TERMinator Suite: Benchmarking Privacy-Preserving Architectures

Dimitris Mouris, Nektarios Georgios Tsoutsos, and Michail Maniatakos

Abstract

Security and privacy are fundamental objectives characterizing contemporary cloud computing. Despite the wide adoption of encryption for protecting data in transit and at rest, *data in use* remains unencrypted inside cloud processors and memories, as computation is not applicable on encrypted values. This limitation introduces security risks, as unencrypted values can be leaked through side-channels or hardware Trojans. To address this problem, encrypted architectures have recently been proposed, which leverage homomorphic encryption to natively process encrypted data using datapaths of thousands of bits. In this case, additional security protections are traded for higher performance penalties, which drives the need for more efficient architectures. In this work, we develop benchmarks specifically tailored to encrypted computers, to enable comparisons across different architectures. Our benchmark suite, dubbed *TERMinator*, is unique as it avoids "termination problems" that prohibit making control-flow decisions and evaluating early termination conditions based on encrypted data, as these can leak information. Contrary to generic suites that ignore the fundamental challenges of encrypted computation, our algorithms are tailored to the security primitives of the target encrypted architecture, such as the existence of *branching oracles*. In our experiments, we compiled our benchmarks for the *Cryptoleq* architecture and evaluated their performance for a range of security parameters.

Index Terms

Benchmarks, data privacy, encrypted computation, termination problem, performance evaluation

I. Introduction

Cloud computing is consistently growing in popularity, since it offers strong computational power for both individuals and companies while providing key advantages for doing business. On the cloud, however, user data can be exposed to attacks from both the cloud provider and third parties. To mitigate these

D. Mouris is with University of Athens, Greece (e-mail: jimouris@di.uoa.gr). N. G. Tsoutsos and M. Maniatakos are with New York University, New York, NY (e-mail: {nektarios.tsoutsos, michail.maniatakos}@nyu.edu).

risks, cloud providers have adopted encryption for data in transit or at rest [4], [10], but all meaningful data manipulations remain unencrypted in cloud processors and volatile memories. Thus, the *data in use* can be exposed to security risks, including side-channel attacks [14], as well as hardware Trojans [2], [11], [12], which can leak sensitive information.

As these risks are inherent in unencrypted computer architectures, including commodity architectures such as `x86` and `ARM`, recent efforts focused on improving hardware security by enabling native processing of encrypted values in the processor pipeline. Indeed, encrypted computer architectures (e.g., [7], [13]) can leverage the properties of homomorphic encryption (e.g., [8]) and manipulate data directly in the encrypted domain. This approach draws a trade-off between security and efficiency, as larger security parameter sizes yield longer ciphertexts (e.g., 2048-bit values), but minimize the attack surface and security risks of outsourcing computations to a third party.

One limitation inherent to encrypted computation is the inability to make runtime decisions when the control values are encrypted. In fact, if the termination condition remains encrypted, a host executing an encrypted program may not be able to decide *if* or *when* the execution ends (i.e., there exists a "termination problem" [3]). In general, the host is unable to make branch decisions using encrypted data, which motivates the use for special constructions, dubbed *BRanching Oracles (BROs)*, which obliviously evaluate the branch outcomes [7].

At the same time, in order to improve the efficiency of existing encrypted architectures, as well as perform comparisons between contemporary and future processor instantiations, there is a need for performance benchmarks. Despite the continuous evolution of benchmarking algorithms, existing suites are optimized for unencrypted computation, without provisions for privacy-preservation during execution [5]. Likewise, existing benchmarks do not consider the existence of BRO constructions and the termination problems are not addressed in the algorithm design. This lack of specialized benchmarks makes it harder to perform meaningful comparisons across different architectures, which is a prerequisite for improving the efficiency of encrypted computation.

Our contribution: To address this problem, in our research we developed a novel benchmark suite for encrypted computer architectures, which avoids termination problems while maintaining data privacy. Our observation is that we can speculatively evaluate alternative execution paths for a given number of iterations, before judiciously combining the results. This is possible since the underlying encrypted architecture supports a privacy-preserving BRO. For our benchmarks, we use the Cryptoleq Enhanced Assembly Language (CEAL) [7] to develop privacy-preserving versions of fourteen algorithms from four benchmark classes (namely synthetic, microbenchmarks, kernels and encoder benchmarks), and in our experiments we evaluate their performance over different security configurations. Contrary to traditional

unencrypted benchmarks (such as `gcc`, `bzip2`, `mcf`, etc.), our algorithms are purposefully structured to prevent data leakage while performing homomorphic operations on ciphertexts.

