# The HEROIC Framework: Encrypted Computation Without Shared Keys

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

***Abstract*—Outsourcing computation to the cloud has recently become a very attractive option for enterprises and consumers, due to mostly reduced cost and extensive scalability. At the same time, however, concerns about the privacy of the data entrusted to cloud providers keeps rising. To address these concerns and thwart potential attackers, cloud providers today resort to numerous security controls as well as data encryption. Since the actual computation is still unencrypted inside cloud microprocessor chips, it is only a matter of time until new attacks and side channels are devised to leak sensitive information. To address the challenge of securing general-purpose computation inside microprocessor chips, we propose a novel computer architecture, and present a complete framework for general-purpose encrypted computation without shared keys, enabling secure data processing. This new architecture, called homomophically encrypted one instruction computation, contrary to the previous work in the area does not require a secret key installed inside the microprocessor chip. Instead, it leverages the powerful properties of homomorphic encryption combined with the simplicity of one instruction set computing. The proposed framework introduces: 1) a RTL implementation for reconfigurable hardware and 2) a ready-to-deploy virtual machine, which can be readily ported to existing server processor architectures.**

***Index Terms*—Cloud computing, encrypted processor, homomorphic encryption, one instruction set computer (OISC), Paillier, virtualization.**

## I. Introduction

AS CLOUD computing services become even more affordable today, the option of outsourcing computationally demanding applications is very appealing. The benefits of performing computation in the cloud typically include great scalability, zero maintenance or upgrade cost, as well as all-in-one and pay-as-you-go service options. Unfortunately, these benefits are sometimes outweighed by concerns about data privacy in the cloud, and security threats are not at all uncommon: on the cloud provider end, there are known attacks to Amazon EC2/S3 and LastPass in 2011, as well as Dropbox in 2012 [1]. Moreover, on the infrastructure end, the known exploits to popular hypervisor technologies keep increasing [2]. In contrast to privately owned datacenters, where many logical and physical controls ensure the privacy of the data and executed programs, in a cloud setting users are asked to trust a third party with full control on their sensitive information [3], [4]. This is only possible as long as end users trust the reputation of the cloud provider itself and have studied the provider's safety record. In case the risk of handing over sensitive information to a cloud provider is not acceptable, users need to incur the usually much higher costs of building and maintaining private datacenters. Thus, it is evident that there is currently a need for protecting the confidentiality of the information processed in the cloud, in a more definitive and effective manner.

A promising solution toward addressing these concerns is the use of a cryptographic algorithm (often referred to as an encryption scheme); this approach renders information unreadable to unauthorized entities, and can protect the confidentiality of sensitive data. Cryptography, in general, is very popular for the storage and transmission of information, but it has not been widely demonstrated to ensure the privacy of instructions and data inside cloud microprocessor chips, without sharing any cryptographic keys with the host. A few secure microprocessor designs have been proposed in the past [5], but they assume a threat model where the processor pipeline is trustworthy. In such approaches, the inputs to the CPU are decrypted before processing, and the CPU outputs are reencrypted, and thus the attack surface is limited to the—typically tamper-proof/resistant—microprocessor chip package. This approach, however, is still theoretically vulnerable to attackers capable of eavesdropping the pipeline or leaking the cryptographic keys from inside the processor, without triggering the tampering protections. Similar attack proposals have already been demonstrated in [6], where a sub-transistor level Trojan is used for extracting sensitive information from the internals of integrated circuits.

Preventing this kind of information leaking from the processor chip would require microprocessors in cloud datacenters to process encrypted information directly, without ever decrypting them. Current computer architectures, however, like reduced instruction set computers (RISC) or complex instruction set computers (CISC), cannot directly support execution of encrypted instructions, since encryption effectively prevents the instruction decoder from determining the correct operations. Contemporary processors are engineered for performance and efficiency, since privacy and security have only recently surfaced as design considerations, and processing encrypted information natively has never been a design objective.

Manuscript received July 8, 2014; revised October 24, 2014 and January 19, 2015; accepted February 28, 2015. Date of publication April 3, 2015; date of current version May 20, 2015. This paper was recommended by Associate Editor S. Bhunia.

N. G. Tsoutsos is with the Department of Computer Science and Engineering, New York University Polytechnic School of Engineering, Brooklyn, NY 11201 USA (e-mail: nektarios.tsoutsos@nyu.edu).

M. Maniatakos is with the Department of Electrical and Computer Engineering, New York University Abu Dhabi, Abu Dhabi 129188, U.A.E. (e-mail: michail.maniatakos@nyu.edu).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TCAD.2015.2419619

Due to this lack of architectural support protecting the confidentiality of information, and in the light of new attacking capabilities that allow an attacker to eavesdrop directly from the transistors of the chip [6], we design a novel homomorphically encrypted one instruction computation (HEROIC) architecture, and propose a complete framework for general-purpose encrypted computation without shared keys. The new framework includes a RTL implementation of the architecture for reconfigurable hardware, as well as a ready-to-deploy virtual machine (VM) written in C/C++, which is compatible with existing server processor architectures. The HEROIC framework can execute instructions and manipulate data directly in the encrypted domain, thus preserving the confidentiality of both data and algorithm. A principal observation in HEROIC is the adoption of a single instruction set architecture, which eliminates the need for different opcodes, and contrary to other secure processor proposals, makes decryption of fetched instructions inside processor boundaries unnecessary. As it will become evident in the following sections, a judicious choice for the single instruction renders one instruction architecture as powerful as a Turing machine. Consequently, under certain configurations, the processing power using the HEROIC framework can be comparable to existing RISC and CISC computers. Furthermore, the use of homomorphic encryption allows meaningful manipulation of data directly in the encrypted domain [7], and eliminates the need for encryption/decryption of data. Therefore, there is no key exchange between the user and the host.

The HEROIC framework, to the best of the authors' knowledge, is the first effort toward a general-purpose encrypted computer architecture for cloud computing, capable of processing encrypted information natively, without the need of sending encryption keys to the host machine. In effect, the proposed architecture is Turing-complete, leveraging the flexibility and simplicity of single instruction architectures (detailed in Section II), and exploits the computational properties of homomorphic encryption (described in Section III). The rest of this paper is organized as follows. Section IV discusses the feasibility analysis and security implications of the proposed architecture, while Sections V and VI present the HEROIC framework and all components (reconfigurable fabric design and VM). This paper also includes a discussion on related work (Section VII) and ends with the conclusion in Section VIII.

## II. Turing-Complete Single Instruction Architectures

Before discussing simple Turing-complete architectures, a brief description of general-purpose computation in the sense of Turing completeness is essential. Turing completeness refers to the property of any system that is powerful enough to recognize all possible algorithms [8]. Apparently, since there exist languages that are not Turing recognizable [9], general-purpose computation is equivalent to whatever an abstract Turing machine can compute, assuming very large—but not infinite—memory (as in bounded-storage machines [10]). According to [11], only three grammar rules (i.e., instructions) are sufficient to construct arbitrary programs and thus perform general-purpose computation; these rules are sequence, selection, and repetition. The sequence rule corresponds to a continuation instruction, essentially a GOTO the next address. The selection rule corresponds to a decision between a GOTO the next address and a GOTO another address out of sequence. Repetition corresponds to a GOTO another address, but without any decision [12].

The three aforementioned rules required for Turing completeness, according to [13], can be implemented using only four basic computer instructions: LOAD, STORE, INCrement, and GOTO. This relation implies that even conditional branching can be emulated using unconditional branching (i.e., GOTO). Hence, any computation model that can implement or simulate these four instructions is equivalent to a Turing machine and can perform general-purpose computation.

Single instruction architectures [also known as one instruction set computers (OISC)] are architectures designed to support only one instruction. As long as the micro-operations of the selected instruction implement the four basic computer instructions discussed earlier, single instruction computer variants are capable of Turing-complete computation [12], [13]. This indicates that OISC computers can be powerful despite the simplicity of the design, and can achieve high throughput in certain parallel computing configurations [14]. Since OISC cores are very compact, several of them can be used in place of a much bigger RISC core. These properties make single instruction computers an appropriate alternative to ordinary RISC computers, especially in a cloud setting where having many small cores provides better granularity for scalability.

Several architecture variants exist, based on the single instruction used. Common Turing-complete variants include subtract and branch if less than or equal to zero (subleq) [15], subtract and branch if negative (sbn) [16], add and branch unless positive (addleq), plus one and branch if equal (p1eq), and reverse subtract and skip if borrow [17]. Even though these variants seem completely different in terms of micro-operations, they all share a common execution pattern: a simple mathematical operation followed by a branch decision based on a condition. The existence of this simple mathematical operation (addition or subtraction) makes single instruction set architectures compatible with homomorphic encryption schemes (detailed in Section III), which allow applying that operation directly over encrypted data. As further discussed in Section V, the HEROIC framework supports both subleq and addleq instructions.

## III. Homomorphic Encryption Primer

In cryptography, a homomorphic scheme is defined as an encryption scheme that supports applying a function directly on encrypted data, so that the decryption of the output equals the result after applying an equivalent function on unencrypted data [18]. Such schemes are very useful since a function is applied after the data is already encrypted, and the output can be decrypted to a correct and meaningful result.

There are several known homomorphic schemes that support different functions (e.g., either modular addition, or modular multiplication, or XOR operations, but not combinations of those), and for that reason they are called partially homomorphic. Lately, new schemes have been added following the invention of fully homomorphic encryption (FHE) and

the Gentry scheme [19], which supports applying a function combination on encrypted data.

One of the criticisms against FHE schemes, however, is that they are not practical for everyday use, due to very high overheads [20]. In addition, the applicability of FHE in a hardware design is yet to be seen, due to the complicated nature of fully homomorphic schemes.

Some known schemes that support partial homomorphism are the RSA scheme [21], the El-Gamal scheme [22], the Paillier scheme [23], and the Goldwasser–Micali scheme [24]. Fully homomorphic schemes are the Gentry scheme and its variants (e.g., the BGV scheme [25], [26] and others [27]–[29]). Partially homomorphic schemes are further categorized based on the supported function, as additive homomorphic (like the Paillier and exponential El-Gamal [30]) and multiplicative homomorphic (like RSA, standard El-Gamal, etc.) for supporting modular addition and multiplication, respectively.

