Investigating the Application of One Instruction Set Computing for Encrypted Data Computation

Nektarios Georgios Tsoutsos[1] and Michail Maniatakos[2]

[1] Computer Science and Engineering,
New York University Polytechnic School of Engineering,
New York City, USA
nektarios.tsoutsos@nyu.edu
[2] Electrical and Computer Engineering,
New York University Abu Dhabi,
Abu Dhabi, UAE
michail.maniatakos@nyu.edu

Abstract. The cloud computing revolution has emphasized the need to execute programs in private using third party infrastructure. In this work, we investigate the application of One Instruction Set Computing (OISC) for processing encrypted data. This novel architecture combines the simplicity and high throughput of OISC with the security of well-known homomorphic encryption schemes, allowing execution of encrypted machine code and secure computation over encrypted data.

In the presented case study, we choose `addleq` as the OISC instruction and Paillier's scheme for encryption, and we extensively discuss the architecture and security implications of encrypting the instructions and memory accesses. Preliminary results in our implemented hardware–cognizant software simulator indicate an average execution overhead of 26 times for 1024–bit security parameter, compared to unencrypted execution of the same OISC programs.

Keywords: Encrypted processor, homomorphic encryption, Paillier, cloud computing.

1 Introduction

In the modern era of computing, the ability to process encrypted data and execute encrypted programs is widely regarded as the holy grail of cloud computing [31,34]. Whether in the form of a private cloud, or in public infrastructures, the confidentiality of the data or the confidentiality of the algorithm itself are of the highest value. Contemporary cloud service providers vouch themselves for the privacy of the user data, as well as the security of the computed results. This is essentially the only foundation for the users' trust. Hence, at their current form, cloud infrastructures are in practice prohibitive for applications where privacy is mandatory and the risk of compromise is unacceptable.

The latest solution to cloud security issues is to use an encryption scheme in order to make the private data unreadable by curious entities or even the cloud providers themselves. Not all encryption schemes, however, provide the ability to manipulate encrypted data and then decrypt to something meaningful; this property is called *homomorphism* and only specific schemes support it. Until recently, all known homomorphic encryption schemes supported only specific manipulations over encrypted data, and the ability to apply arbitrary manipulations remained unsolved. A very important step towards the solution was made in the recent years with the invention of Fully Homomorphic Encryption (FHE), and more specifically the Gentry scheme [13,35,37,16]. Since then, there has been significant progress on the FHE frontier: The authors of [5] propose an approach for secret program execution, based on fully homomorphic encrypted circuits. On the theoretical front, the authors of [6] provide theoretical proof of the correctness of an encrypted processing unit.

While several large corporations, like IBM, invest in such research [7], an encrypted processor based on FHE is not yet available. Some argue that these fully homomorphic schemes are *not yet* practical for everyday use or even for arbitrary manipulation of encrypted data, due to the tremendous overhead of the scheme [33,15]. The release of HELib [18,4] is a step towards the reduction of this overhead. The alternative approach would be to emulate the desired arbitrary manipulation of encrypted data (which is essentially equivalent to encrypted computation) using ordinary *partially homomorphic* encryption schemes, which are significantly faster and practical, compared to fully homomorphic schemes.

Even though homomorphic encryption seems to be very promising, none of the existing computer architectures can leverage the power of developed homomorphic schemes. A major reason for this is that existing computer architectures (including both `RISC` and `CISC`) are in fact designed towards efficiency and speed and not towards the security of the computation. Therefore, architecture specifications need to be redefined to include the security of computation.

To address this problem, our contribution in this paper is a novel idea for an encrypted computer architecture, able to perform arbitrary computation on encrypted data, using encrypted instructions, and thus preserving the security and privacy of both the data and the algorithm. The proposed solution in this paper is based on a Turing complete flavor of a One Instruction Set Computer (OISC) [25,22]. OISC architectures are very appealing for computation of encrypted data, since the stripped design provides great flexibility to incorporate support for homomorphic encryption. Without loss of generality, in this work we focus on `addleq`-based OISC [27,11], which requires a basic computational unit that supports the addition operation. This requirement drives the selection of an additive homomorphic encryption scheme as our candidate method for protecting data and program instructions.

To the best of our knowledge, this is the first effort towards an encrypted computer architecture that can be used in practice and is not based on the very expensive fully homomorphic encryption schemes. We combine the simplicity and high-throughput of OISC with an effective partially homomorphic encryption scheme, as presented in Section 2. Section 3 describes the architecture of the proposed `addleq` computer in the encrypted domain as well as theoretical and practical design considerations. The rest of this paper is organized as follows: Section 4 presents technical details and our experimental setup, while Section 5 provides performance results of the proposed solution. Section 6 features a discussion on the security properties of the proposed solution and future directions, followed by conclusions in Section 7.

2 The OISC Architecture

One Instruction Set Computers (also called Ultimate RISC computers) are architectures that support only one instruction. A careful selection of the aforementioned instruction can provide OISC architectures the capability of Turing–complete computation [12]. OISC computers are very simple but powerful; given their clean design and high throughput at certain configurations [26], they can be proper alternatives to ordinary RISC computers. In addition, having a single instruction renders the instruction operation code unnecessary, and thus only the instruction arguments are required to define a meaningful OISC program.

