This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2017.2722440, IEEE Transactions on Computers

Efficient Detection for Malicious and Random Errors in Additive Encrypted Computation

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

Abstract—Although data confidentiality is the primary security objective in additive encrypted computation applications, such as the aggregation of encrypted votes in electronic elections, ensuring the trustworthiness of data is equally important. And yet, integrity protections are generally orthogonal to additive homomorphic encryption, which enables efficient encrypted computation, due to the inherent malleability of homomorphic ciphertexts. Since additive homomorphic schemes are founded on modular arithmetic, our framework extends residue numbering to support fast modular reductions and homomorphic syndromes for detecting random errors inside homomorphic ALUs and data memories. In addition, our methodology detects *malicious modifications* of memory data, using keyed syndromes and block cipher-based integrity trees, which allow preserving the homomorphism of ALU operations, while enforcing non-malleability of memory data. Compared to traditional memory integrity protections, our tree-based syndrome generation and updating is parallelizable for increased efficiency, while requiring a small Trusted Computing Base for secret key storage and block cipher operations. Our evaluation shows more than 99.999% detection rate for random ALUs errors, as well as 100% detection rate of single bit-flips and clustered multiple bit upsets, for a runtime overhead between 1.2% and 5.5%, and a small area penalty.

Index Terms—Encrypted Computation, Homomorphic Encryption, Memory Integrity, Mersenne Primes, Residue Numbering.

1 Introduction

Since its discovery almost four decades ago, homomorphic encryption (HE) has enabled cryptographers to manipulate encrypted data without decrypting them [1]. One important application of this remarkable property, is the ability to delegate the processing of data, without sacrificing their confidentiality [2]. Even though several encryption algorithms exhibiting homomorphic properties have been proposed in the past (e.g., [3], [4]), it wasn't until 2009 that the academic interest in the area was reignited, following the discovery of fully homomorphic encryption [5]. Indeed, these HE advancements motivated several research directions, and recent work in the area includes message authenticators [6], electronic voting [7], quadratic function evaluation [8], multiparty computation [9], outsourced cloud computing [10], encrypted computation [11], reusable garbled circuits [12], encrypted data sorting [13], as well as verifiable computation [14] [15].

Even though HE has many potential applications, as it is evident from the previous examples, not all algorithms are practical given the computational power of contemporary computers. Indeed, all known fully homomorphic constructions are not yet adequately efficient [16] [17] [18], and only partially homomorphic schemes can be used in practical applications (e.g., [17], [19]). In the latter case, the main homomorphic operation is *modular multiplication*, for which several efficient implementations have been proposed (e.g., [20], [21]). Moreover, considering the significance that cloud computing and private outsourcing has today, our primary focus in this work will be encrypted computation applications (e.g., [22], [11]) that are based on additive HE and the Paillier scheme [23]. Nevertheless, the primitives introduced in this paper are applicable to many different computation models that leverage modular multiplications (e.g., [13]).

In typical encrypted computation scenarios, the end users first encrypt their data with HE, and then a (potentially untrusted) third party process them using an appropriate branching program [24]. As soon as the program is evaluated, the encrypted outputs are returned to the end user for decryption. The homomorphic properties of the underlying encryption scheme guarantee that decrypting these received outputs would yield the correct result, as if encryption was never a part of the computation (i.e., as if the branching program was applied directly to unencrypted data). That way, the confidentiality of data is preserved while data processing is outsourced to another party. Still, HE does not provide any explicit assurance on *data integrity*, and as a result, any malicious or random modification to the data entrusted to the third party may remain undetected by the end user. Specifically, HE schemes are naturally *malleable* in order to support meaningful manipulation of ciphertexts [25], and as soon as malleability is inhibited, an encryption scheme is no longer homomorphic (e.g., [26]). Thus, additional protection mechanisms are required to detect if the integrity of homomorphic data has been compromised.

