arXiv:2204.12201v3 [cs.CR] 27 Jul 2022

Accelerating Fully Homomorphic Encryption by Bridging Modular and Bit-Level Arithmetic

Eduardo Chielle, Oleg Mazonka, Homer Gamil, and Michail Maniatakos
Center for Cyber Security, New York University Abu Dhabi, Abu Dhabi, UAE
{eduardo.chielle,om22,homer.g,michail.maniatakos}@nyu.edu

ABSTRACT

The dramatic increase of data breaches in modern computing platforms has emphasized that access control is not sufficient to protect sensitive user data. Recent advances in cryptography allow end-to-end processing of encrypted data without the need for decryption using Fully Homomorphic Encryption (FHE). Such computation however, is still orders of magnitude slower than direct (unencrypted) computation. Depending on the underlying cryptographic scheme, FHE schemes can work natively either at bit-level using Boolean circuits, or over integers using modular arithmetic. Operations on integers are limited to addition/subtraction and multiplication. On the other hand, bit-level arithmetic is much more comprehensive allowing more operations, such as comparison and division. While modular arithmetic can emulate bit-level computation, there is a significant cost in performance. In this work, we propose a novel method, dubbed *bridging*, that blends faster and restricted modular computation with slower and comprehensive bit-level computation, making them both usable within the same application and with the same cryptographic scheme instantiation. We introduce and open source C++ types representing the two distinct arithmetic modes, offering the possibility to convert from one to the other. Experimental results show that bridging modular and bit-level arithmetic computation can lead to 1-2 orders of magnitude performance improvement for tested synthetic benchmarks, as well as one real-world FHE application: a genotype imputation case study.

CCS CONCEPTS

• **Security and privacy → Domain-specific security and privacy architectures**.

KEYWORDS

fully homomorphic encryption, privacy-preserving computation

ACM Reference Format:
Eduardo Chielle, Oleg Mazonka, Homer Gamil, and Michail Maniatakos. 2022. Accelerating Fully Homomorphic Encryption by Bridging Modular and Bit-Level Arithmetic. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD '22), October 30-November 3, 2022, San Diego, CA, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3508352.3549415

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ICCAD '22, October 30-November 3, 2022, San Diego, CA, USA
© 2022 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-9217-4/22/10…$15.00
https://doi.org/10.1145/3508352.3549415

1 INTRODUCTION

With the ever increasing rates of data generation, digital information is becoming extremely valuable over time. As a result, certain types of attacks have emerged, focusing to capitalize on this paradigm shift. To overcome the aforementioned threats, the use of cryptography has been until now the defacto technological security measure to protect against data leakage. Even though accepted standards such as AES have been successful in protecting data-in-transit and data-at-rest, they fail to provide protection towards data-in-use. Hardware solutions [8, 26] attempt to provide confidentiality and integrity on sensitive computations, however, a number of attacks [1, 13, 16, 18, 19, 30, 32, 33] including Spectre [20], Meltdown [21], and Load Value Injection [31], have raised questions about their effectiveness. Data are eventually decrypted before entering the processor pipeline, and therefore leakage is still possible as recently reported [4].

A solution to the problem of protecting data-in-use is Fully Homomorphic Encryption (FHE), as it allows unconstrained computations on ciphertexts. Numerous FHE schemes have been developed, namely BGV [2], BFV [11], CKKS [5], GSW [12]. GSW exposes homomorphic Boolean gates and programming computation can be constructed bottom-up. Such bit-level arithmetic is universal, but requires many Boolean logic operations. Other schemes, like BGV/BFV, operate directly on integers using modular arithmetic supporting homomorphic addition, subtraction, and multiplication. At the level of integers however, the operations that can be applied on such ciphertexts are limited to those supported by the encryption scheme. If an application requires another type of operation like an integer division or comparison, then all computation must be evaluated in bit-level arithmetic, by using arithmetic modulo 2 or algebraic expressions for Boolean gates.

In 2009, FHE received criticism on its high computational overheads, which made it challenging at the time to employ in practical applications. Improvements have since been performed on various fronts: 1) Improvement of implementation of FHE libraries. For example, HElib was recently overhauled to achieve almost 75× performance improvement [15]; 2) Hardware acceleration. The developers of nuFHE [25] report about 100× acceleration over the software-based implementation of TFHE using GPU hardware. Meanwhile, F1, an ASIC accelerator for the BGV encryption scheme, outperforms software implementations by $5,400\times$ [27]; 3) Dedicated programming frameworks. These frameworks allow users to express programming intent directly in a general-purpose programming language, such as Go, Python, C++. For example, Lattigo [23] implements BFV and CKKS schemes in the Go programming language. PyFHE, PySEAL and Pyfhel [17] are frameworks providing Python interface to FHE operations. E3 framework [6] offers custom C++

data types that can abstract the complexity of FHE library operations. As we observe, native modular arithmetic must be used for logic bit operations; hence program variables must be represented as a sequences of encrypted bits. In such case, *all* programming operations are performed by evaluating Boolean circuits working on encrypted bits - ciphertexts. This transformation makes all computational operations slow, including addition and multiplication. **Our contribution**: In this work, our proposal is to leverage the properties of both (1) bit-level arithmetic (universal, but slower) and (2) modular arithmetic (faster, but not universal) within the same algorithm expressed as a C++ program with custom data types, which allows much better performance of general-purpose programs processing encrypted data. Towards that end, we extend the E3 programming framework [6] and implement *Bridging*, which enables the mixing of different arithmetic abstractions in the same C++ program that processes FHE data. The benefit of using our method is a significant application performance improvement, as we demonstrate with our experiments.

2 PRELIMINARIES

2.1 Homomorphic Encryption

Homomorphic encryption is a special type of encryption that allows meaningful operations in the encrypted domain. Within homomorphic encryption, there are several types of encryption schemes. Partially Homomorphic Encryption (PHE) schemes support only one homomorphic operation over ciphertexts, usually either addition or multiplication. FHE schemes support two orthogonal operations, usually addition and multiplication, theoretically allowing Turing complete computation.

Operations on ciphertexts without decryption are possible using homomorphic functions. Suppose the homomorphic function F over ciphertexts c_x and c_y corresponds to plaintext operation f. Let $f(m_x, m_y)$ denote a two-argument function on plaintexts m_x and m_y; $\mathrm{E}_k(m, r)$ denotes a probabilistic encryption function generated by a random sequence k (key) that maps a plaintext m to a set of ciphertexts c depending on a probabilistic parameter r, and $\mathrm{D}_k(c)$ denotes a deterministic decryption function corresponding to E_k that maps ciphertext c to the corresponding plaintext m. Then, F is defined by the homomorphism of surjective D_k:

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

which converts into the explicit composition over ciphertexts:

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

where H is an arbitrary function generating the randomness value r from the input ciphertexts. This expression establishes a functional requirement for F, which does not decrypt nor re-encrypt the data.