There are several types of OISC computers, depending on the single instruction supported. Common Turing–complete variants include the following: *Reverse subtract and skip if borrow*, *Subtract and branch unless positive*, *Plus one and branch if equal* and *Add and branch unless positive*. While these variants are seemingly different in terms of the operations performed by the single instruction, they share a common pattern: a simple mathematical operation (addition or subtraction) between instruction arguments, followed by a binary decision based on a condition. This straightforward format, as well as the existence of a single addition or subtraction, makes these OISC variants an excellent match to homomorphic encryption schemes, which are capable of preserving these mathematical operations in the encrypted domain.

In order to demonstrate how the OISC architecture can be modified to support encrypted data and encrypted instructions (using homomorphic encryption), and in order to investigate potential design or security issues, we focus on the `addleq` variant. `Addleq`'s single instruction has three arguments (namely A, B and C) and is defined as follows:

1. add the contents of two memory locations defined by arguments A and B,
2. put the results of the addition in the memory location defined by B, and
3. if the result of the addition is not positive, then jump to the instruction that starts at the memory location determined by argument C of the current instruction.

More formally, `addleq` performs the following:

```
Mem[B] = Mem[B] + Mem[A];
if Mem[B] ≤ 0 then goto C
else goto next instruction
```

From the description above, there are three important observations about `addleq`:

(a) `addleq` uses indirect addressing (i.e. instruction arguments are the memory addresses of the memory contents to be added, and the arguments are not actually added themselves),
(b) each instruction requires comparison with zero and
(c) arguments A, B and C should be grouped together.

2.1 Benefits of Using OISC

A major challenge towards an efficient architecture that processes encrypted data is the privacy of the algorithm and the instructions. Depending on the key, the encryption of the same instruction would be different each time. This means that any implementation of an encrypted computer (either in hardware or in software) would always require the decryption key in order to decrypt the instruction operation code (opcode) in order to decide what is the next operation (for example, to determine if the next instruction is a `read, write, add` etc). Providing the decryption key, however, defies the purpose of requiring encrypted computation.

The major contribution of this paper is the use of an OISC architecture to solve this problem; the benefit of this approach is that any implementation of the encrypted computer would not be required to decrypt the opcode of the next instruction: *All instructions are the same* and only the instruction arguments would be different (and still encrypted). Essentially, this proposed solution overcomes the problem of how to discriminate different instructions, while having the entire program in encrypted format. An attacker that may try to guess the algorithm (i.e. the instruction stream) being executed in the ultimate RISC computer, would gain no information, since all instructions are the same and the instruction arguments are already encrypted.

One may envision this architecture, as a standard Harvard architecture, where the instruction memory only contains the instruction arguments in encrypted format (no opcodes are required, since we only have a single instruction code), and a data memory that also contains encrypted data. This idea, however, also works if the architecture uses a unified memory where instruction arguments and data coexist and the code is self-modifying (i.e. instruction arguments and data are treated indistinguishably and instructions are allowed to modify other instruction arguments as well as data). Before analyzing the encrypted `addleq` architecture any further, however, it is necessary to provide basic background information on the homomorphic encryption scheme that is used to protect the privacy of the program data and instructions.

2.2 Homomorphic Encryption Background

Any encryption scheme that allows applying a specific function on encrypted data, so that the output of the function is an encryption of the result that comes from applying the same function directly on unencrypted data, is called *homomorphic* [23,36]. Essentially, the homomorphic property allows applying a function *after* encrypting the data, and the decryption of the result equals the output, if the function was applied on plaintext data. Some existing encryption schemes that support homomorphic properties are the following [14]: the RSA scheme [32], the El Gamal scheme [10], the Paillier scheme [30], the Goldwasser-Micali scheme [17], as well as recent schemes such as the Gentry scheme [15] and variants like the BGV scheme [4] and others [13,35,37].

From those schemes above, some of them only support a single function (either addition, or multiplication but not both) and are referred to as partially homomorphic, while only the Gentry scheme (and its variants) support both addition and multiplication and are referred to as fully homomorphic. The schemes that support a single function are either *additive* homomorphic (like Paillier and exponential El Gamal [8,24]) or *multiplicative* homomorphic (like RSA, standard El Gamal etc). Informally, this means that the applied function yields an output that is a preimage of the addition of the plaintexts (for the additive case) or a preimage of the product of the plaintexts (for the multiplicative case). More formally, homomorphism is defined as follows [23]:

$$Encrypt[m1] \diamond Encrypt[m2] = Encrypt[m1 \circ m2] \tag{1}$$

where ($\circ$) usually is addition (+) or multiplication ($*$) depending on the scheme.

In this paper, without loss of generality, we focus on `addleq` OISC, which uses the addition operation for the purposes of performing arbitrary computation. Addition operation in the encrypted domain is supported by partially homomorphic schemes like the Paillier and exponential El Gamal schemes, as well as fully homomorphic variants of the Gentry scheme. As mentioned earlier in this paper, because fully homomorphic schemes have tremendous overheads (several orders of magnitude [15,33]) and since the exponential El Gamal scheme suffers from high decryption overhead [8,24], we focus on the Paillier encryption scheme.

