# HEROIC: Homomorphically EncRypted One Instruction Computer

Nektarios Georgios Tsoutsos
Computer Science and Engineering
New York University Polytechnic School of Engineering
E-mail: nektarios.tsoutsos@nyu.edu

Michail Maniatakos
Electrical and Computer Engineering
New York University Abu Dhabi
E-mail: michail.maniatakos@nyu.edu

***Abstract*—As cloud computing becomes mainstream, the need to ensure the privacy of the data entrusted to third parties keeps rising. Cloud providers resort to numerous security controls and encryption to thwart potential attackers. Still, since the actual computation inside cloud microprocessors remains unencrypted, the opportunity of leakage is theoretically possible. Therefore, in order to address the challenge of protecting the computation inside the microprocessor, we introduce a novel general purpose architecture for secure data processing, called HEROIC (Homomorphically EncRypted One Instruction Computer). This new design utilizes a single instruction architecture and provides native processing of encrypted data at the architecture level. The security of the solution is assured by a variant of Paillier's homomorphic encryption scheme, used to encrypt both instructions and data. Experimental results using our hardware-cognizant software simulator, indicate an average execution overhead between 5 and 45 times for the encrypted computation (depending on the security parameter), compared to the unencrypted variant, for a 16-bit single instruction architecture.**

***Index Terms*—Encrypted processor, homomorphic encryption, Paillier, cloud computing, one instruction set computer**

## I. Introduction

The decreasing cost of cloud computing services has rendered outsourcing computation a very attractive option for computationally demanding applications. Pay-as-you-go services, zero maintenance/upgrade costs and unlimited scalability are some of the benefits of using cloud computing services. At the same time, however, there is no shortage of concerns about the privacy of the data, as security vulnerabilities are not uncommon: Prominent examples of affected providers include Amazon EC2/S3 and LastPass in 2011, as well as Dropbox in 2012 [1]. While in privately owned datacenters the privacy of the data and executed programs is protected by many logical and physical controls, in the cloud users are asked to assign control over their information to a third party that is assumed to be trusted [2], [3]. In such cases, users essentially have to study the safety record and trust the reputation of the cloud provider itself. Otherwise, they need to incur the –potentially prohibitive– costs of building and maintaining in-house server farms. As a result, there is a need for protecting the confidentiality of the information processed in the cloud, in a more definitive manner.

One popular solution towards addressing these concerns is the use of a cryptographic algorithm (also called *encryption scheme*); this makes information unreadable by unauthorized entities, and thus protects confidentiality. Even though this approach is very useful for the storage and transmission of information, it has not been demonstrated to protect data and instructions *inside* the cloud microprocessors, *without sharing* cryptographic keys with the host. Secure microprocessors have been proposed in the past [4], but their threat model essentially assumes that the pipeline is *trustworthy*. In these proposals, the CPU inputs are decrypted before processing, while CPU outputs are encrypted again, effectively limiting the attack surface to the –usually tamper- proof/resistant– microprocessor chip. In theory, however, this approach *remains vulnerable* to anyone capable of eavesdropping on the pipeline or extracting the cryptographic keys from inside the processor, bypassing tampering protections. Such an attack has already been demonstrated in [5], where a sub-transistor level Trojan is used to extract sensitive information from the chip. These two examples essentially define the threat model that we assume in this work.

In order to prevent this kind of eavesdropping from inside the pipeline, cloud microprocessors need to execute encrypted instructions directly, without ever decrypting them. Modern computer architectures, however, like CISC and RISC, are unable to support execution of encrypted instructions, as the *instruction decoder* in the pipeline is unable to determine the intended operations. Indeed, modern architectures have been designed for efficiency and performance, and not for security and privacy, and thus cannot be used for processing encrypted instructions.

Motivated by the lack of architecture support for the assumed threat model, our contribution is a novel idea for a *general purpose encrypted computer architecture*, called HEROIC. The HEROIC architecture is capable of executing instructions and manipulating data in the encrypted domain, and thus preserves the confidentiality of both the algorithm and the data. The key idea behind HEROIC is to use *a single instruction set architecture*, making instruction decryption unnecessary. With a judicious choice for the single instruction, single set instruction computing is capable of Turing-complete computation; therefore, under certain configurations, the new architecture can provide comparable processing performance to existing CISC and RISC designs. In addition, HEROIC employs *homomorphic encryption*, allowing meaningful manipulation of data directly in the encrypted domain [6].