Formally, homomorphism is defined as follows:

$$Decrypt[Encrypt(m1) \diamond Encrypt(m2)] = m1 \circ m2 \quad (1)$$

where $(\diamond)$ is modular multiplication and $(\circ)$ can be modular addition or multiplication depending on the scheme.

Since FHE schemes suffer from overheads of several orders of magnitude [31], partially homomorphic schemes are currently the only viable candidates for practical encrypted computers. As Turing-complete variants of single instruction architectures require addition or subtraction, the HEROIC framework is required to employ an additive homomorphic scheme. From the additive schemes discussed in the previous paragraphs, the exponential El-Gamal scheme suffers from high decryption overheads since it requires computing a discrete logarithm, which is a hard problem. Thus, the best candidate for HEROIC is the Paillier scheme, where the modular multiplication of the ciphertexts corresponds to modular addition of the plaintexts.

### A. Paillier Scheme

The Paillier scheme is one of the first efficient homomorphic encryption schemes that supports the addition operation [23]. The scheme is mathematically based on the decisional composite residuosity assumption, which states as below.

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

$$z \equiv y^n \pmod{n^2}. \quad (2)$$

The Paillier scheme is categorized as a public key cryptographic scheme and formally is defined as follows.

Let $p$ and $q$ be two large primes of equivalent length, randomly and independently chosen of each other. Let $n = pq$ be the product of these numbers and $\lambda = LeastCommonMultiplier(p-1, q-1)$. Let $g$ be a random integer in $\mathbb{Z}^*_{n^2}$, so that $\mu = (L(g^\lambda \pmod{n^2}))^{-1} \pmod{n}$, $L(x) = (x-1/n)$ exists, where $-1$ power refers to the modular multiplicative inverse. The public key is $(n, g)$ and the private key is $(\lambda, \mu)$. Encryption is defined as follows. Let $m$ be the message to be encrypted, with $m$ in $\mathbb{Z}_n$, and let $r$ be a random integer in $\mathbb{Z}^*_n$. Then, the encryption of message $m$ is

$$Enc[m] = g^m r^n \pmod{n^2}. \quad (3)$$

And the decryption of the ciphertext $c$ is

$$Dec[c] = L\left(c^\lambda \left(\bmod n^2\right)\right) * \mu \pmod{n}. \quad (4)$$

The homomorphism of this scheme is defined as $Dec[Enc[m1] * Enc[m2] \pmod{n^2}] = m1 + m2 \pmod{n}$, which means that the decryption of the modular multiplication result of the encryptions of two messages equals the modular addition of the two messages. This result is significant, since this is the basis for encrypted computation in the HEROIC framework.

## IV. Feasibility and Security Implications of Encrypted Computation

Following the discussion in Section I, our motivation is to protect the privacy of information processed inside cloud microprocessor chips against any information leakage or side channels, without the need of sharing keys or making any assumptions about the physical protections of the chip itself. The use of encryption, which prevents unauthorized entities from accessing sensitive contents and creates an invisible protection container around information, is the foundation of any secure system. Besides the chip itself, encryption can protect against eavesdropping on the memory contents of a computer as well. Before digging into the proposed HEROIC architecture, it is necessary to provide an extended discussion on: 1) the feasibility of encrypted Turing-complete computation; 2) the selection of a single instruction architecture and the benefits of this choice; as well as 3) limitations of general-purpose encrypted computation. The first step in this direction is to define our threat model.

### A. Threat Model

In an effort to address the privacy of sensitive data being processed in the cloud (private keys, session cookies, passwords, etc.), we want to protect the microprocessor internal structures and the memory contents of a cloud computer against: 1) semi-honest cloud service providers (i.e., providers that use the HEROIC framework correctly but will try to peek into the data) and 2) remote adversaries with access to the potentially vulnerable cloud providers. In our model, memory contents can be leaked to, or eavesdropped by, unauthorized entities with remote or physical access to the processor internals and peripherals. In the following paragraphs, different potential threat vectors are elaborated, assuming different levels of attacker resources.

*1) Adversaries With Remote Access:* In this scenario, a software vulnerability would cause computer memory contents to be leaked to an attacker over a remote connection. The attacker would not require a vast amount of resources, since a vulnerable service would leak sensitive information "voluntarily" due to underlying vulnerabilities. The "Heartbleed" [32] vulnerability is a prominent example of such case. This scenario also covers more advanced remote access techniques, such as cache side channels [33].

*2) Adversaries With Physical Access:* In this scenario, an attacker is assumed to have direct access to the underlying hardware and be able to probe memory locations, eavesdrop memory bus signals, or even the registers' values inside

the CPU [6]. This attack is less practical than the remote attack demonstrated above, since it typically requires in-person presence to datacenters that traditionally exist in areas with controlled access; only malicious insiders and internal threats of the cloud provider could potentially execute such attacks.

*3) Adversaries With Access to Chip/IP Supply Chains:* This scenario corresponds to injecting hardware Trojans to the modules used to perform computation. Even though such attacks would require early insider access to the supply chain, such attacks are not impractical and can even incur minimum overhead [6], [34]. A hardware Trojan can leak information directly or through side channels in a potentially stealthy fashion.

*4) Adversaries With Cryptanalysis Capabilities:* In this scenario, potential threats are able to attack the underlying memory encryption, depending on their resources. With unlimited resources, attackers could potentially break the underlying encryption scheme. In our model, we assume that attackers are rational in the sense that the cost of cryptanalyzing the underlying encryption does not exceed the value of the encrypted information. This assumption is important to determine the required strength of the homomorphic encryption scheme used, so that the hardness of decryption has a lower bound.

### B. Encrypted Turing-Complete Computation

Based on the analysis in Section II, it becomes evident that general-purpose computation can be implemented by a computer that implements at least four operations: reading from memory, writing to memory, branching, and incrementing a value. Any computer that can perform these four operations (or equivalents) in the encrypted domain, would also be able to perform general-purpose computation, since Turing completeness is a property oblivious of the abstract domain used (unencrypted or encrypted). The aforementioned four operations can be ported directly from the unencrypted domain to the encrypted domain.

1) `LOAD` and `STORE` memory accesses remain unchanged; addressing can be unencrypted or encrypted (with determinism).
2) `INC`rement (or equivalently `DEC`rement) operations on encrypted values requires additive homomoprhic encryption, which allows direct modification of encrypted values.
3) `GOTO` branch operation is possible using direct address assignment to the program counter (PC) abstraction; addressing can be unencrypted or encrypted (with determinism) as well.

In the next paragraphs, we discuss several issues and potential choices of the mapping presented above.

### C. Encrypted Architecture Challenges

*1) Encryption Granularity:* The first issue related to encrypted computation pertains to the granularity of encryption: one approach would be to perform encryption at the word level, while another approach would encrypt larger memory blocks that may include multiple instructions, arguments, or pieces of data. Encrypting larger blocks, however, would eventually require decryption inside some sorts of a secure container (for example a tamper proof chip) in order for instructions/data to be separated and be processed individually. Using word level granularity does not require having a tamper proof container. In the latter case, using homomorphic encryption allows manipulating the encrypted data words directly, without ever having to decrypt them. Based on our assumed threat model, the cloud microprocessor chip is untrusted, so providing a decryption key would violate our assumptions.

*2) Scope of Encryption:* Another issue relates to the scope of encryption. The first option would be to encrypt only the program data, while having program instructions unencrypted. This option eventually protects sensitive program data; the program algorithm, however, remains in the clear and the algorithmic intellectual property could potentially be available to attackers. The second option would be to encrypt both instructions and data words, so that the algorithm executed is also protected from leaking. In this second case, the computation would require to discriminate different instructions, in order to perform different operations like load, store, or goto, every time the key changes. This limitation can be resolved by using a single instruction architecture [12], [35] as presented in Section II. In the latter case, the single instruction is fully defined by its arguments, so discriminating instructions or "operation codes" is no longer an issue. Practically, in HEROIC, the data is encrypted, while the algorithm is obfuscated [36].

*3) Memory Addressing:* Using a single instruction architecture to perform encrypted computation requires special consideration for memory addressing: since every instruction is encrypted, the instruction arguments are encrypted as well. This applies to arguments that define memory references, and thus memory references would also be encrypted. This is only a minor issue, and resolving it requires that each encrypted memory reference points deterministically to only one physical memory address, assuming that memory contents exist in a unique memory space. In effect, referencing a memory location using an encrypted address should always resolve to the same physical location in memory.

*4) Malleability:* Using homomorphic encryption to protect instructions and data also requires special attention regarding the malleability of homomorphically encrypted values. Malleability is an integral and sometimes necessary property of homomorphic encryption, which is defined as the possibility of transforming a ciphertext into a different ciphertext that decrypts to a related plaintext [37], [38]. The side effect of malleability, however, is the need to protect the integrity of such values against malicious modifications. The homomorphically encrypted values in memory could potentially be manipulated to create other values which would be consistent with the used encryption scheme. This limitation can be mitigated using message authentication or digital signatures to ensure integrity protection [38]. Since, as previously discussed, encrypted computation is also Turing-complete, hashing, message authentication, or digital signing algorithms can be implemented directly in the encrypted domain. Eventually, an encrypted computer could execute integrity verification routines directly on encrypted data without ever decrypting them. By using this approach, the owner/programer of a computer program could identify if an attacker has tampered with the encrypted memory contents in order to alter the computation result.

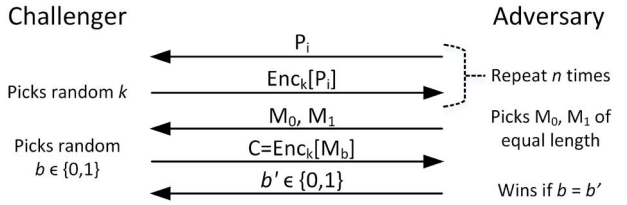


Fig. 1. Representation of the CPA steps. The adversary chooses a plaintext that the challenger encrypts using a key. In case the adversary does not have access to encryptions of chosen plaintexts, the attack may not proceed.

