
 1  #include <iostream>
 2  #include <algorithm>
 3  #include "circle.h"
 4
 5  int main()
 6  {
 7      int max_iter = 30;
 8      SecureInt input = 7_E;
 9      SecureInt i = 0_E, result = 0_E;
10      SecureInt a = 1_E, b = 1_E;
11
12      while( ++i, max_iter-- )
13      {
14          result += (i == input) * a;
15          std::swap(a,b);
16          a += b;
17      }
18      std::cout << result << '\n' ;
19  }
```

Listing 1. Data-oblivious Fibonacci program.

```
class SecureInt{ ... };
constexpr SecureInt operator""_E(unsigned long
     long int x){ ... }
```

Listing 2. An outline example of `circle.h`.

highlighted that data-oblivious programs are naturally more resilient to side-channel attacks [31].

A Usage Example: The computational abstraction levels of our methodology can easily be seen from a real program example. In Listing 1, we present a standard C++ program that computes Fibonacci numbers in the encrypted domain. This program can compile using any standard C++ compiler. The user provides an encrypted index `input` and the algorithm iterates up to a maximum index (in our case `max_iter` is `30`). Listing 2 outlines the elements of the header generated by our framework.

The Fibonacci algorithm is written in a data-oblivious way so that an encrypted variable `result` is updated based on an "encrypted" multiplexer formula that adds the correct output or zeros (line 14). The C++ source uses our developed private integer types that implement all programming operators. Specifically this program uses only ++, +=, ==, $\star$, and <<. The program uses special annotation for private constants (e.g., `7_E`), which are encrypted automatically before compilation to avoid having plaintext constants stored within the binary.

B. THREAT MODEL

In this work, we assume an honest-but-curious host of the user's program and data. The host will execute the program without the intent of corrupting the output but may peek into the data to extract information. Moreover, the hardware and software of the host may have known or unknown vulnerabilities. Therefore, even if the cloud provider does not intentionally try to leak data, data can leak by adversaries using these vulnerabilities (e.g., Heartbleed [5], Meltdown [4], Spectre [3]). We assume that the host computer can have multiple tenants, and the host can keep the program and the data indefinitely.

Once the user program and data are outsourced and reside on the host, security relies on the difficulty of deciphering the decryption function out of the functionality of elements of the program. Since the protected data is never decrypted, the only point of possible attack is the function processing the protected data, which are the functions evaluating `HG`s. We assume the adversary is able to locate/extract these functions, and evaluate them with any arguments. Moreover, we assume the adversary is able to guess the values of ciphertexts constants inside the program (e.g., encrypted 0 or encrypted 1). This attack model will be used in any discussion on security in the following sections.

IV. CIRCUITS FOR PROBABILISTIC ENCRYPTION & DECRYPTION

A. BALANCED CIRCUITS

As discussed in Section II, in order to build a `HG`, we need a pair of functions (sub-circuits) for decryption (D) and encryption (E). Processing common ciphers through BDD engines can be challenging, due to their complexity. To overcome this limitation, we focus on BDD-friendly solutions.

Let c be an encrypted value (ciphertext) and m be the corresponding decrypted value (plaintext). A decryption circuit D represents the function $\mathrm{D}(c) = m$; the value c is the input to D and its bitsize $|c|$ equals to the number of input wires in D. Likewise, the value m is the output of D and its bitsize $|m|$ equals the number of output wires in D.

A necessary condition to provide resilience to common threats such as chosen-plaintext attacks (CPA) [32] requires the usage of probabilistic ciphers, where each plaintext is randomly mapped to one of many equivalent ciphertexts. Hence, we require $|c| > |m|$, and $|c| - |m|$ represents the number of randomness bits added while encrypting m into c. Naturally, $2^{|c|}$ is the cardinality of all possible inputs and $2^{|m|}$ is the cardinality of all possible outputs. If D has the same input range $|c|$ for each output m, then there are exactly $2^{|c|-|m|}$ different ciphertexts mapped to each plaintext output. Let us call a circuit *balanced* if every output value m has exactly $2^{|c|-|m|}$ different inputs mapping to it; otherwise, it is *unbalanced*.[4] A special case circuit with $|c| = |m|$

[4]Here, we use the terminology first introduced in [33] and publicized in subsequent works by others (e.g., [34]).

corresponds to a *bijective* function, also called a one-to-one circuit. We also highlight a distinction between *input values* and *circuit inputs*: an input value is a collection of bits, while a circuit input is a collection of input wires. In our case, if the decryption circuit has $|c|$ input wires then there are $2^{|c|}$ input values.

We can assume that both D and E are constructed as one-to-one circuits; using only a subset of the outputs of D (i.e., m) does not break the aforementioned balancing property due to the following proposition: *Any subset of outputs of a balanced circuit is balanced.* This can be confirmed by the following argument: If the number of output wires is $|m|$ and we select a subset $|s|$, then each s belongs to a subset of $2^{|m|-|s|}$ values of m, since $|m| - |s|$ is the number of bits excluded from s; hence, the number of inputs corresponding to s is $2^{|c|-|m|}2^{|m|-|s|} = 2^{|c|-|s|}$.

Following the proposition, it is possible to complement the balanced subset to build a bijective circuit; it is, however, impossible to complement the unbalanced subset to build a bijective circuit. Therefore, any balanced circuit can be considered as a subset of a one-to-one circuit. Since every one-to-one circuit has an inverse circuit (i.e., a circuit of the inverse function), for any balanced circuit with $|c|$ inputs and $|m|$ outputs there exist circuits with $|m| + (|c| - |m|)$ inputs and $|c|$ outputs. These circuits are the inverses of the original balanced circuit, given that $|m|$ inputs correspond to $|m|$ outputs of the original circuit and the remaining $|c| - |m|$ inputs are additional arbitrary inputs. If the original balanced circuit is our decryption module D, then the inverted circuit is the encryption module E; in the latter case, the $|c| - |m|$ inputs correspond to the probabilistic part of encryption and their functions are complementary: $m = \mathrm{D}(\mathrm{E}(m, r))$ where r is an arbitrary value of $|c| - |m|$ inputs of E.

B. REVERSIBLE OPERATIONS

Let us define a *linear* operation L as

$$y = Lx \equiv Ax \oplus b \tag{2}$$

where x, y, and b are Boolean vectors of size $|c|$, A is a Boolean square matrix, $\oplus$ is the XOR operation and the matrix operation is defined as $(Ax)_i = \bigoplus_{j=1}^{|c|} A_{ij}x_j$. While matrix elements A_{ij} are multiplied with the elements of the vector x_j, there are no AND gates necessary to implement a corresponding circuit since the values of A are constant and they only select a subset of the vector elements in x. It can be proven that if a one-to-one circuit is built with XOR and NOT gates, then its function is L. Such circuits can efficiently be constructed in complementary pairs: L and L^{-1}.

A non-linear transformation F, which retains the balanced property of the circuit, can be selected similarly to an unbalanced Feistel cipher with a random round function as follows: $y = x \oplus f(x)$ and $y_i = x_i \oplus f_i(\{x_{j\neq i}\})$, where x_i and y_i are the input and output i-th bits, $f_i(\{x_{j\neq i}\})$ is an arbitrary function that operates on any subset of x's bits excluding i-th, and $f \equiv \{0, 0, \ldots, f_i, \ldots, 0, 0\}$. Such functions are involution, i.e., $(F \circ F)x = x$.

Having both transformations L and F, it is possible to construct D and E pairs of balanced circuits. In this case, we can employ our novel algorithm for constructing decryption and encryption modules (as elaborated in Section IV-C), which is applicable to any balanced circuit that can be represented as a sequence of linear L and non-linear transformations F. Note that non-balanced functions can be used as ingredients to F, since f_i is an arbitrary function. Both operations L and F complement each other in such a way that: L operates on a vector in parallel, but remains factorizable (as it involves only XOR and NOT gates); F, on the other hand, works only in sequential manner (one bit at a time), but employs non-reversible logic operations (e.g., AND). While L can be considered as a special case of F (since a sequence of F using XORs and Boolean units can express any L), generating L directly using matrices is much simpler and ensures appropriate diffusion properties.

C. ALGORITHM FOR CONSTRUCTING D AND E

The L operation that describes one-to-one circuits has a well defined inverse operation L^{-1}, which implies that the decryption and encryption modules can be expressed as sequence of forward and backward transformations:

$$\begin{aligned} \mathrm{D} &= B_N \circ B_{N-1} \ldots B_2 \circ B_1, \\ \mathrm{E} &= B_1^{-1} \circ B_2^{-1} \ldots B_{N-1}^{-1} \circ B_N^{-1} \end{aligned} \tag{3}$$

where B is defined as either a linear or non-linear operator: $B = \{L \text{ or } F\}$. The form of Eq. 3 ensures that D and E operations are inverse of each other: $\mathrm{D}\circ\mathrm{E} = \mathrm{E}\circ\mathrm{D} = \mathbf{1}$; it also suggests a straightforward general algorithm for generating D and E modules.