To the best of our knowledge, HEROIC is the first effort towards an encrypted computer architecture that can *natively* process encrypted data and can be *practically* used for protecting data privacy in cloud computing services, without ever sharing cryptographic keys with the host machine. In order to

978-3-9815370-2-4/DATE14/©2014 EDAA

achieve that, HEROIC combines the simplicity and flexibility of single instruction architectures, as presented in Section II, along with the computational properties of homomorphic encryption, described in Section III. The rest of the paper is organized as follows: Section IV describes the HEROIC architecture along with design considerations and security implications of creating a 16-bit encrypted architecture. Section V provides experimental performance results of our design, followed by conclusions and future directions in Section VI.

## II. Single Instruction Architectures

Single instruction architectures (also called One Instruction Set Computers – OISC) are architectures designed to support only one instruction. Based on the micro-operations of the selected instruction, single instruction computer variants are capable of *Turing-Complete* computation [7]. This indicates that one instruction set computers are very powerful despite the simplicity of the design, and can achieve high throughput in certain *parallel computing* configurations [7]. Since OISC cores are very compact, several of them can be used in place of a bigger RISC core. These properties make single instruction computers an appropriate alternative to ordinary RISC computers, especially in a cloud setting.

The lack of multiple instructions allows HEROIC to maintain the confidentiality of the instructions and the algorithm as well. On the contrary, having multiple instructions (i.e. different opcodes) would make this impossible, since whenever the encryption key changes, the opcodes would appear completely different. In this scenario, the encrypted computer would be required to decrypt each instruction to identify what is the next execution step. Providing a decryption key, however, is unacceptable in our threat model and would defy the purpose of requiring encryption in the first place. HEROIC never decrypts instruction opcodes, since they are redundant and *all* instructions are the *same* (the computation is determined only by the encrypted instruction arguments). Thus, the proposed solution solves the problem of how to discriminate program instructions while in the encrypted domain. If an eavesdropper would attempt to guess the instruction stream (i.e. the executed algorithm), the eavesdropper would gain no additional information because all instructions are the same and the arguments are already encrypted.

Several architecture variants based on the single instruction used exist. Common Turing-complete variants include *add and branch unless positive* (`addleq`), *subtract and branch unless positive* (`subleq`), *plus one and branch if equal* (`pleq`), and *reverse subtract and skip if borrow*. 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 one instruction set architectures compatible with homomorphic encryption schemes, which allow applying that operation over encrypted data. As further discussed in Section IV, HEROIC implements the `subleq` instruction.

## III. Homomorphic Encryption Background

In cryptography, a *homomorphic* scheme is defined as an encryption scheme that supports applying a function directly on encrypted data, so that the output decryption equals the result after applying another equivalent function on unencrypted data [8]. 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 addition, or 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 [9], 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 [10]. 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, the El-Gamal scheme, the Paillier scheme and the Goldwasser-Micali scheme. Fully homomorphic schemes are the Gentry scheme and its variants (e.g. the BGV scheme and others). Partially homomorphic schemes are further categorized based on the supported function, as *additive* homomorphic (like the Paillier and exponential El-Gamal) and *multiplicative* homomorphic (like RSA, standard El-Gamal, etc) for supporting addition and multiplication respectively.

Formally, homomorphism is defined as follows:

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

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 [11], partially homomorphic schemes are the only viable candidates for a practical encrypted architecture. As Turing-complete variants of single instruction architectures require addition or subtraction, the HEROIC architecture needs 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. 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 detailed explanation of the Paillier scheme is out of the scope of this paper, and the interested reader can find an extensive presentation in [12].

## IV. The HEROIC Architecture

The HEROIC architecture builds upon the use of the `subleq` instruction. `Subleq` requires three arguments (namely A, B and C), and its instruction micro-

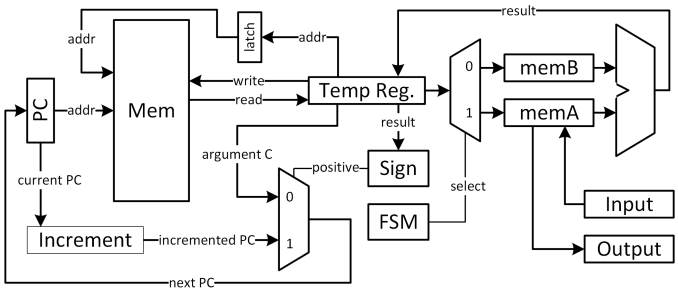


Fig. 1. Basic components and datapath of the `subleq` OISC computer.