### D. Limitations of General-Purpose Computation in the Encrypted Domain

Using homomorphic encryption to protect the confidentiality of sensitive data in memory and potentially the executed algorithm does not come without limitations. In this section, we discuss these limitations in the context of protecting memory contents from unauthorized leakage, assuming the threat model presented in Section IV-A.

*1) Deterministic Encryption:* As discussed in Section IV-C, memory references during program execution should always map to the same physical memory locations. Thus, when encrypted memory addressing is employed, the encryption of an address would have to be the same every time. This means that memory addresses would have to be deterministically encrypted. At first glance, using deterministic encryption could potentially render the underlying encryption scheme open to chosen-plaintext attacks (CPA) [38], presented in Fig. 1. In HEROIC, however, homomorphic encryption (which is public key encryption) is not used in the traditional way: the (public) encryption key that is used for encrypting each data word also remains a secret. Furthermore, in HEROIC, we assume that the program owner is the only entity that has access to the encryption keys, and will not encrypt plaintexts chosen by third parties. In case these assumptions do not hold, a third party may be able to generate encryptions of chosen values, and thus the security of the scheme may be violated.

Typically, in a CPA scenario (following the steps of Fig. 1), an adversary sends chosen plaintexts $P_i$ to the challenger. If the challenger acts as an "encryption oracle" and sends correct encryptions of these plaintexts (which would be equivalent to sharing the encryption key), then the adversary would have a nonnegligible advantage in predicting whether the challenge ciphertext $C$ in Fig. 1, is the encryption of $M_0$ or $M_1$. In the HEROIC case, a CPA could be a risk if the program owner is coerced to encrypt programs or values that are provided by a (malicious) third party.

Another concern that stems from the use of deterministic encryption is related to potential cryptanalysis of the used plaintext range of values. In case a specific range of values is encrypted using deterministic encryption, so that the upper bound of the values is known to (or can be guessed by) an attacker, then, by using homomorphic operations, the attacker could correlate encrypted values in the range with values in the known unencrypted range and potentially leak information. Even though such cryptanalysis becomes exponentially harder as the security parameter of the underlying encryption increases, it is nevertheless an important concern. This potential attack, however, is mitigated by using additional noise in the encrypted memory contents, effectively creating a range of encrypted values with an unknown (or unguessable) upper bound. Specifically, if the plaintext range is $(a, b)$, encrypting additional ranges $(r_i \cdot a, r_i \cdot b)$ for $i \in \mathbb{N}$, random $r_i \in \mathbb{N}$, would effectively increase the complexity of correlating the encrypted values with one particular plaintext range. This approach incurs a tradeoff between necessary memory and degree of assurance against cryptanalysis, which needs to be considered when implementing encrypted computation.

*2) Look-Up Values:* As elaborated in Section II, Turing-complete computation is possible using at least four operations: load, store, increment, and unconditional branching (goto). Instead of unconditional branching, it is also possible (and more efficient) to perform Turing-complete computation using conditional branching, which is a special case of unconditional branching. Using conditional branching, however, requires special consideration: in the encrypted domain, decision information is also encrypted, and one way to perform branch decision is using look-up tables that provide only very limited information about encrypted values. Providing look-up tables requires special care to avoid common cryptanalytic pitfalls: for example, if the encryption of value $V$ is known, then an attacker could potentially build a codebook of encrypted values $\{0, V, 2V, \ldots, kV\}$ or $\{1, V, V^2, \ldots, V^k\}$, $k \in \mathbb{N}$, depending on the applicable homomorphic operation (addition or multiplication). For that matter, any provided look-up tables should be randomly permuted and poisoned with additional (noisy) values that are not actually used by the encrypted computer during runtime. This significantly reduces the probability of predicting a valid value $V$, as adversaries would have to populate all possible program executions for all potential inputs (which is exponentially infeasible as the size of the security parameter increases). This approach effectively presents a tradeoff between the size of used look-up tables (that also include noisy data) and the complexity of cryptanalyzing the provided auxiliary data to create decryption codebooks.

## V. HEROIC Architecture

The HEROIC framework described in this paper is a novel architecture capable of executing encrypted programs and manipulating encrypted data. The HEROIC architecture is based on single instruction architectures (discussed in Section II) and supports both `subleq` and `addleq` as the single instruction. Without loss of generality, for the remaining of the paper, we will describe the architecture assuming the `subleq` instruction, and unless otherwise noted, the same would also hold for the `addleq` instruction.

### A. Subleq Instruction

The `subleq` instruction is fully defined using three arguments (namely A, B, and C), and its micro-operation is specified as follows:

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

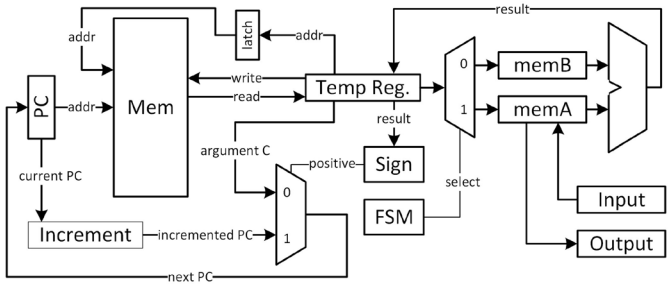


Fig. 2. Basic components and datapath of a sample `subleq`-based computer.

Based on the definition above, we make the following important observations about `subleq`.

1) `subleq` implements all four operations required for Turing completeness, as discussed in Section II.
2) `subleq` uses indirect addressing, meaning that instruction arguments point to the addresses of the two memory locations to be subtracted (arguments are not subtracted directly).
3) `subleq` permits modification of any memory location, and thus one instruction may modify an argument of another instruction (self-modifying code is allowed).
4) Every branch micro-operation requires a comparison of the subtraction result with zero.
5) Arguments A, B, and C are always stored in subsequent memory locations.

A reference `subleq`-based architecture is presented in Fig. 2, which demonstrates all necessary components along with their interconnections. Since one instruction set computing does not support different instructions, a single main memory is sufficient for instruction arguments and data. A temporary register holds values read from memory as well as addresses and values sent to memory, while an arithmetic logic unit (ALU) is responsible for subtracting two given arguments stored in registers "memA" and "memB". Register "memA" is also responsible for I/O operations. The control finite state machine (FSM) and a sign identification unit are responsible for coordination and branching respectively, while a PC along with increment logic complete the design.

As discussed in Sections III and IV-B, HEROIC requires an additive homomorphic scheme, and thus the Paillier scheme (Section III-A) is used to encrypt each memory value in a deterministic configuration. Encrypting memory values using the Paillier scheme would result in values $(2 * n)$ bits wide, where $n$ is the encryption security parameter. Since the baseline `subleq` datapath presented above cannot support the encrypted arguments, major modifications over the baseline are necessary to implement the HEROIC architecture. The next section elaborates on the modifications required to the baseline design to allow encrypted computation, along with challenges and their respective solutions.

### B. Design Considerations

In this section, we present our proposed solutions to several challenges that arise from the use of encrypted instruction arguments. The block diagram of our HEROIC architecture appears in Fig. 3, and provides an abstract view of the additional modules and memories required compared to the baseline presented earlier.

*1) Encrypted Memory Addressing:* In HEROIC, encrypted instruction arguments can be used as indirect references to memory locations. Since HEROIC uses a unified main memory for instructions and data, and because instructions are allowed to reference and homomorphically modify encrypted arguments of other instructions (which later can be used again for memory referencing), the architecture requires all memory addresses to be encrypted. As a result, the PC should use encrypted values to reference instruction arguments in memory. For consistency in addressing, and given that the decryption key is not provided, it is necessary that all memory addresses are encrypted under the same key.

*2) Matching Instruction Arguments:* The transition from the baseline `subleq`-based architecture concept to HEROIC raises another concern: Because HEROIC employs a unified memory for both instructions and data, and since instruction arguments are indistinguishable from data, a mechanism for matching arguments A, B, and C of the same instruction is required. To further complicate this problem, since encrypted addressing is used, a proper sequence of arguments in the unencrypted domain would become permuted in the encrypted domain, as encryption of addresses does not preserve their absolute order.

This issue can be solved by matching each encrypted element inside main memory, with the encrypted address of its next element, essentially providing each memory item with a pointer to the next item (as shown in Fig. 3, top left, last column of the main memory). The value of this pointer is the one used by the PC to find the next instruction argument in the execution trail; simply incrementing the PC would be incorrect.

*3) Out of Range Correction:* In this paper, for simplicity and without loss of generality, we will describe a 16-bit single instruction architecture ported to the encrypted domain (different word sizes will be discussed in the experimental results, in Section VI-D). This means that the width of each memory location—before encryption—would be 16 bits, equal to the size of each memory address. The representation of negative numbers follows the standard two's complement approach, and thus the range of supported numbers is from $-2^{15}$ to $(2^{15}-1)$.

The range difference, however, between the unencrypted (16-bit) and the encrypted (up to 2048-bit, for a security parameter of 1024-bit for Pailler encryptions [23]) creates out of range (OOR) discrepancies. In order to clarify this issue, let us consider a homomorphic addition of two negative numbers. As demonstrated in the example of Fig. 4, the addition of $-42$ with $-1$, which corresponds to adding $(2^{16}-42)$ with $(2^{16}-1)$, would result to the encryption of $(2^{17}-43)$ and not the encryption of $(2^{16}-43)$ (the correct two's complement representation of $-43$). This OOR effect, is an artifact of the different ranges in the encrypted and the unencrypted domain. The inconsistency is eventually corrected by adding the modular multiplicative inverse of the encryption of $2^{16}$ (given to HEROIC with the encrypted program), in order to get the expected result. This operation is shown in Fig. 3 where an OOR correction multiplicative inverse is provided to the ALU.

