Cryptoleq: A Heterogeneous Abstract Machine for Encrypted and Unencrypted Computation

Oleg Mazonka, Nektarios Georgios Tsoutsos, *Student Member, IEEE,*
and Michail Maniatakos, *Member, IEEE*

Abstract

The rapid expansion and increased popularity of cloud computing comes with no shortage of privacy concerns about outsourcing computation to semi-trusted parties. Leveraging the power of encryption, in this paper we introduce Cryptoleq: an abstract machine based on the concept of One Instruction Set Computer, capable of performing general-purpose computation on encrypted programs. The program operands are protected using the Paillier partially homomorphic cryptosystem, which supports addition on the encrypted domain. Full homomorphism over addition and multiplication, which is necessary for enabling general-purpose computation, is achieved by inventing a software re-encryption module written using Cryptoleq instructions and blended into the executing program. Cryptoleq is heterogeneous, allowing mixing encrypted and unencrypted instruction operands in the same program memory space. Programming with Cryptoleq is facilitated using an enhanced assembly language that allows development of any advanced algorithm on encrypted datasets. As a case study, we implemented and evaluated the performance of a typical Private Information Retrieval problem.

Index Terms

Abstract Machine, Compiler, Encrypted Computation, Obfuscation, One Instruction Set Computer, Heterogeneous Computer, Homomorphic Encryption, Paillier.

I. Introduction

Contemporary computing paradigms, such as cloud and pervasive computing, have become increasingly popular as they allow outsourcing computation to a typically more powerful or dedicated set of machines.

O. Mazonka and M. Maniatakos are with the Department of Electrical and Computer Engineering, New York University Abu Dhabi, UAE (e-mail: om22@nyu.edu; michail.maniatakos@nyu.edu).

N. G. Tsoutsos is with the Department of Computer Science and Engineering, New York University, New York (e-mail: nektarios.tsoutsos@nyu.edu).

From Bitcoin mining [1] and Mersenne primes search [2], to commercial cloud services offered by major industry companies, outsourced computation requires code execution in a remote machine. One fundamental concern with such paradigms, however, is the privacy of the outsourced data [3]. In addition to the legitimate third party that performs the outsourced computation, additional concerns arise in light of side channel attacks [4] or even hardware Trojans [5]–[7].

Fortunately, cryptographic primitives such as homomorphic encryption can be leveraged to address those privacy concerns, and eventually return control of the data back to the legitimate information owner [8], [9]. As soon as fully homomorphic encryption (FHE) became theoretically possible [10]–[13], the academic interest in FHE applications has increased accordingly. From secure cloud computation [14] and verifiable computation [15], to multiparty computation [16] and message authenticators [17]. In addition, partial homomorphic encryption (PHE) has recently been leveraged for verifiable computation [18].

Despite the wide range of applications that can benefit from FHE schemes, their efficiency has been a concern, and their practicality has been questioned [19], [20]. More practical implementations of FHE, such as [21], have already been instantiated in the HElib software library, but fully homomorphic operations could still have overheads in the order of seconds [22]. In addition, HElib only recently has evolved to support bootstrapping and rencryption [23], and applications that use this library to implement generic/interactive computer programs that process homomorphic data are yet to be seen.

On the other hand, PHE schemes are more practical than their FHE counterparts, and what the former lack in range of supported operations, they gain in efficiency. Indeed, PHE schemes typically require straightforward operations on ciphertexts, such as modular multiplication, which can be implemented very efficiently (e.g. [24]–[27]). Our observation is that PHE could be sufficient for practical applications of outsourced computation, where the applicable threat model can afford obfuscation as an adequate mitigation control to protect the privacy of processed data. Of course, PHE is less powerful than FHE in terms of computational completeness, and end-to-end encryption is traded for performance in computations. This tradeoff will allow us to design and implement a new programming language capable of processing homomorphically encrypted data with practical computational cost.

Problem formulation: This work addresses the problem of protecting the privacy of sensitive data, when these data are being processed within semi-trusted containers and the computation is outsourced. In addition, this work combines privacy preservation with a practicality requirement, which, so far, is difficult to achieve through FHE schemes. Likewise, this work also addresses the usability problem of theoretical approaches, which prevents them from seeing wide deployment and being used in everyday applications.

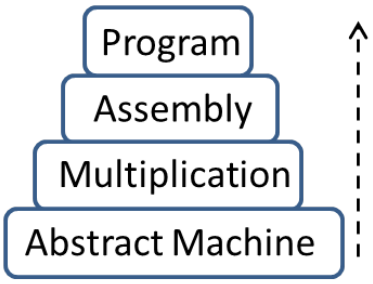

Fig. 1. Cryptoleq abstraction layers.

Our contribution: In this work we propose Cryptoleq, a new programming language based on a single instruction computer architecture, which processes homomorphic data natively. Cryptoleq defines a universal computer for processing encrypted and unencrypted data together within the same program memory space. Data encryptions are generated using Paillier PHE, and only homomorphic addition is natively supported. Universal computation, however, requires support for both addition and multiplication, and in this work, we simulate multiplication with obfuscated re-encryption implemented using Cryptoleq instructions. Overall, we claim the following:

- *Design and implementation of Cryptoleq,* which expands single instruction computing with native support for homomorphic data using a novel bit layout representation. Cryptoleq supports programs written without privacy protections, as well as protected execution using encrypted data under full encryption or obfuscation modes, depending on the need to multiply encrypted values.
- *A practical framework for Cryptoleq* with extended assembly language, compiler, and emulator for executing Cryptoleq programs on different platforms (e.g. x86, ARM).

Roadmap: The paper is organized on a bottom-up traversal of Cryptoleq abstraction layers (fig. 1). Following a preliminaries discussion on homomorphic encryption and single instruction architectures (section II), the Cryptoleq abstract machine is presented in section III. Multiplication on encrypted operands is then presented in section IV, while section V discusses the Cryptoleq Enhanced Assembly Language (CEAL) that accelerates program development in Cryptoleq. In section VI, we evaluate the performance of Cryptoleq using a Private Information Retrieval case study, while related work and conclusions appear in sections VIII and IX.

II. PRELIMINARIES

A. Homomorphic Encryption

Homomorphic encryption allows the application of mathematical operations directly on encrypted data so that the results after decryption would correspond to applying matching operations on unencrypted data. This can be expressed as:

$$\mathbf{v} = f(\mathbf{u}) \qquad \Longleftrightarrow \qquad \text{Enc}(\mathbf{v}) = g(\text{Enc}(\mathbf{u})) \tag{1}$$

where $\mathbf{v}$ and $\mathbf{u}$ are some vectors in the unencrypted domain, $\text{Enc}(\cdot)$ is the encryption operation over each vector element, and f and g are related functions. For example, if $\text{Enc}(z) = a^z$ then $\text{Enc}(x) \cdot \text{Enc}(y) = \text{Enc}(x+y)$ so the homomorphism is between addition in unencrypted data and multiplication in encrypted. In this case $\mathbf{u}$ is a vector of two elements, $\mathbf{v}$ is a scalar, f is scalar addition, and g is scalar multiplication.

Having one homomorphic mathematical operation is not sufficient for universal computation, as not all functions can be expressed through that operation. On the other hand, when we have both addition and multiplication, there exists a zero element that is both an additive identity and multiplicative absorbing element, and this allows evaluating any function. Indeed, addition and multiplication on a binary ring would define a Turing complete pair of logic gates [28], and such pair is sufficient to construct a universal machine. A homomorphic encryption scheme supporting only one operation is called partially homomorphic, while a scheme supporting two orthogonal operations is called fully homomorphic.

In this work, we employ the Paillier additive PHE scheme that uses a modulus N as its security parameter [29]. This modulus is defined as the product of two primes, and message m is encrypted as $r^N g^m \bmod N^2$. In Paillier, g is an encryption base parameter, while r is randomly selected to provide semantic security to the probabilistic scheme [30].

B. One Instruction Set Computer

One instruction set computer (OISC) is a computer architecture which supports only one instruction and is able to perform universal computation [31]. It operates on memory organized as a sequence of memory cells, while processor instructions and data reside in unified memory space, following the von-Neumann model. There exist three OISC categories [32]: (i) Transport Triggered Architecture Machines, (ii) Bit Manipulation Machines, and (iii) Arithmetic Based Machines.