2.2 Data-oblivious programming

Data-oblivious computation is a type of computation where the input data does not influence the behavior of the program. Under "behavior" we imply execution branching or memory access. Operations performed on encrypted data have to be in a data-oblivious form since ciphertexts are never decrypted, therefore the program is not capable of making decisions based on their plaintext values. In some fields, this property is also termed as *constant-time programming*, and has also been used for protecting against side-channel analysis.

```
int fibonacci(int in)
{
    int i=0, a=0, b=1;
    while( i++ != in )
    {
        std::swap(a,b);
        a += b;
    }
    return a;
}
```

Listing 1: Simple Fibonacci function.

```
int fibonacci(int in)
{
    int i=0, a=0, b=1;
    int r=0;
    int max_iter = 10;
    while( max_iter-- )
    {
        r += (i++ == in) * a;
        std::swap(a,b);
        a += b;
    }
    return r;
}
```

Listing 2: Data-oblivious Fibonacci function.

Consider a simple function in Listing 1. The iterations are interrupted once the index `i` reaches the input value `in`. Assume that computation is now protected so the inputs and outputs must remain encrypted. Since variable `in` cannot participate in any decisions to interrupt iterations, it must be processed in a data-oblivious way. We replace the interrupt condition with a multiplexer accumulator `r` and introduce a fixed number of iterations `max_iter` not depending on the input, as shown in Listing 2. The number of iterations should exceed the input index, otherwise the accumulator will not reach the input index and will not get updated. The program shown in Listing 2 works in constant time regardless of the input.

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

2.3 FHE Batching

The FHE batching technique offers the ability to pack several plaintexts into "slots" within a single ciphertext. This feature is supported by some FHE schemes, and in practice enables parallel processing of plaintexts, which are part of the same ciphertext (in a Single Instruction Multiple Data (SIMD) style) [29]. Notably, such SIMD processing enables significant performance improvements for algorithms with parallel computation properties. Each ciphertext variable is effectively a vector of values and any unary or binary operation has the effect of array operations on all elements of these vectors. Using batching, integral data types naturally have arrays

[1]Computer-assisted program transformation to data-oblivious variants is an active topic of research, focusing currently on domain-specific languages and compilers [3, 9].

of bit values inside each bit of the variable, and gate operations on each separate bit have the effect of the gate operation on all bit values. BGV, BFV, and CKKS all support batching which enables parallel processing of encrypted vectors of data.

3 PROPOSED BRIDGING METHODOLOGY

We define new data types to abstract the low-level complexity of the encryption scheme and Boolean circuits. This is possible by extending the E3 framework [6], which is one level of abstraction above FHE libraries and enables portability of code across different libraries.

3.1 Modular and bit-level arithmetic

Some FHE schemes define arithmetic addition, subtraction, and multiplication on numeric rings, but not other arithmetic such as division or comparison. While some applications may not require any other operation, a programmer is accustomed to having all programming operations available in their programs. For example, the C++ statement "`if(a<b)c+=a`" and its corresponding data-oblivious form: "`c+=(a<b)*a`", cannot be evaluated with an FHE scheme where the comparison operation is not defined. Notably, addition, subtraction, and multiplication on integers is an incomplete set of arithmetic operations; for instance, the comparison operation cannot be reduced to these three operations. However, when operating on bits, the same set of operations is universal, having addition and subtraction correspond to logical `XOR` and multiplication to logical `AND`.

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

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

$$\begin{aligned} x \,\mathtt{AND}\, y &= xy, & x \,\mathtt{NAND}\, y &= 1 - xy \\ x \,\mathtt{OR}\, y &= x + y - xy, & x \,\mathtt{NOR}\, y &= 1 - (x \,\mathtt{OR}\, y) \\ x \,\mathtt{XOR}\, y &= x + y - 2xy, & x \,\mathtt{XNOR}\, y &= 1 - (x \,\mathtt{XOR}\, y) \\ \mathtt{NOT}\, x &= 1 - x, & \mathtt{MUX}(x, y, z) &= x(y - z) + z, \end{aligned}$$

where `MUX` is the multiplexer operation (as in $x?y{:}z$). The set of values $\{0, 1\}$ is closed under the above set of expressions. In this paper, we use the term *homomorphic gates* to describe logic gates operating on encrypted bits. Given these homomorphic gates, higher level operations (comparison, addition, multiplication) can be built using standard combinational arithmetic circuit design, which allows to perform any programming operation, i.e. giving the full spectrum of C++ arithmetic. Gate equations simplify for plaintext modulo 2; e.g. XOR gate does not require multiplication, as x `XOR` $y = x + y \bmod 2$.

```
 1  SecureUint<8> fibonacci(SecureUint<8> in)
 2  {
 3      SecureUint<8> i=_0_E, a=_0_E, b=_1_E;
 4      SecureUint<8> r=_0_E;
 5      int max_iter = 10;
 6      while( max_iter-- )
 7      {
 8          r += (i++ == in) * a;
 9          std::swap(a,b);
10          a += b;
11      }
12      return r;
13  }
```

Listing 3: Secure version of Fibonacci function. Type `SecureUint` **behaves as native** `unsigned int`**.**

Data types: We define three data types for bit-level arithmetic, `SecureUint`, `SecureInt`, and `SecureBool`, and one for modular arithmetic, `SecureMod`. A `SecureUint` or `SecureInt` variable is an array of ciphertexts, each encrypting either zero or one. `SecureUint` is comparable to `unsigned int`, while `SecureInt` corresponds to `int`. The third type is `SecureBool`, which is a type derived from `SecureUint<1>`. It is the secure equivalent of `bool`. For modular arithmetic, there is only the native type, which operates modulo the plaintext modulus. We call this type `SecureMod`.

With bit-level arithmetic, we can express programs with operations not supported by modular arithmetic. Listing 3 demonstrates the transition from plaintext computation to secure computation: Integer variables are declared with the secure type `SecureUint`; the rest of the code remains the same. In encrypted computation, increasing the number of bits of integral types can lead to a dramatic increase in performance overhead. For this reason, we define the bit size as a template specialization. Explicit size declaration can be found in functional programming languages, so the programmer may already be familiar with this practice. In the example of Listing 3, all variables are 8 bits (lines 3, 4).