Before correcting an OOR effect, however, it must be detected first. This is achieved by using an OOR lookup memory that matches the encryptions of all numbers from 0 to $2^{17}$

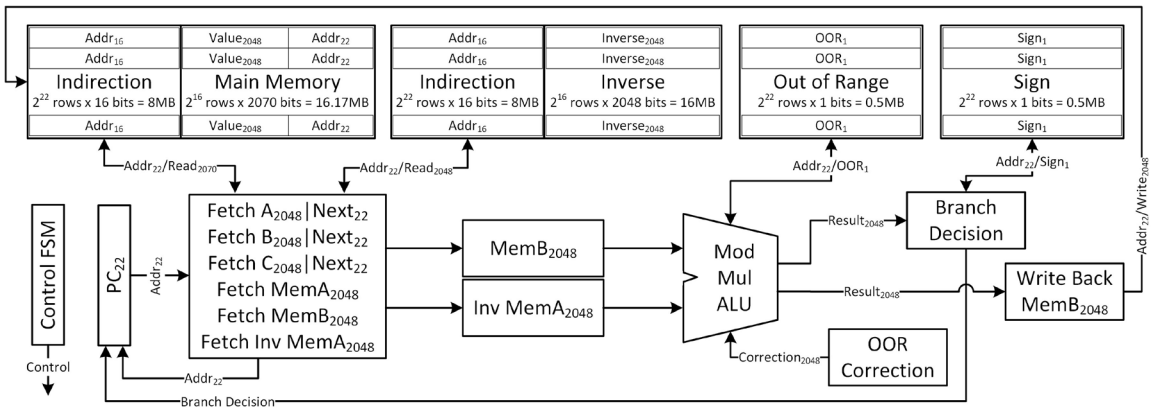


Fig. 3. Abstract view of additional units and memories of the HEROIC encrypted computer (security parameter 1024-bits).

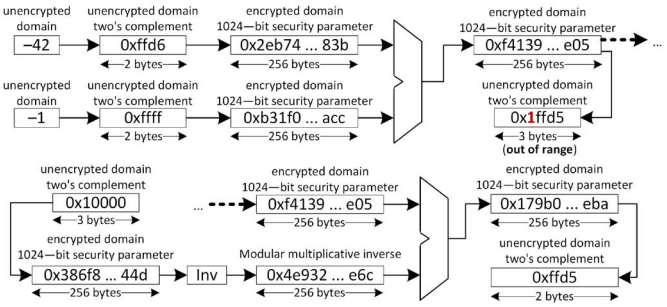


Fig. 4. Out of range correction for homomorphic addition of representations of negative numbers.

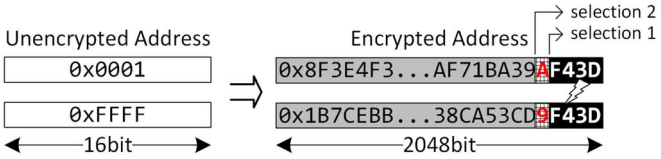


Fig. 5. Identifying the proper number of lower bits required to differentiate encrypted memory addresses.

with one bit of information indicating "above $2^{16}-1$" or below. As soon as the ALU result is matched with an entry above $2^{16}-1$, a secondary addition with the modular multiplicative inverse corrects the result. The OOR lookup memory is shown in Fig. 3 as well (top right).

*4) Homomorphic Subtraction:* When the HEROIC architecture uses the `subleq` instruction, it requires an ALU that performs homomorphic subtraction. The Paillier scheme, however, ensures that the modular multiplication of two ciphertexts would generate a value, which when decrypted, corresponds to the modular addition of the respective plaintexts (i.e., homomorphic addition). Thus, in order to achieve homomorphic subtraction, the modular multiplicative inverse of the subtrahend needs to be homomorphically added to the minuend, and this operation still preserves the homomorphic properties of the scheme.

The modular multiplicative inverse, however, cannot be retrieved algebraically given an encrypted value. Thus, an inverse lookup memory (similar to the OOR lookup memory presented earlier) is necessary. This memory returns the multiplicative inverse of the encrypted value of the subtrahend (shown at the top center of Fig. 3). The retrieved inverse, along with the minuend, is used by the ALU to perform modular multiplication and generate the expected homomorphic subtraction output. This result is then subject to "OOR" correction as described earlier (Fig. 4).

Alternatively, when the HEROIC architecture uses the `addleq` instruction, the subtraction operation is replaced by addition. In this case, the inverse lookup memory is not needed, since the addition operation is directly supported by the Paillier scheme. A potential setback when using `addleq` is that it is significantly harder to program because a high level compiler is not currently available, and is less efficient in terms of programming compared to `subleq` [39].

*5) Memory Addressing Size:* A security parameter size of 1024 bits combined with encrypted memory addressing requires memory address support of 2048 bits in width. Such memory would have a prohibitive cost and is not actually necessary, since, in the unencrypted domain, 16 bits of address size suffice for proper execution. In this paper, we implement three optimizations, which effectively reduce the required memory addressing size to a practical implementation.

*a) Optimization 1:* Since the unencrypted domain is assumed 16 bits, the unencrypted main memory should support 16-bit addressing. While Paillier encryptions can be 2048-bit wide, in the encrypted domain we can use only the lower bits of these 2048 bits to request memory addresses. Fig. 5 shows how the lower bits of an encrypted address can be used to differentiate $2^{16}$ main memory addresses. As shown in the example, if only 16 bits are used to differentiate memory addresses (selection 1), there would be a collision, as the last 16 bits of the encrypted values of addresses `0x0001` and `0xFFFF` may very well be the same. If the selection included 20 bits (selection 2), this would (most probably) allow proper separation of all memory locations.

Fig. 6 presents the minimum number of address bits necessary to discriminate at most $2^{17}$ different encrypted addresses (since the OOR lookup memory uses addresses up to $2^{17}$), given 100 random keys per security parameter size and confidence interval $\leq 9.8$ for confidence level 95%. The results indicate that using only 22 bits of address size is sufficient to discriminate $2^{17}$ encrypted addresses with 90% probability. In case of a collision, memory reencryption with another key is necessary. It should be emphasized that collisions can only

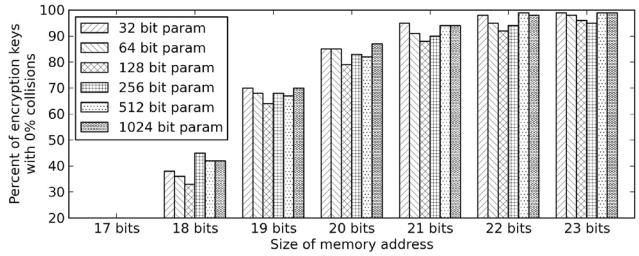


Fig. 6. Percentage of encryption keys with 0% collisions in $2^{17}$ encryptions for different memory addressing sizes and security parameter sizes.

happen during initial encryption (i.e., before execution), so this hazard is not applicable during runtime. This optimization reduces the memory address size (and thus the number of memory locations) to $2^{22}$, down from $2^{2048}$ (for the highest security parameter size). Without this optimization, HEROIC would be impractical.

*b) Optimization 2:* A second observation is that, for the main memory, only $2^{16}$ out of $2^{22}$ addresses are used. Since every address points to a 2070-bit value (i.e., 2048 + 22 bits for argument matching, assuming 1024-bit security parameter), there are many memory locations of very large width that remain unused. In order to reduce the number of these very wide encrypted memory locations, we use one level of indirection between two memories of different address sizes and content widths. The first (original) memory would have 22 bits of addressing size but only 16 bits of content width, pointing to the second (new) memory. The second memory would have 16 bits of addressing size but the content widths would be much larger to fit an encrypted value. This optimization effectively limits wasting memory locations of very large content width, reducing the main memory size down to 24.17 MB, instead of approximately 1 GB without the optimization. Similarly, after this optimization, the multiplicative inverse memory is reduced to 24 MB instead of 1 GB. The two-level memory optimization is presented at the block diagram of the HEROIC architecture (Fig. 3 top left and top center).

*c) Optimization 3:* A further optimization to the memory indirection presented above, is to use our collision resolve unit (CRU) to reduce the size requirement of that memory by about five times. In more detail, CRU translates the 22-bit (truncated) input address to a valid 16-bit output index for the secondary memory discussed earlier, by using a tagging scheme based on the six most significant bits of the input. Specifically, CRU requires a smaller memory, addressed by the 16 least significant bits of the input, which stores several tag-index pairs on every memory line. To select the correct output, CRU retrieves the memory line corresponding to the aforementioned 16 least significant bits of the input, and then checks all 6-bit tags in the line to find a valid output index match; because CRU essentially implements a permutation, one match is guaranteed to be found.

An important observation that allows CRU to reduce the memory requirements, is that each memory line has an expected upper limit for the number of necessary tag-index pairs per line. The 22-bit inputs are truncations of encrypted values, and thus, they are uniformly distributed and indistinguishable from truly random values. Furthermore, the number of such inputs is exactly $2^{16}$ (i.e., there are holes in the 22-bit range). The expected maximum number of collisions per CRU line on this set of inputs, using an analysis similar to [40, Ch. 5.5], is $\Omega(\log n/\log\log n)$ (where $n$ is the set size), while the probability of having more than $t$ collisions per CRU line is less than $2^{16}e^t t^{-t}$. Since the probability of having 11 collisions is about 1%, we can provision up to ten tag-index pairs per memory line, with very high probability of success.

Using this optimization, we are able to replace the $2^{22}$ by 16 bits indirection memory, with a $2^{16}$ by 220 bits CRU memory, which amounts to about 5× in savings. Furthermore, CRU does not impose any execution overhead whatsoever to the implementation of HEROIC, since checking the 6-bit tags from a retrieved memory line is performed very efficiently in software, as well as in parallel in hardware. All memory indirections in HEROIC can benefit from the CRU optimizations.

*6) Jump Decisions in the Encrypted Domain:* Another challenge in the HEROIC architecture is determining the mathematical sign of the ALU output, in order to make the necessary branch decisions. Since the ALU result is also encrypted, its sign is unknown, and algebraically it is not possible to compare the result with zero: if we were able to compare an encryption with known values, we would easily break the encryption scheme, by performing a binary search. Even though order preserving encryption schemes exist [41], these are not homomorphic and cannot be used in our case study.