D and E Generation: First, a B operation is randomly selected (either L or F). For L, we generate a random matrix A and a random vector b, along with the inverse matrix A^{-1} and vector $A^{-1}b$. For F, we select a non-balanced scalar function f of $n < |c| - 1$ arguments, and then randomly select a subset of n indices for the input to function f, as well as one index excluded from the subset to be the output of f. Once the function (L or F) is determined, it is named B_1 and its inverse B_1^{-1}. The same process is repeated N times defining a sequence of operations as in Eq. 3.

There exists a simple and efficient method for constructing the random matrices A and A^{-1} at the same time, following the steps of the Gauss-Jordan matrix elimination method. Starting with the unit matrix of size n, we sequentially select the top first and second rows and randomly apply $A_{2j} = A_{1j} \oplus A_{2j}$, followed by the first and third ($A_{3j} = A_{1j} \oplus A_{3j}$), repeating until the first and n-th. Then the process continues with the second and third, until second and n-th. The iterations continue until the last selection of the n − 1-th and n-th; at that point, we follow the same steps in reverse order (from the bottom up). This process has finite steps (the complexity is $O(n^2)$) and will always generate A with saturated randomness. If the exact same sequence is repeated in the reverse order with the same random application $A_{ij} = A_{kj} \oplus A_{ij}$, the generated matrix would be A^{-1}.

A Trivial Numerical Example: As a simple example demonstrating how D and E are constructed in practice, let us limit the size of the ciphertext to 3, and select the *FLF* formula. Let us choose as non-linear operation f a logical function NAND denoted as NAND$(x, y) = !(xy)$, working on random indices of the vector. For the first F in the formula, we generate 3 random[5] indices: $(1, 0, 2)$. These indices specify the operation $x_1 := x_1 \oplus !(x_0 x_2)$. Next, for the function L, matrix A and vector b are randomly generated:

$$A = \begin{pmatrix} 1 & 1 & 0 \\ 1 & 0 & 0 \\ 1 & 1 & 1 \end{pmatrix} \qquad b = \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix}$$

Their inverse L^{-1} is obtained as described above:

$$A^{-1} = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix} \qquad A^{-1}b = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}$$

Finally, the second F is generated as: $(2, 1, 0)$, which is the operation $x_2 := x_2 \oplus !(x_1 x_0)$.

Next, following the forward and backward formulas, our tool `Circle` generates the following circuits:

$$D = \begin{bmatrix} b := x_1 \oplus !(x_0 x_2) \\ d := x_0 \oplus b \\ e := x_0 \oplus b \oplus x_2 \\ y_0 := !d \\ y_1 := x_0 \\ y_2 := !e \oplus !(y_1 y_0) \end{bmatrix} \qquad E = \begin{bmatrix} c := x_2 \oplus !(x_1 x_0) \\ d := x_0 \oplus x_1 \\ g := x_0 \oplus c \\ y_0 := x_1 \\ y_2 := g \\ y_1 := !d \oplus !(y_0 y_2) \end{bmatrix}$$

where variables x are inputs and y are outputs, and the remaining names are internal wires of the circuits. The transformation from Fs and L into the final D and E may not be obvious due to a slight optimization done by `Circle`, but it can be easily verified in this simple example. The first F clearly appears in the first line of D, and the second F in the first line of E.

V. SECURITY CONSIDERATIONS FOR HOMOMORPHIC CIRCUITS

A. INTRODUCTION

As mentioned in Section I, the security of our methodology is based on the innumerability of Boolean circuits with sufficiently large number of inputs and outputs, similar to logic encryption [27] and IC camouflaging [28] security guarantees. The best known algorithm for solving such problems is the use of SAT solvers. Therefore, the security of our encryption is estimated directly by the performance of the best known algorithms breaking the encryption, presented in Section VIII.

In this section, we explore some theoretical aspects of security of our method and implications of selecting arbitrary attributes for the encryption.

B. PRIVATE AND EVALUATION KEYS

In our methodology, unlike white-box cryptography [35], there is no obfuscation procedure to hide the encryption keys: The algorithm proposed in Section IV-C generates two complementary operations based on a random sequence, which are consequently embedded inside the `HG`. In a sense, the private key is the module D, while E along with digest module H remain protected encryption elements. Breaking the encryption implies recovering module D or its functional equivalent. Since D and E are operators that uniquely define encryption (key), any set of `HG`s based on the same D and E would share the same encryption. And the opposite is true: different pairs of D and E could be used in the same computation process, but would not be compatible. BDD-processed `HG`s, necessary executing the protected program, constitute the evaluation key, i.e., they are parts that are supplied with the program in order to evaluate functions on the protected values.

C. PRAGMATICS OF SECURITY

Having encryptions and decryption modules E and D, an `HG` can be constructed according to Eq. 1 with the help of some digest function H. A universal gate NAND (equally NOR) ultimately is enough for any computation. Its BDD-processed homomorphic construction (`hNand`) can be used inside the circuits for standard operations, e.g., arithmetic multiplication can be performed using `hNand`. Therefore, each bit is first encrypted with E, the encrypted bits (ciphertext) are arranged into arrays representing integers of the protected type, and the arithmetic is executed on encrypted bits with `hNand` as if they are ordinary bits processed by NAND gates.

In our work, we assume that the user prepares the protected executable which is released to an adversary. The program may have encrypted constants inside, but is also able to dynamically encrypt data on the fly. Without revealing the protected operation E, an encryption can be constructed out of `hNand`, for example an encryption of unit $\tilde{1}$ as:

$$\tilde{1} = \mathrm{E}'(1, x) = \texttt{hNand}(x, \texttt{hNand}(x, x)) \qquad (4)$$

and $\tilde{0} = \texttt{hNand}(\tilde{1}, \tilde{1})$ as its inverse. Here, E$'$ is not related to encryption function E; E$'$ takes λ bits as a random input x, contrary to E taking $\lambda - 1$. With a function implementing Eq. 4 the program can input both plaintext as well as ciphertext, and operate on ciphertext. Yet, the output from the program has to pass through the D module for decryption. Once the program is compiled, its execution can be outsourced to an untrusted party. The compiled executable contains a function evaluating `hNand`. Since the adversary may be able to build an efficient encrypter out of `hNand` the encryption has to be strong against CPA. Note that the adversary is not able to learn anything more about the encryption from the built encrypter than to learn from `hNand`.

D. HOMOMORPHIC SECURITY

1) ENCRYPTION PRIMITIVES

Section IV describes the method for generating encryption and decryption modules. We construct an `HG` `hNand` by selecting the following ingredients:

1) the encryption formula;

[5]All generations are done by our `Circle` tool. Different seeds would produce different results.

2) ciphertext size, λ;
3) digest function H; and
4) random sequence and its seed, R.

The encryption formula defines the complexity of the encryption, namely the length of the sequence of L and F operations and their type, as described in Eqs. 3. In our setup by default we use formula `FLF` which means that D and E modules have two non-linear and one linear operations as described in Section IV. Basically if the formula is complex enough, i.e., given the pairs of input and output bits no probabilistic correlation can be found, then the black-box access to modules E or D would not help to recover function D. Although black box access leaks some information about function D, with sufficiently big λ it does not leak enough to recover the function. We have empirically tested the diffusion property of the generated encryption to make sure that the there is no statistical bias in ciphertext bits. In our implementation, the random matrix A (for L) is filled with 50 percent probabilities of zeros and ones, which ensures that the probability of dependency of one arbitrary input bit and one arbitrary output bit is exactly $1/2$.

The ciphertext size λ is the security parameter controlling the strength of the encryption with respect to brute-force attacks. Digest function H generates the random part r for re-encryption of the result of `HG`. This function is required to mix bits without statistical bias. We choose to generate H (by `Circle`) in the same random way as D and E, with the difference that it is generated for size 2λ (i.e., for the size of two ciphertexts) and only the first $\lambda - 1$ (size of the random noise) outputs are used for r. The random sequence R, which should be seeded by a passphrase only known to the user, is used for generating specific instances of D, E, and H functions and their modules.

We define the encryption scheme as the following set of encryption primitives:

- Private: D, E, H, R;
- Public: λ, `hNand`.