2.3 Paillier Scheme

The Paillier scheme is the first efficient additive homomorphic scheme [30]. The Paillier scheme is based on the *decisional composite residuosity assumption*, which states the following:

Given a composite number n and an integer z, it is *hard* to decide whether there exists y such that

$$z \equiv y^n \pmod{n^2} \tag{2}$$

The Paillier scheme is a public key cryptographic scheme and is defined as follows [23]:

Let p and q be two large prime numbers of equal length, randomly and independently chosen of each other, and $n = pq$ the product of these numbers (where $\log_2 n$ is the security parameter of the scheme); the knowledge of p and q would be part of the private key and n would be part of the public key. In addition, let

$$\phi(n) = (p-1)(q-1) \tag{3}$$

be Euler's Totient function for n. Then, using the Paillier scheme, the encryption of a plaintext message "m", and the decryption of ciphertext "c" are defined as follows:

$$Encrypt[m] = (n+1)^m * r^n \pmod{n^2} \tag{4}$$

for random r in the multiplicative group $\mathbb{Z}_n^*$.

$$Decrypt[c] = \frac{(c^{\phi(n)} \pmod{n^2}) - 1}{n} \phi(n)^{-1} \pmod{n} \tag{5}$$

where $\phi(n)^{-1}$ is the modular multiplicative inverse of $\phi(n)$ in $\mathbb{Z}_n^*$.

The scheme supports the following additive homomorphic properties:

$$Encrypt[m1] * Encrypt[m2] = Encrypt[m1 + m2 \pmod{n}] \tag{6}$$

which essentially means that the multiplication of the encryptions of two messages is a preimage of the encryption of the addition of these messages. Therefore, the encryption of the sum of two plaintexts equals the result of multiplying the ciphertexts.

3 Addleq in the Encrypted Domain

As already discussed in the introduction section of this paper, our goal is to define an architecture for a computer that is capable of performing computations on encrypted data, as well as executing encrypted instructions, in order to protect the privacy of the algorithm and the program itself. We investigate an architecture that exploits the simplicity of `addleq` OISC computer as well as the additive homomorphic properties of the Paillier encryption scheme.

3.1 Basic Components of the Design

A starting point in designing an encrypted `addleq` computer would be to determine all necessary components. Since `addleq` supports self–modifying code by design, a single *main memory* is necessary. This main memory would contain both data and instructions, and would incorporate a memory control unit responsible for addressing, as well as for reading and writing bytes. In addition, the design should have an *ALU unit*, that will be used for performing *modular multiplication* in the encrypted domain (equivalent to normal addition in the unencrypted domain). Program execution is controlled by a *control finite state machine (FSM)* responsible for execution and fetching operations, as well as the *program counter*. For storing memory data and addresses, a *temporary register* is also necessary. Since `addleq` needs to branch if the ALU output is not positive, a *sign identification unit* is required as well.

These basic components are presented in Fig. 1 and the `addleq` datapath is described in Section 3.2.

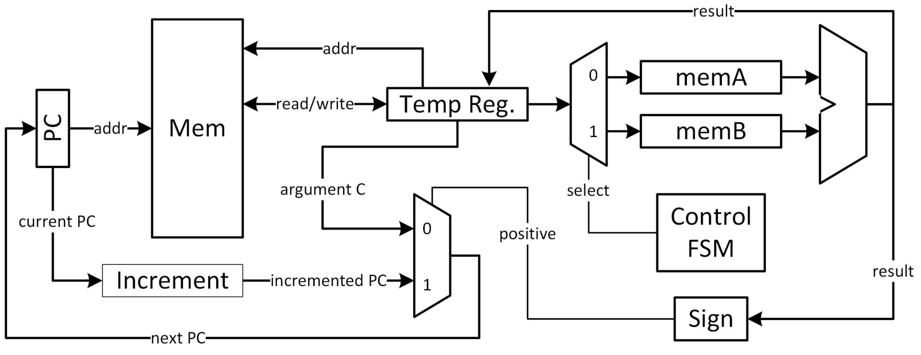

Fig. 1. Basic components and datapath of the `addleq` OISC computer

3.2 Addleq Datapath

The datapath of *addleq* is inherently simple, as seen in Fig. 1. In order to execute a single `addleq` instruction (that has three arguments), these steps are followed:

i. Initially, the program counter address is used to read from main memory and the returned value is stored to the temporary register (fetching of argument A).
ii. The contents of temporary register are used as the address sent to main memory for reading, and the returned value is stored back to the temporary register (fetching of *memA*, which is the ALU input referenced in memory by argument A).
iii. Through a demultiplexer controlled by the control FSM, the contents of the temporary register are stored in the *first* input register of the ALU.
iv. The program counter value goes through the increment unit and a multiplexer, and the address of the next memory location is written back to the program counter.
v. The program counter address is used to read from main memory and the returned value is stored to the temporary register (fetching of argument B).
vi. The contents of the temporary register are used as the address sent to main memory for reading, and the returned value is stored again to temporary register (fetching of *memB*, which is the ALU input referenced in memory by argument B).
vii. Through a demultiplexer controlled by the control FSM, the contents of the temporary register are stored in the *second* input register of the ALU.
viii. The modular multiplication ALU uses the two register inputs and generates a result that is then sent back to the temporary register as well as to the sign identification unit.
ix. The contents of the temporary register are sent to the main memory to be stored at the same location where *memB* was fetched from, as the program counter still points to that address (location referenced in memory by argument B).

x. The program counter value goes through the increment unit and a multiplexer, and the address of the next memory location is written back to the program counter.
xi. The program counter address is used to read from main memory and the returned value is stored to the temporary register (fetching of argument C).
xii. The sign identification unit determines if the last ALU result is positive and configures the program counter multiplexer. The program counter value goes through the increment unit and the multiplexer selects between the incremented program counter and the contents of the temporary register (essentially argument C) and the selected value is written back to the program counter.