Since OISC has only one instruction, it is an attractive choice for homomorphic computations for several reasons. In particular, OISC does not require the use of opcodes, since there is no need to discriminate between different instructions; the single instruction is fully defined by its arguments and the micro-operations associated with that instruction are predefined. On the other hand, if the underlying architecture

has multiple opcodes, they could either be unencrypted, so information about the executed algorithm can be leaked, or could be encrypted, so the encrypted processor would have to obliviously compute all possible opcodes for the given arguments and homomorphically combine their results. Furthermore, since OISC architectures typically apply a simple mathematical operation over their arguments, this operation can be directly ported to the encrypted domain by identifying its homomorphic counterpart. Thus, due to their simplicity, OISC architectures are naturally compatible with homomorphic encryption relations (eq. 1) and can be extended to support processing of encrypted arguments.

One popular Turing-complete OISC is Subleq, which stands for subtract and branch if less than or equal to zero [33]. Its abstract machine is easily implemented both in hardware and emulating software, as it is sufficiently simple and computationally efficient compared to other OISC variants. In this work, we leverage some Subleq principles for constructing our computational model, as presented in the following section.

III. Cryptoleq Abstract Machine

A. Description of the Model

The Cryptoleq abstract machine is a processor model operating on a sequence of memory cells. Each memory cell has an address and a value, while any cell value may also be used as a memory address. The processor has an instruction pointer IP and executes instructions read from memory. Each such instruction consists of three operands, named A, B and C.

Let $[\cdot]$ denote a dereference operation so that $[X]$ represents a memory cell with address X, as well as its value. The three instruction operands are fetched from memory as follows:

$$A = [\text{IP}]; \quad B = [\text{IP} + 1]; \quad C = [\text{IP} + 2]. \tag{2}$$

As soon as A and B are fetched, referenced memory cells $[A]$ and $[B]$ are accessed using the values of A and B as addresses (fig. 2). Then, the instruction modifies $[B]$ and IP values according to the following two steps:

$$\begin{aligned} [B] =& O_1([A], [B]) \\ \text{IP} =& \begin{cases} C, & \text{if } \ O_2([B]) \leq 0 \\ \text{IP} + 3, & \text{otherwise} \end{cases} \end{aligned} \tag{3}$$

where O_1 and O_2 are operations that will be explicitly defined in the following paragraphs (eq. 5). O_1 results in a "cell type" value, while O_2 results in an integer.

As soon as the first step of eq. 3 is executed (i.e. operation O_1), the result is stored back to the memory cell pointed by B. Next, operation O_2 is performed on the updated value of $[B]$ and the result

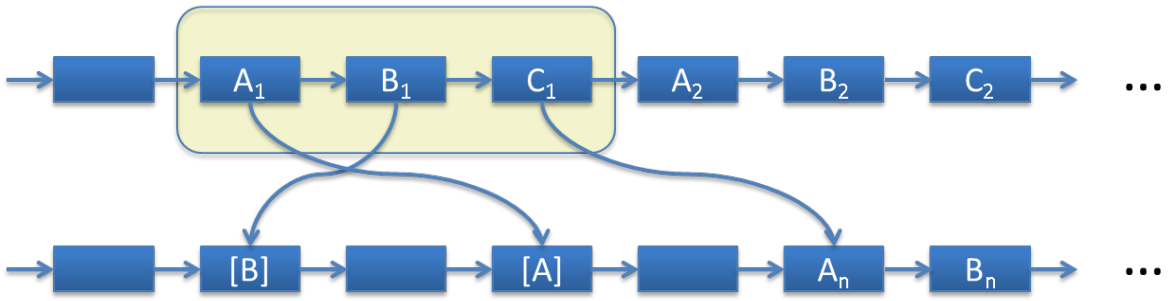

Fig. 2. Instruction and data organization in Cryptoleq.

is compared with zero. We further define a "Less or equal to zero" compound Boolean operation, named Leq, over a cell value argument:

$$\text{Leq}(x) \stackrel{\text{def}}{=} (O_2(x) \leq 0). \tag{4}$$

If Leq($[B]$) test is *True*, then value C is assigned to IP. This is effectively a control flow branch to the address of the cell where C points to. On the other hand, if Leq($[B]$) is *False*, the IP is updated with the address of the cell immediately after C (i.e. if IP was pointing to A, then it is increased by 3) and the execution sequence is repeated for the next instruction. Additional details regarding the increment of IP and address sequence organization are presented in section III-B.

***O* operations:** In Cryptoleq, operations O_1 and O_2 are defined as follows:

$$O_1(x, y) = x^{-1} \cdot y \bmod N^2, \qquad O_2(x) = \left\lfloor \frac{x-1}{N} \right\rfloor. \tag{5}$$

Throughout this paper we use exponent (-1) for reciprocal[1] in the corresponding modulus; to avoid ambiguity we also use $(\cdot)_{\xi}^{-1}$, where placeholder subscript ξ explicitly defines the modulus for the reciprocal. The floor brackets notation $\lfloor \cdot \rfloor$ refers to the integer part of the fraction in the equation above.

Operations $O_{1,2}$ have been judiciously selected in order to:

1) support computation on encrypted operands, compatible with the chosen encryption scheme (as it will become evident in section III-B), and
2) support computation on unencrypted operands, compatible with existing Turing-complete single instruction languages (further discussed in section III-D)

[1]Modular inversion is an expensive operation; section III-E presents how this operation can be avoided.

B. Encryption Scheme

Let N be a cryptographic parameter equal to the product of two primes. The Paillier encryption scheme is defined as a unique correspondence between a value x from $\mathbb{Z}^*_{N^2}$ and values m and r from $\mathbb{Z}_N$ and $\mathbb{Z}^*_N$ respectively:

$$x = r^N g^m \mod N^2 \tag{6}$$

where g is a generator in $\mathbb{Z}^*_{N^2}$. We select $g = 1 + Nk$ with a random k coprime to N. Our selection of g is less general than the original Paillier encryption definition [29], where g can be any number from $\mathbb{Z}^*_{N^2}$ with its order being nonzero and multiple of N. This selection of g is equally secure to [29] and sufficient for our scheme, with the order of g being N, and the rest of encryption and decryption being the same up to notation. In our encryption scheme, r serves as the probabilistic part of encryption, while m is the plaintext value to be protected. Based on our selection of g, eq. 6 becomes:

$$x = \text{Enc}(m) \equiv r^N(1 + Nkm) \mod N^2. \tag{7}$$

Here m can be written inside the brackets because modulus N^2 reduces all higher powers of N. Moreover, decryption requires knowledge of k as defined above, and ϕ that is the value of Euler's totient function of N:

$$\begin{aligned} m &= \text{Dec}(x) = \frac{x^{\phi(k\phi)^{-1}_N} - 1}{N} \mod N, \\ r &= (xg^{-m})^{N^{-1}_\phi} \mod N. \end{aligned} \tag{8}$$

Eqs. (7) and (8) provide a clear method to encrypt and decrypt numbers comprising the data and executable code of a program, while operations $O_{1,2}$ in eq. 5, along with the abstract machine description, define a method to evaluate a program.

Operation O_1 (eq. 5) is homomorphic to subtraction of m's with recombination of random parts, as follows:

$$x^{-1}y = (r_x^{-1} r_y)^N g^{m_y - m_x} \mod N^2. \tag{9}$$

Thus, multiplication of y by the inverted x corresponds to subtraction of their unencrypted values. Having the subtraction operation also brings the addition functionality to the programming paradigm, considering $x + y = x - (0 - y)$.

TS-notation: Our observation about the aforementioned encryption scheme is that the encrypted value x is also bijective to a unique pair of values t and s:

$$x = 1 + Nt + s \tag{10}$$

with $t = [0, N-1]$, $s = [0, N-2]$, and $(s+1)$ being coprime to N. If eq. 10 reconstructs x from t and s, the converse is:

$$t = \left\lfloor \frac{x-1}{N} \right\rfloor \qquad \text{and} \qquad s = ((x-1) \bmod N). \tag{11}$$

To distinguish these two equivalent representations, in this work we name *X value* the left-hand side of eq. 10 and *TS value* the corresponding t and s pair.