As defined before, `hNand` is BDD-processed `HG` NAND, constructed according to Eq. 1. Element D must be protected as its function decrypts any ciphertext. Element E has to be protected because it gives the adversary access to r, c pairs which, as shown in Section VIII-C, gives advantage in SAT attack. Element H has to be protected because of malleability attacks explored in Section V-E. Finally, R must be protected since the adversary given the same algorithms implemented in software can reproduce exactly the same generation of D.

2) SECURITY OF BDD-PROCESSED CIRCUITS

For each Boolean function there is only one canonical representation as a reduced ordered Binary Decision Diagram [26]. A direct consequence of this result is that if two Boolean circuits have the same truth table (i.e., they implement the same function), their BDD representations are identical and independent of the sub-circuits and sub-modules within each

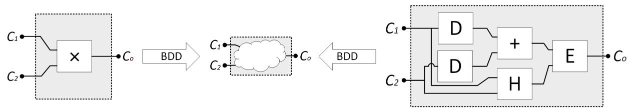


FIGURE 2. Schematic demonstration of BDD-processing of the Paillier modular multiplication operation (left) and its functional decomposed equivalent (right). When processed through BDD, both converge to the same circuit (center).

of the given circuits; in effect, there exists a unique circuit that corresponds to the BDD representation of that function.

Proposition: If black-box access to E is CPA secure and if there exists at least one secure homomorphic logic operation (e.g., NAND) with respect to E, then a BDD-processed `HG` *constructed by Eq. 1 for the same operation (e.g.,* `hNand`*) is also secure.*

Corollary: If a BDD-processed `HG` constructed by Eq. 1 is not secure given that E is CPA secure, then the construction of corresponding secure homomorphic logic operation is theoretically impossible.

Proof: First, let us note that with a universal gate (or a universal set of gates) such as NAND, it is possible to generate predefined encrypted values. For example, the construction of Eq. 4 is always encryption of 1 for any value of x. This means that the adversary is always able to construct a function E' and use CPA on E'. If `hNand` is secure, then E' can be analyzed by the adversary only as a black box. This means that we have to require black-box security for E. Next, let S be a secure `HG` and $\mathcal{B}$(`HG`) be a BDD-processed `HG`. The adversary can process S by BDD obtaining a new $\mathcal{B}(\mathrm{S})$. Since the functions of any `HG` representation are equivalent and since the BDD representation for the same order of BDD variables is canonical, then $\mathcal{B}$(`HG`) and $\mathcal{B}(\mathrm{S})$ have exactly the same form. Thus, if S is secure, then so is $\mathcal{B}(\mathrm{S})$, and therefore $\mathcal{B}$(`HG`) is also secure. ■

The proposition above supports the argument that the BDD-processed `HG` construction by Eq. 1 is secure with appropriate selection of E, D, H, and λ. The user is free to generate functions D and E as complex as possible. With sufficiently big λ the encryption is CPA secure. On the other hand, questioning the security of the BDD-processed `HG` is same as stating that no equivalent secure homomorphic operation is possible. The above proposition and its proof gives a better intuition on the security requirements: D and E have to be complex enough for bit statistical distribution (e.g., bit-flipping attack) and λ has to be big enough for resistance to brute-force attacks. Thus, canonical representation of the circuits using BDD processing is secure given these two requirements. Assuming CPA security of E and existence of secure homomorphic operation circuit, the above proposition proves the security of our method.

3) EXAMPLE OF PAILLIER SCHEME

To clarify the idea of canonical transformation, such as BDD-processing, we explicitly demonstrate it on the Paillier homomorphic scheme (Fig. 2). This scheme supports homomorphic addition operations and is proven to be secure. If we construct a circuit of the same operation using Eq. 1 and process it by a BDD engine, then, according to the above proposition, the BDD-processed circuit is also secure; i.e., the adversary is not able to get any advantage in breaking the cryptosystem when given a *BDD-processed construction of Eq. 1*. If this is not correct, then BDD-processing of the original secure operation would break the cryptosystem, which is impossible by definition.

Similar convergence occurs in arithmetic as shown in the following derivation. Let N be the Paillier encryption modulus. Selecting the generator as $1+N$ we get:

$$c = E(m, r) = r^N(1 + Nm)$$
$$D(c) = (c^\alpha - 1)/N = m$$

where r is the random part of encryption, $\alpha = \phi\phi_N^{-1}$, ϕ is Euler's function and ϕ_N^{-1} is its inverse in N. Here (and below) we imply the arithmetic modulo N^2, and the plaintext is always modulo N. We define the digest function as multiplication of the random parts of the encryption:

$$H(c_1, c_2) = r_1r_2 = c_1c_1^{-\alpha}c_2c_2^{-\alpha}$$

Indices 1 and 2 are used for two operands. A construction analogous to Eq. 1 with addition operation on plaintexts reads:

$$\begin{aligned}&E(D(c_1) + D(c_2), H(c_1, c_2))\\&\quad= E((c_1^\alpha + c_2^\alpha - 2)/N, c_1c_1^{-\alpha}c_2c_2^{-\alpha})\\&\quad= c_1c_2c_1^{-\alpha}c_2^{-\alpha}(c_1^\alpha + c_2^\alpha - 1) = c_1c_2\end{aligned}$$

because $(c_1^\alpha + c_2^\alpha - 1) = c_1^\alpha c_2^\alpha$. This result shows how a construction of decryption, addition, and re-encryption transforms into a function which effectively hides the decryption.

E. SELECTION OF DIGEST FUNCTION

`hNand` is constructed using an independent digest function generated by `Circle` as a random operation L (as described in Section IV) on 2λ input, which is the full array of input bits to `hNand`. This selection is not arbitrary, as other naive choices may lead to breaking the scheme.

Since function D is an inverse to E, `hNand` can potentially use recovered random bits, blend them and use the result for re-encryption. Namely, $\mathrm{D}(c_1) = (m_1, r_1)$, $\mathrm{D}(c_2) = (m_2, r_2)$, and $c_{\text{out}} = \mathrm{E}(\mathrm{NAND}(m_1, m_2), \mathrm{H}(r_1, r_2))$. If function H is known to the adversary, then the adversary can construct specific combinations of input ciphertexts so that $\mathrm{H}(r_1, r_2)$ is known. The simplest case would be $\mathrm{H}(r_1, r_2) = r_1 \oplus r_2$, where $\oplus$ is a vectored XOR. Then the attacker submits the unknown ciphertext c as $\texttt{hNand}(c, c) = \mathrm{E}(m, 0)$. The result obviously falls into 2 possible values, effectively breaking the encryption scheme. Even permutations of r_1 and r_2 before XOR can not be considered secure because the order of permutations grows at best as the Landau permutation function, leaving countable numbers of possible values. Taking a subset of bits from r_1 and r_2, as an alternative solution, would introduce a statistical bias for the distribution of encrypted values. In this situation, the adversary's strategy is to reduce the entropy in the ciphertext.

Having a truly random function H of the same strength as D would not give an adversary any benefit. However, D (or E) is not a good solution since it has λ inputs, but H has $2\lambda - 2$ inputs. If it is used twice with subsequent blending, then the same attack strategy as above can be used. To avoid this, H has to be generated for all its inputs. And since H digests bits with the same complexity as D, there is no point to use r_1 and r_2, while c_1 and c_2 can be used instead.

Another security concern is the usage of the same H in different `HG`s. In our current setup only `hNand` is used for calculations. There is no real obstacle to use several BDD-processed `HG`s. However, if the same randomness is used for re-encryption of different logical operations, then the adversary can build an efficient decryption function. For example, if `hAnd` and `hXor` implement AND and XOR with the same digest function, then for an arbitrary ciphertext c: $c_a = \texttt{hAnd}(c, c)$ and $c_x = \texttt{hXor}(c, c)$; hence, $c_a = c_x$ when $\mathrm{D}(c) = 0$.

Section VIII explores the security of the proposed scheme experimentally.

VI. PROTECTING HIGH-LEVEL PROGRAMS

The ultimate goal of this research direction is to provide to a non-crypto-savvy programmer the toolchain to develop any application computing on encrypted data. To this end, we have developed a comprehensive framework that can be used with C++ source files. A detailed description of the framework is outside of the scope of this research paper. Instead, we provide a short summary of the toolchain and the steps required to compile a C++ program that can compute on encrypted values. During development, the programmer needs to perform the following steps:

1) Identify variables in the program to be protected, their types, and their bit sizes; and convert, if necessary, the execution flow into data oblivious code. The latter can potentially be a tedious process. While some applications and algorithms are naturally data-oblivious (e.g., AES, database queries, etc.), others will need considerable effort to be converted assuming worst-case execution scenarios.
2) Define the ciphertext size of the private types. As shown in the experimental results, the ciphertext size has a direct impact on program performance, and the programmer should adjust it to their threat model.