3.3 Design Challenges

Encrypted Memory Addressing. Since instruction arguments and data are in a unified memory, then *memory addressing should also be encrypted.* Instruction arguments and program counters would obviously require to reference data locations or other arguments (for self-modifying code, as discussed in Section 2.1), and the same program should be oblivious of the encryption key (i.e. every encryption of the instruction arguments and the data should be able to reference any memory location). Since any implementation of the encrypted computer would not have access to the decryption key, the architecture should work on encrypted memory addressing. Our proposed architecture, however, also addresses this concern and allows encrypted addressing and encrypted program counters.

Matching Instruction Arguments. Porting `addleq` to the encrypted domain, raises another issue: Since `addleq` uses a single memory space for instruction arguments and data (indistinguishable to each other), it is required to match arguments A, B and C. This issue can be solved by storing inside the program memory the encryptions of each element (either instruction argument or datum) along with the encrypted address of the next encrypted element. Essentially, each element also points to the next one, by using encrypted references.

For example, without loss of generality, we assume that the program counter contains the encrypted address of a memory location that contains instruction argument A and the encrypted address of the next element (i.e. the address of instruction argument B). The `addleq` computer retrieves from memory the contents of the memory location pointed by argument A and then the program counter loads the encrypted address of the next element. The `addleq` computer then retrieves the memory contents pointed by argument B similarly. The `addleq` ALU multiplies the retrieved memory contents (pointed by arguments A and B) and if Paillier encryption has been used, due to additive homomorphism this corresponds to simply adding these two values in the unencrypted domain. The result is stored to the memory location pointed by argument B, and the `addleq` computer decides if a branch is required or not. If a branch is required, the program counter becomes equal to argument C, otherwise the program counter becomes equal to the address of the next argument.

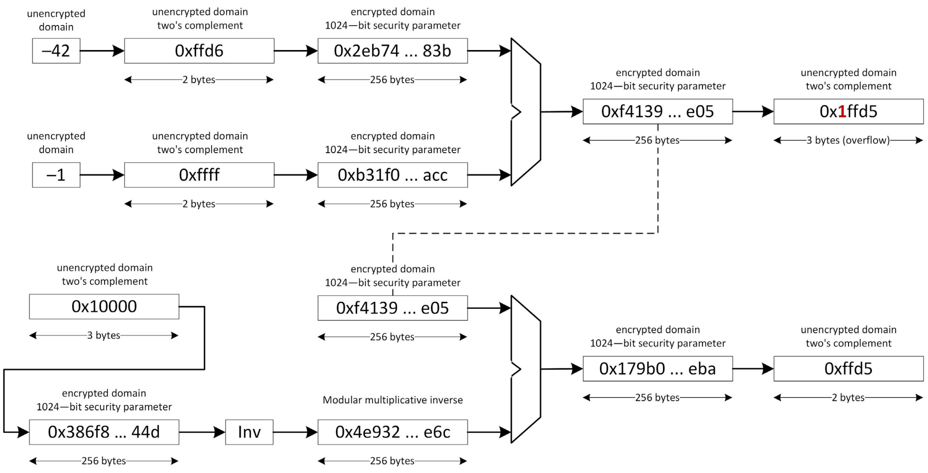
unencrypted domain
−42
unencrypted domain two's complement
0xffd6
2 bytes
encrypted domain 1024—bit security parameter
0x2eb74 ... 83b
256 bytes
unencrypted domain
−1
unencrypted domain two's complement
0xffff
2 bytes
encrypted domain 1024—bit security parameter
0xb31f0 ... acc
256 bytes
encrypted domain 1024—bit security parameter
0xf4139 ... e05
256 bytes
unencrypted domain two's complement
0x1ffd5
3 bytes (overflow)
unencrypted domain two's complement
0x10000
3 bytes
encrypted domain 1024—bit security parameter
0x386f8 ... 44d
256 bytes
Inv
Modular multiplicative inverse
0x4e932 ... e6c
256 bytes
encrypted domain 1024—bit security parameter
0xf4139 ... e05
256 bytes
encrypted domain 1024—bit security parameter
0x179b0 ... eba
256 bytes
unencrypted domain two's complement
0xffd5
2 bytes

Fig. 2. Overflow correction for homomorphic addition of representations of negative numbers

Addition Overflow Detection. In this paper, for our `addleq` case study, we are investigating 16–bit memory addressing for the unencrypted programs. This implies that the size of each memory location would be 16 bits, equal to the size of a memory address. For representing negative numbers, the standard two's complement approach is used and the proposed range of supported numbers is set from -2^{15} to $(2^{15} - 1)$. One reason why the use of two's complement to represent negative numbers is mandatory, is the fact that Paillier's scheme only supports the encryption of *positive* values (for example, using two's complement for a range of 2^{16} numbers, it means that –1 is represented as $(2^{16} - 1)$).

Due to subtle homomorphic addition properties, however, adding the representations of two negative numbers in the encrypted domain, would cause a result *out of range*. For example, adding –42 with –1, which corresponds to adding $(2^{16} - 42)$ with $(2^{16} - 1)$, would result to the encryption of $(2^{17} - 43)$ instead of the encryption of $(2^{16} - 43)$ (the representation of signed number –43). This inconsistency occurs because in the encrypted domain the range of numbers is much higher (e.g. with 1024–bit security parameter size, the encrypted range is 2048 bits), compared to 2^{16} in the unencrypted domain. The issue is addressed, however, by homomorphically subtracting the encryption of 2^{16} (or equivalently, by adding the *modular multiplicative inverse* of the encryption of 2^{16}, which is always the same and can be precomputed) to get the correct result (i.e. $(2^{16} - 43)$ in the previous example). An elaboration of this example is shown in Fig. 2.