The view of an encrypted value x (X value) as t and s (TS value) has additional benefits: First, as elaborated in section III-C, a simple definition of negative values for O_2 is possible. Examining O_2 in eq. 5 and t in eq. 11, we observe that comparing t with zero is equivalent to Leq$(\cdot)$ comparison in eq. 4. In addition, TS-values are useful when s is zero, since subtraction of t values demonstrates the same homomorphic property as the encrypted m in eq. 9:

$$\begin{aligned} x^{-1}y &= (1 - Nt_x)(1 + Nt_y) = \\ &= 1 + N(t_y - t_x) \mod N^2. \end{aligned} \tag{12}$$

Leveraging this property, the Cryptoleq processor can execute instructions using TS values and generate the same result as if the instructions were executed on a backwards compatible Subleq processor.

Moreover, a TS pair with $s \neq 0$ can still be used in addition and subtraction because a special "unit" for each particular s can be defined. Let $U = 1 + Nu$ be a unit that increases t values by 1. Then, combining this definition with eq. 10 and operation O_1 (eq. 5), we can solve for u:

$$\begin{aligned} &U^{-1} \cdot (1 + Nt + s) = 1 + N(t+1) + s \mod N^2, \\ &u = (1+s)^{-1} \mod N. \end{aligned} \tag{13}$$

Using this definition, U can be used as a unit to predictably change the t part of any encrypted value.

Since memory cell addresses in Cryptoleq are of the same type as cell values, arranging these cells in a sequence without introducing the notion of U, is problematic: when an operand is treated as an address, naive homomorphic addition or subtraction of a constant does not correspond to an equal increment or decrement in the address value. Introducing the U unit is essential for defining the notion of sequence in encrypted values and use them as memory addresses. Hence, address arithmetic in eqs. (2) and (3) becomes a simple modular multiplication with a power of U; moreover, if TS values are used, address arithmetic is expressed as a direct increment or decrement of the t part.

C. Range of values

The Leq operation in eq. 4 is required to make a decision on what values within the range $(0, N)$ are considered less than zero. Thus, to introduce the concept of "negative" numbers, we break the values in range $(0, N)$ into two groups:

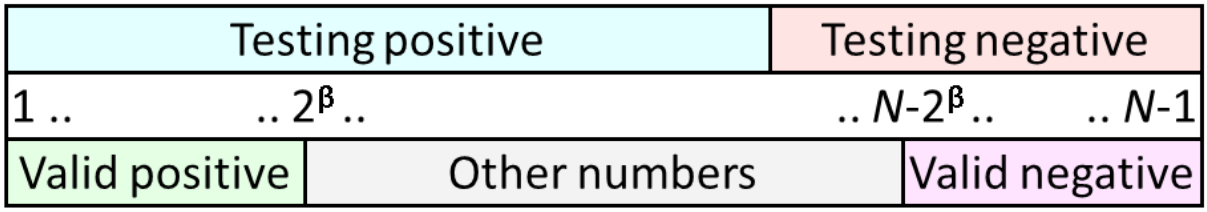

Fig. 3. Schematic display of Cryptoleq valid number range.

- **positive** is a number whose most significant bit position is less than the most significant bit position of N, and
- **negative** is a number whose most significant bit is at the same bit position as the most significant bit of N.

This grouping is natural, as it is similar to the commonly accepted bit representation of numbers in conventional computers. Since N is not a power of 2, the range of negative numbers is actually smaller than the range of positive numbers. To ensure symmetry between positives and negatives, as expected in computer arithmetic, we further restrict the range of valid positive numbers. Therefore, we define a "valid" positive number as a number that:

1) has a corresponding negative counterpart, congruent to modulus N, and
2) does not become a negative number when doubled.

Since $2^{\lfloor \log_2 N \rfloor}$ limits the range of positive numbers, $N - 2^{\lfloor \log_2 N \rfloor}$ defines the range of negative numbers. A natural parameter governed by the above definition of valid positive numbers can be expressed as the largest power 2^β with:

$$\beta = \left\lfloor \log_2 (N - 2^{\lfloor \log_2 N \rfloor}) \right\rfloor \tag{14}$$

so that any number greater than zero and less that 2^β is a valid positive number in Cryptoleq programs. Hence, the whole range of integers in a program is divided into the following four classes shown in fig. 3: (i) zero, (ii) valid positive numbers $< 2^\beta$, (iii) valid negative numbers, and (iv) other numbers with undefined behavior if used in the program arithmetic.

The β parameter introduced in eq. 14 defines numbers available to the arithmetic of Cryptoleq programs and provides a strong guarantee that each positive number has a corresponding negative, so that the Leq test (eq. 4) is defined correctly. Furthermore, β is required by the multiplication algorithm presented in section IV-B. It is worthwhile to note that some N moduli may correspond to a relatively small β value,

which may not be sufficient to accommodate the required range of numbers in a program; selecting such an N modulus, however, has very low probability as the bitsize of N increases.

Bit representation of numbers in Cryptoleq is somewhat different from conventional representations. Programs can use a value either encrypted (as eq. 7), or unencrypted, as a TS value with $t = m$ and $s = 0$. We name this latter case *open* representation:

$$\overline{x} = \text{Open}(m) = 1 + Nm. \tag{15}$$

Typical bitwise operations over this bit representation of numbers are not naturally supported and require emulation within the program using multiplication and division.

Using our definitions for operations $O_{1,2}$, the notion of negative numbers, and the ability to organize memory with sequential addresses, we can fully define our Cryptoleq abstract machine. Cryptoleq is able to obliviously execute code independent of whether values are constructed using encryption (eq. 7) or the open representation (eq. 15). This is beneficial since:

- it provides backward compatibility to other single instruction architectures, and
- both encrypted and unencrypted operands can be mixed within the same program to allow heterogeneous execution.

D. Mixed-mode execution

A Cryptoleq program can operate on open values (eq. 15), as well as encrypted ones (eq. 7). For the former case, program instructions effectively perform homomorphic subtraction (eq. 12) of the t parts (recall that open values have $s = 0$ by definition), and *compare the t value directly with zero*, as this is equivalent to Leq$(\cdot)$ comparison (eq. 4) since $O_2(x)$ (eq. 5) and t (eq. 11) yield the same value for the same x. Hence, the effective operation that instructions perform on open values is the same as if plaintext values were processed, and if operations $O_{1,2}$ in eq. 5 were redefined as:

$$O_1^{\text{Subleq}}(x, y) = y - x \qquad \text{and} \qquad O_2^{\text{Subleq}}(x) = x. \tag{16}$$

Indeed, if the set of eqs. (2) to (5) formally defines the Cryptoleq abstract machine, then redefining $O_{1,2}$ as in eq. 16 would be sufficient to define a Subleq abstract machine instead. Therefore, we can apply eq. 15 on all instructions and data of a Subleq program and run it natively using Cryptoleq.

In practice, encrypted Cryptoleq programs can use open values in several cases (e.g. for iteration or array indexing). Coexisting open and encrypted values in a program does not affect correctness since their interrelated usage is limited: while an encrypted value can be multiplied by an open value to produce an encrypted result, they cannot be added together or converted from one to the other. Indeed, a property of

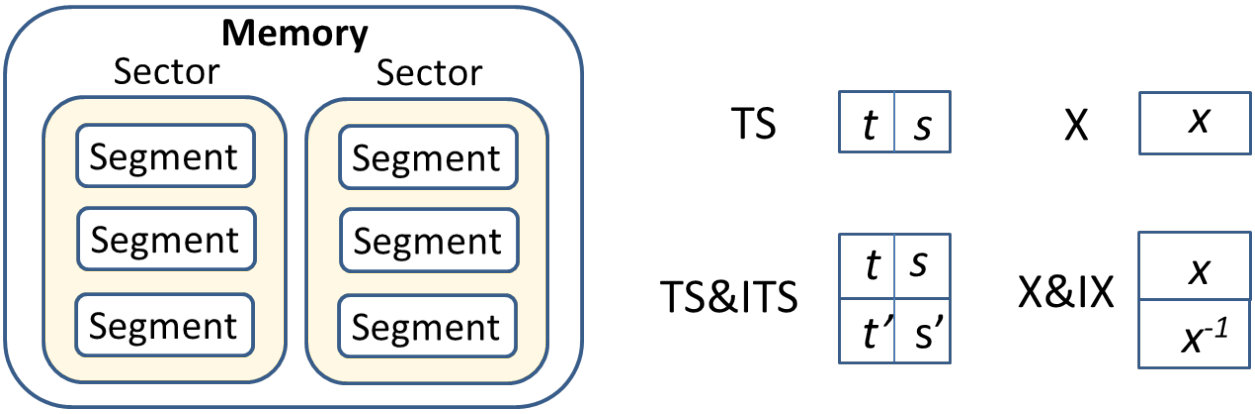

Fig. 4. Memory organization and inverted values.

secure encryption schemes is that encrypted values exist in their own domain and cannot be manipulated to leak data. To actually break the barrier between the encrypted and unencrypted domains, mixing open and encrypted values, obfuscated decryption and re-encryption capabilities can be used (section IV-A).