The rest of the paper is organized as follows: In Section II we discuss background notions, while in Section III we elaborate on three examples of privacy-preserving algorithms included in our benchmarks suite. Our experimental evaluation is presented in Section IV and our conclusions summarized in Section V.

II. Preliminaries

Homomorphic Encryption: There exist encryption algorithms that support meaningful manipulation of encrypted data, such as addition (e.g., [8]) or multiplication (e.g., [9]). In this case, a homomorphic function can be applied directly on ciphertexts so that when the result is eventually decrypted, it will be the same as applying addition or multiplication on plaintexts. This enables users to encrypt their data and outsource all manipulations to a third party, without sacrificing privacy. Using homomorphic operations in a processor pipeline, however, requires special care to ensure that branching does not reveal any sensitive data by observing side-channel information (e.g., the branch target). This risk is mitigated using BROs that return an encryption of the correct branch target [7].

Termination problems: As observed in [3], an inherent limitation of homomorphic processing is that an encrypted computation host remains oblivious to any termination conditions of the executed program. Indeed, if a loop iterates on a condition over an encrypted variable (e.g., $while(x > 0)$), the host should not learn if $x > 0$, or side-channel information may be leaked. In effect, executing programs with encrypted variables cannot depend on early termination conditions or ciphertext-dependent decisions. Thus, before encrypting sensitive program variables, the algorithm should be transformed to its privacy-preserving counterpart, which trades efficiency for side-channel resistance by obliviously evaluating all possible iterations.

Threat Model: To formalize our assumptions about the applicable risks to user data, we introduce a concrete threat model. In our model, program evaluation is outsourced to a *rational, honest-but-curious* third party, which implements the encrypted computer architecture correctly, but has incentives to eavesdrop sensitive user data (e.g., to sell targeted advertisements). Likewise, we assume that adversaries may compromise a vulnerable third party and eavesdrop on sensitive data in volatile or non-volatile memory. In our developed benchmarks, we explicitly define which variables are sensitive and should remain encrypted to preserve data privacy.

BRO Instantiation: Regarding ciphertext-controlled branch decisions, we assume that third parties and adversaries can recover side-channel information by observing branch outcomes. To mitigate this risk,

we inherit the security assumptions of BROs, which is treated as decision-making black boxes, and all inputs and outputs are encrypted. Without loss of generality, our BRO instantiation employs *function G* introduced in [7, Section IV-B]), which outputs a re-encryption of its second argument if the plaintext value of its first argument is positive, otherwise outputs the encryption of integer zero; due to probabilistic encryption guarantees, these outputs are indistinguishable and protected against eavesdropping.

III. Algorithms in Benchmark Suite

In this section, we introduce our benchmark suite comprising fourteen privacy-preserving benchmarks without termination problems: Simon, Speck, Jenkins, Private Information Retrieval (PIR), Insertion Sort, N-Queens, Private Set Intersection (PSI), Tak function, Deduplication, Fibonacci, Factorial, Matrix Multiplication, Set Permutations, and Primes (Sieve of Eratosthenes).

Classes: Our benchmarks are categorized in four classes, depending on the type and features of the main loop iteration:

- *Synthetic:* This class comprises of primitive recursive benchmarks (such as Tak [6] and N-Queens), which allow assessing the universality of an abstract machine with respect to encrypted computation, as well as the performance of encrypted data structures (e.g., a stack).
- *Microbenchmarks:* This class evaluates the performance of homomorphic addition and multiplication, which are critical micro-operations of encrypted abstract machines. Examples in this class include Factorial (multiplication-intensive), Fibonacci (addition-intensive) and PIR (both addition- and multiplication-intensive).
- *Kernels:* This class focuses on evaluating essential core loops of different real-life applications, which combine memory swaps, branch decisions and arithmetic operations. Example benchmarks in this class include Insertion Sort, PSI, Deduplication (i.e., Set Union), Matrix Multiplication, Primes (i.e., Sieve of Eratosthenes) and Permutations.
- *Encoder Benchmarks:* This class comprises three real-life cryptographic and hash applications (namely Speck, Simon [1] and Jenkins), which are demanding in terms of bitwise operations and branch decisions on encrypted values, and allow assessing the BRO of the target abstract machine.