Listing 1 presented a ready-to-compile program with our framework. The header `circle.h` is included, so the secure types become available, and `int` variables have been replaced by `SecureInt`. The range of secure types and the security parameter have been explicitly defined in the configuration file, not shown here.

As illustrated in Fig. 3, software modules (a) automate the identification and conversion of program constants to their private counterparts, (b) generate the `SecureInt` class, and (c) output its definition into a header file to be included

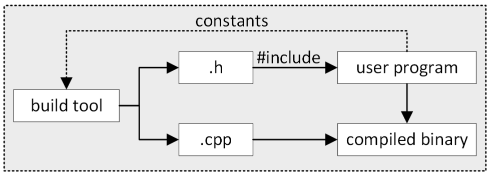


FIGURE 3. Automatic generation of C++ classes. The build tool generates secure type data classes (.h) and their operator implementations (.cpp), which are linked into the executable.

by the user program. The definitions of the class operators are written in the implementation file that is linked together with the user program into the final executable. In more detail, when the user compiles, our framework performs the following steps:

1) Generates decryption (D) and encryption (E) modules using the algorithm from Section IV-C.
2) Converts these modules into C/C++ functions.
3) Generates an `HG` for each supported operation and converts it into a C/C++ function.
4) Builds C++ classes for private data types.
5) Builds overloaded operators for the target program using the functions generated in step 3.
6) Converts constants and program input data to ciphertexts using the encryption function generated in step 2.

At the end of the process, the user receives a ready-to-execute binary, which can be executed on a non-trusted computational device. Once executed, the program outputs an encrypted result. The user can obtain the result and decrypt it using the D function generated in step 2.

A key component of our framework is `Circle` that automates the tasks associated with Boolean circuits (steps 1, 2, and 3). `Circle` (a) reads input files describing a logic circuit in the textual form of Boolean expressions, (b) performs a set of operations on the input circuit, and (c) outputs a processed circuit. One of these operations is the invocation of a BDD engine. In our framework, we employ a third-party BDD engine provided by the CUDD library [36]. In addition, `Circle` implements our algorithm for constructing balanced decryption and encryption circuits (Section IV-C), it supports conversions between truth tables and circuits, allows comparing two circuits, and can invert a circuit to its functional reverse.

VII. EXPERIMENTAL EVALUATION

A. SETUP

Our methodology can be applied to any universal set of operations. From the point of view of the construction and performance evaluation simplicity, we select only one Boolean logic operation NAND as the universal set. We construct `hNand` according to Eq. 1 and security recommendations from Section V. Now all computation logic can be expressed via appropriate connections between `hNand`s. As an efficiency improvement, we also construct `hNot` out of `hNand` by rewiring its inputs. It should be noted that the construction of `hNot` does not reveal any new information to the adversary, because in constructing `hNot` no private encryption primitives are used. After evaluating the performance of `hNot` and `hNand` as a function of security parameter (λ) in software and in FPGA, we perform a similar evaluation on circuits for common programming operations for different word sizes. Finally, we report performance of our methodology as compared to the state-of-the-art libraries for several benchmarks.

Our experiments are performed on a desktop computer equipped with Intel i7-4790 3.60GHz processor with 16 GB RAM running Ubuntu 18.04.2, using GCC 7.3.0 C/C++ compiler. The circuits are synthesized using Xilinx Vivado 2018.3 for the xc7a100tcsg324-3 board of Artix-7 family. The TFHE library commit 3319e2c [29] is used for comparison. As a baseline for the security parameter λ we select value 80. In Section VIII we justify this selection by showing the effectiveness of different attacks.

B. HOMOMORPHIC GATES IN C++ PROGRAMS

Having BDD-processed `HG`s expressed as functions gives the opportunity to run encrypted programs on ordinary computers without requirements of specific hardware. The program enhanced by privacy-preserving data types would not have any parts written outside standard C++.

To explore the practicality of the BDD-processed `HG`s, we performed experiments to determine the trend of evaluation time of the smallest `HG`, `hNot`, and the universal `HG`, `hNand`, for ciphertext size 4 – 100. For a particular size of ciphertext λ, we compiled the circuits for several values of seeds and we plotted the average execution time required by the gates as a function of increasing ciphertext size in Fig. 4. As expected, higher security parameters increase the performance overhead of BDD-processed `HG`s. Still, even for $\lambda = 80$, we observed the time required to execute `hNand` and `hNot` is 2.51×10^{-5} and 5.36×10^{-6} seconds respectively, thus facilitating the use of our methodology in ordinary computers. As it will be further explored in Section VII-E.2, our solution is 3-4 orders of magnitude faster compared to the fastest FHE library.

C. HOMOMORPHIC GATES IN FPGA

Since FPGAs operate natively on Boolean circuits, we further investigated the possibility of such acceleration. We performed the same experiments executing `hNand` and `hNot` on hardware for ciphertext size varying from 4 – 100. BDD-processed `HG`s and their corresponding Verilog codes were generated using `Circle`, and their circuits were synthesized with timing constraints. We consider the post-synthesis time as reported by Vivado as an estimate of the execution time for the BDD-processed `HG`s. Fig. 5 shows the performance of `hNand` and `hNot` gates synthesized for FPGA as a function of λ. The execution time required for $\lambda = 80$ is 4.5×10^{-8} and 2.1×10^{-8} seconds for `hNand` and `hNot` respectively. Therefore, by FPGA acceleration, we were able to improve performance of BDD-processed `HG`s by almost 3 orders of magnitude. This can be attributed to the fact that each operation in software is executed in sequence,

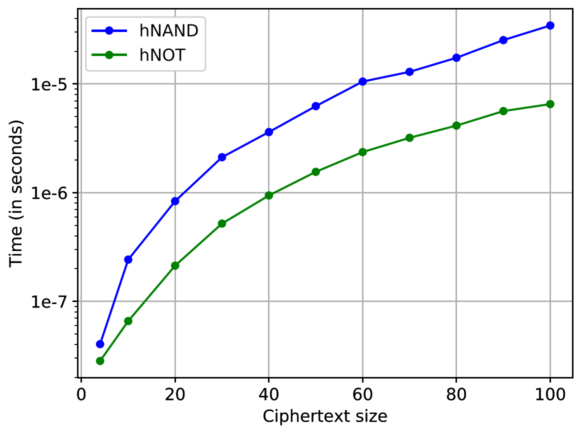


FIGURE 4. Execution time of `hNot` and `hNand` implemented in software.

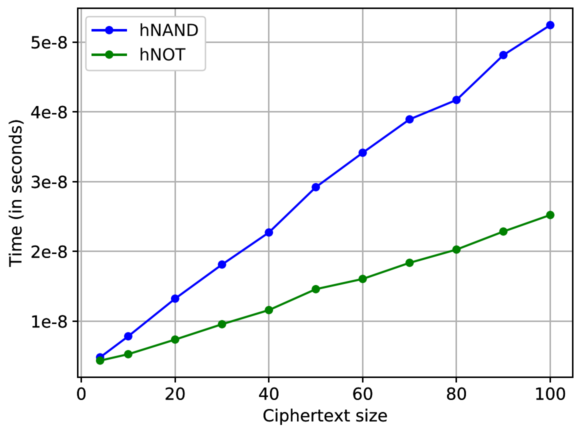


FIGURE 5. Execution time of `hNot` and `hNand` synthesized for an FPGA.

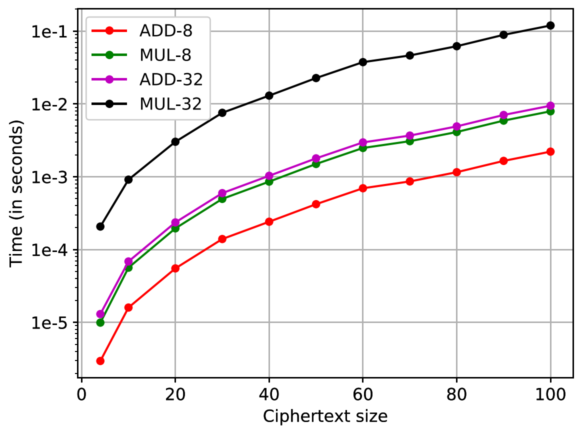


FIGURE 6. Execution time of ADD and MUL for 8 and 32 bit word size in software.