Cryptoleq instructions always treat open and encrypted values the same, as they are indistinguishable to the processor. In addition, modular multiplication of two open or two encrypted values is always homomorphic to adding the corresponding plaintexts. Nevertheless, Leq$(\cdot)$ comparisons (eq. 4) are expected to be performed only on open values, as applying this comparison on encrypted values with an unknown random part r would yield an unpredictable result. In some cases, the Leq comparison can be applied to encrypted values as well, but only if the compiler or the programmer can track modifications of the random part in encrypted values. In the latter case, it is possible to discriminate between expected encryptions.

Mixed-mode provides the flexibility to prioritize the variables to be kept private, given specified performance constraints, experimentally demonstrated in section VI.

E. Implementation remarks

Memory organization: Since the Cryptoleq abstract machine represents memory addresses using encrypted values, any potential implementation with standard memory modules would require a translation from these encrypted value to natural memory indices. In this work, we propose three different memory organization approaches:

1) In the first approach, the memory is organized as a collection of sectors (fig. 4). Each sector is a collection of continuous segments (i.e. sequences of memory cells) and all cell addresses within

one sector share the same s value, while all cell addresses within one segment have sequential t values. Incrementing a t value by a unit (eq. 13) returns the next cell address.

2) In the second approach, we employ the x values as addresses. Since the size of x is much larger compared to physical addresses, a "hash map" is used for translation.
3) The third memory organization also uses x values of addresses but implements a self-balancing binary search tree (e.g. red-black tree) data structure to allow efficient search of the physical address corresponding to an x value.

The first memory organization type is a natural selection when implementing the memory in hardware, while the second and third types are good matches for software emulations of Cryptoleq that leverage standard libraries (e.g. C++ STL).

Operand representation in memory: A second implementation decision is related to the representation of Cryptoleq operands in memory, as TS or X values. The first memory organization type discussed above (i.e. use of sectors) is by design compatible with TS storage, since access to a memory location is direct. The second and the third organization types (i.e. use of hash map and binary search tree respectively) are generally faster when the X representation is used, since x values are used directly for address mapping. A benefit of the X representation is its convenience for multiplying operands (eq. 5), while Leq comparisons (eq. 4) can be implemented using comparisons with precomputed values as:

$$\text{Leq}(x) = (x < 1 + N) \vee (x > N \cdot 2^{\lfloor \log_2 N \rfloor}). \tag{17}$$

On the other hand, the TS representation requires temporary transformation to X before multiplying operands, but it is more convenient for Leq comparisons. An extensive comparison of the various options appears in section VI-B1.

Avoiding modular inversion: Another important optimization is avoiding the modular inversions required in eq. 5, since such inversions incur high performance overheads. In this work, we propose to precompute and store the inverted values alongside the non-inverted ones. Then, a runtime inversion can be implemented by swapping the stored pair. In this case, the Cryptoleq processor must update both values in the pair when $[B]$ is updated, so an additional modular multiplication is necessary: for example, if one stored pair is $\{x, x^{-1}\}$ and another is $\{y, y^{-1}\}$, operation O_1 in eq. 5 should be applied twice to create an updated pair $\{x^{-1} \cdot y, x \cdot y^{-1}\}$. The benefit of this optimization is the evasion of expensive modular inversions, but this is traded with the aforementioned additional multiplication and doubling of required memory. This inverted representation can be used with both TS and X notations; thus, there are four distinct operand representation configurations (as presented in fig. 4): (i) TS, (ii) X, (iii) TS with

inverted TS (i.e. TS&ITS), and (iv) X with inverted X (i.e. X&IX). Note that t' and s' for inverted TS are not the modular inverses of t and s, but are computed using eq. 11 on x^{-1}.

Selecting β: Eq. 14 essentially defines the maximum ranges for valid positive and negative numbers in the program's arithmetic. Nevertheless, if β is always set to maximum using this equation, increasing the size of N (i.e. the security parameter) would actually reduce efficiency, since the maximum possible β increases as well. In practice, Cryptoleq allows programmers to fix β to a value smaller than the one in eq. 14. Indeed, an arithmetic range up to 64 bits for example is sufficient for most programs. As demonstrated by our experiments in section VI-B, restricting β to standard program arithmetic sizes has a significant improvement on execution overheads for large N.

F. Completeness of Cryptoleq

Based on the previous discussion, our Cryptoleq abstract machine supports programs that homomorphically add or subtract values and can perform conditional jump based on non-encrypted values (these are the same basic operations as a typical OISC architecture [31]). An important concept that has not been introduced yet, however, is support for multiplication[2] and comparison within programs. Other arithmetic OISC implementations address this problem at the software level, using multiplication algorithms expressed in native instructions. The same can be done in Cryptoleq. Multiplication algorithms, however, are optimally implemented using conditional jumps and in Cryptoleq this is available only for non-encrypted values (i.e. only such values can support multiplication). This limitation exists since conditional jumps over encrypted values could immediately leak side channel information about those values. Since multiplication (along with addition) is necessary for universal computation, one of our contributions is designing a special software function G that is used to implement a multiplication algorithm for encrypted values without conditional jumps, as well as to support encrypted comparisons.

IV. Multiplication over Encrypted Values

A. Function G

Function G is a software module that performs obfuscated decryption and re-encryption. Our goal is to define the simplest mathematical function which is sufficient for designing any other complex algorithm in Cryptoleq, such as multiplication or comparison.

[2]Division can be built on top of multiplication.

Definition: Let x be an encrypted value as in eq. 7, and $\overline{x}$ be the corresponding open value as in eq. 15. Let $\widetilde{y}$ be defined as $\widetilde{y} = r^N \cdot y$ for some random r. If $\widetilde{0}$ represents an encryption of *zero*, then function G is defined as follows:

$$G(x,y) = \begin{cases} \widetilde{0}, & \text{if} \quad \text{Leq}(\overline{x}) \\ \widetilde{y}, & \text{otherwise.} \end{cases} \tag{18}$$

Function G takes two encrypted arguments as inputs and returns the second argument modified with new randomization when the unencrypted value of the first argument is positive; otherwise, it returns an encryption of zero. In both cases, a different randomization parameter is used each time.

Our observation is that function G can be expressed using Cryptoleq instructions. Moreover, since Cryptoleq's O_1 operation is multiplication (eq. 5), exponentiation is also easy to implement as a combination of squaring and multiplication operations based on the bit expansion of the exponent. In Cryptoleq, if an encrypted value is raised to exponent $\phi(k\phi)_N^{-1}$, it actually decrypts into an open representation (as in eq. 15):

$$\begin{aligned} x^{\phi(k\phi)_N^{-1}} &= (r^N(1+Nkm))^{\phi(k\phi)_N^{-1}} = \\ &= (1+Nkm)^{\phi(k\phi)_N^{-1}} = \\ &= 1 + Nk\phi(k\phi)_N^{-1}m = 1 + Nm \mod N^2. \end{aligned} \tag{19}$$

Note that $r^{N\phi} = 1 \bmod N^2$. Cryptoleq programs do not have to retain a copy of $\phi(k\phi)_N^{-1}$, as its obfuscated bit expansion is sufficient to define the sequence of multiplications in calculating the result of function G; this sequence can be statically generated during program compilation. In case programmers choose to use a fixed β, as mentioned at the end of section III-E, any plaintext value can be multiplied by a *random* coefficient $s < 2^{\beta_{max} - \beta_{fixed}}$ *without affecting the result of* $\text{Leq}(\overline{x})$ in eq. 18. Hence, as a security enhancement, Cryptoleq programs can retail the obfuscated bit expansion of $s \cdot \phi(k\phi)_N^{-1}$ and calculate the result of function G correctly.