Rationale: Our benchmarks are specifically tailored for private computation in the encrypted domain, and an important motivation for our selections is the ability to eliminate branch decisions over encrypted control values (i.e., avoid termination problems and side-channel leakage). For that matter, in this work we ensure that the control flow of our algorithms does not rely on runtime data dependencies, such as early termination conditions within a loop (e.g., $break$ statements based on sensitive values). As a result, our benchmarks can preserve the privacy of encrypted values, leveraging the security primitives of the

underlying abstract machine, such as a BRO, which can be founded on hardware root of trust, as well as cryptographic primitives, depending on the security assumptions of the target architecture itself.

Since our benchmark selection is targeting encrypted computation processors, our objective is to assess the universality of the underlying execution engine (e.g., the ability to evaluate primitive recursive functions), the ability to support data structures in the encrypted domain (e.g., an encrypted stack), as well as the ability of oblivious branch decisions (e.g., encrypted multiplexing). These are fundamental properties of encrypted computation, and their runtime overhead can be assessed by our algorithms. In fact, our benchmarks are tied to real-life privacy applications: for example, encrypted matrix multiplication is beneficial in deep learning with private coefficients, PSI has applications in computing collision courses of aerial objects privately (e.g., military satellites), while Deduplication is a key algorithm for removing redundant files in cloud storage.

Moreover, our benchmarks are oblivious of the underlying encryption scheme (which depends solely on the underlying encrypted architecture that is being measured), and allow evaluating the microarchitectural features of a target encrypted processor implementation. Since encrypted computation relies on randomly permuted memory spaces, our algorithms enable assessing different cache architecture levels, sizes and replacement policies, as well as different branch prediction strategies. In addition, our algorithms allow evaluating the performance of different homomorphic ALU implementations (e.g., pipelined ALUs), as well as memory performance using intensive data transfers (e.g., permutations).

In the next paragraphs, we elaborate on the design and privacy-preservation challenges of three representative benchmarks: Tak, Speck and Insertion Sort.[1]

Notation: We employ the same mathematical notation as earlier work [7, Section IV-A], where $\widetilde{X}$ corresponds to the encryption of X, while $\hat{+}$, $\hat{-}$, $\star$ and **D2** represent homomorphic addition, subtraction, multiplication and integer division by 2 (i.e., right shift [7, Alg. 2]) respectively. Likewise, $\hat{=}$ represents *private equality* based on *function G*, and returns $\widetilde{1}$ if the operands map to equal plaintexts, or $\widetilde{0}$ otherwise [7, Eq. 23].

A. Synthetic: Private Tak Function Benchmark

The *Tak function* is a synthetic benchmark, often used to demonstrate recursion performance. The textbook version uses variables x, y, z as inputs, and unless $x \leq y$, each invocation spawns three recursive calls, where each variable is reduced by one [6].

[1]For conciseness, we elaborate on the implementation of a set of representative benchmarks; the source of all fourteen benchmarks, along with experimental results and documentation, is available at *https://github.com/momalab/TERMinatorSuite*.

Algorithm 1 Private Tak Function

Private Vars: $x, y, z, sel, x_{old}, y_{old}$

1: **procedure** TAK($x, y, z, iter$)
2: $sel \leftarrow \mathbf{G}(x \mathbin{\hat{-}} y, \widetilde{1})$
3: **while** $iter > 0$ **do**
4: $iter \leftarrow iter - 1$
5: $x_{old} \leftarrow (\widetilde{1} \mathbin{\hat{-}} sel) \star x_{old} \mathbin{\hat{+}} sel \star x$
6: $y_{old} \leftarrow (\widetilde{1} \mathbin{\hat{-}} sel) \star y_{old} \mathbin{\hat{+}} sel \star y$
7: $x \leftarrow (\widetilde{1} \mathbin{\hat{-}} sel) \star x \mathbin{\hat{+}} sel \star \text{TAK}(x \mathbin{\hat{-}} \widetilde{1}, y, z, iter)$
8: $y \leftarrow (\widetilde{1} \mathbin{\hat{-}} sel) \star y \mathbin{\hat{+}} sel \star \text{TAK}(y \mathbin{\hat{-}} \widetilde{1}, z, x_{old}, iter)$
9: $sel \leftarrow \mathbf{G}(x \mathbin{\hat{-}} y, \widetilde{1})$
10: $z \leftarrow (\widetilde{1} \mathbin{\hat{-}} sel) \star y \mathbin{\hat{+}} sel \star \text{TAK}(z \mathbin{\hat{-}} \widetilde{1}, x_{old}, y_{old}, iter)$
11: **return** z