In order to detect an overflow, an *overflow lookup table* is introduced. This lookup table matches the encryption of numbers from 0 to $(2^{17} - 1)$ with one bit that indicates "over $(2^{16} - 1)$" or "not over $(2^{16} - 1)$". If the ALU result matches

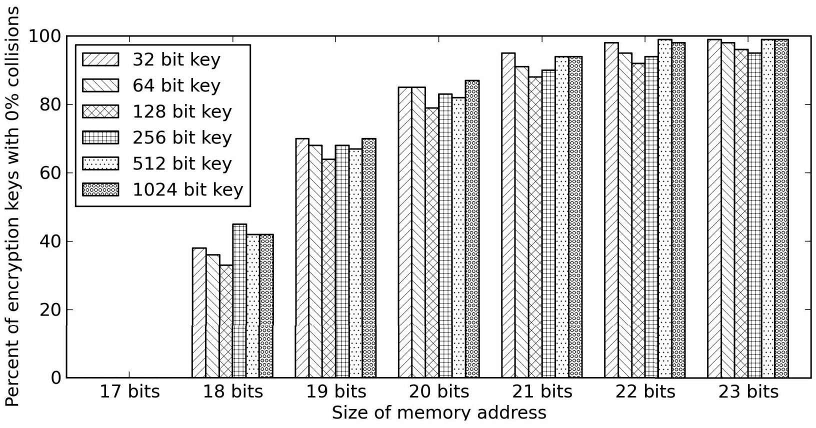

Fig. 3. Percentage of encryption keys with 0% collisions in 2^{17} encryptions for given memory addressing sizes

a value in this overflow lookup table, then a correction by "subtracting" 2^{16} is performed.

Memory Addressing Size. Given a security parameter of 1024 bits and encrypted memory addressing, the actual implementation of the proposed architecture would require memory addresses of 2048 bits, as shown by Eq. 4. Such memory, besides its prohibitive cost, is also unnecessary as in the unencrypted domain we only need 16-bit memory addressing. In this work, we propose to use a fraction of this 2048 bit address to successfully identify the requested memory address. Specifically, we seek to identify how many bits are required to discriminate memory addresses with high probability and for a high percentage of different encryption keys. As previously discussed, for 16–bit memory addressing in the unencrypted domain, a total of 2^{17} unique encryptions are addressed in the overflow lookup table, while up to 2^{16} unique encryptions are addressed in the main program memory.

Fig. 3 demonstrates the number of memory address bits required to discriminate the maximum number of unique encryptions (i.e. 2^{17}) with 0% collisions, for several encryption key sizes (i.e. security parameter sizes). Depending on the Paillier encryption security parameter size, the actual encrypted values have twice as many bits (e.g. for 256–bit security parameter, the encrypted value is 512 bits). As the diagram demonstrates, if we use memory addresses of 22 bits, we can accommodate all 2^{17} different encryptions for at least 90% of the encryption keys used (the analysis is based on a random sample of *100 keys* for each key size, and for confidence level 95% the confidence interval is ≤ 9.8). Of course, in this case there is about 10% probability that this addressing size would not be enough for a specific encryption key, and in this scenario memory re-encryption *with a different key* would be required.

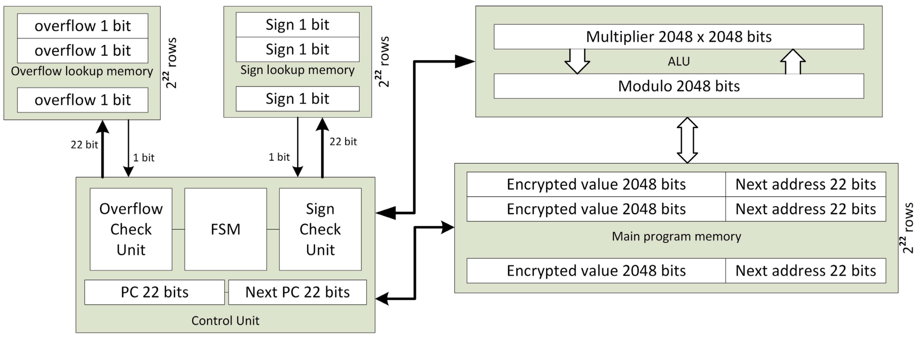

Fig. 4. High level view of additional units of the *addleq* encrypted computer to address sign identification and overflow detection in the encrypted domain (security parameter 1024–bits)

Jump Decisions in the Encrypted Domain. An additional problem, already mentioned in the previous paragraphs, is the problem of determining if the result of the ALU modular multiplication is less than or equal to zero (i.e. not positive). Since the computation is performed on encrypted data, there is no straightforward way of comparing an encrypted value with zero without decrypting it. There are proposed schemes that allow *order-preserving encryption* [1], but these schemes are not additive homomorphic and the Paillier scheme is not order-preserving either. Furthermore, if an encryption scheme allows comparing an encrypted value with zero, this would automatically allow decryption, simply by doing a *bit–by–bit binary search*. To solve this problem, in this work we propose the use of a *sign lookup table* that contains that mathematical sign for the encryptions of a range of numbers. Essentially, this table matches encrypted values with the sign of the corresponding decryptions.