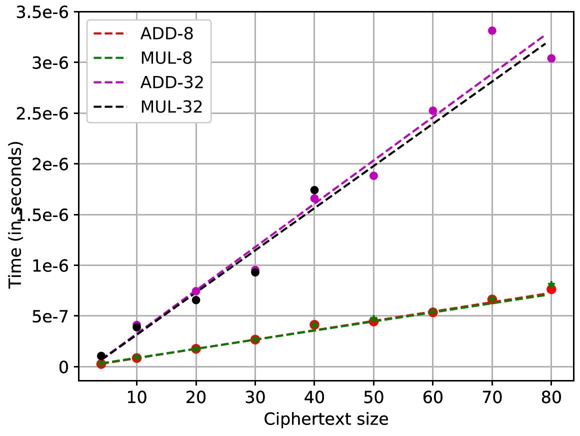


FIGURE 7. Execution time of ADD and MUL for 8 and 32 bit word size synthesized for an FPGA. Dashed lines show 2-parameter interpolations.

so the overhead is correlated to the size of the BDD-processed HG, while in hardware there is intrinsic parallelism, and the overhead is correlated to the depth of the circuit.

Another observation is that when comparing the rate of execution time increase between BDD-processed HGs in C++ and FPGA, we notice that the performance penalty for higher security parameters is much less for FPGAs. This is expected, since the circuit depth does not grow linearly, and for bigger circuits more parallelism can be exploited.

D. CIRCUITS OF PROGRAMMING OPERATIONS

Regular C++ programs do not operate on bits, but typically use arithmetic operations on integers. Thus, in this subsection, we report the experiments performed on homomorphic equivalents of addition (ADD) and multiplication (MUL), for 8-bit and 32-bit word sizes for each homomorphic circuit.

We evaluate the ADD and MUL circuits for λ 4-100. The performance of these circuits is shown in Fig. 6. The experiments demonstrate that for $\lambda = 80$ and 8-bit word size, the execution times required for ADD and MUL are 1.63×10^{-3} and 5.83×10^{-3} seconds respectively. Similarly, for 32 bit word size, the execution times for ADD and MUL are 6.95×10^{-3} and 8.78×10^{-2} seconds respectively.

As with evaluation of gates in FPGA, the performance of the programming operations can be improved using FPGAs. We synthesize the circuits for 8-bit and 32-bit word size. Post-synthesis timing, as estimated by Vivado, is reported in Fig. 7. For 32-bit MUL circuits, Vivado runs out of memory and is not able to synthesize circuits for λ above 40 bits. For 8-bit word size, the execution time for both ADD and MUL is approximately 7×10^{-7} seconds. Extrapolating the performance graphs for circuits with 32-bit word size, we observe that the execution time of ADD and MUL for $\lambda = 80$ is approximately 3.2×10^{-6} seconds. The results presented in this section further prove that the performance of homomorphic circuits can also be improved by 3 orders of magnitude with FPGA acceleration.

We have also evaluated the performance of division (DIV) as it is the most complex C++ arithmetic operation. Our results show that, in software, DIV behaves similarly to MUL, with a 20% performance degradation. On our FPGA, DIV is 4 times slower than MUL. The trends of degradation for different ciphertext sizes are very similar, so we refrain from plotting DIV to avoid cluttering the figures.

TABLE 2. Runtime in seconds for the TERMinator Suite: This work vs TFHE.

8 bits								
	bsort	**fact**	**fib**	**isort**	**jen**	**matrix**	**sieve**	**speck**
This work	0.0249	0.0724	0.0429	0.00994	0.0195	0.157	0.0695	0.185
TFHE	119	284	139	47.6	84.5	632	274	787

32 bits								
	bsort	**fact**	**fib**	**isort**	**jen**	**matrix**	**sieve**	**speck**
This work	0.101	0.797	0.171	0.0404	0.112	1.98	0.294	1.18
TFHE	512	3554	625	205	547	9050	1215	5236

E. COMPARISON TO STATE-OF-THE-ART

1) HELIB

Comparing the performance of our scheme to state-of-the-art FHE is not straightforward. One of the widely accepted standard FHE libraries, HElib [37], provides an API for exploration of FHE capabilities. For general-purpose computations, HElib can be used via logic gates as we do in our framework. Unfortunately, this usage of HElib is impractical. To use HElib at its maximum potential, the programmer has to implement higher order operation primitives by optimizing computation. This involves batching, parallelism, circuit static analysis, and possibly developing ancillary algorithms. There is an ongoing effort for streamlining and automating these procedures to the extent possible.

In order to make a fair comparison between our work and HElib in terms of performance, we use AES as a benchmark, as a publication exists where the authors have meticulously optimized AES in order to showcase the potential of HElib [38]. In that work, the authors use AES-128 as a circuit. While our framework allows the use of a C++ AES implementation directly instead of a circuit, the high-level synthesis would add performance penalty; since [38] uses AES as a circuit, we do the same. Thus, we use an AES-128 circuit from OpenCores.[6] Afterwards, we synthesize this circuit to include only NAND and NOT gates, and we convert it into a C++ function, which relies on calls to our `hNand` and `hNot`.

The execution time of one AES encryption using our methodology for $\lambda = 80$ is 4.75 seconds. In contrast, the non-bootstrappable implementation in [38] takes 245 seconds to perform the same operation, while the bootstrappable implementation takes 1050 seconds. It should be emphasized that we use our methodology out of the box; we do not implement batching, thread-level parallelism, or manual optimization on the elements of the AES algorithm. Instead, we rely on the Verilog circuit compiler optimizer. This offers a usability advantage over HElib. Future work will explore further performance improvements using batching and parallelism.

2) TFHE

Another state-of-the-art FHE library, TFHE [29], does actually provide an API to Boolean logic gates. Hence, comparison of our approach with TFHE is natural. We compare the performance of our framework with TFHE for two reasons: First, to the best of our knowledge, TFHE is the fastest FHE library that can be used as the underlying engine for general-purpose computations. Second, TFHE provides homomorphic gate access in the same way as in our work, that makes comparison fair – batching, parallel execution, or manual preprocessing are outside of the comparison.

For comparison on general-purpose computation, we use a set of data-oblivious benchmarks from the TERMinator Suite [39] that are grouped into three categories:

1) Basic - heavily based on arithmetic and logical operations: Bubble Sort (bsort), Insertion Sort (isort), Matrix Multiplication (matrix), and Sieve of Erastothenes (sieve);
2) Encoder - implementing bitwise-intensive cryptographic and hash applications: Jenkins (jen) and Speck Cipher (speck); and
3) Microbenchmarks: the addition-intensive Fibonnaci (fib) and the multiplication-intensive Factorial (fact).

Table 2 presents the execution time (in seconds) for the benchmarks running on 8-bit and 32-bit SecureInt variables with $\lambda = 80$ against TFHE. As the results demonstrate, the proposed methodology is 3 to 4 orders of magnitude faster than TFHE, irrespective of the type of benchmark used or the word size of the variables.

In terms of storage and memory requirements, our methodology expands each bit to λ bits. TFHE, on the other hand, represents each bit as a 2KB ciphertext. Furthermore, with regards to the evaluation functions that are added to the binaries, TFHE requires approximately 78MB of storage. Our methodology, assuming the presence of only `hNand` and `hNot`, requires 1.5MB for $\lambda = 80$.

VIII. EXPERIMENTAL SECURITY ANALYSIS

A. ANALYSIS OF INVARIANT VARIABLES

Here we explore the ability of the BDD representation to obfuscate the processed function. Specifically, since the original circuit is constructed with distinct decryption and encryption blocks as submodules, we inspect whether any of the internal wires of the `hNand` explicitly reveals any of the decrypted bits of any of the two operands. In order to test for leakage, we consider the following algorithm: For the first input, we enumerate all possible values. For the second input, we enumerate all possible encryptions of a specific plaintext value (in this example, encryptions of 1). Should a decrypted bit appear in any wire, it should remain constant for all possible iterations of the above algorithm, since the second decrypted input is always the same plaintext value.

[6] https://github.com/jeremysalwen/combinatorial_aes

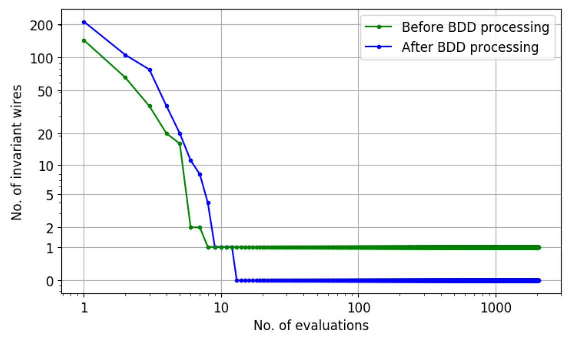


FIGURE 8. Number of invariants (Y-axis) vs number of evaluations (X-axis) for constructed `HG` before and after BDD-processing (`hNand`). The number of `hNand`'s invariant variables reaches zero at the 13th iteration.