Algorithm 2 Private Insertion Sort

Private Vars: $array, x, y, diff, max, min$

1: **procedure** INSERTIONSORT($array$[N])
2: **for** $i \in \{1, \ldots, \text{N} - 1\}$ **do**
3: $j \leftarrow i$
4: **while** $j \neq 0$ **do**
5: $x \leftarrow array[j-1], \quad y \leftarrow array[j]$
6: $max \leftarrow \mathbf{G}(x \mathbin{\hat{-}} y, x) \mathbin{\hat{+}} \mathbf{G}(y \mathbin{\hat{-}} x, y)$
7: $max \leftarrow max \mathbin{\hat{+}} (x \mathbin{\hat{=}} y) \star x$
8: $min \leftarrow max \mathbin{\hat{-}} \mathbf{G}(x \mathbin{\hat{-}} y, x \mathbin{\hat{-}} y) \mathbin{\hat{-}} \mathbf{G}(y \mathbin{\hat{-}} x, y \mathbin{\hat{-}} x)$
9: $array[j-1] \leftarrow min,\ array[j] \leftarrow max,\ j \leftarrow j - 1$
10: **return** $array$

Threat Model & Challenges: In our benchmark, variables x, y, z are encrypted to preserve their privacy. This requirement, however, prevents evaluating the termination condition, as $x \leq y$ comparisons are not meaningful over ciphertexts.

Privacy-Preserving Algorithm: To enable meaningful comparisons between ciphertexts x and y, we leverage *function G* to compute an encrypted bit sel (Algorithm 1, lines 2 & 9). The latter allows multiplexing two variables (e.g., x and y) by evaluating an expression homomorphic to $(1-sel)\cdot x + sel \cdot y$

Algorithm 3 Private Speck32 Encryption Algorithm

Private Vars: $x, y, xor, bit, \{x, y\}_{LSB}, L, R, subkey[\text{RNDS}]$

1: **procedure** XOR(x, y) ▷ The word size WS $= 16$ for Speck32
2: $xor \leftarrow \widetilde{0}$
3: **for** $i \in \{0, \ldots, \text{WS} - 1\}$ **do** ▷ For all bits $x_{LSB} \oplus y_{LSB}$
4: $x_{LSB} \leftarrow x \mathbin{\hat{-}} \mathbf{D2}(x) \mathbin{\hat{-}} \mathbf{D2}(x)$ ▷ Compute LSB(x)
5: $y_{LSB} \leftarrow y \mathbin{\hat{-}} \mathbf{D2}(y) \mathbin{\hat{-}} \mathbf{D2}(y)$ ▷ Compute LSB(y)
6: $bit \leftarrow (x_{LSB} \mathbin{\hat{+}} y_{LSB}) \star (1 \mathbin{\hat{-}} (x_{LSB} \star y_{LSB}))$ ▷ $\oplus$
7: **for** $j \in \{0, \ldots, i - 1\}$ **do** $bit \leftarrow bit \mathbin{\hat{+}} bit$
8: $xor \leftarrow xor \mathbin{\hat{+}} bit$ ▷ Add corresponding bit
9: $x \leftarrow \mathbf{D2}(x), \quad y \leftarrow \mathbf{D2}(y)$ ▷ Continue with next bits
10: **return** xor
11: **procedure** ROR(x, POSITIONS)
12: **for** $i \in \{0, \ldots, \text{POSITIONS} - 1\}$ **do** ▷ Rotate right N times
13: $lsb \leftarrow x \mathbin{\hat{-}} \mathbf{D2}(x) \mathbin{\hat{-}} \mathbf{D2}(x), \quad y \leftarrow lsb$
14: **for** $i \in \{0, \ldots, \text{WS} - 2\}$ **do** $y \leftarrow y \mathbin{\hat{+}} y$
15: $x \leftarrow y \mathbin{\hat{+}} \mathbf{D2}(x \mathbin{\hat{-}} lsb)$
16: **return** x
17: **procedure** SPECK32ENCRYPT($L, R, subkey[\text{RNDS}]$)
18: **for** $i \in \{0, \ldots, \text{RNDS} - 1\}$ **do**
19: $R \leftarrow \text{XOR}(\ \text{ROR}(R, 7) \mathbin{\hat{+}} L,\ subkey[i]\ )$
20: $L \leftarrow \text{XOR}(\ \text{ROR}(L, 14),\ R\ )$
21: **return** L, R ▷ Ciphertext output