To address this issue, HEROIC uses a sign lookup memory (loaded along with the encrypted program) that returns the mathematical sign of any encrypted (two's complement) value within a range of encrypted numbers. For the assumed unencrypted domain of 16 bits, the sign lookup memory can return the sign of all numbers in the range from 0 to $(2^{16} - 1)$, given their respective encryption as the lookup address (shown in top right of Fig. 3). Following the analysis of Section V-B5, the sign lookup memory can also benefit from the first address truncation optimization and use only 22 bits of memory address size.

## VI. HEROIC Framework Implementation

To evaluate the applicability and performance of the HEROIC architecture, we implemented two different variants of the HEROIC architecture: 1) a RTL implementation for reconfigurable fabric and 2) a software VM. Each part of this framework is presented in the following paragraphs, followed by performance and memory requirements evaluation.

### A. HEROIC in Reconfigurable Fabric

A recent trend for boosting performance in data centers and make microprocessors even more versatile is to use FPGAs in collaboration with ASIC CPUs [42], [43]. Following this trend, a RTL implementation of the HEROIC architecture has been developed for our framework. This implementation targets reconfigurable hardware (i.e., FPGA chips) and is fully parameterizable with regards to security parameter, memory

word size, collision protection, and supported machine code (`subleq` or `addleq`). The RTL implementation also takes advantage of the memory saving optimizations (discussed in Section V-B5), and fast modular multiplication using the Montgomery algorithm [44]. Additional details about the RTL design are presented in the following paragraphs.

*1) RTL Design:* The RTL design of HEROIC is implemented to be scalable and completely modular. This approach allows more robust testing and flexibility for future changes and further optimizations (e.g., parallel computing configurations). At a high level, the design consists of the CPU core connected to three memories.

a) *The Main Memory:* In this memory, the instruction arguments and data values are stored encrypted, along with a pointer to the next element in sequence. This memory uses the indirection optimizations discussed in Section V-B5.
b) *The Inverse Memory:* In this lookup memory, the modular multiplicative inverses of a range of encrypted values are stored to allow homomorphic subtraction (enabled only if `subleq` is used). This memory can potentially benefit from the indirection optimizations as well.
c) *Sign and Out-of-Range Memory:* In this lookup memory, the sign and out-of-range information necessary for branch decisions and out-of-range correction is stored as a pair of bits. Since the row width of this memory is small, it is not beneficial to use any memory indirection optimization.

The RTL design is using an input/output controller (IOCTRL) that is interpolated between the CPU and main memory. This controller is responsible for interfacing with the outside world and communicating input and outputs to/from the CPU over the memory bus, using reserved addresses through memory mapping. The controller uses its own state machine and supports double handshaking for input and output, which supports arbitrary I/O delays and makes the design very flexible.

The CPU module itself consists of three main units and two auxiliary units.

a) *The Fetch Unit:* This unit consists of three "direct fetch" modules, each being responsible for retrieving arguments A, B, and C of each instruction from the main memory, as well as two "indirect fetch" module that retrieve indirect arguments Mem[A] and Mem[B] (based on A and B retrieved earlier). Each module in the fetch unit is optimized for performance and is self-contained with individual state machines, so no separate control is necessary. All modules inside the fetch unit are synchronized using double handshaking and multiplexed over the same memory bus.
b) *The ALU Unit:* This unit consists of two modular Montgomery multipliers (one for homomorphic addition and one for out-of-range correction), a fetch module for sign/out-of-range lookup information and an optional fetch module for multiplicative modular inverse information (in case `subleq` is used). The Montgomery multipliers are efficient (performance equivalent to [45]) and a result is produced in $h+2$ clock periods, when $h$ is the bit-size of the given arguments. Furthermore, the sign information is used in this unit for branch decisions, which effectively determine the new PC value.
c) *The Writeback Unit:* This unit consists of a single module responsible for writing the updated Mem[B] value, received from the ALU unit, back to memory. The writeback module is optimized as well, and communicates with the ALU using the double handshaking synchronization protocol discussed earlier. The module employs its own FSM and is multiplexed on the main memory bus as well.
d) *Two Bus Multiplexers:* These auxiliary units are used to selectively connect different fetch units over the same memory bus, so that memory modules with only one port can be used. The first bus multiplexer is 5-to-1 and is used during argument fetching by the Fetch unit. The second multiplexer is 2-to-1 and is used for switching between the 5-to-1 multiplexer (used for reading from memory), and the Writeback unit (used for writing back to memory). These multiplexers are not implementation specific, but necessary for universal compatibility, since reconfigurable HW does not always support tristate buffers required for bus mastering.

The RTL design operates as follows.

a) Initially, the program owner compiles the high level program source and generates single instruction machine code. Then, the owner generates a secret key, which is used to homomorphically encrypt each value in the machine code (i.e., data values and instruction arguments) along with sequencing information (Section V-B2). The encryption/decryption key is only known to the owner and is never sent to the HEROIC processor. Since each encrypted value is explicitly linked with the next one, it can be encrypted individually and out of order under the same key (i.e., the HEROIC program encryption process is embarrassingly parallel). Note that in case there is a collision, this is detected at this stage and reencryption under a different key is necessary.
b) The program owner post-processes the encrypted machine code, adding to each value a truncated address pointer to the next value in sequence, and randomly permuting the order of all values. After adding memory saving indirection, this becomes the main program memory.
c) The program owner also generates sign/out-of-range memories, and in case of `subleq`, modular inverse memories (potentially using an indirection optimization).
d) All memories are loaded to the HEROIC processor, along with the first PC value for execution. The processor naturally generates encrypted outputs, which are collected by the program owner; these outputs, when decrypted, correspond to the expected program outputs.

The HEROIC design for reconfigurable hardware incorporates units that are designed to support different security parameter strengths. With trivial modifications, unit configuration can be dynamic during runtime for all units, while it is also possible to leverage dynamic reconfiguration features of modern FPGAs. A high level block diagram of the RTL implementation is provided in Fig. 7.

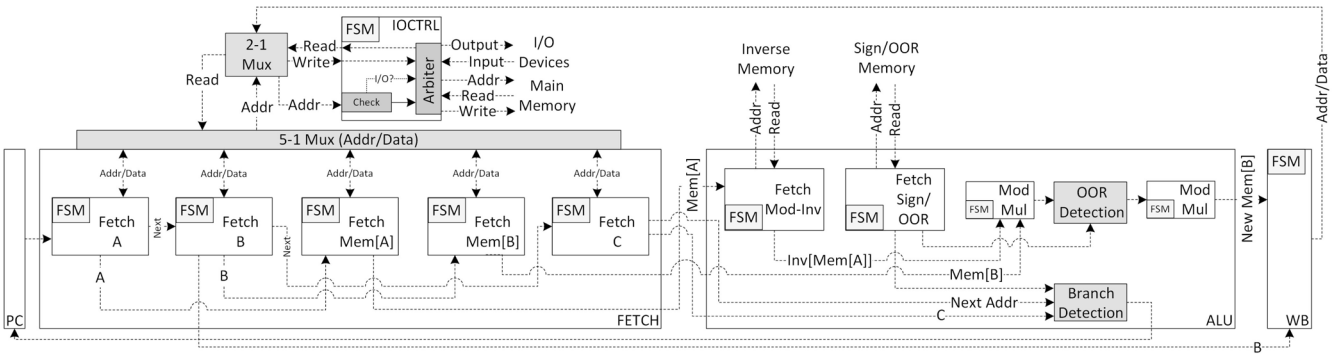


Fig. 7. Abstract view of HEROIC encrypted computer implementation for reconfigurable fabric.

### B. HEROIC Virtual Machine

The HEROIC RTL implementation presented in the previous paragraphs provides protection against eavesdropping, but eventually requires an FPGA to run on. To overcome this constraint, we leverage virtualization, which generally provides a translation layer for porting one instruction set architecture (ISA) over another, and we develop a novel HEROIC VM that can run on any commodity CPU. Our VM is implemented in `C` and runs natively (i.e., without emulating any hardware states or modules) using the GNU GMP multiple precision arithmetic library [46] that is portable and already optimized for performance on different CPU architectures. The implemented VM is capable of executing encrypted HEROIC programs, receive encrypted inputs at runtime, and generate encrypted outputs to program owners. On the program owner end, our VM implementation is supported by a special HEROIC Python library that wraps around encryption routines from [47]. This Python library can be used for program main memory encryption, input/output encryption and decryption, memory randomization/permutation, memory noise generation for extra security, memory indirection, intelligent collision detection, truncation of next-element-address pointers, and modular inverse memory generation. The VM can also be used for event logging, statistics generation, and execution trail generation (useful for debugging).

A typical usage model for this VM would be to run (standalone or parallelized in multiple instances) on a cloud server, where users can provide encrypted computation images and receive results. Since modern cloud servers offer great scalability, the use of HEROIC VM is not subject to any memory size limitations that a single system may impose. Another usage model would be to act as a secure co-processor, where only security sensitive parts of a computation (e.g., proprietary algorithms/IP) are offloaded to the VM. The VM eventually provides a ready-to-use HEROIC implementation that can be executed as any other application on a server, while providing a secure and private environment for sensitive data.

*1) VM Design:* The VM implementation consists of five main parts.

a) *Memory Routines:* These routines initialize and implement all memory-related functionality, using our Python backend library. These routines implement encrypted memory file parsers (for main memory, sign/out-of-range memory, and optionally modular inverse memory), encrypted value converters to GMP multiprecision format arrays and memory lookup routines that fetch lookup values by directly indexing GMP memory arrays.

b) *Homomorphic Operation Routines:* These `C` routines are used for homomorphic addition/subtraction with out-of-range and randomness correction. The most important function in this category is naturally the modular multiplication operation, which is also performed directly by the ISA-optimized GMP routines.

c) *Input/Output Routines:* These `C` routines are responsible for communicating with the outside world using encrypted values. The program owner can optionally use our Python backend library to receive/decrypt the VM outputs, or generate/send encrypted inputs to the VM.

d) *Main Execution Loop:* This `C` routine implements all steps of HEROIC execution, including: reading arguments, calling homomorphic operations, making branch decisions, updating the PC, and writing the main memory with computed values.

e) *Support Routines:* These `C` routines perform memory `free` operations (for memory leakage prevention), logging and generate debug information.