The postfix `_E` after each constant provides encrypted values used in variable initialization of integral type variables. E3 automatically encrypts and replaces the plaintext constants with their encrypted representations, so the final binary does not have information about the initial values used in the program.

3.2 Bridging modular and bit-level arithmetic

Declaring variables with a protected integral type and using solely bit-level arithmetic as in Listing 3 has a potential drawback: When an FHE scheme provides fast modular arithmetic operations, the usage of circuits operating on separate bits is slow. This paper brings the idea of *Bridging* - mixing both modular and bit-level arithmetic in one program with the ability to convert variables from one type to the other. Some variables can be declared using a protected type supporting only modular arithmetic, while others with another secure type supporting bit-level arithmetic. In bridging mode, a type of bit-level arithmetic declares a conversion function into a type of modular arithmetic, and vice-versa. In all cases, the encryption of the two different C++ types must share the same FHE keys, which is ensured by our proposed methodology and framework.

```
 1  SecureMod fibonacci(SecureUint<8> in)
 2  {
 3      SecureUint<8> i=_0_E;
 4      SecureMod a=_0_M, b=_1_M, r=_0_M;
 5      int max_iter = 10;
 6      while( max_iter-- )
 7      {
 8          r += (i++ == in) * a;
 9          std::swap(a,b);
10          a += b;
11      }
12      return r;
13  }
```

Listing 4: Bridging (i.e., mixing `SecureMod` and `SecureUint` types) enables performance improvement. The postfix _M denotes encrypted variable for the `SecureMod` type.

3.2.1 `SecureUint` *to* `SecureMod`. For performance reasons, it is desirable to execute the entire computation in modular arithmetic, since it is much faster than bit-level. If however, a program requires an operation not supported in modular arithmetic (e.g., comparison), then, without Bridging, the whole program must perform all computations using bit-level arithmetic, severely degrading performance. In effect, Bridging enables the isolation of the parts of the computation requiring bit-level arithmetic. For example, the expression "`c+=(a<b)*a`" can use bit-level arithmetic for the comparison only. The variables required by the comparison (i.e., `a` and `b`) must be of integral type. Nevertheless, the operands of the multiplication can be cast to our modular type, allowing multiplication and addition to be executed in modular arithmetic, resulting in a variable `c` of modular type.

Listing 4 demonstrates the code of the Fibonacci function of Listing 3 with *Bridged* arithmetic: Only the input `in` and counter `i` are declared as integral type `SecureUint`, while the others are replaced with the faster `SecureMod` type. Line 8 does implicit conversion from `SecureUint` to `SecureMod`; in this way, bit-level multiplication (which is slow) is not executed. Instead, only the native (much faster) multiplication of ciphertexts is needed. Specifically, the comparison between `i` and `in` is the slowest operation in the program, and its result is one encrypted bit which can naturally be casted to `SecureMod`, since the set $\{0, 1\}$ is a subset of the plaintext range. The operation for casting an integral type (bit-level) into modular (FHE native) is a summation of the encrypted bits of the integral type. In fact, the binary representation of a value X can be reorganized by *Horner's scheme* in a set of additions over its s bits x_i:

$$\begin{aligned} X &= 2^{s-1}x_{s-1} + 2^{s-2}x_{s-2} + \ldots + 2x_1 + x_0 = \\ &= (\ldots((x_{s-1}) \cdot 2 + x_{s-2}) \cdot 2 + \ldots + x_1) \cdot 2 + x_0 \end{aligned}$$

Evaluating the right hand side of the above equation yields the value corresponding to the bit sequence. This evaluation is an efficient way to convert a program variable of type `SecureUint` into a `SecureMod` value. Listing 5 shows the C++ implementation of such casting. We emphasize that this conversion requires no ciphertext multiplications, only additions. Specifically, $2 \cdot (s - 1)$ ciphertext additions with a maximum additive depth of $s - 1$, where s is the number of encrypted bits.

Line 8 of Listing 4 does implicit conversion of `SecureBool` to `SecureMod`. Note that `SecureBool` is a derived class from `SecureUint<1>`. To observe a more complex scenario, consider the expression "`c+=(a==b)*a`" that actually requires conversion of

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

Listing 5: Casting from `SecureUint` to `SecureMod`. Since the former is a set of ciphertexts representing encrypted bits, it is possible to access each bit individually.

```
template <int Size>
SecureMod to_SecureMod(SecureInt<Size> v)
{
    SecureUint<Size> u(v);
    auto pos = to_SecureMod(u);
    int max = 1 << Size;
    auto neg = SecureMod::t - max + pos;
    SecureMod isNeg = v[Size-1];
    return isNeg * neg + (1-isNeg) * pos;
}
```

Listing 6: Casting from `SecureInt` to `SecureMod`. The `SecureInt` variable is converted to `SecureUint`, which is then converted to `SecureMod`.

`SecureUint`. Comparison between `a` and `b` must be evaluated in bit-level. This implies that types of `a` and `b` must be `SecureUint`. The comparison is done on a bit-by-bit manner using homomorphic gates following the gate equations described in Section 3.1. The gates correspond to normal logic gates, but operating on ciphertexts instead of ordinary bits. The result of the comparison is one encrypted bit represented by type `SecureBool`. Multiplication between a `SecureBool` and a `SecureUint` is evaluated as a multiplexer operation with s AND gates, where s is the number of encrypted bits in the `SecureUint` variable, resulting in a `SecureUint` type. The type of variable `c` can be chosen as `SecureMod`. The addition in the expression is then performed on variables of `SecureMod` and `SecureUint` types: "`c=c+t`", where "`t=(a==b)*a`". Implicit conversion evokes our function of Listing 5 from the constructor of `SecureMod` type out of `SecureUint`. Then the addition operation follows on two variables, both of the `SecureMod` type.

It should be noted that all these conversions and evaluations are done obliviously to the user and do not require special attention; the user writes only "`c+=(a==b)*a`". A more efficient way to perform this computation is to explicitly convert argument `a` in this expression to `SecureMod`. In such case, the multiplication is done in modular arithmetic which is around s times faster than a multiplication between `SecureBool` and `SecureUint` (and many more times faster than multiplying two `SecureUint`s).[2] The corresponding expression becomes "`c+=(a==b)*SecureMod(a)`". Same way as above, the constructor calls the conversion function (Listing 5), this time one step earlier - before the multiplication - resulting in having a larger portion of the computation in modular arithmetic, hence improving the performance. It it worth noting that while there is automatic conversion from `SecureUint` to `SecureMod`, it is the programmer's task to define each variable's type and, in some cases, call conversion explicitly for better performance.