We extend the `Circle` tool to support the presented algorithm and monitor all wires, reporting if any remains constant. We have analyzed the invariants for different values of λ and confirmed no invariants remain for all λs tested. As an example, we demonstrate a circuit NAND `HG` (as Fig. 1, a) with $|c| = 6$. For this circuit, which has 12 input wires, we generate a list of all possible combinations of the first input (2^6), and all possible encryptions of 1 for the second argument (2^5). Therefore we have $2^6 \cdot 2^5 = 2048$ inputs.

Fig. 8 presents the invariant counts for all 2048 iterations (in random order). Before we process the circuit using our BDD engine, the invariant analyzer identifies an invariant at the end of the 2048 rounds, and manual inspection reveals that the decrypted '1' value clearly appears in the internal wire. On the contrary, after BDD processing (blue line of Fig. 8), no internal value (wire) remains constant at the end of 2048 iterations. Indeed, after 13 iterations, as denoted by the arrow of Fig. 8, no invariants remain. The lack of constant values after all iterations indicates that the decrypted value (or any of its bits) never appears as plaintext in the `hNand`.

While the decrypted value does not appear in any of variables inside the circuit, we also investigated potential statistical bias of a wire towards a decrypted value, even though it is not identified as invariant. Using `Circle` we have explored the statistics of the values of the internal circuit variables, but could not find any correlation that points to a bias.

B. BRUTE-FORCE ATTACK

As mentioned in earlier sections, the `hNand` function is included in the program (either binary or source code), and thus can be identified by an attacker statically or dynamically. We assume that the attacker can locate these processing functions and can feed inputs to them to observe the output. The attacker can generate an encryption module, with the help of which the decrypted plaintext can be recovered. We call this *Constructed Encrypter attack*. Fig. 9 shows an example of such an encrypter circuit. The construction `hNand`$(x,$ `hNand`$(x, x))$ is always $\tilde{1}$ for any value of x (Eq. 4). Considering x as random noise to the encryption, the encrypted value can be obtained out of the inverted $\tilde{1}$ (i.e., $\tilde{0}$ = `hNand`$(\tilde{1}, \tilde{1})$) and a multiplexer selecting either

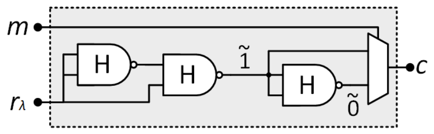


FIGURE 9. An encrypter constructed from three `hNand`s and a multiplexer. r_λ is a random ciphertext of size λ, *m* is a plaintext, *c* is an encryption of *m*, $\tilde{0}$ and $\tilde{1}$ are encrypted values of 0 and 1.

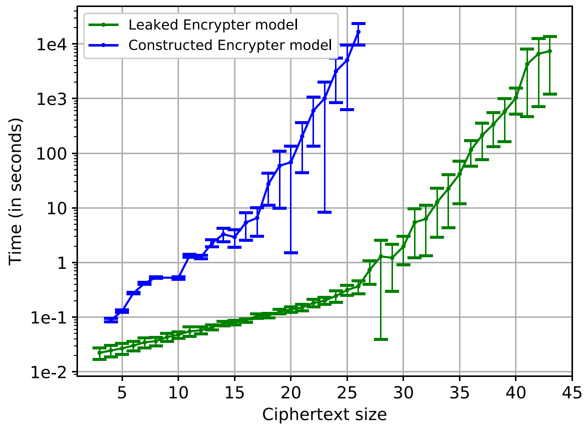


FIGURE 10. The time required to decrypt a ciphertext using SAT-attack, as a function of ciphertext size. The experiments are performed on leaked encrypter model as well as constructed encrypter model for `hNand`.

$\tilde{1}$ or $\tilde{0}$. In this way, different encryptions of bits can be generated.

The memory requirements for a brute force attack can be estimated by assuming the adversary is able to generate a comprehensive truth table by constructing an encrypter out of `hNand`. If the ciphertext is the address of a single bit, then a 13-bit ciphertext requires 1kB of memory, a 43-bit requires 1TB, and a 80-bit requires more than 10^{11}TB. On the other hand, from sequential evaluation of the constructed encrypter, we estimate the time required to run all possible combinations: $2^{1.06\lambda-18}$s, which results in $\approx 10^{20}$ seconds for $\lambda = 80$. Comparing the evaluation time of AES-128 on the same computer, we roughly estimate that $1.08\lambda - 2.2$ is the number of AES bits that have the same strength as λ bits with regards to brute-force attacks.

C. SAT ATTACK

Boolean Satisfiability attacks (SAT-attacks) have been proven to be effective in breaking logic encryption [40] and camouflaging schemes [41]. SAT-attacks can be used to decrypt ciphertexts by setting the desired ciphertext to the outputs of the encrypter and ask a SAT-solver to generate the inputs. In this section, we quantify the time needed for a SAT-solver to decrypt a ciphertext given the constructed encrypter and different sizes of random noise bits, and report the time complexity.

To find the time complexity of breaking the scheme as a function of the ciphertext size, we perform SAT attacks using `Z3` [42] on a 2.00GHz Intel Xeon CPU machine equipped with 64 GB of RAM and 16 CPU cores running CentOS 6.10. Fig. 10 shows the results for our Constructed Encrypter

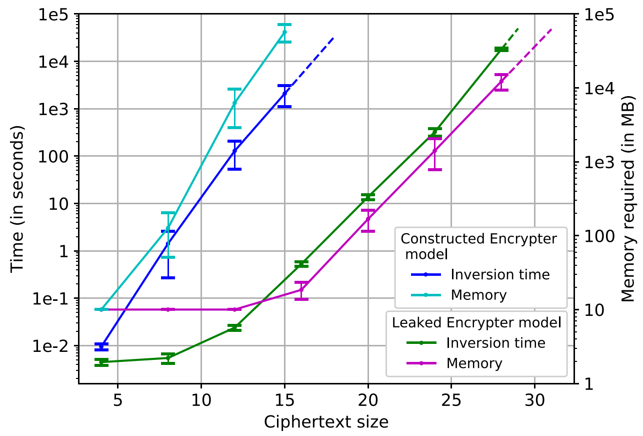


FIGURE 11. The time and memory required for an encrypter circuit to get inverted. The experiments are performed on leaked encrypter model as well as constructed encrypter model for `hNand`**.**

attack for ciphertext sizes up to 26 bits. From the results, we observe that the time needed for the SAT-solver to decrypt a value increases exponentially with the ciphertext size. This enables selecting the desired security strength, considering that a higher number of noise bits requires longer processing times. Extrapolating the results to 80 bits of ciphertext, we find the best-fit functions in the form $\exp(ax+b)$ derived from our experimental values, and report that the SAT solver would need roughly 10^{14} years to decrypt, which we consider intractable.

To better understand the resilience of our construction to SAT-attacks, we also consider a stronger attack model, where the attacker is assumed to have access to the BDD-processed original encryption circuit: we call this scenario the *Leaked Encrypter attack*. In other words the 'leaked encrypter' is the original encrypter separated from the rest of the homomorphic gate and processed by BDD. It should be noted that this attack contradicts our attack model outlined in Section III-B and in this scenario, the adversary is assumed to have acquired a part of the private key. Since the complexity of the circuit is significantly reduced, `Z3` requires less time to solve for higher ciphertext sizes compared to the previous attack model. However, as shown in Fig. 10, the complexity is still exponential. Fig 10 depicts the time needed by an adversary to decrypt ciphertexts up to 43 bits. Extrapolating again to 80 bits of ciphertext size, the SAT solver would need roughly 10^7 years to decrypt.

D. INVERSE CIRCUIT ATTACK

Both brute-force and SAT attacks can be used to decrypt ciphertexts, but cannot be used to recover the decryption function. In this section, we explore the possibility to break the encryption by inverting the constructed encrypter (Fig. 11) using the circuit reversing function of `Circle`. The algorithm implemented in the tool is briefly described below.

Any state of a combinational circuit can be defined by the values of the minimal *set* of wires necessary to calculate the output. Initially, it is the input wires. Then the set is updated: The value of a new wire is computed by a logic gate which is added to the set. At the same time the wires that are no longer to be used are excluded from the set. Let function G describe the forbidden states of the set of wires. Starting from $G_0 = 0$, each computation of a new wire value and each exclusion of a wire value form a sequence of functions G_i by the rules: $G^+_{i+1} = \text{OR}(\text{XOR}(f(x, y), z), G_i)$ when z is created with a gate f out of x and y; and $G^-_{i+1} = \text{AND}(G_i(z = 0), G_i(z = 1))$, when excluding z. These rules ensure the property of G in its definition.