As it appears in the next section IV-B function G is adequate to implement a multiplication algorithm on encrypted values.

B. Multiplication Algorithm

The multiplication algorithm[3] is a software library procedure that produces an encrypted product given two encrypted factors. The requirements for this algorithm are: (i) it should be based only on addition and subtraction, (ii) it uses function G, and (iii) it cannot use any conditional jumps on encrypted values.

[3]This should not be confused with multiplication used for executing instructions (i.e. O_1 in eq. 5), which is homomorphic to addition in the unencrypted domain. Here, the presented algorithm is built on top of this homomorphic addition.

Algorithm 1 Multiplication: top level

```
1: procedure MULTIPLY(x, y)
2:     sum ← 0̃
3:     for (β + 1) times do
4:         z ← Div2(x)                          ▷ Div2 to be defined in alg. 2
5:         bit ← x − (z + z)
6:         sum ← sum + G(bit, y)
7:         y ← y + y
8:         x ← z
9:     end for
10:    return sum
11: end procedure
```

The non-branching constraint ensures that the algorithm iterations are always deterministic and do not depend on any argument values. Moreover, section III-C defines the ranges of valid positive and negative values for Cryptoleq programs so that the β parameter can be used to determine the algorithm iterations without any risk of overflowing positive values. Another observation is that it is sufficient to construct a multiplication algorithm for positive values only, since multiplying arbitrary values reduces to multiplying positives:

$$\begin{aligned} xy &= z_1 z_3 - z_2 z_3 \\ z_1 &= G(x, \tilde{1})G(y, \tilde{1}) + G(-x, \tilde{1})G(-y, \tilde{1}) \\ z_2 &= G(x, \tilde{1})G(-y, \tilde{1}) + G(-x, \tilde{1})G(y, \tilde{1}) \\ z_3 &= |x| \cdot |y| \end{aligned} \tag{20}$$

using the following formula for absolute values:

$$|x| = G(x, x) + G(-x, -x). \tag{21}$$

Eq. 20 expresses the multiplication of two arbitrary numbers x and y, based on the multiplication of their absolute values as well as multiplications of function G outputs (either $\widetilde{0}$ or $\widetilde{1}$ in this case). In addition, eq. 20 uses negations, which are defined as the modular additive inverses of the given values.

Our proposed non-branching multiplication procedure is presented in alg. 1. The algorithm also uses the Div2() auxiliary procedure that returns the division of an integer by 2 (alg. 2). The latter is also expressed as a non-branching algorithm and uses the Half2() auxiliary procedure (alg. 3). The input to

Algorithm 2 Division by 2

procedure DIV2(x)
  $sum \leftarrow \widetilde{0}$
  $p_2 \leftarrow \widetilde{2^\beta}$
  for β times **do**
    $p_2 \leftarrow$ Half2(p_2) ▷ Half2 to be defined in alg. 3
    $y \leftarrow sum + p_2$
    $y \leftarrow y + y - x$
    $sum \leftarrow sum + G(\tilde{1} - y, p_2)$
  end for
  return sum
end procedure

Algorithm 3 Division by 2 of power of 2

procedure HALF2(x)
  $sum \leftarrow \widetilde{0}$
  $p_2 \leftarrow \widetilde{1}$
  for β times **do**
    $y \leftarrow p_2 + p_2 - x$
    $y \leftarrow G(|y|, \tilde{1})$ ▷ $|\cdot|$ - eq. 21
    $y \leftarrow G(\tilde{1} - y, p_2)$
    $sum \leftarrow sum + y$
    $p_2 \leftarrow p_2 + p_2$
  end for
  return sum
end procedure

Half2() can only be a power of 2 and the procedure divides its input by 2, using function G and absolute values as in eq. 21. An alternative implementation for Half2() would be to use a precalculated lookup table, which is expected to be faster, and this optimization was used in our approach.

V. CRYPTOLEQ ENHANCED ASSEMBLY LANGUAGE

A. Overview

Cryptoleq is complemented by a developed piece of software named Cryptoleq Enhanced Assembly Language (CEAL). It is designed to aid in writing extended programs. Specifically, since the algorithms enabling universal computation on encrypted data have to be written in Cryptoleq, their complexity requires more expressive language than mere sequence of Cryptoleq instructions. CEAL allows writing symbolic notation avoiding explicit memory address manipulation. A CEAL compiler automates several procedures required for proper program execution, including, but not limited to, memory arrangement, address resolution, expression evaluation and macro definition substitution. It can also generate function G, as discussed in section IV-A. This section discusses the fundamental differences of CEAL compared to typical compilers[4].

CEAL provides syntax for supporting encrypted values, either directly hardcoding them in a program or generating them during compilation time. The compiler can also generate random encrypted values to be stored in memory or values to be used as memory addresses. So the program data can be easily initialized with encrypted values, as well as assigned to randomly selected memory addresses.

B. Arrays

Similar to other high level programming languages, CEAL can allocate memory arrays and access memory cells by either indexes or pointers. This is supported by Cryptoleq instructions via self-modifying code. In order to access memory indirectly – via a pointer – for reading or writing, an instruction is used to write the address into the corresponding operand of the target instruction. The CEAL compiler allocates the required amount of memory, initializes it and assigns the memory addresses. An array can be placed in a continuous block of memory with all elements sharing the s part of their addresses. CEAL syntax provides three options for placing an array: (i) in the default block of memory with $s = 0$, (ii) starting from an explicitly defined address, or (iii) at a random position. In the last two cases, the addresses of the array elements are indistinguishable from encrypted values. Array element access is facilitated using pointer arithmetic, and more specifically modification of t part: the program uses the unit value, as discussed in eq. 13, to navigate through the array elements.

[4]A detailed description of CEAL is out of scope of this paper and can be found at [34].

C. Library, Multiplication and Function G

Macros are introduced into CEAL to simplify repetition of instruction sequences, to make a local scope for names, and to build more complex constructions, such as functions. CEAL defines functions as special constructs built by small macros passing the execution to specific points in the program – function entries – while copying the return address in advance, so when the function finishes execution the control flow is passed back to the calling point. These macros also prepare the arguments and return values.

In order to reuse useful general macros, a library written in CEAL can be attached to a program by an include directive. This is similar, for example, to C include directive and C standard library linking. The CEAL library implements several macros and algorithms, most notably multiplication of encrypted and unencrypted values, function G, and equal operation on encrypted values. Specifically, the developed CEAL library defines two multiplication algorithms as functions: one for encrypted and one for unencrypted values. The reason the algorithms are different is that multiplication of unencrypted values can use conditional jump, hence can be implemented in a more optimal way. On the contrary, encrypted multiplication in Cryptoleq is implemented by code written in CEAL implementing the three algorithms of section IV-B .

The current implementation of CEAL compiler and the accompanying library uses a specific built-in directive to generate the function G. This directive takes three arguments: the first is a constant expression, and the other two are names of macros. The compiler replaces the directive with a list of macros constructed by binary expansion of the value of its first argument using those names. The decryption function (eq. 19) can be generated by providing the value $\phi(k\phi)_N^{-1}$ as the first argument and macro names of two operations corresponding to bit 0 and bit 1. The resulting code, a new macro, takes an argument x and performs the following operations: squares the argument x for bit 0; multiplies the accumulator by the argument x; and finally squares the argument x for bit 1. This simple procedure produces the open value of x. Finally, depending on the Leq test result on x (eq. 4), the developed function G returns either a newly generated random $\widetilde{0}$, or this newly generated $\widetilde{0}$ multiplied by the second argument.