[2]The exact speed-up compared to multiplying two `SecureUint`s depends on the variables' bit size.

3.2.2 `SecureInt` *to* `SecureMod`. So far, we have discussed unsigned numbers. However, bit-level arithmetic also supports signed numbers following the two's complement arithmetic. On the other hand, modular arithmetic only supports numbers in $\mathbb{Z}_t$, where t is the plaintext modulus. Nevertheless, it is possible to emulate negative numbers in modular arithmetic in the programmer's domain as in Cryptoleq [22], where lower values are considered positive and large values are interpreted as negative numbers. In this case, the conversion from a signed bit-level arithmetic type `SecureInt` to `SecureMod` is defined as:

$$X = \begin{cases} t - 2^s + \sum_{i=0}^{s-1} (2^i \cdot x_i), & \text{if } x < 0. \\ \sum_{i=0}^{s-1} (2^i \cdot x_i), & \text{otherwise.} \end{cases} \tag{1}$$

where s is the number of bits and x_i is the bit of x at position i. The condition $x < 0$ is determined by the most significant bit of x; thus, we can use it as a multiplexer between the two cases. Listing 6 presents the algorithm for converting a `SecureInt` into a `SecureMod`. First, the `SecureInt` is interpreted as a `SecureUint` (line 4). In line 5, this value is converted into a `SecureMod` using the algorithm of Listing 5. Then in line 7, we perform the subtraction of plaintexts t and `max` (i.e. 2^s), and then do a ciphertext addition with the `SecureMod` variable `pos`. At this point, we have generated two `SecureMod` variables: `pos`, representing the value in case it is positive, and `neg` containing the value in case it is a negative number. This totals $2s - 1$ ciphertext additions with a additive depth of s. Finally, we select between `neg` and `pos` using the most significant bit of the `SecureInt` input (lines 8-9). If the bit is one, it means it is a negative number; thus, we select `neg`; otherwise, we select `pos`. For selection we need ciphertext multiplications. The entire conversion from `SecureInt` to `SecureMod` requires two ciphertext multiplications and $2s + 1$ ciphertext additions with a multiplicative depth equal to one. By comparing Listings 5 and 6, we can see that converting signed numbers to `SecureMod` is less efficient than converting unsigned numbers to `SecureMod`. Furthermore, t must be large enough ($t \geq 2^s$) to accommodate the converted value.

3.2.3 `SecureMod` *to* `SecureUint`. Conversion from `SecureMod` to `SecureUint` is theoretically possible using the expression:

$$X = \sum_{i=1}^{t-1} (i \cdot \texttt{SecureBool}(1 - (x - i)^{t-1})) \tag{2}$$

where x is the `SecureMod` variable to be converted, t is the plaintext modulus and is prime, i is a `SecureUint` counter, and X is the resulting `SecureUint`. Due to the properties of modular arithmetic and the parameters used in homomorphic encryption, the exponentiation to $t-1$ results in zero in case the base is zero, and one otherwise. When $i = x$, the expression in the summation results in i, while in all other cases it will result in zero, since $1 - (x - i)^{t-1} = 0 \; \forall \; i \neq x$.

```
SecureMod pow(SecureMod b, int e)
{
    if (e == 0) return SecureMod(1);
    if (e == 1) return b;
    auto r = pow(b * b, e >> 1);
    if (e & 1) r *= b;
    return r;
}
```

Listing 7: Homomorphic exponentiation function.

```
template <int Size>
SecureUint to_SecureUint<Size>(SecureMod x)
{
SecureMod one(1);
    auto & t = SecureMod::t;
    vector<SecureUint<Size>> v;
    for (int i = 1; i < t; i++)
    {
        auto si = SecureUint<Size>(i);
        auto eq = one - pow(x-i, t-1);
        v.push_back(si * SecureBool(eq));
    }
    return sum(v);
}
```

Listing 8: Casting from `SecureMod` to `SecureUint`.

```
template <int Size>
SecureInt to_SecureInt<Size>(SecureMod x)
{
    auto u = to_SecureUint<Size>(x);
    SecureInt<Size> pos(u);
    auto max = 1 << s;
    auto diff = max - SecureMod::t;
    u = to_SecureUint<Size>(diff + x);
    SecureInt<Size> neg(u);
    auto & isNeg = pos[s-1];
    return isNeg * neg + (1-isNeg) * pos;
}
```

Listing 9: Casting from `SecureMod` to `SecureInt`.

An implementation of the fast exponentiation algorithm is presented in Listing 7. The number of multiplications is given by $\lfloor \log_2 e \rfloor + \omega(e) - 1$ and the multiplicative depth is $\lceil \log_2 e \rceil$, where e is the exponent and $\omega(\cdot)$ is a function that calculates the Hamming weight. The conversion from `SecureMod` to `SecureUint` can exploit this property of the exponentiation to $t - 1$ resulting in zero or one to create an equality function, where the equality function is given by $1 - (x - i)^{t-1}$. With this information, we can build a linear search to find the `SecureUint` i that is equal to the `SecureMod` x using the result of the equality as a selector. Listing 8 presents the algorithm that performs this conversion. The number of multiplications is given by $t \cdot (s + \lfloor \log_2 (t-1) \rfloor + \omega(t-1) - 1)$, while the multiplicative depth is equal to $\lceil \log_2 (t-1) \rceil + 1$. One can notice that this algorithm is only practical for small plaintext moduli t. Once t becomes large, the linear search makes it impractical. Since `SecureMod` requires a large t for it to be useful, this conversion should not be used in practice and should be avoided, as later presented and discussed in Section 4.2.

3.2.4 `SecureMod` *to* `SecureInt`. Conversion from `SecureMod` to `SecureInt` is possible by applying Eq. 3 to the result of Eq. 2:

$$Y = (1 - X_{s-1}) \cdot X + X_{s-1} \cdot (2^s - t + X) \tag{3}$$

X is the result of Eq. 2, Y is the `SecureInt` output, s is the number of encrypted bits in X and Y, t is the plaintext modulus, and $2^s \geq t$. Listing 9 shows the algorithm for this conversion. It leverages the conversion described in Listing 8 and adjusts for the sign. This algorithm requires $2 \cdot t \cdot (s + \lfloor \log_2 (t-1) \rfloor + \omega(t-1) - 1) + 2$ multiplications with a multiplicative depth of $\lceil \log_2 (t-1) \rceil + 2$.