The proposed sign lookup table provides matching for the encryptions of 16-bit values with the corresponding sign (0 for less or equal zero, 1 for positives). In theory, this lookup table would require memory address size equal to the encryption of each number in range from 0 to $(2^{16} - 1)$, which depending to the Paillier security parameter may be up to 2048 bits long. Instead, we propose a truncation down to 22 bits (similar to the overflow lookup table addressing truncation described earlier), since as seen in Fig. 3 this is sufficient for discriminating 2^{17} different values for the vast majority of different random encryption keys.

Fig. 4 features a high level view of the encrypted addleq architecture, including the additional sign and overflow lookup memories.

Memory Space Requirements. From the above description it becomes evident why a limitation in the possible encrypted values is required. Allowing 2^{16} possible values to be encrypted, ultimately requires two additional memories (sparsely filled) of 22 bit address size each, to support the encrypted addleq functionality. So, the addleq main memory (the one that contains encrypted instruction arguments and data) can only contain the encryptions of values up to $(2^{16} - 1)$, as well as addressing for up to $(2^{16} - 1)$ instruction arguments. This is not a significant problem, however, since addleq programs by nature use larger values progressively as the program size increases, and the vast majority of programs can easily fit in this proposed range.

In addition, it should be stressed out that the main memory of addleq needs to have adequate size to fit the encryptions of data, as well as instruction arguments A, B and C, along with the truncated address for the next element; in practice, if the Paillier security parameter is 1024 bits (so each encrypted value would be 2048 bits), and the additional bits for the truncated address of the next element are 22, this adds to 2060 bits in each memory line (each line is addressed with 22 bits). In order to execute encrypted programs, the owner of the program should encrypt (using Paillier's scheme) the main program memory (which is given by the compiler of the addleq assembly), as well as provide the two mandatory lookup tables for signs and overflows; all three memories are specific to the encryption key used each time.

4 Experimental Setup

In order to evaluate the performance of the proposed architecture, a simulator of the proposed encrypted addleq architecture has been developed. All experiments were performed using Python 2.7.4 on a virtual 64–bit Ubuntu 13.04 host running at 2.6GHz ($2 *$ i7–3720QM Intel cores) with 2GB of memory. For the implementation of Paillier's homomorphic encryption scheme, an openly available educational Python library from [21] was used.

For our experiments, four `addleq` assembly programs have been used from [27]; these programs are written directly in assembly language and after being converted to machine code, they have been homomorphically encrypted for execution in the simulator (using different security parameter sizes). Furthermore, for hardware performance figures, we assumed a hardware implementation running at 200MHz (for example, in an FPGA) and the following estimates have been used:

(a) *memory access delay* has been estimated to less than 100ns [19] for memories up to 16GB (which corresponds to 20 clock cycles at 200Mhz),
(b) *modular multiplication* has been estimated to $(h + 3)$ cycles for argument size h [9] (which corresponds to 2051 clock cycles for 2048–bit arguments using security parameter $n = 1024$ bits) based on *Montgomery* algorithm [29,2,28], and
(c) *addition* (using *Kogge-Stone* fast adder design) for 16–bit unencrypted arguments (used for comparison) is estimated to 6ns [20] (which requires 2 clock cycles at 200Mhz) .

Table 1. Simulated execution clock cycles (security parameter 1024 bits)

Benchmark	Unencrypted	Encrypted	Overhead
Program 1	$4.44 * 10^5$	$1.35 * 10^7$	30 x
Program 2	$6.28 * 10^5$	$1.67 * 10^7$	27 x
Program 3	$3.78 * 10^2$	$6.99 * 10^3$	19 x
Program 4	$4.10 * 10^4$	$1.09 * 10^6$	27 x

Table 2. Average overhead for different security parameter sizes

Security Parameter	Average Overhead
32 bits	3 x
64 bits	4 x
128 bits	5 x
256 bits	8 x
512 bits	14 x
1024 bits	26 x

5 Results

As a proof of concept, preliminary results generated using the developed simulator are presented in Tables 1 and 2. The simulator is parameterized to compare the performance of normal (unencrypted) `addleq` with the performance that our proposed encrypted `addleq` design would have had in a standard FPGA running at 200MHz, using the figures presented in Section 4. Even though memory access times are not affected by different security parameter sizes, modular multiplications have different delays, depending on the size of the arguments. Table 2 presents the average (over all four assembly programs used) encrypted execution overhead depending on the security parameter size. The results show that as the security parameter size increases, the impact of the modular multiplication operations in the encrypted OISC computer becomes higher, and the overall overhead is *26 times* for security parameter 1024 bits.

Table 1 provides information on the number of clock cycles required for encrypted program execution, compared to unencrypted execution. These results indicate that encryption requires about 2 orders of magnitude more clock cycles, primarily due to the multicycle ALU operations and memory lookup operations.

6 Discussion on Security

The proposed encrypted computer is designed to solve the problem of executing programs and manipulating data in the cloud. It is designed to address honest but curious cloud service providers that would support the proposed architecture but may try to look inside the processor. Our goal is to protect the confidentiality of the execution steps as well as the data.

The security of the system is based on the security assumptions of the Paillier scheme; the proposed architecture works with any program that is the result of `addleq` assembly compilation. However, since the Paillier scheme is a probabilistic scheme, it incorporates a random *helper value* "r" in the encryption of each plaintext. This helper value is necessary in order to have different encryptions each time, for the same plaintext, and essentially have semantic security in the cryptographic sense of *Indistinguishability against Chosen Plaintext attacks (IND-CPA)* [23]. OISC computers, however, cannot be probabilistic; it would not be possible to construct an OISC computer where the same plaintext has different encryptions, all at the same time: when that computer generates a result that would be used as a reference to a memory location, this reference always needs to be the same, since there would be only one actual location in memory. So, it is not possible to use Paillier homomorphic encryption in a probabilistic fashion, since intermediate values computed by `addleq`, to be used as memory address references, should always be the same.