operation is defined as `Mem[B] = Mem[B] - Mem[A]; if Mem[B] ≤ 0 then goto C else goto next;`

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

(a) `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)
(b) `subleq` permits modification of any memory location, and thus one instruction may modify an argument of another instruction (*self-modifying code* is allowed)
(c) every branch micro-operation requires a comparison of the subtraction result with zero
(d) arguments A, B and C are always stored in subsequent memory locations

A typical `subleq`-based architecture is presented in Fig. 1, which demonstrates all necessary components along with their interconnections. In this case study, without loss of generality, we explore a 16-bit single instruction architecture (32-bit architecture is possible as well). 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 ALU is responsible for subtracting two given arguments stored in registers memA and memB. Register memA is also responsible for encrypted I/O operations. The control finite state machine (FSM) and a sign identification unit are responsible for coordination and branching respectively, while a program counter (PC) along with increment logic complete the design.

In HEROIC, each memory value should be encrypted using the Paillier scheme. Paillier arguments are $(2 * n)$ bits wide, where $n$ is the security parameter. The basic datapath presented earlier, however, cannot support the encrypted arguments of HEROIC without major modifications. The next section elaborates on the encrypted design challenges and their respective solutions.

### A. Design Considerations

*1) Encrypted Memory Addressing:* Since HEROIC uses a unified main memory for instructions and data, and because instructions are allowed to modify the arguments of other instructions, the architecture requires encrypted memory addressing. The program counter references memory locations

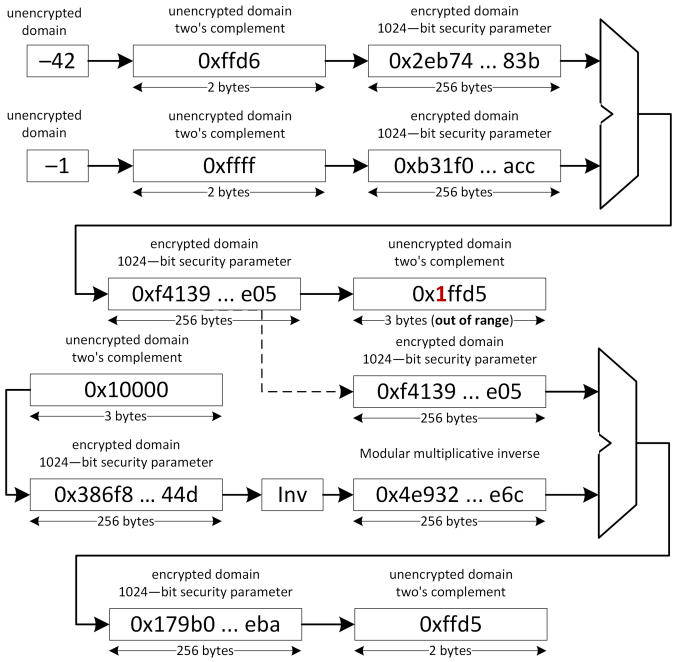


Fig. 2. Out of range correction for homomorphic addition of representations of negative numbers.

directly, while instruction arguments reference locations indirectly. Given that these instruction arguments are already encrypted, and the decryption key is not provided, it is necessary that all memory addresses are encrypted under the same key. Our proposed HEROIC architecture, however, properly satisfies this requirement and supports encrypted program counters as well as encrypted addressing.

*2) Matching Instruction Arguments:* Going from the standard `subleq` design concept into HEROIC immediately 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 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. The value of this pointer is used by the program counter (PC) to find the next instruction argument in the execution trail, since, simply incrementing the PC would be incorrect.

*3) Out of Range Correction:* In this work, without loss of generality, we are investigating a 16-bit architecture ported in the encrypted domain. This means that the width of each memory location 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 (2048-bit based on security parameter of 1024-bit for Pailler encryptions [12]) creates *out of range* discrepancies. In order to clarify this problem, let us consider a homomorphic addition of two negative numbers. As demonstrated in the example of Fig. 2, 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 out of range 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 with the encrypted program), in order to get the expected result.

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

*4) Homomorphic Subtraction:* The HEROIC computer 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 out of range lookup memory presented above) is necessary. This memory returns the multiplicative inverse of the encrypted value of the subtrahend. This 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 out of range correction as described earlier (Fig. 2).