4 EXPERIMENTAL RESULTS

4.1 Overview and setup

We evaluate bridging using three sets of experiments:

(1) Conversion from `SecureUint` and `SecureInt` to `SecureMod`, and vice-versa, using different bit sizes.

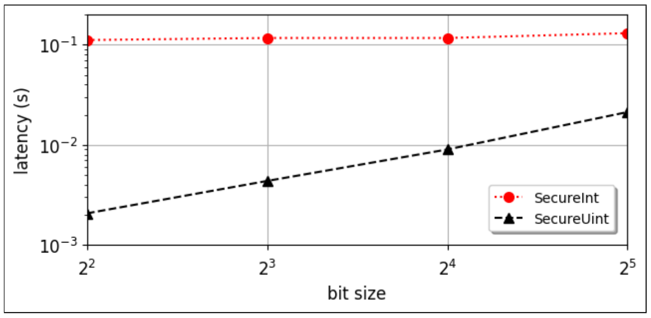

Figure 1: Conversion latency time from `SecureUint` and `SecureInt` to `SecureMod` for different bit sizes. Throughput is up to $n = 2^{15}$ times faster since a ciphertext fits n plaintexts.

(2) Performance comparison of bridging and bit-level arithmetic using synthetic benchmarks.
(3) Case study analysis using one real-world FHE application.

The results presented in Section 4 were collected using a single thread (Sections 4.2 and 4.3) and 24 threads (Section 4.4) of an Intel Xeon Silver 4214R CPU @ 2.40GHz with 24 cores and 1 TB of memory running on RHEL 7.9, with GCC 7.3.1. We use the E3 framework (commit #9fb718f) with SEAL 3.3.2 [28] as underlying FHE library, and BFV as the encryption scheme. We set the encryption parameters as: polynomial degree $n = 2^{15}$, as it provides the largest noise budget for encrypted computation, and plaintext modulus $t = 2^{16} + 1$, as this is the smallest t that enables batching for the chosen n. While a smaller t would provide less noisy homomorphic operations, virtually all practical FHE applications rely on efficient utilization of batching. Nevertheless, although $t = 2$ does not enable batching on BFV, we also test it on the benchmarks (Section 4.3) since some homomorphic gates are simpler in modulo 2. The remaining parameters are automatically defined by SEAL given the required security level of 128 bits.

4.2 Conversion

Fig. 1 presents the conversion latency time from `SecureUint` and `SecureInt` to `SecureMod` for different bit sizes following the algorithms of Listings 5 and 6. The number of multiplications for `SecureUint` and `SecureInt` stays constant at 0 and 2, while the multiplicative depth is 0 and 1, respectively, for all bit sizes. . As expected, conversion from both `SecureUint` and `SecureInt` becomes slower for larger bit sizes due to the larger number of additions required. The slowdown is the same for both types. It is less noticeable for `SecureInt` due the log-scale graph and the fact that the multiplications dominate the `SecureInt` latency time.

We also evaluate the conversion latency time from `SecureMod` to `SecureUint`/`SecureInt` using the algorithms presented in Listings 8 and 9 with different plaintext moduli and bit sizes ($s = \{4, 8, 16, 32\}$). Fig. 2 summarizes the findings. The results are what we expect from analysing the algorithms in Listings 8 and 9: When converting from `SecureMod` to `SecureUint`, the latency increases linearly to the bit size due to the operation on line 11 of Listing 8. Nevertheless, the dominant factor is the plaintext modulus t. A minor logarithmic effect comes from the exponentiation function (line 10) where the number of multiplications increases, while a major linear effect comes from the larger number of iterations in the

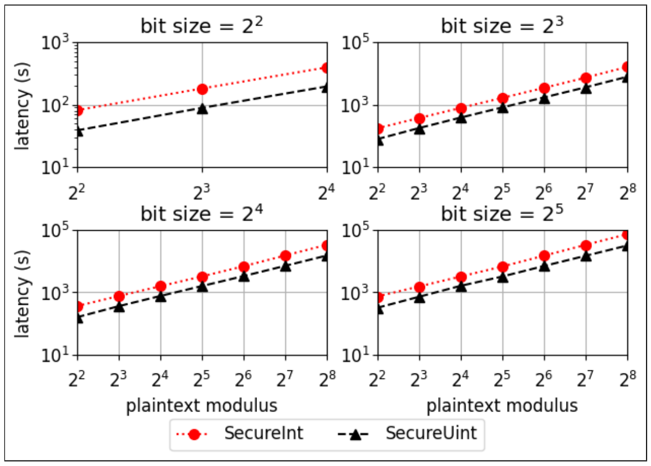

Figure 2: Conversion latency time from `SecureMod` to `SecureUint` and `SecureInt` for different plaintext moduli and bit sizes.

`for` loop (line 7). The conversion from `SecureMod` to `SecureInt` (Listing 9) takes roughly twice the time since it consists of two calls to the `SecureMod` to `SecureUint` conversion (lines 4 and 8) plus two multiplications (line 11).

4.3 Benchmarks

In order to evaluate different execution modes (bit-level arithmetic and bridging), we use six data-oblivious benchmarks, some of which were adapted from the TERMinator Suite [24]. These benchmarks are developed to be data-oblivious and manipulate sets of encrypted variables adjusted to $\{4, 8, 16\}$-bit size. The algorithms are: (FIB) Fibonacci, an additive-intensive algorithm, (LOG) Logistic Regression, where the data must be capped before inference, (MAX) Maximum, commonly used as non-linear function in ML applications, (MUX) Multiplexer, a simple operation similar to the ternary operator that replaces branch conditions on encrypted data, (PKS) Private Keyword Search, which searches privately for an item in a list or database, and (SOR) Sort, a sorting algorithm with low multiplicative depth, designed for FHE.

Effect of plaintext modulus: Although $t = 2$ does not enable batching in BFV, and therefore, it would not be used in practice, we compare it against the smallest plaintext modulus that enables batching ($t = 2^{16} + 1$) for $n = 2^{15}$. The plaintext modulus does not affect the latency time to execute a homomorphic operation, but some homomorphic gates simplify in modulo 2 (e.g. XOR), as we discuss in Section 3.1. Thus, a Boolean circuit used in bit-level arithmetic should have lower latency for $t = 2$ if it contains XOR or XNOR gates. In Fig. 3, we can see that in fact some applications are faster in modulo 2 (`SecureUint-2` and `SecureInt-2` for unsigned and signed bit-level arithmetic with $t = 2$). MUX is the simplest benchmark, containing in bit-level arithmetic one equality, two `SecureBool-SecureUint` multiplications (or `SecureBool-SecureInt` for signed numbers), one negation, and one addition. A `SecureBool-SecureUint` multiplication is entirely composed of XOR gates, and around half of the gates in the equality are XOR or XNOR. These operations are mainly responsible for the speed-up.