### C. Performance Evaluation

We evaluated the performance of the HEROIC framework by using four computationally intensive benchmarks, namely primes, bubbleshort, factorial, and fibonacci. We compiled all four programs from `C` to `subleq` machine code using the compiler from [39], and each instruction argument was subsequently encrypted using the HEROIC auxiliary library and the Paillier encryption routines from [47]. The HEROIC auxiliary library was also configured to add 25% additional random values into the look-up memory ranges, in order to thwart potential cryptanalysis (Section IV-D2). We evaluated the framework performance in terms of execution time by executing our benchmarks inside the HEROIC VM running on a modern PC, as well as on a Xilinx FPGA board. Specifically, the VM was evaluated on a dual core Intel i5-4288U CPU running at 2.60 GHz with 2 GB of memory, while the FPGA implementation was evaluated on a Kintex-7 XC7K325T-2FFG900C running at 200 MHz, with 1 GB DDR3 PC3-12800 RAM. The hardware complexity of the HEROIC FPGA implementation was measured at 11 245 slice LUTs and 7059 slice registers, corresponding to 5.5% and 1.8% resources respectively; in addition, the Xilinx memory controller IP required for our experiments consumed an additional 3.8% slice LUTs and 1.6% slice registers. The execution

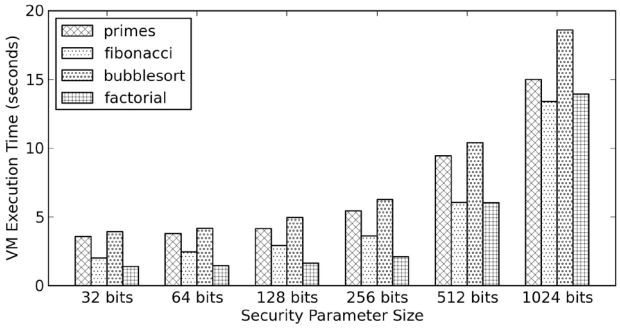


Fig. 8. Measured execution time on the HEROIC VM, for different security parameter sizes.

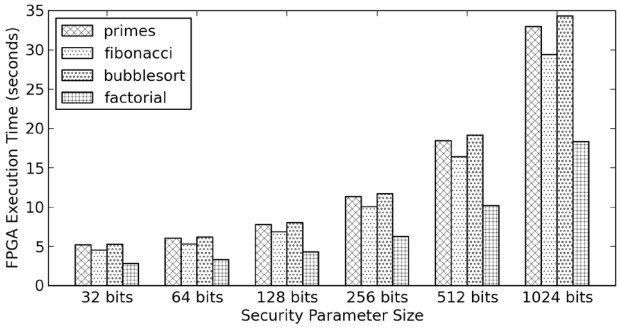


Fig. 9. Measured execution time on the HEROIC reconfigurable fabric, for different security parameter sizes.

TABLE I
EXECUTION STATISTICS FROM USED BENCHMARKS

| Benchmark | Out of range (%) | Executed Instructions |
|---|---|---|
| **primes** | 47.152 | 1855799 |
| **fibonacci** | 51.170 | 1617294 |
| **bubblesort** | 51.784 | 1882234 |
| **factorial** | 50.228 | 1011994 |

time for different security parameter sizes is presented in Figs. 8 and 9, while additional benchmark statistics are given in Table I.

Fig. 8 presents the actual CPU time for executing each benchmark, and Table I provides the number of encrypted `subleq` instructions executed, as well as the percentage of instructions that required out-of-range correction. From the results, we make the following observations.

1) As the security parameter increases, the execution time increases as well, but not necessarily linearly. After the 256-bit security parameter size, the VM incurs a significant performance degradation. This is attributed to the GMP memory management routines for large arithmetic, as the computation operands differ significantly from the native width of 64 bits.
2) The ratio of out-of-range corrections is close to 50%, as expected, since the encrypted values are distributed randomly in the encrypted domain and the probability of an out-of-range result is consequently evenly distributed.

Likewise, Fig. 9 shows the execution time for the FPGA implementation. One can observe that the FPGA results scale

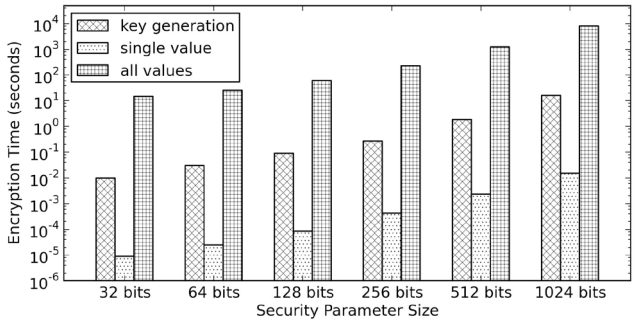


Fig. 10. Compilation one-time overhead, for generating and post-processing all necessary encrypted memory ranges.

smoothly as the security parameter increases, contrary to what was discussed about the VM. An interesting observation is that the FPGA performance is between 1.7× and 2.5× slower than the i5 CPU: we conjecture that this is attributed to the FPGA clock rate, which is 13× slower than the CPU, as well as the FPGA memory latency in light of using a flat memory hierarchy. Indeed, the available FPGA memory controller IP is optimized for large data transfers (i.e., bursts) of 512 bits at a time; these transfers have high latency (around 150 ns per burst), and applications with intensive access patterns for smaller transfers are impacted. In the HEROIC architecture case, each instruction requires multiple accesses to auxiliary memories (small data transfer), and thus every access incurs the aforementioned burst overhead. Given an FPGA with higher clock speeds, the RTL is expected to outperform the VM.

A comparison between the VM and the FPGA implementation in terms of average instructions per second is presented in Fig. 12. This figure demonstrates the performance difference discussed earlier, as well as the steep drop in VM performance above the 256-bit mark. In addition, Fig. 10 presents the average, one-time compilation overhead for generating and post-processing all necessary encrypted memory ranges [i.e., steps a) to c) discussed at the end of Section VI-A1] in our experiments. It should be noted that the overhead of encrypting individual values is heavily dependent on the underlying encryption routines [47], while the size of ranges and post-processing requirements are associated with the HEROIC framework functional requirements for different word sizes and security parameters.

In addition, a comparison of the encrypted HEROIC VM and FPGA implementation against unencrypted x86 and one instruction code running on a commodity processor is given in Fig. 11. The experimental results indicate that the overhead of the unencrypted one instruction execution compared to the encrypted HEROIC execution varies from 3.7× to 22.3× for the VM, and 6.4× to 40.9× for the FPGA on average, for different security parameters. Furthermore, the overhead of the encrypted HEROIC execution compared to the unencrypted x86 execution running natively, ranges from 787× to 4344× for the VM, and from 1297× to 8336× for the FPGA, on average. These results also indicate that the average overhead attributed only to the use of one instruction architecture is approximately 200×; as elaborated in the second challenge of Section IV-C, however, a single instruction architecture is necessary to remove the limitation of discriminating different instructions in the encrypted domain.

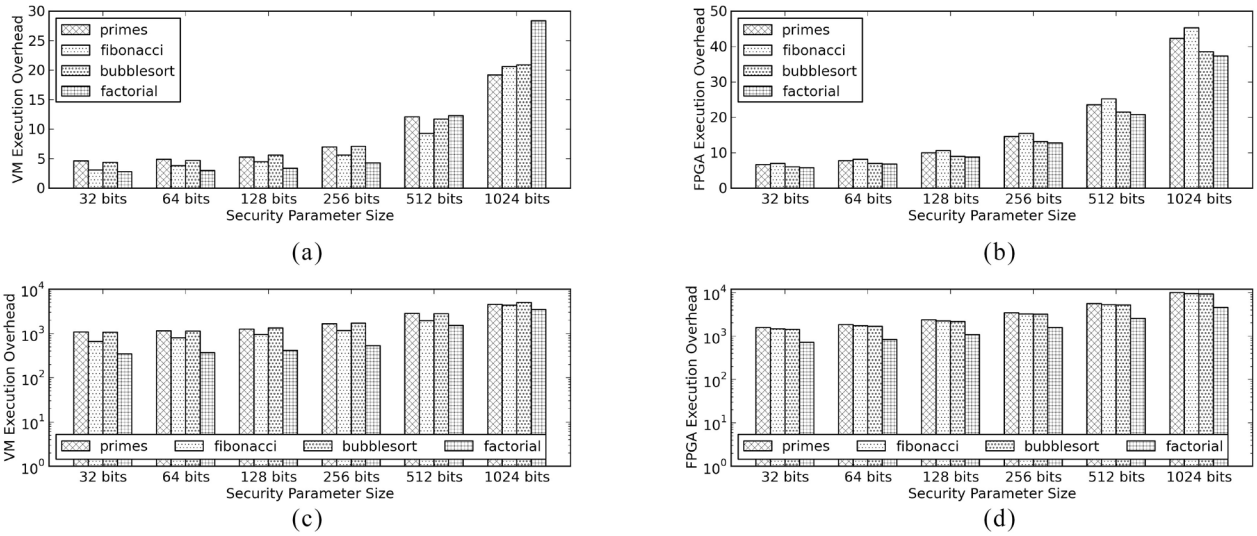


Fig. 11. Comparison of encrypted HEROIC execution overhead against unencrypted native x86 and one instruction code running on a commodity processor. (a) VM against unencrypted one instruction code. (b) Reconfigurable fabric against unencrypted one instruction code. (c) VM against unencrypted x86. (d) Reconfigurable fabric against unencrypted x86.

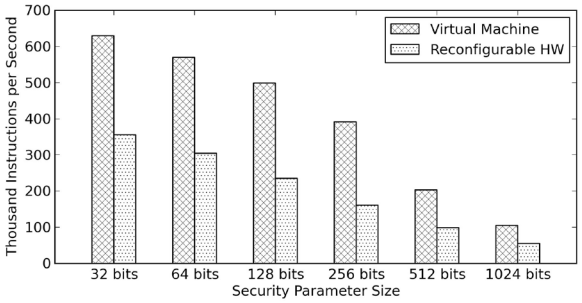


Fig. 12. Average IPS for VM and FPGA implementations (for different security parameters). On average, the VM performance is 1.7–2.5× faster than the FPGA.