An alternative approach would have been to use concepts of the `addleq` one instruction set computing variant, which does not require subtraction. In this case, the inverse lookup memory is not needed, since the addition operation in `addleq` is directly supported by the Paillier scheme. A major setback with `addleq`, however, is that it is significantly harder to program, because a high level compiler is not yet available, and is less efficient in terms of programming compared to `subleq` [13].

*5) Memory Addressing Size:* Given a security parameter size of 1024 bits, combined with encrypted memory addressing, would require HEROIC to support memory addresses 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 work, we propose two optimizations, which effectively reduce the required memory addressing size to a practical implementation.

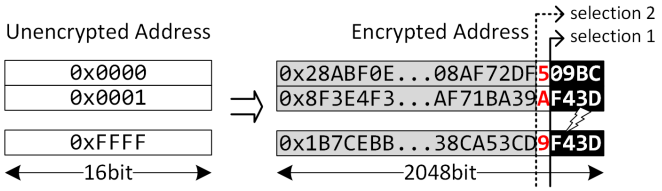


Fig. 3. Identifying the proper number of lower bits required to differentiate encrypted memory addresses.

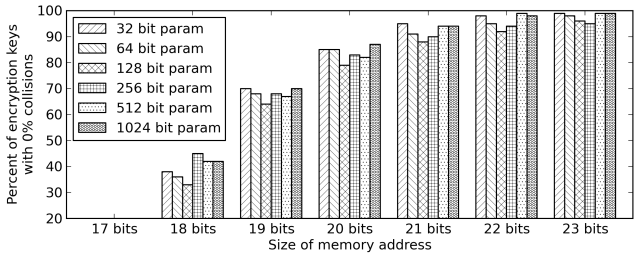


Fig. 4. Percentage of encryption keys with 0% collisions in $2^{17}$ encryptions for different memory addressing sizes and security parameter sizes.

Since the unencrypted domain is 16 bits, the main memory should support 16-bit addressing. While Paillier encryptions can be 2048-bit wide, we can use only the *lower bits* of these 2048 bits to request memory addresses. Fig. 3 shows how the lower bits of an encrypted address can be used to differentiate 16-bit main memory addresses. As shown in the example, if 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` are the same. If the selection included 20 bits (selection 2), this would allow proper separation of all memory locations.

Fig. 4 presents the minimum number of address bits necessary to discriminate *at most* $2^{17}$ different encrypted addresses (since the out of range 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 re-encryption* with another key is necessary. It should be emphasized that collisions can only 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.

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 2060-bit value (for the highest security parameter size), there are many memory locations of *very large width* 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 24.09MB, instead of approximately 1GB without the optimization. Similarly, the multiplicative inverse memory has now a size of 24MBs instead of 1GB. The two-level memory optimization is presented at the block diagram of the HEROIC architecture (Fig. 5).

*6) Jump Decisions in the Encrypted Domain:* Another challenge in the HEROIC design 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 [14], these are not homomorphic and cannot be used in our case study.

The proposed solution to this problem would be to use 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 our case study, the sign lookup memory would return the sign for encryptions of numbers in the range from 0 to $(2^{16}-1)$. Following the analysis of Section IV-A5, the sign lookup memory can also benefit from the first address truncation optimization and use only 22 bits of memory address size.

Fig. 5 provides an abstract view of the additional units and memories required in the HEROIC architecture, based on the previous discussion.

### B. Security Implications

The threat model of our proposed architecture is a *rational, curious* but *honest* cloud service provider, that would support HEROIC, but may also try to snoop inside the processor. One security implication of the design is the use of a *deterministic* variant of Paillier's encryption scheme. This is mandatory, since memory references in OISC computers must always be the same, to succeed. In theory, determinism eliminates *semantic security*, but in our context, *chosen-plaintext* attacks are not practical: the second half of Paillier's public key (needed for encryption of chosen values) is never revealed to the cloud provider or anyone else besides the program owner. This eventually prevents an attacker from obtaining encryptions of known values, making it significantly harder to guess the information sent to HEROIC. Additional details on chosen-plaintext attack feasibility and the need for determinism can be found in the security discussion section of [15].

Another implication is that *ciphertext-only* attacks exploiting *frequency analysis* may still be possible, due to the additive homomorphism over the used plaintext range and the effects of deterministic encryptions. Furthermore, the provided lookup memories for *sign*, *out of range* and *modular inverse* identification may assist frequency analysis as well. Ongoing work explores the addition of noise to the encrypted values to effectively reduce the practicality of the aforementioned attacks, while ensuring execution correctness.