(lines 5-8 & 10). The maximum recursion depth is controlled by variable $iter$, which depends only on the maximum potential range of inputs (i.e., the execution steps are oblivious to the actual private inputs). Still, any additional recursions are innocuous, as homomorphic multiplexors maintain the correct result.

B. Kernels: Private Insertion Sort Benchmark

Insertion Sort enables in-place sorting of an input array, by left-shifting each array element to its correct (sorted) position. The textbook algorithm iterates over all array elements and compares the j-th element with its previous one; if the higher-index element is larger, the algorithm swaps the two elements.

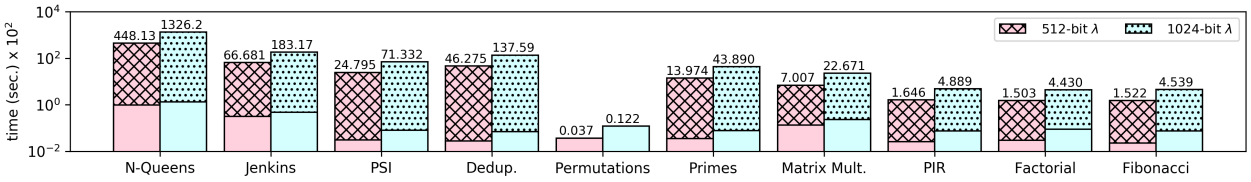

Fig. 1. Measured execution time of additional benchmarks, using a 512- and 1024-bit security parameter size and input parameters from Table I.

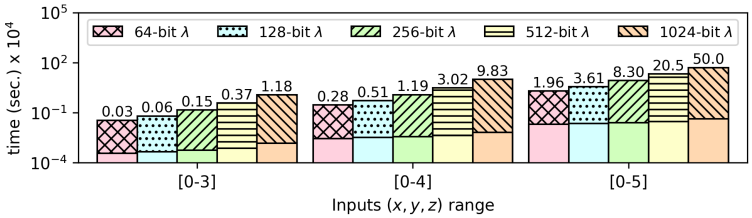

Fig. 2. Measured execution time for the Tak function benchmark, for different security parameter sizes and different input ranges ($\beta = 8$).

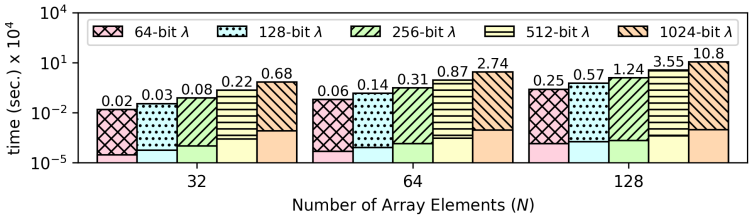

Fig. 3. Measured execution time for the Insertion Sort benchmark, for different security parameter and input array sizes ($\beta = 16$).

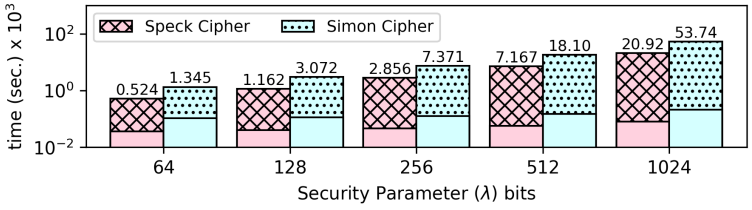

Fig. 4. Measured execution time for the Speck & Simon cipher benchmarks using different security parameter sizes, based on the 32-bit input vector and $\beta = 16$.