The aforementioned integrity concerns become more evident, if we take into account that several encrypted computation components, such as modular multiplication ALUs and memory cells, are mapped directly to hardware circuits. In this case, these components are susceptible to a wide range of transient or permanent (random) faults that can undermine the trustworthiness of encrypted computation results. As demonstrated in [11], encrypted computation pipelines share many similarities with standard processor

- *N. G. Tsoutsos is with the Department of Computer Science and Engineering, New York University, New York. E-mail: nektarios.tsoutsos@nyu.edu*
- *M. Maniatakos is with the Department of Electrical and Computer Engineering, New York University Abu Dhabi, UAE. E-mail: michail.maniatakos@nyu.edu*

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2017.2722440, IEEE Transactions on Computers

architectures; yet, there exist key differences that render efficient error detection a non-trivial problem. For example, due to the special use of modular arithmetic in HE operations, existing techniques such as Residue Numbering Systems (RNS) [27] cannot be applied directly, as they eventually *require revealing the secret factorization of cryptographic moduli*. Moreover, due to the large bit lengths of encrypted arguments (typically, in the order of thousands), integrity techniques like parity checks or ECC *would incur prohibitive overheads in terms of bit redundancy*. Likewise, the applicability of hash functions is also limited, as digesting very long arguments incurs proportionally large performance delays; at the same time, since hashes are non-homomorphic by nature and cannot propagate through HE ALUs, they should be recomputed after each intermediate ALU operation.

Besides the aforementioned random-fault scenarios, we can also assume a broader threat model where adversaries are allowed to actively manipulate encrypted memory cells. In the latter case, all previous mitigations are rendered ineffective, since adversaries may inject specially crafted faults that are designed to circumvent random-fault detection techniques. For example, consider an adversary that manages to replay an outdated value; in this case, random-fault detection techniques would accept the replayed value as valid again, which, however, can have devastating consequences in the computation context. Hence, our goal in this work is to protect encrypted computation both against random faults, as well as active adversaries. Towards that end, our observation is that, since many HE algorithms are built on top of modular arithmetic, we can exploit their intrinsic mathematical properties and create efficient error detection primitives. These primitives are designed to be compatible with common homomorphic operations in encrypted computation ALUs, and can prevent active adversaries from forging encrypted data under several scenarios, while minimizing the size of required metadata. Overall, we claim the following contributions:

a) Development of an efficient error detection method using keyed syndromes (dubbed *Exponential Syndromes*), which are homomorphic and can protect critical components of encrypted computation, namely the encrypted data memory and the homomorphic ALU, against malicious and random faults respectively.
b) Construction of a non-malleable integrity tree structure (dubbed *Simon Tree*), which leverages the lightweight `Simon` block cipher [28], to enable concurrent detection of malicious modifications in encrypted data memory, such as replay and reordering attacks.
c) Design of a *fast modular reduction algorithm* that leverages the properties of Mersenne number moduli [29], and enables significant performance enhancements for the proposed error detection framework.

The rest of the paper is organized as follows: in Sections 2 and 3 we elaborate on the assumed threat model and discuss our extension of residue numbering to modular arithmetic. Section 4 presents our fast modular reduction algorithm, while Section 5 elaborates on our error detection method for random faults. Additionally, in Section 6 we expand our detection method to malicious errors, in line with our threat model, and in Section 7 we present our experimental setup, along with a discussion of our results using Monte Carlo simulation and HDL implementation. Finally, we discuss related work in Section 8, and our concluding remarks are presented in Section 9.

2 Threat Model

In this Section, we elaborate on the threat scenarios that we consider possible, as well as the different attacks that we want to protect against. Our target application is encrypted computation that is based on additive HE (e.g., [22]), and without loss of generality we assume that Paillier encryption [23] is used as the underlying scheme[1]. In Paillier, the modular multiplication of ciphertexts is homomorphic to the modular addition of plaintexts, so hardware implementations of encrypted computation require modular multiplication ALUs (e.g., [31] implements a Montgomery ALU). Given that the typical size of each Paillier ciphertext is 2048 bits, these modular multiplication ALUs normally require a large number of clock cycles to generate a result, while occupying a substantial area on the silicon as well. Hence, the probability of a transient or permanent fault affecting the ALU outputs is non-negligible, and we consider this as an integrity threat for encrypted computation results. We further assume that such random faults can occur during any step involved in the ALU result calculation.