## V. Performance Results

### A. Experimental Setup

The HEROIC architecture presented in this paper has been evaluated using a hardware-cognizant software simulator developed in Python. All experiments were executed on a virtual Ubuntu system with 2GB RAM and two Intel i7 cores at 2.6GHz. For encrypting experimental programs, an open-source educational Python implementation of Paillier's cryptosystem was used from [16], in deterministic configuration.

In order to evaluate the performance and correctness of encrypted execution, four sample programs have been compiled to assembly language from original `C` code using a `subleq` compiler from [13], and then instructions were encrypted using Paillier's scheme with different security parameter sizes (the encryption key is necessary during compilation). The software simulator was configured to emulate an FPGA implementation running at 200MHz, using the following delay figures:

(a) *memory access* delay has been modeled at 100ns for memories up to 16GB [17],
(b) *modular multiplication* delay has been modeled at $h+3$ clock cycles (i.e. $5h+15$ ns), for argument size of $h$ bits, based on the fast *Montgomery* algorithm [18], and
(c) *subtraction* delay in the unencrypted domain assumed a *Kogge-Stone* implementation and has been modeled at 6ns [19] for 16-bit arguments (used for comparing the unencrypted design with HEROIC).

### B. Simulation Results

The experimental results from the developed simulator are presented in Table I. The simulator has been configured to compare the performance of the proposed HEROIC computer against the unencrypted `subleq` variant. As it becomes evident from the results of that Table, the size of the security parameter has an impact on the overall performance, since the delay of the modular multiplication in the ALU is correlated with the size of encrypted arguments. The results of Table I also indicate that depending on the security parameter size, the average performance overhead of the proposed HEROIC design is from about *5* up to about *45 times* slower, compared to the unencrypted `subleq` design.

In addition, Table I provides information on the number of clock cycles required for execution of each experimental program, using multiple security parameter sizes. The results suggest that there is a maximum difference of *two* orders of magnitude in terms of clock cycles between the encrypted HEROIC and the unencrypted `subleq`. This difference is primarily attributed to the expensive modular multiplications and the memory read/write operations of HEROIC. Ongoing work explores hiding the memory latency by prefetching the next instruction arguments, while modular multiplication takes place; this effectively makes the ALU the only element in the critical path, potentially increasing performance. Furthermore, since HEROIC cores may be parallelized as in [7], encryption overheads can be concealed using parallel configurations.

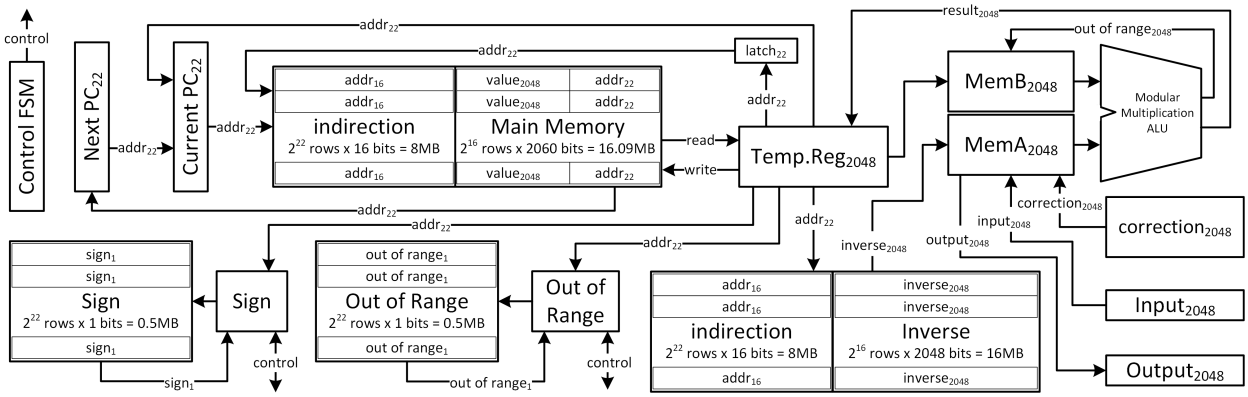


Fig. 5. Abstract view of additional units and memories of the HEROIC encrypted computer (security parameter 1024-bits).