### D. Memory Requirements Evaluation

In this section, we discuss the memory size requirements of our proposed HEROIC architecture. As previously presented, there are four different memories in the architecture: 1) main; 2) indirection; 3) sign/out-of-range; and 4) inverse lookup.

Section V-B5 and Fig. 6 demonstrated that, for a 16-bit architecture, the optimal number of additional bits $c$ required to differentiate encrypted memory addresses while maximizing the number of usable keys is six bits. Therefore, for a 16-bit word size, 22 bits are required. Additional experiments using different word size revealed that the number of additional bits is independent of the word size of the architecture; hence, using exactly six extra bits is sufficient. The number of extra bits does not scale with the security parameter. This observation is crucial, since it affects the addressing size requirements (i.e., wordsize + six bits) of the indirection memories.

Analogous to a RISC/CISC architecture, the HEROIC architecture will use the memory available to the CPU, generating out-of-memory requests for programs and memory accesses that violate the bounds. Given a memory of size $m$ bits, the indirection memory required to support the main memory is $2^{\lceil log(m) \rceil + c}$, where $c$ is the number of extra bits for addressing. Using the CRU optimization discussed in Section V-B5, this size can be reduced several times.

TABLE II
MEMORY SIZE REQUIREMENTS FOR AUXILIARY MEMORIES, FOR DIFFERENT SECURITY PARAMETER SIZES, AND ARCHITECTURE WORD SIZES

| Word size | Memory | Security Parameter Size 32 | 64 | 128 | 256 | 512 | 1024 | Unit |
|---|---|---|---|---|---|---|---|---|
| 8 | Sign/Out-of-range | 4 | 4 | 4 | 4 | 4 | 4 | KB |
| | Modular Inverse | 2 | 4 | 8 | 16 | 32 | 64 | KB |
| 16 | Sign/Out-of-range | 1 | 1 | 1 | 1 | 1 | 1 | MB |
| | Modular Inverse | 0.5 | 1 | 2 | 4 | 8 | 16 | MB |
| 32 | Sign/Out-of-range | 64 | 64 | 64 | 64 | 64 | 64 | GB |
| | Modular Inverse | 32 | 64 | 128 | 256 | 512 | 1024 | GB |
| 64 | Sign/Out-of-range | 256 | 256 | 256 | 256 | 256 | 256 | EB |
| | Modular Inverse | 128 | 256 | 512 | 1024 | 2048 | 4096 | EB |

The memory size bounds for the auxiliary memories of HEROIC, namely the sign/out-of-range and modular inverse lookup, for different security parameter sizes, are presented in Table II. The sign/out-of-range memories depend only on the unencrypted wordsize (8, 16, 32, or 64 bits) and thus do not scale with the security parameter size. On the other hand, the size of the modular inverse memory is directly related to the security parameter, as it correlates to the size of the encrypted operands. As the memory requirements increase exponentially with the wordsize, any size above 32-bits is prohibitive using today's hardware (the 64-bit architecture would require several exabytes of memory). While the 8- and 16-bit architectures comfortably fit in a modern FPGA or host CPU, the 32-bit architecture requires careful design and, probably, extensive operating system support for paging, as its auxiliary memory requirements lie in the region of 96–1088 GB, for security parameters of 32- and 1024-bits, respectively. A modern cloud computing environment, however, is a good candidate to run the HEROIC VM for 32-bit architecture programs, considering

the vast amounts of memory available that extend to several hundred gigabytes.

## VII. Related Work

Protecting sensitive memory contents against leaking has always been a concern. Lately, several different approaches have been proposed addressing different aspects of the same problem. Fletcher *et al*. [5] proposed a secure computer processor that employs a block cipher for encrypting the outputs of the CPU, as well as decrypting the inputs to the unit, incurring a 12–13.5× slowdown. In such an approach, the memory contents are protected by the block cipher, while the processor chip itself is considered a tamper proof container. Similarly, Suh *et al*. [48] proposed a tamper-evident and tamper-resistant processor, which incurs a 20%–60% execution overhead and includes a decryption key inside the chip, since data reside unencrypted inside the processor. Both these approaches, however, do not protect against side channel attacks to the processor, which can leak the internal unencrypted information, and at the same time introduce an additional key management problem in the process. Furthermore, these approaches have limitations against adversaries with early supply chain access or adversaries eventually able to bypass the physical tamper protection. In [49], an approach for secret program execution based on fully homomorphic encrypted circuits is proposed. This paper, however, does not provide any experimental evidence or applicability figures, and does not solve the problem of input/output communication. On the theoretical front, Breuer and Bowen [50] provided a conceptual proof of the correctness of an encrypted processing unit, but the proposal do not actually solve the problem of the ALU implementation, and does not provide support for self-modifying code.

On the virtualization frontier, several approaches have been proposed, that protect VMs against rogue or vulnerable hypervisors. Szefer and Lee [51] proposed architectural changes in the CPU to use lookup tables that enforce memory protection, while in [52] it is proposed to use address independent seed encryption and Bonsai Merkle tree for memory encryption and integrity protection. Both proposals, however, are limited against threats with physical access, as in the former case, the data remains unencrypted in memory, while in the latter case the protection key is vulnerable to eavesdropping and side channels. Similarly, Zhang *et al*. [53] proposed to use a lightweight security monitor under the hypervisor for security protection, while in [54] hardware extensions are proposed to isolate VMs from the hypervisors. Both approaches, however, are vulnerable to side channels and probing attacks (such as reading DRAM after power-off or probing external buses) and only assume an honest cloud service provider. Additionally, the framework proposed in [55] protects VMs against malicious hypervisors through build-in integrity monitoring based on a dynamic root of trust, but does not consider threats with physical access to the CPU and the trusted computing base.

Another approach for protecting sensitive information, that is currently being used commercially, is to use a secure coprocessor inside the central processing unit system on chip. As discussed in [56], hardware level isolation and secret keys provisioned at fabrication level could ensure protection of sensitive information against network threats, even in light of the software kernel being compromised due to underlying vulnerabilities. This approach, however, is limited against side channel attacks that could leak the internal key, as well as threats with early supply chain access. At the instruction set architecture level, the current state-of-the-art is to use an inverse sandbox approach [57], [58]. This approach essentially creates a secure container (usually dubbed as an "enclave") that would protect sensitive information from being attacked by another process that has escalated privileges. In this design, however, physical attacks are not considered, and the secret key may also be leaked through side channels.

## VIII. Conclusion

In this paper, we presented the HEROIC framework, consisting of: 1) VM that runs on commodity processors and 2) an encrypted processor RTL design for FPGAs. Based on our assumed threat model, HEROIC can protect cloud computing against a range of threats that potentially have access to the internals of microprocessors and can leak sensitive information. The HEROIC architecture supports general-purpose computation using homomorphic encryption and is based on a single instruction architecture free of operation codes. Contrary to the previous work, this approach removes the need for shared keys that can be a potential target for attackers.

## References


[1] C. Barron, H. Yu, and J. Zhan, "Cloud computing security case studies and research," in *Proc. World Congr. Eng.*, London, U.K., 2013, pp. 1287–1291.

[2] IBM Security Solutions. (Aug. 2010). "2010 Mid-year trend and risk report," pp. 49–56. [Online]. Available: http://www-05.ibm.com/fr/pdf/IBM_X-Force2010_Mid-Year_Trend_and_Risk_Report.pdf, accessed Jun. 6, 2014.

[3] S. Pearson, "Taking account of privacy when designing cloud computing services," in *Proc. ICSE Workshop Softw. Eng. Challenges Cloud Comput.*, Vancouver, BC, Canada, 2009, pp. 44–52.

[4] H. Takabi, J. B. Joshi, and G.-J. Ahn, "Security and privacy challenges in cloud computing environments," *IEEE Security Privacy*, vol. 8, no. 6, pp. 24–31, Nov./Dec. 2010.

[5] C. W. Fletcher, M. van Dijk, and S. Devadas, "A secure processor architecture for encrypted computation on untrusted programs," in *Proc. ACM Workshop Scal. Trust. Comput.*, Raleigh, NC, USA, 2012, pp. 3–8.

[6] G. T. Becker, F. Regazzoni, C. Paar, and W. P. Burleson, "Stealthy dopant-level hardware Trojans," in *Proc. 15th Int. Workshop Cryptograph. Hardw. Embedded Syst. Workshop*, Santa Barbara, CA, USA, 2013, pp. 197–214.

[7] C. Fontaine and F. Galand, "A survey of homomorphic encryption for nonspecialists," *EURASIP J. Inf. Security*, vol. 2007, no. 1, pp. 26–35, 2007.

[8] A. Teller, "Turing completeness in the language of genetic programming with indexed memory," in *Proc. 1st IEEE Conf. Evol. Comput.*, Orlando, FL, USA, 1994, pp. 136–141.

[9] M. Sipser, *Introduction to the Theory of Computation*. Boston, MA, USA: Cengage Learn., 2013.

[10] Esolangs. (Jun. 2014). *Bounded–Storage Machine*. [Online]. Available: http://esolangs.org/wiki/Bounded-storage_machine

[11] C. Böhm and G. Jacopini, "Flow diagrams, Turing machines and languages with only two formation rules," *Commun. ACM*, vol. 9, no. 5, pp. 366–371, 1966.

[12] W. F. Gilreath and P. A. Laplante, *Computer Architecture: A Minimalist Perspective*. New York, NY, USA: Springer, 2003, pp. 55–69.

[13] R. Rojas, "Conditional branching is not necessary for universal computation in von Neumann computers," *J. Univ. Comput. Sci.*, vol. 2, no. 11, pp. 756–768, 1996.

[14] O. Mazonka and A. Kolodin, "A simple multi-processor computer based on subleq," *ArXiv ePrint*, arXiv:1106.2593 [cs.DC], 2011.