In practice, this last observation means that for our proposed encrypted `addleq` architecture, the used Paillier's scheme implementation has to be slightly modified in order to produce the same encrypted value for the same input (i.e. become partially deterministic). In theory, this essentially makes the processor susceptible to Chosen–Plaintext Attacks (i.e. the semantic security property is partially lost), and the scheme has the same security properties as textbook RSA [3,23], which is also semantically non-secure. However, the threat of a Chosen–Plaintext Attack is not a major concern in our context of cloud computing, since the encryption of all values is performed by the program owner who knows all public key parameters (as well as private key parameters) to be able to encrypt arbitrary values; using Paillier's encryption, only the value of n (which is one half of the public key) is required by the addleq computer ALU to perform modular multiplication, and only this half needs to be revealed (in the currently used Paillier's scheme implementation, the second half of the public key is correlated with the first half, however, in general uncorrelated halves can be used as well). Thus, no third party or even the cloud provider has access to the entire public key in order to calculate encryptions of known values, and to launch a Chosen–Plaintext Attack to the processor and the encrypted memory contents.

Another concern comes from the fact the program owner provides 2 lookup tables with information about the sign of 2^{16} encrypted values as well as information if an encrypted value is larger or smaller than 2^{16}. In the strict cryptographic sense, the first lookup table reveals 1 bit of the plaintext and the second table also reveals "some" information. However, the information revealed in indeed very little, compared to the entropy of the encrypted value, and in practice the confidentiality of the information is not threatened. Of course revealing even 1 bit is less than optimal, but any Turing complete computer needs to make runtime decisions, and in this case we use the minimum amount of information to perform this mandatory task.

Finally, one concern relates to the order with which the encrypted instruction arguments are provided. If these arguments are given in sorted ordering, an attacker could guess the program counter sequence and thus launch a Chosen–Plaintext Attack. This means that if the program counter starts from (encryption of) address 0x0000 for argument A and then goes to (encryption of) address 0x0001, an attacker could potentially guess this and find the encryption of plaintext "1" and ultimately generate a *codebook* (using homomorphism, the attacker finds $1 + 1$, $1 + 1 + 1$ etc). To prevent this, the proposed architecture supports "spaghetti" memory: having that each memory location is accompanied by a reference to the next address, these addresses can be randomly chosen, without loss of generality. Essentially, the program can start from (encryption of) address zero and jump to random locations in a spaghetti code fashion; the trail is preserved since each location points to the next one, but this looks unintelligible to any third–party observer.

6.1 Future Directions

In this work we present a novel approach to solve the problem of encrypted computation for protecting the confidentiality of the program and the data. Without loss of generality of the proposed solution, we use a publicly available implementation of Paillier's scheme in Python [21]. Because this particular implementation is not recommended for "production" use (due to the quality of prime numbers generated and the randomness sources used), future work will include cryptographically safe implementations of Paillier's scheme. Our proposed architecture would work with any other implementation of the Paillier's encryption scheme, provided that the implementation is slightly modified to use deterministic encryptions (i.e. for a specific plaintext, the same helper value r is used each time, instead of a random one each time).

In addition, in this paper, for simplification of a certain subtleties of homomorphic multiplication, the same r value was used for every plaintext. The proposed architecture, however, supports having a different helper value r for each different plaintext, with minimal modifications; this option, that provides extra security, would be explored in future work.

Future directions also include exploring other OISC variants, like `subleq`; using a different variant requires only minor technical modifications, as the proposed architecture is generic. Furthermore, performance improvements using cache memories, pipelining and incorporating more than one execution units are also part this continued investigation. A hardware implementation of the proposed architecture would provide more conclusive performance figures.

7 Conclusion

In this paper we presented a novel idea for an encrypted computer architecture, capable of performing computations on encrypted data as well as executing encrypted programs written in addleq machine code. Our key contribution is a new design that combines OISC computer design with homomorphic encryption, which is an important step towards achieving privacy in cloud computing. Experimental results corroborate that the proposed architecture has the potential to be an effective solution, incurring an average overhead of 26 times compared to unencrypted OISC computation.