D. Equal function

In addition to the four basic operations already discussed (i.e. addition, subtraction, multiplication and division), Cryptoleq programs require additional arithmetic operations over encrypted values. A prominent example is equivalence of two encrypted values that is homomorphic to equality of their

Algorithm 4 Private Information Retrieval

```
1: procedure PIR(input)
2:     array table[6][2]
3:     sum ← 0̃
4:     for i = 0 to 5 do
5:         key ← table[i][0]
6:         val ← table[i][1]
7:         sum ← sum + val · Equal(key, input)
8:     end for                                   ▷ Equal returns 0̃ or 1̃ (eq. 22)
9:     return sum
10: end procedure
```

plaintexts. Therefore, we define the equivalence comparison operation as follows, using tilde $\tilde{\cdot}$ to denote an encrypted value:

$$\text{Equal}(x, y) = \begin{cases} \widetilde{1}, & \text{if } \text{Dec}(x) = \text{Dec}(y) \\ \widetilde{0}, & \text{otherwise.} \end{cases} \tag{22}$$

This function can be easily expressed via function G:

$$\text{Equal}(x, y) = \tilde{1} - G(x - y, \tilde{1}) - G(y - x, \tilde{1}). \tag{23}$$

From a theoretical standpoint addition and multiplication operations are sufficient for Turing completeness. Hence, equality checks in the encrypted domain can be implemented using an bitwise OR operation over all bit encryptions of the difference of two numbers. In this work, the introduction of the Equal function is actually a practicality feature and not a theoretical requirement.

VI. Case Study Evaluation

A. PIR Example

Private Information Retrieval (PIR) is a classic example of applications which require private computation. In the simplest scenario, the user maintains an encrypted database on an untrusted server. At some point, the user desires to retrieve some data from the database, without revealing any information about the inquiry itself, data stored in the database or the result of the inquiry.

As a simple example, the database can be visualized as a table of key-value pair entries, e.g. {1:6, 2:7, 3:8, 4:9, 5:0, 6:1}. An inquiry to the database is a particular key and the expected output is the corresponding value. So, in this example, when the key 3 is requested the output returned to the user

TABLE I

COMPARISON OF DIFFERENT MEMORY MODELS AND OPERAND REPRESENTATIONS (IN SECONDS).

	TS	X	TS&ITS	X&IX
Sectors	3244	3236	79.3	57.0
Hash	3208	3201	61.9	16.3
Red-Black tree	3222	3195	61.8	15.1

should be `8`. As mentioned before, both the key input and the return result are encrypted. Therefore, PIR entails a brute-force search through all encrypted entries, secretly comparing database keys with the encrypted input, eventually returning the encrypted value when the keys match.

A straightforward algorithm of the PIR example appears in alg. 4. The flow of the algorithm ensures that (i) no data are ever decrypted; and (ii) there is no branching based on sensitive data. This algorithm is implemented in CEAL in a similar fashion: the program iterates through the entries of the table, accumulating the result of the multiplication of the value of each entry and the Boolean result of the equal function between the encrypted key and the encrypted input. Alg. 4 is ported to CEAL using 24 lines of code, and when compiled it occupies approximately 30000 memory cells.

B. Performance Analysis

1) Memory options: Section III-E describes the various options for memory organization and operand representation. Table I shows the execution time of the PIR example running with 32-bit N for various configurations. The times are shown in seconds. The table columns define the memory cell type: TS, X, TS & inverted TS (TS&ITS), and X & inverted X (X&IX). For the last two configurations, as mentioned in section III-E, the memory storage requirement doubles since inverted values are also stored along the original values. The rows of table I correspond to the selected memory type: Sector type, Hash map, or Red-Black tree.

The results show that Hash and Red-Black tree with inverted X memory outperform all other configurations. Therefore, for the rest of the results, we select Red-Black tree and X&IX as the baseline configuration. It should be emphasized that the poor performance of Sectors is expected, as Cryptoleq performance is evaluated in emulation mode. Sectors are expected to outperform the complex Hash and RB-trees when implemented in real hardware.

2) Size of N: The next set of results investigates the effect of the security parameter size to the performance of Cryptoleq. Runtime (in seconds) and number of instructions (in 10^5 units) are jointly presented in fig. 5, for security parameter N ranging from 16 to 1024 bits. Since the data type range,

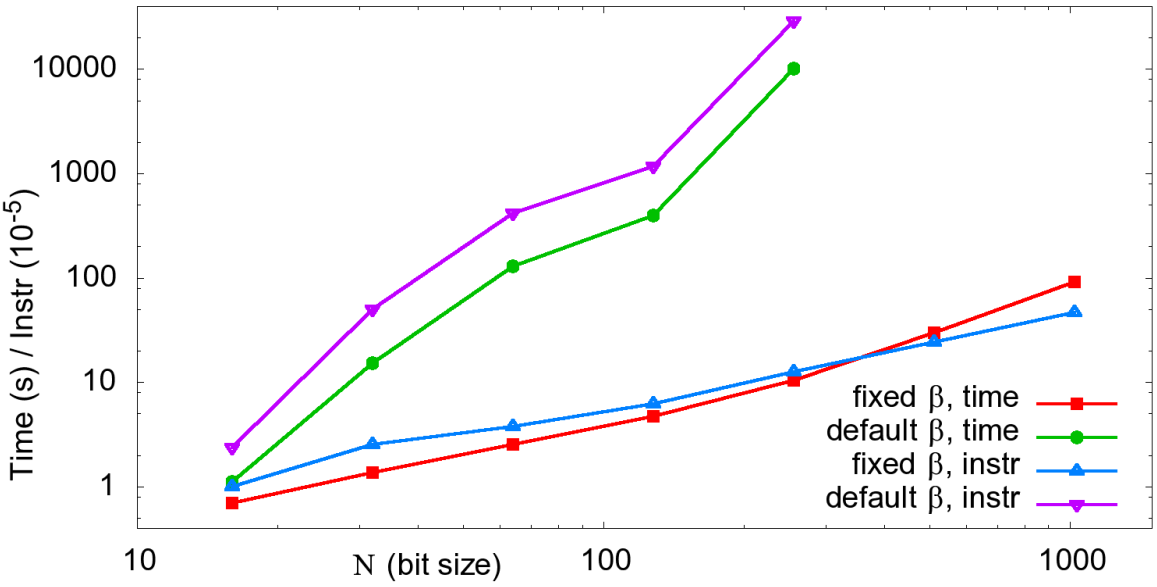

Fig. 5. Calculation time (in seconds) and Instruction count (in 10^5 units) vs N bit size.

defined by β (section III-E), affects the number of instructions executed, we perform two sets of experiments: default β, calculated to be exactly N bit size minus 3, and a restricted $\beta = 8$.

With regards to unlimited β, we can quickly observe that the instruction and time overhead is prohibitive even for small sizes of N. Fortunately, as data type values over 64-bit do not natively exist in modern computer architectures and have to be emulated through big number libraries, the infeasibility of using unlimited β does not affect the practicality of Cryptoleq.

Indeed, the fixed β results of fig. 5 showcase that when β is restricted to 8 bits, the performance and instruction overhead remain in the feasible range. An interesting observation is the difference in the rate of the increase of execution time to the number of instructions. Specifically, the runtime performance overhead increases roughly 3 times, while the instruction overhead grows approximately 2 times for each doubling of the size of N. This is attributed to the fact that operand sizes more than 64 bits require extensive use of big number libraries, affecting runtime performance given the same number of instructions.

3) β performance: The multiplication algorithm (section IV-B) encompasses two enclosed loops proportional to β; therefore, for each doubling of β, multiplication should take roughly 4 times longer. In this subsection we explore the performance implications of various β commonly used as native data type sizes, namely 8, 16, 32 and 64 bits. All experiments have been executed for security parameter size $N = 1024$, which is the minimum standard, not yet broken size. Table II presents the number of calls

TABLE II

PERFORMANCE IMPACT OF β SELECTION.

β	Calls to G	Instructions executed
8	498	4688612
16	1746	16696340
32	6546	64671092
64	25362	266779700

to function G, as well as the total number of instructions, for different β.

The quadratic dependency to β, along with some expected overhead, can be extracted from the results. Since N is the same for all four cases, the running time is exactly proportional to the total number of instructions. The results corroborate that data type sizes typically used in programming languages (8-64 bits) can be practically used in Cryptoleq for acceptable, with regards to their security, key sizes.