[15] P. J. Nürnberg, U. K. Wiil, and D. L. Hicks, "A grand unified theory for structural computing," in *Metainformatics*. Berlin, Germany: Springer, 2004, pp. 1–16.
[16] W. L. van der Poel, "The essential types of operations in an automatic computer," *Nachrichtentechnische Fachberichte*, vol. 4, pp. 144–145, 1956.
[17] Esolangs. (Apr. 2014). *One Instruction Set Computer*. [Online]. Available: http://esolangs.org/wiki/OISC
[18] D. Micciancio, "A first glimpse of cryptography's Holy Grail," *Commun. ACM*, vol. 53, no. 3, p. 96, 2010.
[19] C. Gentry, "Fully homomorphic encryption using ideal lattices," in *Proc. 41st Annu. ACM Symp. Theory Comput.*, Bethesda, MD, USA, 2009, pp. 169–178.
[20] X. He, M.-O. Pun, and C.-C. Kuo, "Secure and efficient cryptosystem for smart grid using homomorphic encryption," in *Proc. IEEE Innov. Smart Grid Technol.*, Washington, DC, USA, 2012, pp. 1–8.
[21] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Commun. ACM*, vol. 21, no. 2, pp. 120–126, 1978.
[22] T. ElGamal, "A public key cryptosystem and a signature scheme based on discrete logarithms," *IEEE Trans. Inf. Theory*, vol. 31, no. 4, pp. 469–472, Jul. 1985.
[23] P. Paillier, "Public-key cryptosystems based on composite degree residuosity classes," in *Advances in Cryptology–EUROCRYPT*. Berlin, Germany: Springer, 1999, pp. 223–238.
[24] S. Goldwasser and S. Micali, "Probabilistic encryption & how to play mental poker keeping secret all partial information," in *Proc. 14th Annu. ACM Symp. Theory Comput.*, San Francisco, CA, USA, 1982, pp. 365–377.
[25] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, "(Leveled) fully homomorphic encryption without bootstrapping," in *Proc. Innov. Theor. Comput. Sci. Conf.*, Gwangju, Korea, 2012, pp. 309–325.
[26] S. Halev and V. Shoup. (2013). *Design and Implementation of a Homomorphic-Encryption Library*. [Online]. Available: https://github.com/shaih/HElib
[27] J. Fan and F. Vercauteren, "Somewhat practical fully homomorphic encryption," *Cryptology ePrint Archive*, Rep. 2012/144, 2012.
[28] D. Stehlé and R. Steinfeld, "Faster fully homomorphic encryption," in *Advances in Cryptology–ASIACRYPT*. Berlin, Germany: Springer, 2010, pp. 377–394.
[29] M. van Dijk, C. Gentry, S. Halevi, and V. Vaikuntanathan, "Fully homomorphic encryption over the integers," in *Advances in Cryptology–EUROCRYPT*. Berlin, Germany: Springer, 2010, pp. 24–43.
[30] R. Cramer, R. Gennaro, and B. Schoenmakers, "A secure and optimally efficient multi-authority election scheme," *Eur. Trans. Telecommun.*, vol. 8, no. 5, pp. 481–490, 1997.
[31] C. Gentry, "A fully homomorphic encryption scheme," Ph.D. dissertation, Dept. Comput. Sci., Stanford Univ., Stanford, CA, USA, 2009.
[32] B. Schneier. (Apr. 2014). *Schneier on Security: Heartbleed*. [Online]. Available: https://www.schneier.com/blog/archives/2014/04/heartbleed.html
[33] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Cross-VM side channels and their use to extract private keys," in *Proc. Conf. Comput. Commun. Security (CCS)*, Raleigh, NC, USA, 2012, pp. 305–316.
[34] N. G. Tsoutsos and M. Maniatakos, "Fabrication attacks: Zero-overhead malicious modifications enabling modern microprocessor privilege escalation," *IEEE Trans. Emerg. Topics Comput.*, vol. 2, no. 1, pp. 81–93, Mar. 2013.
[35] O. Mazonka. (Apr. 2014). *Subleq*. [Online]. Available: http://mazonka.com/subleq/
[36] B. Barak *et al.*, "On the (im) possibility of obfuscating programs," in *Advances in Cryptology–CRYPTO*. Berlin, Germany: Springer, 2001, pp. 1–18.
[37] M. Prabhakaran and M. Rosulek, "Homomorphic encryption with CCA security," in *Automata, Languages and Programming*. Berlin, Germany: Springer, 2008, pp. 667–678.
[38] J. Katz and Y. Lindell, *Introduction to Modern Cryptography*. Boca Raton, FL, USA: CRC Press, 2008.
[39] O. Mazonka. (2009). *Higher Subleq Compiler into OISC Language*. [Online]. Available: http://mazonka.com/subleq/hsq.html
[40] C. Stein, R. L. Drysdale, and K. P. Bogart, *Discrete Mathematics for Computer Scientists*. Boston, MA, USA: Pearson, 2011.
[41] R. Agrawal, J. Kiernan, R. Srikant, and Y. Xu, "Order preserving encryption for numeric data," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, Paris, France, 2004, pp. 563–574.
[42] A. Putnam *et al.*, "A reconfigurable fabric for accelerating large-scale datacenter services," in *Proc. 41st Annu. Int. Symp. Comput. Archit. (ISCA)*, Minneapolis, MN, USA, 2014, pp. 13–24.
[43] D. Bryant. (Jun. 2014). *Disrupting the Data Center to Create the Digital Services Economy*. [Online]. Available: https://communities.intel.com/community/itpeernetwork/datastack/blog/2014/06/18/disrupting-the-data-center-to-create-the-digital-services-economy
[44] P. L. Montgomery, "Modular multiplication without trial division," *Math. Comput.*, vol. 44, no. 170, pp. 519–521, 1985.
[45] A. Daly and W. Marnane, "Efficient architectures for implementing Montgomery modular multiplication and RSA modular exponentiation on reconfigurable logic," in *Proc. ACM/SIGDA Int. Symp. Field-Program. Gate Arrays*, Monterey, CA, USA, 2002, pp. 40–49.
[46] Free Software Foundation. (Apr. 29, 2014). *The GNU Multiple Precision Arithmetic Library*. [Online]. Available: https://gmplib.org/
[47] M. Ivanov. (2011). *Pure Python Paillier Homomorphic Cryptosystem*. [Online]. Available: https://github.com/mikeivanov/paillier
[48] G. E. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas, "AEGIS: Architecture for tamper-evident and tamper-resistant processing," in *Proc. 17th Annu. Int. Conf. Supercomput.*, Florence, Italy, 2003, pp. 160–171.
[49] M. Brenner, J. Wiebelitz, G. von Voigt, and M. Smith, "Secret program execution in the cloud applying homomorphic encryption," in *Proc. Digit. Ecosyst. Technol. Conf. (DEST)*, Daejeon, Korea, 2011, pp. 114–119.
[50] P. T. Breuer and J. P. Bowen, "A fully homomorphic crypto-processor design," in *Engineering Secure Software and Systems*. Berlin, Germany: Springer, 2013, pp. 123–138.
[51] J. Szefer and R. B. Lee, "Architectural support for hypervisor-secure virtualization," *ACM SIGARCH Comput. Archit. News*, vol. 40, no. 1, pp. 437–450, 2012.
[52] Y. Xia, Y. Liu, and H. Chen, "Architecture support for guest-transparent VM protection from untrusted hypervisor and physical attacks," in *Proc. 19th Int. Symp. High Perform. Comput. Archit. (HPCA)*, Shenzhen, China, 2013, pp. 246–257.
[53] F. Zhang, J. Chen, H. Chen, and B. Zang, "CloudVisor: Retrofitting protection of virtual machines in multi-tenant cloud with nested virtualization," in *Proc. Symp. Oper. Syst. Prin.*, Cascais, Portugal, 2011, pp. 203–216.
[54] S. Jin, J. Ahn, S. Cha, and J. Huh, "Architectural support for secure virtualization under a vulnerable hypervisor," in *Proc. 44th Annu. IEEE/ACM Int. Symp. Microarchit.*, Porto Alegre, Brazil, 2011, pp. 272–283.
[55] A. Vasudevan *et al.*, "Design, implementation and verification of an extensible and modular hypervisor framework," in *Proc. IEEE Symp. Security Privacy (SP)*, Berkeley, CA, USA, 2013, pp. 430–444.
[56] Apple Inc. (Apr. 2014). *iOS Security*. [Online]. Available: http://images.apple.com/iphone/business/docs/iOS_Security_Feb14.pdf
[57] Intel Corporation. (2013). *Software Guard Extensions Programming Reference*. [Online]. Available: https://software.intel.com/sites/default/files/329298-001.pdf (Accessed: 4/29/14)
[58] F. McKeen *et al.*, "Innovative instructions and software model for isolated execution," in *Proc. 2nd Int. Workshop Hardw. Archit. Support Security Privacy (HASP)*, Tel-Aviv, Israel, 2013, pp. 1–8.

**Nektarios Georgios Tsoutsos** (S'13) received the five-year Diploma degree in electrical and computer engineering from the National Technical University of Athens, Athens, Greece, and the M.Sc. degree in computer engineering from Columbia University, New York, NY, USA. He is currently pursuing the Ph.D. degree in computer science with the New York University Polytechnic School of Engineering, New York.

His current research interests include computer security, encrypted computation, computer architecture and embedded systems.

**Michail Maniatakos** (S'08–M'12) received the B.Sc. and M.Sc. degrees in computer science and embedded systems from the University of Piraeus, Piraeus, Greece, and the M.Sc., M.Phil., and Ph.D. degrees from Yale University, New Haven, CT, USA.

He is an Assistant Professor of Electrical and Computer Engineering with New York University (NYU) Abu Dhabi, Abu Dhabi, U.A.E., and a Research Assistant Professor with the NYU Polytechnic School of Engineering, New York, NY, USA. He is the Director of the MoMA Laboratory, NYU Abu Dhabi. His current research interests include robust microprocessor architectures, hardware security, and heterogeneous microprocessor architectures. He has authored several publications in IEEE transactions and conference papers.