References

1. Agrawal, R., Kiernan, J., Srikant, R., Xu, Y.: Order preserving encryption for numeric data. In: Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data, pp. 563–574. ACM (2004)
2. Blum, T., Paar, C.: Montgomery modular exponentiation on reconfigurable hardware. In: Proceedings of the 14th IEEE Symposium on Computer Arithmetic, pp. 70–77. IEEE (1999)
3. Boneh, D., Joux, A., Nguyen, P.Q.: Why textbook ElGamal and RSA encryption are insecure. In: Okamoto, T. (ed.) ASIACRYPT 2000. LNCS, vol. 1976, pp. 30–43. Springer, Heidelberg (2000)
4. Brakerski, Z., Gentry, C., Vaikuntanathan, V.: (Leveled) fully homomorphic encryption without bootstrapping. In: Proceedings of the 3rd Innovations in Theoretical Computer Science Conference, pp. 309–325. ACM (2012)
5. Brenner, M., Wiebelitz, J., von Voigt, G., Smith, M.: Secret program execution in the cloud applying homomorphic encryption. In: Proceedings of the 5th IEEE International Conference on Digital Ecosystems and Technologies Conference (DEST), pp. 114–119. IEEE (2011)
6. Breuer, P.T., Bowen, J.P.: Typed assembler for a RISC crypto-processor. In: Barthe, G., Livshits, B., Scandariato, R. (eds.) ESSoS 2012. LNCS, vol. 7159, pp. 22–29. Springer, Heidelberg (2012)
7. Cooney, M.: IBM touts encryption innovation (2009), `http://www.computerworld.com/s/article/9134823/IBM_touts_encryption_innovation?taxonomyId=152&intsrc=kc_top&taxonomyName=compliance`
8. Cramer, R., Gennaro, R., Schoenmakers, B.: A secure and optimally efficient multi-authority election scheme. European Transactions on Telecommunications 8(5), 481–490 (1997)
9. Daly, A., Marnane, W.: Efficient architectures for implementing montgomery modular multiplication and RSA modular exponentiation on reconfigurable logic. In: Proceedings of the 2002 ACM/SIGDA Tenth International Symposium on Field-Programmable Gate Arrays, pp. 40–49. ACM (2002)
10. ElGamal, T.: A public key cryptosystem and a signature scheme based on discrete logarithms. IEEE Transactions on Information Theory 31(4), 469–472 (1985)
11. Esolangs: Addleq Turing complete OISC language, `http://esolangs.org/wiki/Addleq`
12. Esolangs: One Instruction Set Computer, `http://esolangs.org/wiki/OISC`
13. Fan, J., Vercauteren, F.: Somewhat practical fully homomorphic encryption. IACR Cryptology ePrint Archive, 2012 144 (2012)
14. Fontaine, C., Galand, F.: A survey of homomorphic encryption for nonspecialists. EURASIP Journal on Information Security 2007 (2007)
15. Gentry, C.: A fully homomorphic encryption scheme. Ph.D. thesis, Stanford University (2009)

16. Gentry, C.: Fully homomorphic encryption using ideal lattices (2009)
17. Goldwasser, S., Micali, S.: Probabilistic encryption & how to play mental poker keeping secret all partial information. In: Proceedings of the Fourteenth Annual ACM Symposium on Theory of Computing, pp. 365–377. ACM (1982)
18. Halevi, S., Shoup, V.: Design and implementation of a homomorphic-encryption library (2012)
19. Hennessy, J.L., Patterson, D.A.: Computer architecture: a quantitative approach, pp. 72, 96–101. Elsevier (2012)
20. Hoe, D.H., Martinez, C., Vundavalli, S.J.: Design and characterization of parallel prefix adders using FPGAs. In: 2011 IEEE 43rd South eastern Symposium on System Theory (SSST), pp. 168–172. IEEE (2011)
21. Ivanov, M.: Pure Python Paillier homomorphic cryptosystem (2011), `https://github.com/mikeivanov/paillier`
22. Jones, D.W.: The ultimate RISC. ACM SIGARCH Computer Architecture News 16(3), 48–55 (1988)
23. Katz, J., Lindell, Y.: Introduction to modern cryptography. CRC Press (2008)
24. Lange, A.: An overview of homomorphic encryption (2011), `http://www.cs.rit.edu/~arl9577/crypto/alange-presentation.pdf`
25. Mavaddat, F., Parhami, B.: URISC: the ultimate reduced instruction set computer. Faculty of Mathematics. University of Waterloo (1987)
26. Mazonka, O., Kolodin, A.: A simple multi-processor computer based on subleq. arXiv preprint arXiv:1106.2593 (2011)
27. Mazonka, O.: Addleq (2009), `http://mazonka.com/subleq/`
28. Mclvor, C., McLoone, M., McCanny, J.V.: 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, pp. 379–384. IEEE (2003)
29. Montgomery, P.L.: Modular multiplication without trial division. Mathematics of Computation 44(170), 519–521 (1985)
30. Paillier, P.: Public-key cryptosystems based on composite degree residuosity classes. In: Stern, J. (ed.) EUROCRYPT 1999. LNCS, vol. 1592, pp. 223–238. Springer, Heidelberg (1999)
31. Parann-Nissany, G.: The holy grail of cloud computing – maintaining data confidentiality (2012), `http://www.wallstreetandtech.com/technology-risk-management/the-holy-grail-of-cloud-computing-maint/240006774`
32. Rivest, R.L., Shamir, A., Adleman, L.: A method for obtaining digital signatures and public-key cryptosystems. Communications of the ACM 21(2), 120–126 (1978)
33. Schneier, B.: Homomorphic encryption breakthrough (2009), `http://www.schneier.com/blog/archives/2009/07/homomorphic_enc.html`
34. Simonite, T.: Computing with secrets, but keeping them safe (2010), `http://www.technologyreview.com/news/419344/computing-with-secrets-but-keeping-them-safe/`
35. Stehlé, D., Steinfeld, R.: Faster fully homomorphic encryption. In: Abe, M. (ed.) ASIACRYPT 2010. LNCS, vol. 6477, pp. 377–394. Springer, Heidelberg (2010)
36. Stuntz, C.: What is homomorphic encryption, and why should I care? (2010), `http://blogs.teamb.com/craigstuntz/2010/03/18/38566/`
37. van Dijk, M., Gentry, C., Halevi, S., Vaikuntanathan, V.: Fully homomorphic encryption over the integers. In: Gilbert, H. (ed.) EUROCRYPT 2010. LNCS, vol. 6110, pp. 24–43. Springer, Heidelberg (2010)