4) Mixed-mode simulation: The results presented in the previous subsections combined the overhead of all stages required for the execution of the PIR example, including compilation time and loading time. As expected, compilation and loading times are proportional to the size of the program. Table III breaks down the execution time to the individual stage overheads, and presents several statistics for different implementations of the PIR algorithm:

- `Secure`: The originally implemented PIR algorithm.
- `Mixed 1`: Key values of the databases are now open (i.e. function Equal compares two open values and outputs encrypted 1 or 0).
- `Mixed 2`: Open key values (`Mixed 1`), and removal of the secure multiplication (line 7) when key is not equal to input.
- `Open`: All values are open (not encrypted).

For the following experiments, security parameter size N is 1024 bits and β is fixed at 8 bits.

While the total execution time of the `Secure` mode is 91 seconds, only 35.8 seconds is the running time. Compilation time incurs significant overhead as the compiler is generating the encrypted representation of the operands and program constants. When a smaller number of encrypted values is used, compilation times decreases proportionally, as shown by the compilation overheads of `Mixed 1`, `Mixed 2` and `Open`. Similarly, loading time is also considerable for the `Secure` mode, due to the selected baseline configuration for memory cell representation: X&IX mode requires calculation of the inversion of encrypted memory cells during loading time. The overhead of the loading process remains

TABLE III

STATISTICS FOR DIFFERENT OPERATION MODES.

	Secure	Mixed 1	Mixed 2	Open
Compilation, s	26.5	8.8	8.8	5.1
Loading, s	28.6	21.5	21.5	8.8
Execution, s	35.8	34.1	8.8	0.4
Time total, s	91.0	64.5	39.2	14.3
G calls	498	486	81	0
Open mult	0	0	0	6
Secure equal	6	0	0	0
Secure mult	6	6	1	0
Instr open	95847	13014	2499	1803
Instr secure	3053659	2990930	8981	0
Instr mixed	1535106	1499425	5173	0
Instr total	4688612	4503369	16653	1803

significant for `Mixed 1` and `Mixed 2`, due to the oblivious way the processor is treating open and encrypted values (modular inversions for open values are also computed).

The difference in the execution times of the various versions of the PIR algorithm can be explained by the individual statistics shown in table III. Open instructions are classified as instructions where both A and B operands use open representation (i.e. their s part is zero), secure instructions have both operands encrypted while mixed instructions include one operand from each category. As expected, in the `Secure` mode, both secure multiplication and equal functions were called 6 times since the PIR database contains 6 entries. Furthermore, when the security requirements are reduced (modes `Mixed 1` and `Mixed 2`), number of function G calls and secure instructions decreases, allowing faster computation. The latter highlights the heterogeneous property of the presented Cryptoleq abstract machine, where security requirements can be fine-tuned and prioritized according to performance implications, all within the same abstract machine supporting seamless manipulation of encrypted and unencrypted operands.

VII. CRYPTOLEQ SECURITY CONSIDERATIONS

A. Design Objectives & Protection Strategy

A major objective for the development and design of the Cryptoleq framework is native support for data privacy when computation is outsourced to semi-trusted parties. Our strategy to address this requirement is to protect memory confidentiality using two methods: (i) *probabilistic encryption*, and (ii) *obfuscated*

decryption and re-encryption, depending on the type of mathematical operations required by the program. Thus, the following alternative protection modes are supported:

1) Encryption mode, where all memory locations are protected through Paillier PHE encryption, proven asymptotically secure as the security parameter increases. This mode is applicable when the mathematical operations in a program can reduce to subtraction of values. Prominent examples in this mode are algorithms for secure elections systems that tally votes.
2) Obfuscation mode, where all memory locations are also protected through encryption (identical to the previous one), and only in case a program needs to perform multiplication or comparison operations, obfuscated decryption and re-encryption is performed on the fly. Prominent examples in this mode are sorting and searching algorithms or Z-transforms on discrete-time signals.

For mixed-mode execution and backward compatibility, Cryptoleq also supports an unencrypted mode, where memory contents are stored in plaintext format.

B. Additional Discussion on Obfuscation Mode

As a general protection mechanism, obfuscation can be leveraged for creating public-key cryptosystems from secret-key ones, as well as for converting public-key cryptosystems to homomorphic ones [35]. For the former, one could allow public encryption but private decryption, by obfuscating the encryption function corresponding to the secret key. Similarly, for the latter, full homomorphism can be added by creating an obfuscated algorithm that decrypts, applies a function on plaintexts, and re-encrypts the result using the cryptosystem's key-pair.

An important negative result from [35], however, is that *strong* obfuscator programs do not generally exist, as there are inherently unobfuscatable functions. Conversely, constructing obfuscators for specific point functions is theoretically possible [36]. Obfuscation can also be approached heuristically, based on high performance on evaluation metrics, such as potency and resilience, at a reasonable cost. Indeed, it is possible to attain a realistic threat model where heuristic approaches are sufficient for most privacy-aware applications. In this work, we focus on a heuristic approach for obfuscation, and function G is designed under those assumptions.

C. Security Model & Limitations

Based on the description of our heterogeneous framework, we can define our assumed security model. In the case of encryption and obfuscation modes, all stored memory values of Cryptoleq programs are protected using a secure probabilistic PHE scheme. Following the security guarantees of that scheme, it is not generally possible to directly leak information about the plaintexts of encrypted values stored to

memory locations. Moreover, in our case we do not consider active adversaries [37], as data confidentiality is the asset to be protected and allowing malleability is a requirement for data manipulation. Instead of attacking the encryption scheme, a powerful adversary may attempt to analyze side-channel information of the Cryptoleq execution itself, such as IP trace patterns, memory access events or obfuscated decryptions, and extrapolate information about the protected values or the type of the running algorithm.

Naturally, the security guarantees of the underlying encryption scheme are not continuously applicable when function G is invoked to perform obfuscated decryption and re-encryption[5]. Indeed, obfuscation can provide heuristic guarantees compared to a provably secure cryptosystem, and thus our threat model is considering rational, semi-honest adversaries that are unable to deobfuscate function G routines or extract obfuscated bit-expansions of decryption key material stored inside the executable. An adversary may attempt to analyze the bit-expansion of $\phi(k\phi)_N^{-1}$ in an effort to recover ϕ and ultimately the PHE private key. Using a *random* blinding coefficient s, however, as mentioned in section IV-A, would make recovering ϕ from the bit expansion of $s \cdot \phi(k\phi)_N^{-1}$ significantly harder. It may also be sufficient for an adversary that holds a copy of the executable, to detect the entry point and argument locations of the obfuscated function G and reuse this code until after decryption happens; nevertheless, when the blinding coefficient s is used, adversaries can only retrieve random multiples of the actual plaintexts. Since the resilience of the underlying obfuscation is critical, Cryptoleq can leverage self-modifying code, on-the-fly code generation, and oblivious mixing of encrypted computation with unencrypted, to increase obfuscation protections. Specifically for the blinding coefficient s, self-modifying code allows runtime re-randomization of the bit expansion of $s \cdot \phi(k\phi)_N^{-1}$ with new random s, without affecting correctness.

As the previous discussion shows, there exist cases where deobfuscation could be a concern for the security of the system, as we employ heuristic obfuscation methods when provably secure encryption cannot be applied. Still, in the rational, semi-honest adversarial model, where cryptanalysis or deobfuscation threats combined with memory snooping are not applicable, the heuristic guarantees of obfuscation are sufficient for practical applications. Moreover, if the protected program does not use any multiplication or comparison operations, no obfuscated decryption routines are included in the executable and memory contents are protected with provably secure encryption.

VIII. Related Work

In the area of encrypted computation both theoretical and practical approaches have been proposed in the past. The authors of [38] have designed Ascend, which is a secure processor that employs Oblivious

[5]Re-encryption also requires a trustworthy source of randomness.

RAM to obfuscate memory accesses to encrypted external memory. Specifically, in Ascend, a block cipher accelerator decrypts all oblivious RAM read operations before instructions and data move into the processor caches, while data evicted from these caches are re-encrypted before oblivious memory storage. The processor contains a shared block cipher key and the chip itself is considered tamper-proof. Similarly, in Aegis [39], the authors propose a secure processor, which supports integrity attestation of executed software similar to Trusted Platform Modules (tamper-evident execution), and provides privacy protection for off-chip memory using non-malleable symmetric encryption (tamper-resistant execution). The Aegis chip contains a permanent private key that is necessary to decrypt other symmetric keys used to protect data and instructions within program binaries.