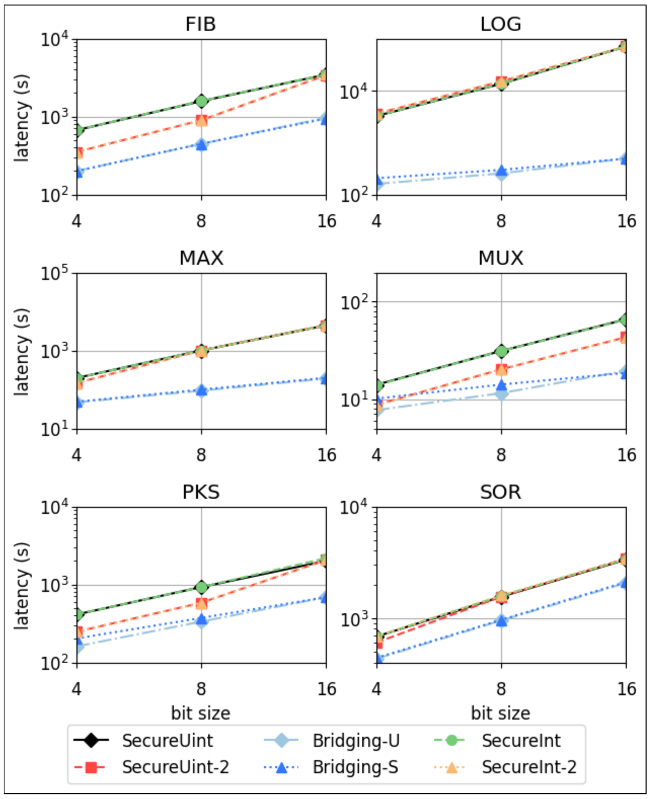

Figure 3: Latency time for six benchmarkswith $t = 2^{16} + 1$.

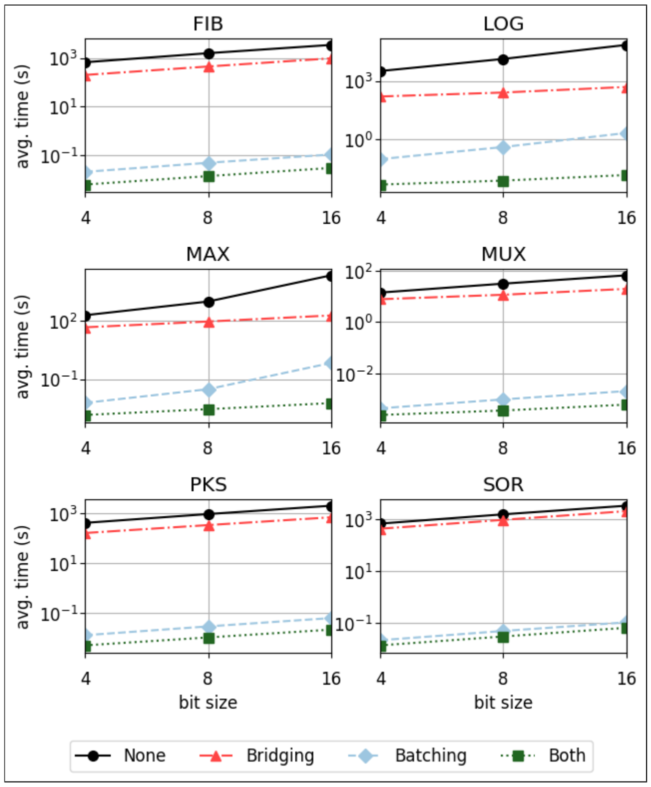

Figure 4: Average execution time for six benchmarks.

Bit-level arithmetic vs bridging: Fig. 3 presents the latency time comparison between bit-level arithmetic and bridging. Regarding unsigned numbers (`SecureUint` vs `Bridging-U`), results show that bridging outperforms bit-level arithmetic for all benchmarks and bit sizes. For some applications, like SOR, the performance improvement is limited (around 60% speed-up). This happens because this algorithm requires many non-native operations (comparisons) in most stages. Only in the last part of the algorithm bridging can be employed, since using bridging before would require the inefficient conversion from `SecureMod` to `SecureUint` in order to execute the latter comparisons. Conversely, the logistic regression benefits a lot from using bridging with a speed-up of more than two orders of magnitude. This is possible because the non-native operations required by the filtering function are executed first. Therefore, at the filtering stage it is already possible to employ bridging and perform the remaining computation using the faster modular arithmetic.
Signed numbers: Also in Fig. 3 we compare how bridging behaves with signed numbers. We can see some degradation in PKS (4 bits) and MUX (4 and 8 bits). In PKS, there is conversion from `SecureInt` to `SecureMod` in every iteration of the loop. The slowest operation in the loop is the comparison. This comparison operation is faster for smaller circuits (4 bits); therefore, the proportional latency of the two ciphertext multiplications required for the `SecureInt` to `SecureMod` conversion is higher. For larger circuits, the comparison becomes more costly, thus, amortizing the conversion cost.
Bridging can be used with batching: Batching is a powerful technique used in FHE to pack many plaintexts into a ciphertext. The number of plaintexts that can be packed into a ciphertext is equal to the polynomial degree ($n = 2^{15}$ in our case). Although carefully crafted algorithms using batching and rotations can reduce program latency, the main benefit of batching is increased throughput, since it is possible to process n plaintexts at once. Bridging on the other hand always reduces latency, and consequently improves throughput. Bridging is a technique independent of batching and both can be used in tandem. In Fig. 4 we present the average execution time (latency/# plaintexts packed) for six benchmarks and four cases: no batching and no bridging (None), bridging-only (`Bridging`), batching-only (`Batching`), and batching and bridging together (`Both`). As expected, the throughput improvement provided by batching is impressive. However, for this technique to be used at its fullest, it is necessary to fill all ciphertext slots with plaintexts, which is the case with embarrassingly parallel workload. If the load is less than n, the performance will degrade. Bridging does not suffer from this problem since it focuses on reducing the latency. Nevertheless, bridging can be used in combination with batching. In this case, the speed-up of both techniques is added, leading to an even faster runtime (acceleration by more than 6 orders of magnitude (LOG)).
Insights: Bridging demonstrates substantial performance improvement for algorithms that allow mixing `SecureUint` and `SecureMod`, while in the worst case, bridging is equivalent to bit-level arithmetic. The comparison heavy SOR benefits less from bridging since these operations occur throughout the program and near the output, which precludes using the `SecureMod` type in an earlier stage. On the contrary the benchmarks containing non-native operations

near the beginning exhibit more significant performance improvements (MAX, FIB, PKS). Bridging benefits are maximized when non-native operations (e.g. comparisons) are at the beginning of the computation and the remaining of the computation can be done in modular arithmetic. The non-native operation would require the whole computation to work on bit-level arithmetic, but with bridging it becomes much faster since only the non-native operations are performed with `SecureUint`, while the remaining run using `SecureMod`. This is demonstrated by our logistic regression benchmark, which provides more than two orders of magnitude (precisely 143 times) of performance improvement.