In addition to the ALU, we also consider soft errors in memory cells storing encrypted data. In particular, our threat model considers Single Event Upsets (SEUs) and Multiple Bit Upsets (MBUs) as in [32], where up to 4 consecutive bits are set or cleared together (i.e., clustered faults). Such faults can affect the integrity of the encrypted arguments while they remain in storage, which can yield corrupted results as soon as the faulty values are loaded and processed. In our model, we assume that the probability of such soft errors is non-negligible, given that memory cells occupy large areas on silicon. Conversely, we consider that the probability of a soft error in a single register inside a processing core is negligible.

Given the security implications of encrypted computation, it is also expected that active adversaries may be motivated to tamper with the encrypted memory cells. In effect, we assume that adversaries may have direct access to the memory modules storing encrypted arguments (or can leverage memory disclosures [33]), and have the capacity to inject faults in a precise and judicious fashion to multiple bits at the same time (e.g., [34]). Hence, we assume that adversaries are able to bypass common linear or cyclic error-detection codes, since these are designed to protect against accidental data modifications (e.g., [35], [36]).

In general, our security objective is to authenticate the contents of all memory cells with respect to any potential corruption. Thus, our threat model also considers adversaries capable of modifying any data word of arbitrary size. For example, adversaries can swap the contents of memory locations (e.g., swap two 2048-bit arguments), copy one memory value over another, reorder memory contents,

1. It should be noted that exponential ElGamal [30] is also additive homomorphic, but decryptions require solving a discrete logarithm problem. Moreover, the generalized Damgård-Jurik-Nielsen cryptosystem [7] is also applicable.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2017.2722440, IEEE Transactions on Computers

homomorphically generate new values using modular multiplication, as well as replay previously stored values to the same or different memory location. In fact, powerful adversaries are allowed to keep a "transcript" of how each memory location is updated over time, and then replace the most recent memory value with any outdated one. Likewise, another extreme example would be to replace the entire data memory contents with an earlier snapshot, or swap the memory modules with different ones altogether.

Besides the encrypted memory, however, our threat model does not allow active tampering with the processor pipeline (e.g., the program counter, the registers, the ALU, etc.), as meaningful attacks are largely impractical. Indeed, encrypted processors typically reside within dedicated, tamper-proof ICs, to minimize the risk of probing or modifying the processor's internal state without detection. Moreover, encrypted processors have randomized execution states (i.e., program counters are updated in "spaghetti" fashion due to encrypted addressing), while the use of encrypted instruction arguments prevents meaningful analysis of encrypted binaries [11]. Consistent with the *semi-honest* adversarial model of existing encrypted computation applications, our threat model also excludes intentional interference and denial-of-service attacks on the processor.

Evidently, in order to ensure the integrity of additive encrypted computation under the aforementioned threat assumptions, we need to combine cryptographic protections with random fault detection methodologies. An important observation, however, is that cryptographic integrity protections differ from random fault detection methods, in that the former requires explicit knowledge of secret information (i.e., secret keys) to generate and verify integrity metadata (dubbed *message authentication tags* [25]). Use of secret keys is mandatory, to ensure that *only a holder of the keys* can authenticate of integrity tags, and prevent adversaries from forging metadata for (chosen) non-authentic values or manipulating existing ones without detection. Thus, our threat model requires a *Trusted Computing Base (TCB)* within the encrypted computation processor, which cannot be accessed or tampered with by any adversary, and is used generate and verify integrity tags, as well as store secret keys for integrity. Essentially, we require that hardware implementations of encrypted computation cores incorporate an isolated secure region that is responsible for keyed integrity operations, and this will be our assumed root of trust[2].

In the next Section, we elaborate on how residue numbering can be extended to support modular multiplication.

3 Residue Numbering for Modular Multiplication

Residue numbering is a popular and powerful method for fault detection in basic ALU operations [39]. It provides an alternative representation of arithmetic values using weightless residues over a predefined "moduli set" [27], rather than traditional *digits* of positional weight. This allows parallelizable and carry-free addition and multiplication, providing concurrent error detection for the ALU through adding or multiplying the residues of ALU inputs and verifying equality with the residue of ALU outputs.