A theoretical approach on encrypted computation, leveraging FHE, is presented in [40], where the authors discuss an encrypted processing unit that supports static programs and provide proof of its correctness. Similarly, the authors of [14] use FHE circuits for outsourcing execution to the cloud, and further discuss the requirement of selective decryptions in order to detect termination of FHE encrypted programs (called the *termination problem*). In [41], the authors demonstrate general function evaluation and zero-knowledge protocols focusing on universally composable security, while [15] leverages homomorphic hashing for verifiable computation delegation.

Encrypted computation using secure containers has emerged as a protection strategy for commodity processors as well. In [42], the authors present new architectural features that allow the creation of encrypted *enclaves*, to provide privacy and integrity protection from potentially malicious processes running on the same system with elevated privileges. In this case, a memory encryption engine protects all traffic between the processor and the system main memory. In the same direction, the hardware enforced isolation discussed in [43] provides integrity protection even in case the operating system kernel is under attack, using a cryptographic coprocessor provisioned with secret keys during fabrication.

IX. Conclusions

In this paper, we have presented a new computational model based on the concept of single instruction architecture, able to execute programs whose instruction operands have been encrypted using Paillier PHE scheme. Universal computation is achieved by introducing a software function, which adds multiplication to the abstract machine's native addition and subtraction operations. This function is expressed using the only available instruction. We have also developed an enhanced assembly language to facilitate the development of complex programs, in addition to a compiler and an emulator. We evaluated this framework and our experimental results show that Cryptoleq incurs practical overhead when used with typical range of valid numbers.

Cryptoleq allows for several future improvements with regards to performance and security. The former can be improved through the introduction of high-radix representations (e.g. Montgomery), and advanced runtime techniques (such as automatic detection of open values to replace homomorphic multiplication with plaintext addition). Similarly, binary obfuscation is also a heavily researched topic and future work will explore the application of such techniques to Cryptoleq binaries to enhance the obfuscation offered by our framework.

RESOURCES

The executable files for the CEAL compiler and emulator, as well as sample Cryptoleq programs can be found at [34].

REFERENCES

[1] M. B. Taylor, "Bitcoin and the age of bespoke silicon," in *International Conference on Compilers, Architectures and Synthesis for Embedded Systems*. IEEE Press, 2013, p. 16.

[2] G. Woltman and S. Kurowski, "The great internet mersenne prime search," 2004.

[3] R. Chow, P. Golle, M. Jakobsson, E. Shi, J. Staddon, R. Masuoka, and J. Molina, "Controlling data in the cloud: outsourcing computation without outsourcing control," in *Cloud Computing Security Workshop*. ACM, 2009, pp. 85–90.

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

[5] N. G. Tsoutsos, C. Konstantinou, and M. Maniatakos, "Advanced techniques for designing stealthy hardware trojans," in *Design Automation Conference (DAC)*, 2014, pp. 1–4.

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

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

[8] K.-M. Chung, Y. Kalai, and S. Vadhan, "Improved delegation of computation using fully homomorphic encryption," in *Advances in Cryptology–CRYPTO 2010*. Springer, 2010, pp. 483–501.

[9] N. G. Tsoutsos and M. Maniatakos, "The HEROIC Framework: Encrypted Computation Without Shared Keys," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 34, no. 6, pp. 875–888, 2015.

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

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

[12] C. Gentry, "A fully homomorphic encryption scheme," Ph.D. dissertation, Stanford University, 2009.

[13] N. Smart and F. Vercauteren, "Fully homomorphic encryption with relatively small key and ciphertext sizes," Cryptology ePrint Archive, Report 2009/571, 2009, http://eprint.iacr.org/.

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

[15] D. Fiore, R. Gennaro, and V. Pastro, "Efficiently verifiable computation on encrypted data," in *Computer and Communications Security (CCS)*. ACM, 2014, pp. 844–855.

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

[17] R. Gennaro and D. Wichs, "Fully homomorphic message authenticators," in *Advances in Cryptology-ASIACRYPT 2013*. Springer, 2013, pp. 301–320.

[18] Y. Zhang, C. Papamanthou, and J. Katz, "Alitheia: Towards practical verifiable graph processing," in *Computer and Communications Security (CCS)*. ACM, 2014, pp. 856–867.

[19] M. Naehrig, K. Lauter, and V. Vaikuntanathan, "Can homomorphic encryption be practical?" in *Cloud Computing Security Workshop*. ACM, 2011, pp. 113–124.

[20] B. Schneier, "Homomorphic encryption breakthrough," [Online]. Available: http://www.schneier.com/blog/archives/2009/07/homomorphic_enc.html, 2009, (Accessed: 11/13/15).

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

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

[23] ——, "Helib: Design and implementation of a homomorphic-encryption library," [Online]. Available: https://github.com/shaih/HElib.

[24] A. Daly and W. Marnane, "Efficient architectures for implementing montgomery modular multiplication and RSA modular exponentiation on reconfigurable logic," in *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2002, pp. 40–49.

[25] T. Blum and C. Paar, "Montgomery modular exponentiation on reconfigurable hardware," in *Proceedings of the 14th IEEE Symposium on Computer Arithmetic*. IEEE, 1999, pp. 70–77.

[26] C. Mclvor, M. McLoone, and J. V. McCanny, "Fast Montgomery modular multiplication and RSA cryptographic processor architectures," in *Conference Record of the Thirty-Seventh Asilomar Conference on Signals, Systems and Computers*, vol. 1. IEEE, 2003, pp. 379–384.

[27] P. L. Montgomery, "Modular multiplication without trial division," *Mathematics of computation*, vol. 44, no. 170, pp. 519–521, 1985.

[28] Y. Hu, W. J. Martin, and B. Sunar, "Enhanced flexibility for homomorphic encryption schemes via CRT," *Proc. ACNS, Springer Verlag, LNCS*, pp. 93–110, 2012.

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

[30] S. Goldwasser and S. Micali, "Probabilistic encryption," *Journal of computer and system sciences*, vol. 28, no. 2, pp. 270–299, 1984.

[31] W. F. Gilreath and P. A. Laplante, *Computer Architecture: A Minimalist Perspective*. Springer, 2003, pp. 55–69.

[32] O. Mazonka, "Bit copying - the ultimate computational simplicity," *Complex Systems*, vol. 19, pp. 1–23, 2009.

[33] O. Mazonka and A. Kolodin, "A simple multi-processor computer based on subleq," *arXiv preprint arXiv:1106.2593*, 2011.

[34] "CEAL implementation examples," [Online]. Available: https://github.com/momalab/cealdemo/.

[35] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. Vadhan, and K. Yang, "On the (im) possibility of obfuscating programs," in *Advances in Cryptology–CRYPTO 2001*, 2001, pp. 1–18.

[36] H. Wee, "On obfuscating point functions," in *ACM Symposium on Theory of Computing*. ACM, 2005, pp. 523–532.

[37] P. Paillier and D. Pointcheval, "Efficient public-key cryptosystems provably secure against active adversaries," in *Advances in Cryptology-ASIACRYPT99*. Springer, 1999, pp. 165–179.

[38] C. W. Fletcher, M. v. Dijk, and S. Devadas, "A secure processor architecture for encrypted computation on untrusted programs," in *Scalable Trusted Computing Workshop*, 2012, pp. 3–8.

[39] G. E. Suh, D. Clarke, B. Gassend, M. Van Dijk, and S. Devadas, "Aegis: architecture for tamper-evident and tamper-resistant processing," in *International Conference on Supercomputing*. ACM, 2003, pp. 160–171.

[40] P. T. Breuer and J. P. Bowen, "A fully homomorphic crypto-processor design," in *Engineering Secure Software and Systems*. Springer, 2013, pp. 123–138.

[41] R. Canetti, A. Jain, and A. Scafuro, "Practical UC security with a global random oracle," in *Computer and Communications Security (CCS)*. ACM, 2014, pp. 597–608.

[42] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar, "Innovative instructions and software model for isolated execution," in *Hardware and Architectural Support for Security and Privacy Workshop (HASP)*. ACM, 2013, pp. 1–10.

[43] Apple Computer, "iOS Security (whitepaper)," [Online]. Available: https://www.apple.com/business/docs/iOS_Security_Guide.pdf, 2015.