In the second rule (G^-_{i+1}) function G_i has all the information about z. Therefore, z can be calculated as $z = G_i(z = 0)$ or $z = \text{NOT}(G_i(z = 1))$. Both expressions give the same result if forward computation of z is injective. Otherwise, two different values of z are both valid and either expression can be used for reconstructing the previous state of the variable set. `Circle` calculates the sequence of G_i, and working backward computes one by one all excluding variables until the input ones. This computation is symbolic and results in constructing a circuit having the inverse functionality to the original circuit. We performed experiments for λ from 4 to 30 for constructed encrypter model and leaked encrypter model. The experiments were performed on a 3.33 GHz Intel Xeon CPU with 96GB of RAM and 24 CPU cores. Fig. 11 shows the results of inverting a constructed and a leaked encrypter in terms of time and memory required. The memory required for the constructed encrypter model increased very quickly and was completely exhausted after $\lambda = 16$. Thus, the small amount of data can only give a rough estimate of the time required to invert a circuit for $\lambda = 80$ which was found to be 10^{26} years, based on extrapolation from the current data. For the leaked encrypter model, the memory was not exhausted but the time required to invert the circuit increased exponentially. From extrapolation of the best fit curve, we find that the time to invert the circuit is 10^{15} years.

E. SECURITY SUMMARY

The security of the proposed method resides in a) the definitions of the encryption scheme and the attack model; b) the algorithms for constructing the encryption primitives; and c) finding the limits for breaking the encryption in our attempts using different methods. It is easy to see that private elements D and E are black-box secure given enough complexity in both λ parameter and the function formula. We prove the security of BDD-processed circuit under the assumption that at least one example of secure homomorphic scheme exists for defined D and E.

Conceptual analysis reveals insights about the security parameter λ. Our experiments to break the encryption using a SAT solver [42] demonstrate impracticality for the attack for λ values above 30 because of the time complexity. Memory required to perform inverse circuit attack becomes prohibitive for λ values more than 20. Attack methods based on precooking re-encryption randomness clarify limitations on selecting secure digest functions as explained in section V-E. Since the choice of security parameter (λ) is crucial to performance,

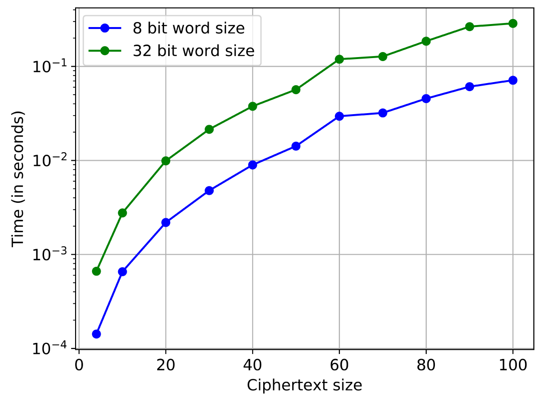


FIGURE 12. Fibonacci running on 8-bit and 32-bit SecureInt variables for different ciphertext sizes.

the user can select the trade-off between security and performance overhead given their threat model. In the case of mass data leakage through microarchitectural vulnerabilities or software/OS bugs, smaller λ values would add enough burden to adversaries towards deciphering values, but the application would incur minimal performance overhead. Fig. 12 shows the trade-off between λ values and performance, for different protected type ranges. In case nation-states are in the threat model, then λ values more than 80 would be needed.

IX. CONCLUDING REMARKS

In this paper, we present a methodology that enables private computation based on composed homomorphic operations leveraging the properties offered by the BDD transformation of Boolean functions. To automate circuit manipulations, we develop `Circle`, which assists the development of linear and non-linear decryption and probabilistic encryption functions using our novel algorithm, as well as building circuits for `HG`s. Using our developed class of private integers and overloaded operators, end-users can effortlessly manipulate sensitive variables in C++ programs.

We executed experiments measuring performance on different levels of computation, starting from the gate operation to full C++ programs. Our framework can execute data-oblivious benchmarks 3 to 4 orders of magnitude faster compared to the fastest state-of-the-art FHE schemes (Table 2).

We also provided arguments supporting the security of the proposed scheme as well as recommendations on the construction and usage of the encryption components. Furthermore, we performed experiments on the security of our homomorphic construction demonstrating exponential growth of difficulty to break the encryption given the increase of the security parameter. Using the best-known algorithm for breaking similar schemes, ciphertexts of 80-bit size require millions of years to be broken.

ACKNOWLEDGEMENT

The work of Esha Sarkar was supported by the NYU Abu Dhabi Global Ph.D. Fellowship.

REFERENCES

[1] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage, "Hey, you, get off of my cloud: Exploring information leakage in third-party compute clouds," in *Proc. 16th ACM Conf. Comput. Commun. Secur.*, 2009, pp. 199–212.

[2] D. Lyon, "Surveillance, Snowden, and big data: Capacities, consequences, critique," *Big Data Soc.*, vol. 1, no. 2, Jul. 2014, Art. no. 205395171454186.

[3] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2019, pp. 1–19.

[4] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown: Reading kernel memory from user space," in *Proc. 27th USENIX Secur. Symp. USENIX Secur.*, 2018, pp. 973–990.

[5] N. G. Tsoutsos and M. Maniatakos, "Trust no one: Thwarting' heartbleed' attacks using privacy-preserving computation," in *Proc. IEEE Comput. Soc. Annu. Symp. VLSI*, Jul. 2014, pp. 59–64.

[6] L. Bilge and T. Dumitraş, "Before we knew it: An empirical study of zero-day attacks in the real world," in *Proc. ACM Conf. Comput. Commun. Secur.*, 2012, pp. 833–844.

[7] G. Chen, S. Chen, Y. Xiao, Y. Zhang, Z. Lin, and T. H. Lai, "SgxPectre attacks: Stealing intel secrets from SGX enclaves via speculative execution," 2018, *arXiv:1802.09085*. [Online]. Available: https://arxiv.org/abs/1802.09085

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

[9] A. Acar, H. Aksu, A. S. Uluagac, and M. Conti, "A survey on homomorphic encryption schemes: Theory and implementation," *ACM Comput. Surv.*, vol. 51, no. 4, p. 79, Jul. 2018, doi: 10.1145/3214303.

[10] P. Paillier, "Public-key cryptosystems based on composite degree residuosity classes," in *Advances in Cryptology*. Berlin, Germany: Springer, 1999, pp. 223–238.

[11] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Commun. ACM*, vol. 26, no. 1, pp. 96–99, Jan. 1983, doi: 10.1145/357980.358017.

[12] T. ElGamal, "A public key cryptosystem and a signature scheme based on discrete logarithms," in *Advances in Cryptology (CRYPTO)* (Lecture Notes in Computer Science), vol. 196, G. R. Blakley and D. Chaum, Eds. Berlin, Germany: Springer, 1985, pp. 10–18.

[13] I. Damgård, M. Jurik, and J. B. Nielsen, "A generalization of Paillier's public-key system with applications to electronic voting," *Int. J. Inf. Secur.*, vol. 9, no. 6, pp. 371–385, 2010.

[14] D. Boneh, E.-J. Goh, and K. Nissim, "Evaluating 2-DNF formulas on ciphertexts," in *Theory of Cryptography*, Berlin, Germany: Springer, 2005, pp. 325–341.

[15] P. S. Pisa, M. Abdalla, and O. C. M. B. Duarte, "Somewhat homomorphic encryption scheme for arithmetic operations on large integers," in *Proc. Global Inf. Infrastruct. Netw. Symp. (GIIS)*, Dec. 2012, pp. 1–8.

[16] C. Gentry, "Fully homomorphic encryption using ideal lattices," in *Proc. 41st Annu. ACM Symp. Symp. Theory Comput.*, 2009, pp. 169–178.

[17] N. P. Smart and F. Vercauteren, "Fully homomorphic encryption with relatively small key and ciphertext sizes," in *Public Key Cryptography* Berlin, Germany: Springer, 2010, pp. 420–443.

[18] M. Van Dijk, C. Gentry, S. Halevi, and V. Vaikuntanathan, "Fully homomorphic encryption over the integers," in *Proc. 29th Annu. Int. Conf. Theory Appl. Cryptograph. Techn.* (Lecture Notes in Computer Science), vol. 6110. Springer, May/Jun. 2010, pp. 24–43.

[19] C. Gentry, A. Sahai, and B. Waters, "Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based," in *Advances in Cryptology-(CRYPTO)* (Lecture Notes in Computer Science), vol. 8042, R. Canetti and J. A. Garay, Eds. Berlin, Germany: Springer, 2013, pp. 75–92.

[20] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, "(Leveled) fully homomorphic encryption without bootstrapping," in *Proc. Innov. Theor. Comput. Sci. Conf.*, 2012, pp. 309–325.