TABLE I
SIMULATED EXECUTION CLOCK CYCLES AND AVERAGE OVERHEAD FOR DIFFERENT SECURITY PARAMETER SIZES

| Benchmark | Unencrypted clock cycles | Encrypted clock cycles for security parameter size | | | | | |
|---|---|---|---|---|---|---|---|
| | | 32-bit | 64-bit | 128-bit | 256-bit | 512-bit | 1024-bit |
| **primes** | $2.28 * 10^8$ | $1.24 * 10^9$ | $1.53 * 10^9$ | $2.12 * 10^9$ | $3.29 * 10^9$ | $5.64 * 10^9$ | $1.03 * 10^{10}$ |
| **factorial** | $5.84 * 10^6$ | $3.16 * 10^7$ | $3.92 * 10^7$ | $5.43 * 10^7$ | $8.45 * 10^7$ | $1.45 * 10^8$ | $2.66 * 10^8$ |
| **bubblesort** | $1.06 * 10^7$ | $5.77 * 10^7$ | $7.14 * 10^7$ | $9.89 * 10^7$ | $1.54 * 10^8$ | $2.64 * 10^8$ | $4.84 * 10^8$ |
| **fibonacci** | $1.89 * 10^7$ | $1.02 * 10^8$ | $1.27 * 10^8$ | $1.76 * 10^8$ | $2.74 * 10^8$ | $4.71 * 10^8$ | $8.63 * 10^8$ |
| **Average Overhead (times)** | | **5.4** | **6.7** | **9.3** | **14.4** | **24.8** | **45.4** |

## VI. CONCLUSION AND FUTURE DIRECTIONS

In this work we presented a novel architecture for an encrypted general purpose computer design, that combines the simplicity and flexibility of one instruction set architectures with the security properties of homomorphic encryption. The proposed design is capable of *natively* executing encrypted programs written in `subleq` assembly as well as processing encrypted data directly. Experimental results indicate that HEROIC has the potential to be an effective solution to the problem of confidentiality and privacy in cloud computing, having an average execution overhead between 5 to 45 times, compared to the unencrypted `subleq` execution.

The introduction of HEROIC opens up several research directions: These include exploring performance improvements of HEROIC using cache memories, branch prediction, prefetching, pipelining, as well as incorporating more than one execution units in the design. Currently, an FPGA implementation of HEROIC is being developed in order to obtain more accurate performance figures. In addition, a multicore version of this architecture that exploits parallelism and significantly reduces performance overhead will also be explored.

## REFERENCES


[1] C. Barron, H. Yu, and J. Zhan, "Cloud computing security case studies and research," in *Proceedings of the World Congress on Engineering*, 2013, pp. 1287–1291.

[2] S. Pearson, "Taking account of privacy when designing cloud computing services," in *ICSE Workshop on Software Engineering Challenges of Cloud Computing*, 2009, pp. 44–52.

[3] 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, 2010.

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

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

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

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

[8] D. Micciancio, "A first glimpse of cryptography's holy grail," *Communications of the ACM*, vol. 53, no. 3, pp. 96–96, 2010.

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

[10] X. He, M.-O. Pun, and C.-C. Kuo, "Secure and efficient cryptosystem for smart grid using homomorphic encryption," in *Innovative Smart Grid Technologies*, 2012, pp. 1–8.

[11] C. Gentry, *A fully homomorphic encryption scheme*, Ph.D. thesis, Stanford University, 2009.

[12] P. Paillier, "Public-key cryptosystems based on composite degree residuosity classes," in *Advances in cryptology–EUROCRYPT99*, 1999, pp. 223–238.

[13] O. Mazonka, "Higher subleq compiler into OISC language," [Online]. Available: http://mazonka.com/subleq/hsq.html.

[14] R. Agrawal, J. Kiernan, R. Srikant, and Y. Xu, "Order preserving encryption for numeric data," in *ACM SIGMOD International Conference on Management of Data*, 2004, pp. 563–574.

[15] N. G. Tsoutsos and M. Maniatakos, "Investigating the Application of One Instruction Set Computing for Encrypted Data Computation," in *Security, Privacy, and Applied Cryptography Engineering*, 2013, pp. 21–37.

[16] M. Ivanov, "Pure Python Paillier homomorphic cryptosystem implementation," [Online]. Available: https://github.com/mikeivanov/paillier.

[17] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, pp. 72, 96–101, Elsevier, 2012.

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

[19] D. H. Hoe, C. Martinez, and S. J. Vundavalli, "Design and characterization of parallel prefix adders using FPGAs," in *IEEE Southeastern Symposium on System Theory*, 2011, pp. 168–172.