4.4 Case-study: Genotype imputation

Genotype imputation is the process of filling missing information in DNA sequencing with the use of statistical methods. P-Impute [14] converts an inefficient statistical model based on correlation of similar individuals into a Private Information Retrieval (PIR) problem, which has efficient solutions in FHE, using the BFV encryption scheme. In this section, we discuss the performance of p-Impute without (`SecureUint`) and with bridging taking into consideration the multiplicative depth for defining more efficient encryption parameters. Batching is used to fit n plaintexts into a ciphertext.

We set each run to use 24 threads and report the query time, i.e., the time performing encrypted computation, and the number of ciphertext additions, multiplications, and subtractions in Table 1. We use the same plaintext modulus as in the other experiments ($t = 2^{16} + 1$), but we vary the polynomial degree ($n \in \{2^{13}, 2^{14}, 2^{15}\}$) in order to evaluate the extra performance improvement provided by bridging for requiring a lower multiplicative depth. Without bridging, we were only able to run with $n = 2^{15}$, since lower polynomial degrees do not provide enough noise budget to run the application without bootstrapping. With bridging, we were able to reduce n to 2^{13} without corrupting the result. As n reduces, the number of batching slots in a ciphertext reduces since it is equal to the polynomial degree. However, the reduction in latency for the homomorphic operation compensates for the reduction in slots. Experimental results show that when n is halved, the latency of the ciphertext multiplication reduces by at least 4 times, and can be amplified depending on the behavior of cache memories. Consequently, one halving of n increases throughput by at least 2x and reduces latency by 4x.

For the same polynomial degree ($n = 2^{15}$), bridging is around 6.82x faster than bit-level arithmetic. This is due to the reduced number of operations on encrypted data, as one can see in Table 1. Nevertheless, bridging requires less noise budget to operate, which allows us to use smaller polynomial degrees. In the fastest case, bridging has the throughput improved by 62.2x, while the latency reduces by around 249 times.

5 DISCUSSION

Polynomial degree: We compare bridging with other accelerations of homomorphic computation at the programming level. As we show in Section 4.4, the polynomial degree affects the performance and noise budget. A smaller polynomial degree makes operations a few times faster. At the same time, it reduces the noise budget, allowing fewer computations before data corruption or bootstrapping. It also reduces the number of plaintexts that can be packed in a ciphertext when batching. Nevertheless, the performance improvement from a smaller polynomial degree outweighs the fewer batching slots in the ciphertexts.

Table 1: Number of ciphertext operations and execution time for p-Impute [14].

Bridging	n	add	mul	sub	time (s)
no	2^{15}	53288	66152	51256	5026
yes	2^{15}	14248	9072	4536	737
no	2^{14}	NA	NA	NA	NA
yes	2^{14}	28496	18144	9072	196
no	2^{13}	NA	NA	NA	NA
yes	2^{13}	56992	36288	18144	80

Batching enables SIMD usage of ciphertexts [29]. It can provide several orders of magnitude performance improvement for SIMD-compatible applications, however not all FHE schemes support it [7, 10]. Due to its significant computational benefits, batching should always be used where possible.

Bridging makes possible to use the comprehensive bit-level arithmetic and the fast modular arithmetic in the same program. It increases the expressivity of programs previously limited to modular arithmetic, and improves performance of complex programs implemented using bit-level arithmetic. At the same time, it reduces noise, since the depth of the datapath is shorter due to fewer homomorphic operations being executed, which may enable using a smaller polynomial degree, further improving performance. Therefore, the performance improvement provided by bridging comes from two factors: 1) Reduced number of homomorphic operations since Boolean circuits performing additions and multiplications in bit-level arithmetic are replaced by a single operation (native addition or multiplication), and 2) reduced multiplicative depth, since when Boolean circuits are replaced by a single instruction, the multiplicative depth reduces. This reduces the noise budget required by the application, enabling the use of a smaller polynomial degree, further improving performance. In addition, bridging is independent of encryption parameters, apart from the plaintext modulus which should not be too small. This makes bridging perfectly compatible with batching and any polynomial degree.

6 CONCLUSIONS

In this work, we presented a new methodology combining universal bit-level arithmetic and faster modular arithmetic computation, dubbed *bridging*, to accelerate applications using fully homomorphic encryption. Experiments demonstrate significant performance improvements when using both bridging and batching modes. Bridging by itself can offer several orders of magnitude performance improvement, depending on the type of application. In the benchmark evaluation, bridging improved performance by more than 2 orders, and 6 orders of magnitude when combined with batching. Furthermore, the case-study private genotype imputation became two orders of magnitude faster due to reduced number of homomorphic operations and multiplicative depth, allowing us to use more efficient encryption parameters.

RESOURCES

Bridging has been integrated with the E3 framework and is available on github.com/momalab/e3 from commit #88a323f.

REFERENCES

[1] Sally Adee. 2008. The hunt for the kill switch. *IEEE Spectrum* 45, 5 (2008), 34–39.

[2] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. 2014. (Leveled) fully homomorphic encryption without bootstrapping. *ACM Transactions on Computation Theory (TOCT)* 6, 3 (2014), 1–36.

[3] Sunjay Cauligi, Gary Soeller, Brian Johannesmeyer, Fraser Brown, Riad S Wahby, John Renner, Benjamin Grégoire, Gilles Barthe, Ranjit Jhala, and Deian Stefan. 2019. FaCT: a DSL for timing-sensitive computation. In *ACM Conference on Programming Language Design and Implementation*. 174–189.

[4] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H Lai. 2018. SGXspectre Attacks: Leaking Enclave Secrets via Speculative Execution. *arXiv preprint arXiv:1802.09085* (2018).

[5] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. 2017. Homomorphic encryption for arithmetic of approximate numbers. In *ASIACRYPT*. Springer, 409–437.