Threat Model & Challenges: In our benchmark, all elements of the input array are encrypted to protect their privacy. As a result, the algorithm cannot compare two array elements and decide if these elements should be swapped.

Privacy-Preserving Algorithm: In Algorithm 2, we are able to compare two encrypted values x and y

(i.e., find the min and max) using *function G* and the homomorphic difference $x \hat{-} y$. The algorithm exhausts all possible iterations, so the number of executed steps does not depend on the input array elements.

C. Encoders: Speck32 Cipher Benchmark

Speck is a lightweight block cipher that is based on the Add-Rotate-XOR (ARX) paradigm [1]. Essentially, each round of Speck encryption uses bitwise rotations and XOR operations to compress an input block and round subkey into an output block, and multiple rounds are cascaded resembling a standard "Feistel" structure. By design, Speck's performance is optimized for software implementations.

Threat Model & Challenges: In our benchmark, all inputs and outputs are encrypted to preserve their privacy (e.g., against side-channel leakage). An important challenge is the ability perform bitwise operations (such as XOR and rotations) that do not have homomorphic counterparts, as well as oblivious runtime decisions based on ciphertext bits.

Privacy-Preserving Algorithm: To enable bitwise operations in the encrypted domain, we leverage the homomorphic equivalent of integer division by 2 (**D2**) and privately iterate over all bits in a block using two new helper functions, namely XOR and ROR (Alg. 3). According to [1], a 32-bit input block requires $RNDS = 22$ rounds and a 64-bit key that is expanded to 22 round subkeys using Speck's *key schedule*.

IV. EXPERIMENTAL EVALUATION

For our evaluation, we instantiated all fourteen benchmarks in our suite using the CEAL programming language [7], which offers native support for function G.[2] In our experiments, we varied the input length (N), the bit-size of the public encryption key (λ), as well as the number of *precision bits* for integer datatypes (β).[3] CEAL public keys are defined as the product of two primes, so larger λ values provide better resistance to cryptanalysis; however, as λ increases, homomorphic operations and *function G* invocations become slower. Likewise, since the complexity of homomorphic multiplication ($\star$) is $\mathcal{O}(\beta^2)$ [7, Section IV-C], in our experiments we opted for the minimum β fitting the input of each benchmark.

Experimental Setup: We measured the runtime performance of our benchmarks in both the encrypted and unencrypted domain, using the Cryptoleq virtual machine implemented within the CEAL compiler,

[2]All benchmarks are also available in `C/C++` to enable flexible integration with different encrypted architectures and BRO instantiations.

[3]In CEAL programs, the range of positive integers is limited by 2^β [7, Fig. 3], and each positive integer has a negative counterpart.

configured with the GNU GMP library for arbitrary-precision arithmetic. All experiments were performed on Ubuntu 16.04, running on a 3.40 GHz Intel i7-6700 system with 16 GBs of memory.

Runtime Performance Results: In Fig. 2 we present our performance evaluation for the synthetic class example (Section III-A), while our Kernel and Encoder benchmark examples (Sections III-B and III-C) are presented in Figs. 3 and 4 respectively. An overview of the runtime performance of our additional benchmarks is also reported in Fig. 1. As we observe from our experiments, the *Tak* runtime performance is exponentially dependent on the input value range (i.e., the value of $iter$), while our insertion sort experiments demonstrate a superlinear dependence on the input array size. Likewise, the runtime overhead of Speck increases superlinearly with the size of λ, but is always faster compared to the Simon cipher from the same family [1]. For comparison, our graphs also include the unencrypted runtimes (solid-color bars) overlaid on the corresponding encrypted runtimes (i.e., patterned bars).

Benchmark Characterization: We further configured the Cryptoleq virtual machine to collect execution statistics for all fourteen benchmarks. As reported in Table I, our suite covers a diverse set of algorithms, ranging from $5 \cdot 10^3$ to $3 \cdot 10^6$ *function G* invocations, as well as 0 to 10^3 *private equality* invocations. Such diversity renders TERMinator a suitable candidate for evaluating and comparing encrypted architecture implementations.