[21] M. Varia, S. Yakoubov, and Y. Yang, "HEtest: A homomorphic encryption testing framework," in *Financial Cryptography and Data Security*, vol. 1326. New York, NY, USA: Springer, 2015, pp. 213–230.

[22] T. Lepoint and M. Naehrig, "A comparison of the homomorphic encryption schemes FV and YASHE," in *Proc. 7th Int. Conf. Cryptol. Afr. Prog. Cryptol. (AFRICACRYPT)*, in Lecture Notes in Computer Science, Marrakesh, Morocco, vol. 8469. Cham, Switzerland: Springer, May 2014, pp. 318–335.

[23] L. Kruger, S. Jha, E.-J. Goh, and D. Boneh, "Secure function evaluation with ordered binary decision diagrams," in *Proc. 13th ACM Conf. Comput. Commun. Secur.*, 2006, pp. 410–420, doi: 10.1145/1180405.1180455.
[24] H. Kim, S. Hong, B. Preneel, and I. Verbauwhede, "Binary decision diagram to design balanced secure logic styles," in *Proc. IEEE 22nd Int. Symp. On-Line Test. Robust Syst. Design (IOLTS)*, Jul. 2016, pp. 239–244.
[25] S. B. Akers, "Binary decision diagrams," *IEEE Trans. Comput.*, vol. C-27, no. 6, pp. 509–516, Jun. 1978.
[26] Bryant, "Graph-based algorithms for Boolean function manipulation," *IEEE Trans. Comput.*, vol. C-35, no. 8, pp. 677–691, Aug. 1986.
[27] M. Yasin, A. Sengupta, M. T. Nabeel, M. Ashraf, J. J. Rajendran, and O. Sinanoglu, "Provably-secure logic locking: From theory to practice," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, New York, NY, USA, 2017, pp. 1601–1618, doi: 10.1145/3133956.3133985.
[28] M. Li, K. Shamsi, T. Meade, Z. Zhao, B. Yu, Y. Jin, and D. Z. Pan, "Provably secure camouflaging strategy for IC protection," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 38, no. 8, pp. 1399–1412, Aug. 2019.
[29] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachéne. (Mar. 2018). *TFHE: Fast Fully Homomorphic Encryption Library*. [Online]. Available: https://tfhe.github.io/tfhe/
[30] A. Sahai and B. Waters, "Fuzzy identity-based encryption," in *Proc. 24th Annu. Int. Conf. Theory Appl. Cryptograph. Techn.* Berlin, Germany: Springer-Verlag, 2005, pp. 457–473, doi: 10.1007/11426639_27.
[31] J. Yu, L. Hsiung, M. El Hajj, and C. W. Fletcher, "Data Oblivious ISA Extensions for Side Channel-Resistant and High Performance Computing," *IACR Cryptol. ePrint Arch.*, vol. 2018, p. 808, 2018.
[32] S. Goldwasser and S. Micali, "Probabilistic encryption," *J. Comput. Syst. Sci.*, vol. 28, no. 2, pp. 270–299, Apr. 1984.
[33] S. Amoroso and Y. Patt, "Decision procedures for surjectivity and injectivity of parallel maps for tessellation structures," *J. Comput. Syst. Sci.*, vol. 6, no. 5, pp. 448–464, Oct. 1972. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0022000072800138
[34] S. Capobianco, J. Kari, and S. Taati, "Post-surjectivity and balancedness of cellular automata over groups," *Discrete Math. Theor. Comput. Sci.*, vol. 19 no. 3, Sep. 2017. [Online]. Available: https://dmtcs.episciences.org/3918, doi: 10.23638/DMTCS-19-3-4.
[35] L. Goubin *et al.*, "How to reveal the secrets of an obscure white-box implementation," *J. Cryptograph. Eng.*, 2019, doi: 10.1007/s13389-019-00207-5.
[36] F. Somenzi, "CUDD: Cu decision diagram package release," Dept. Elect., Comput., Energy Eng., Univ. Colorado Boulder, Boulder, CO, USA, Tech. Rep., Dec. 2015. [Online]. Available: https://pdfs.semanticscholar.org/cb3c/a92ebe93b1076aef5fcd6c8f215a06694424.pdf
[37] S. Halevi and V. Shoup, "Bootstrapping for HElib," in *Advances in Cryptology*, E. Oswald and M. Fischlin, Eds. Berlin, Germany: Springer, 2015, pp. 641–670.
[38] C. Gentry, S. Halevi, and N. P. Smart, "Homomorphic evaluation of the AES circuit," in *Proc. 32nd Annu. Cryptol. Conf. Adv. Cryptol. (CRYPTO)*, vol. 7417. Berlin, Germany: Springer-Verlag, 2012, pp. 850–867.
[39] D. Mouris, N. G. Tsoutsos, and M. Maniatakos, "TERMinator suite: Benchmarking privacy-preserving architectures," *IEEE Comput. Arch. Lett.*, vol. 17, no. 2, pp. 122–125, Jul. 2018.
[40] P. Subramanyan, S. Ray, and S. Malik, "Evaluating the security of logic encryption algorithms," in *Proc. IEEE Int. Symp. Hardw. Oriented Secur. Trust (HOST)*, May 2015, pp. 137–143.
[41] M. E. Massad, S. Garg, and M. V. Tripunitara, "Integrated circuit (IC) Decamouflaging: Reverse engineering camouflaged ICs within minutes," in *Proc. 22nd Annu. Netw. Distrib. Syst. Secur. Symp., (NDSS)*, San Diego, CA, USA, Feb. 2015, pp. 1–14.
[42] L. De Moura and N. Bjørner, "Z3: An efficient SMT solver," in *Proc. 14th Int. Conf. Tools Algorithms Construct. Anal. Syst.* Berlin, Germany: Springer-Verlag, 2008, pp. 337–340.

OLEG MAZONKA received the Ph.D. degree in theoretical physics from the National Center for Nuclear Research, Poland, in 2000. He worked as a Software Engineer and Architect, taught mathematics with a school, and lectured software engineering with the University of South Australia. He is currently a Research Associate with the Department of Electrical and Computer Engineering, New York University Abu Dhabi, UAE.

ESHA SARKAR (Student Member, IEEE) received the B.Tech. degree in electrical engineering from the National Institute of Technology, India, in 2012. She is currently pursuing the Ph.D. degree with the NYU Tandon School of Engineering. Her current research interests are in hardware security, security analysis of data-protection schemes, machine learning for vulnerability discovery, and industrial control system security.

EDUARDO CHIELLE received a jointly-supervised Ph.D. degree in microelectronics from the Universidade Federal do Rio Grande do Sul (UFRGS) and in informatics from the Universidad de Alicante (UA), the M.Sc. degree in computer science from UFRGS, and a five-year bachelor's degree in computer engineering from the Universidade Federal do Rio Grande (FURG). He placed first in the TTTC's McCluskey Best 2016 Latin American Ph.D. Thesis Contest for his work on microprocessor soft error mitigation. His interests include computer architecture, privacy-preserving computation, embedded devices, hardware design, and fault tolerance.

NEKTARIOS GEORGIOS TSOUTSOS (Member, IEEE) received the Ph.D. degree in computer science from New York University and the M.Sc. degree in computer engineering from Columbia University. He is currently an Assistant Professor with the Department of Electrical and Computer Engineering, University of Delaware, with a joint appointment in the Department of Computer and Information Sciences. His research interests are in cybersecurity and applied cryptography, with a special focus in hardware security, trustworthy computing, and privacy outsourcing. He holds a patent on encrypted computation using homomorphic encryption. He has authored multiple articles in the IEEE Transactions and conference proceedings, and serves in the program committee of several international conferences. He is also the organizer of the International Embedded Security Challenge (ESC) that is held annually during the Cyber Security Awareness Worldwide (CSAW) event.

MICHAIL MANIATAKOS (Senior Member, IEEE) received the B.Sc. and M.Sc. degrees in computer science and embedded systems from the University of Piraeus, Greece, and the Ph.D. degree in electrical engineering and the M.Sc. and M.Phil. degrees from Yale University, New Haven, CT, USA. He is currently an Associate Professor of electrical and computer engineering with New York University (NYU) Abu Dhabi, Abu Dhabi, UAE, and a Research Assistant Professor with the NYU Tandon School of Engineering, New York, NY, USA. He is also the Director of the MoMA Laboratory, NYU Abu Dhabi. His research interests, funded by industrial partners, the US Government, and the UAE Government, include robust microprocessor architectures, privacy-preserving computation, smart cities, as well as industrial control systems security. He has authored several publications in IEEE transactions and conferences, holds patents on privacy-preserving data processing, and also serves in the technical program committee for various international conferences.