[6] Eduardo Chielle, Oleg Mazonka, Nektarios Georgios Tsoutsos, and Michail Maniatakos. 2018. E^3: A Framework for Compiling C++ Programs with Encrypted Operands. Cryptology ePrint Archive, Report 2018/1013. Online: https://eprint.iacr.org/2018/1013, GitHub repository: https://github.com/momalab/e3.

[7] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachene. 2016. Faster fully homomorphic encryption: Bootstrapping in less than 0.1 seconds. In *ASIACRYPT*. Springer, 3–33.

[8] Victor Costan and Srinivas Devadas. 2016. Intel SGX Explained. *IACR Cryptology ePrint Archive* 2016 (2016), 086.

[9] Eric Crockett, Chris Peikert, and Chad Sharp. 2018. Alchemy: A language and compiler for homomorphic encryption made easy. In *ACM Conference on Computer and Communications Security (CCS)*. 1020–1037.

[10] Léo Ducas and Daniele Micciancio. 2015. FHEW: bootstrapping homomorphic encryption in less than a second. In *EUROCRYPT*. Springer, 617–640.

[11] Junfeng Fan and Frederik Vercauteren. 2012. Somewhat Practical Fully Homomorphic Encryption. *IACR Cryptology ePrint Archive* 2012 (2012), 144.

[12] Craig Gentry, Amit Sahai, and Brent Waters. 2013. Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based. In *CRYPTO*. Springer, 75–92.

[13] Judah Ari Gross. 2018. Ending a decade of silence, Israel confirms it blew up Assad's nuclear reactor. https://www.timesofisrael.com/ending-a-decade-of-silence-israel-reveals-it-blew-up-assads-nuclear-reactor/.

[14] Gamze Gürsoy, Eduardo Chielle, Charlotte M. Brannon, Michail Maniatakos, and Mark Gerstein. 2021. Privacy-preserving genotype imputation with fully homomorphic encryption. *Cell Systems* (2021). https://doi.org/10.1016/j.cels.2021.10.003

[15] Shai Halevi and Victor Shoup. 2018. Faster Homomorphic Linear Transformations in HElib. Cryptology ePrint Archive, Report 2018/244. https://eprint.iacr.org/2018/244.

[16] David Hély, Maurin Augagneur, Yves Clauzel, and Jérémy Dubeuf. 2012. Malicious key emission via hardware Trojan against encryption system. In *International Conference on Computer Design (ICCD)*. IEEE, 127–130.

[17] Alberto Ibarrondo and Alexander Viand. 2021. Pyfhel: PYthon For Homomorphic Encryption Libraries. In *Proceedings of the 9th on Workshop on Encrypted Computing; Applied Homomorphic Cryptography* (Virtual Event, Republic of Korea) *(WAHC '21)*. Association for Computing Machinery, New York, NY, USA, 11–16. https://doi.org/10.1145/3474366.3486923

[18] Yier Jin, Michail Maniatakos, and Yiorgos Makris. 2012. Exposing vulnerabilities of untrusted computing platforms. In *International Conference on Computer Design (ICCD)*. IEEE, 131–134.

[19] Ramesh Karri, Jeyavijayan Rajendran, Kurt Rosenfeld, and Mohammad Tehranipoor. 2010. Trustworthy hardware: Identifying and classifying hardware trojans. *Computer* 43, 10 (2010), 39–46.

[20] Paul Kocher, Jann Horn, Anders Fogh, , Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2019. Spectre Attacks: Exploiting Speculative Execution. In *40th IEEE Symposium on Security and Privacy (S&P'19)*.

[21] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown: Reading Kernel Memory from User Space. In *27th USENIX Security Symposium (USENIX Security 18)*.

[22] Oleg Mazonka, Nektarios Georgios Tsoutsos, and Michail Maniatakos. 2016. Cryptoleq: A Heterogeneous Abstract Machine for Encrypted and Unencrypted Computation. *IEEE Transactions on Information Forensics and Security* 11, 9 (2016), 2123–2138. https://doi.org/10.1109/TIFS.2016.2569062

[23] Christian Mouchet, Jean-Philippe Bossuat, Juan Ramón Troncoso-Pastoriza, and Jean-Pierre Hubaux. 2020. Lattigo: a Multiparty Homomorphic Encryption Library in Go.

[24] Dimitris Mouris, Nektarios Georgios Tsoutsos, and Michail Maniatakos. 2018. TERMinator Suite: Benchmarking Privacy-Preserving Architectures. *IEEE Computer Architecture Letters* 17, 2 (2018), 122–125.

[25] nuFHE 2018. NuCypher fully homomorphic encryption (NuFHE). https://github.com/nucypher/nufhe. NuCypher.

[26] Sandro Pinto and Nuno Santos. 2019. Demystifying Arm TrustZone: A Comprehensive Survey. *ACM Comput. Surv.* 51, 6, Article 130 (Jan. 2019), 36 pages. https://doi.org/10.1145/3291047

[27] Nikola Samardzic, Axel Feldmann, Aleksandar Krastev, Srinivas Devadas, Ronald Dreslinski, Christopher Peikert, and Daniel Sanchez. 2021. F1: A Fast and Programmable Accelerator for Fully Homomorphic Encryption. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture* (Virtual Event, Greece) *(MICRO '21)*. Association for Computing Machinery, New York, NY, USA, 238–252. https://doi.org/10.1145/3466752.3480070

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

[29] Nigel P. Smart and Frederik Vercauteren. 2014. Fully homomorphic SIMD operations. *Designs, codes and cryptography* 71, 1 (2014), 57–81.

[30] Nektarios Georgios Tsoutsos and Michail Maniatakos. 2014. Fabrication attacks: Zero-overhead malicious modifications enabling modern microprocessor privilege escalation. *IEEE Transactions on Emerging Topics in Computing* 2, 1 (2014), 81–93.

[31] Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lipp, Marina Minkin, Daniel Genkin, Yarom Yuval, Berk Sunar, Daniel Gruss, and Frank Piessens. 2020. LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection. In *41th IEEE Symposium on Security and Privacy (S&P'20)*.

[32] K Xiao, D Forte, Y Jin, R Karri, S Bhunia, and M Tehranipoor. 2016. Hardware Trojans: Lessons Learned after One Decade of Research. *ACM Transactions on Design Automation of Electronic Systems (TODAES)* 22, 1 (2016), 6.

[33] Kaiyuan Yang, Matthew Hicks, Qing Dong, Todd Austin, and Dennis Sylvester. 2016. A2: Analog malicious hardware. In *IEEE Symposium on Security and Privacy (S&P)*. IEEE.