TABLE I

Function G & *Private Equality* INVOCATIONS

Type	**Benchmark**	β	**Inputs**	**Invocations**	
				G Fun	***Eq.*** ($\hat{=}$)
Snth	**N-Queens**	16	N = 4	2988500	0
Snth	**Tak**	8	$x, y, z \in [0-3]$	260544	0
Encd	**Speck Cipher**	16	SPECK32 Test V.	466752	0
Encd	**Simon Cipher**	16	SIMON32 Test V.	1204736	0
Encd	**Jenkins**	32	"ab, cd"	400448	0
Krnl	**Insertion Sort**	16	$N = 32$	146320	496
Krnl	**PSI**	16	$[16] \cap [16]$	61416	1024
Krnl	**Deduplication**	16	$N = 32$	288260	497
Krnl	**Permutations**	16	4 elements	0	0
Krnl	**Eratosthenes**	16	256 primes	73984	0
Krnl	**Matrix Mult.**	16	$[8\times8] \times [8\times8]$	93312	0
Micr	**PIR**	16	$db\_size = 32$	4656	16
Micr	**Factorial**	16	fact(8), $iter = 16$	9280	16
Micr	**Fibonacci**	16	fib(24), $iter = 32$	9312	32

V. Concluding Remarks

Encrypted computation comes at a high cost, and development of efficient computer architectures that enable native processing of encrypted values can significantly improve runtime performance. In this work, we developed an open-source benchmark suite over a diverse set of privacy-preserving algorithms, which avoids termination problems using a branching oracle primitive, and enables comparisons across different encrypted architectures. As our case study, we report baseline runtime measurements and execution statistics using the Cryptoleq virtual machine and GMP arbitrary-precision library.

Acknowledgment

This work was partially sponsored by the NYU Abu Dhabi Global Ph.D. Student Fellowship program. D. Mouris thanks Orestis Polychroniou for the fruitful discussions.

References

[1] R. Beaulieu, D. Shors, J. Smith, S. Treatman-Clark, B. Weeks, and L. Wingers, "The SIMON and SPECK lightweight block ciphers," in *Design Automation Conference (DAC)*. ACM, 2015, pp. 1–6.

[2] G. T. Becker *et al.*, "Stealthy Dopant-Level Hardware Trojans," in *Cryptographic Hardware and Embedded Systems*. Springer, 2013, pp. 197–214.

[3] M. Brenner *et al.*, "Secret Program Execution in the Cloud Applying Homomorphic Encryption," in *Digital Ecosystems and Technologies Conference*. IEEE, 2011, pp. 114–119.

[4] R. Chow *et al.*, "Controlling Data in the Cloud: Outsourcing Computation without Outsourcing Control," in *Workshop on Cloud Computing Security*. ACM, 2009, pp. 85–90.

[5] J. L. Henning, "SPEC CPU2006 Benchmark Descriptions," *ACM SIGARCH Computer Architecture News*, vol. 34, no. 4, pp. 1–17, 2006.

[6] D. E. Knuth, "Textbook examples of recursion," *Artificial Intelligence and Mathematical Theory of Computation*, pp. 207–230, 1991.

[7] O. Mazonka, N. G. Tsoutsos, and M. Maniatakos, "Cryptoleq: A Heterogeneous Abstract Machine for Encrypted and Unencrypted Computation," *IEEE Transactions on Information Forensics and Security*, vol. 11, no. 9, pp. 2123–2138, 2016.

[8] P. Paillier, "Public-Key Cryptosystems Based on Composite Degree Residuosity Classes," in *Advances in Cryptology*. Springer, 1999, pp. 223–238.

[9] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Communications of the ACM*, vol. 21, no. 2, pp. 120–126, 1978.

[10] 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.

[11] N. G. Tsoutsos, C. Konstantinou, and M. Maniatakos, "Advanced Techniques for Designing Stealthy Hardware Trojans," in *Design Automation Conference*. ACM/EDAC/IEEE, 2014, pp. 1–4.

[12] N. G. Tsoutsos and M. Maniatakos, "Fabrication Attacks: Zero-Overhead Malicious Modifications Enabling Modern Microprocessor Privilege Escalation," *IEEE Transactions on Emerging Topics in Computing*, vol. 2, no. 1, pp. 81–93, 2014.

[13] ——, “The HEROIC Framework: Encrypted Computation Without Shared Keys,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 34, no. 6, pp. 875–888, 2015.

[14] Y. Zhang *et al.*, “Cross-VM Side Channels and Their Use to Extract Private Keys,” in *Computer and Communications Security*. ACM, 2012, pp. 305–316.