One of our goals in this work is to detect faults in ALUs that perform modular multiplications, using the same simple and efficient residue-based checks as in standard ALUs. In residue numbering, however, modular multiplication, division, and comparison are not straightforward, and often require algorithms of excessive complexity [40]. Additionally, porting the traditional residue checks directly to modular multiplication, would yield incorrect results.[3]

Recall that Paillier's public parameter N is the product of prime factors (see also Appendix A). A sufficient extension for residue check correctness would be to use the prime factors of the multiplication modulus N itself, also as the "moduli set" of residue numbering. At the same time, when modulus N is the product of only two large primes u and v, revealing this factorization would completely break Paillier's security. Thus, to achieve both security and fault detection using a simple residue check, our observation is that we can add a third factor z to modulus N. As in the multi-prime RSA scheme case [41], the third factor z in Paillier must be a prime to ensure correctness, and at least u and v should remain secret at all times.

The following paragraphs provide theoretical support for extending residue numbering to modular multiplication.
Representation: Using residue numbering, any non-negative integer can be represented uniquely by a set of (typically smaller) residues, each corresponding to a specific modulus. In more details, if a set of k positive moduli $\{m_1, m_2, \cdots, m_k\}$ is chosen so that:

$$gcd(m_i, m_j) = 1, \quad \forall i, j \in \mathbb{N}^+_{\leq k}, i \neq j \tag{1}$$

(i.e. all moduli are pairwise co-primes), a non-negative integer X less than $M = \prod m_i$ can be represented as a set of k integers $\{x_1, x_2, \cdots, x_k\}$ so that $x_i = X \bmod m_i$.
Uniqueness: The residue numbering representation is unique due to the following theorem (based on the Chinese Remainder Theorem).

Theorem 1. Given a set of relatively prime positive moduli $\{m_1, m_2, \cdots, m_k\}$, then for any integer $X \in [0, M)$, where $M = \prod m_i$ for $1 \leq i \leq k$, the set of residues $X \bmod m_i$ is unique.

Proof: If the theorem did not hold, there would exist distinct integers A and $B \in [0, M)$ with identical residue representations. Thus, for all i we would have $A = c_i \cdot m_i + a_i$, $B = d_i \cdot m_i + b_i$, $c_i \neq d_i$ and $a_i = A \bmod m_i = B \bmod m_i = b_i$, so $A - B$ would be a multiple of m_i:

$$A - B = (c_i - d_i) \cdot m_i, \quad \forall i \text{ and } c_i \neq d_i. \tag{2}$$

This also means that $A - B$ is a multiple of $lcm(m_1, m_2, \cdots, m_k)$, which equals M, since m_i are pairwise co-primes. But if $A - B$ is a multiple of M, then A and B cannot be both in the interval $[0, M)$ (*contradiction*). □
Simple Arithmetic: In residue numbering, we can directly apply carry-less addition, subtraction or multiplication on

2. Note that there exist commercial processor designs also featuring secure enclaves (e.g., [37], [38]).

3. For example, the modular multiplication of 20 times 30 modulo 9, using RNS modulus 7, is $((20 * 30) \bmod 9) \bmod 7 = (600 \bmod 9) \bmod 7 = 6 \bmod 7 = 6$, which does not match $(((20 \bmod 7) * (30 \bmod 7)) \bmod 9) \bmod 7 = ((6 * 2) \bmod 9) \bmod 7 = (12 \bmod 9) \bmod 7 = 3$.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2017.2722440, IEEE Transactions on Computers

the residues of two operands, and get the residue representation corresponding to the result of that same operation. More formally, if $\{a_1, a_2, \cdots, a_k\}$ and $\{b_1, b_2, \cdots, b_k\}$ are residue representations of integers A and B respectively, then the residue representation $\{x_1, x_2, \cdots, x_k\}$ of $X = (A \diamond B) \bmod M$ would be $x_i = (a_i \diamond b_i) \bmod m_i$, where $M = \prod m_i$, and $\diamond$ is addition, subtraction or multiplication.

Nested Reductions: Based on the above, we now prove another theorem that is applicable when we have *nested reductions* (for example, when we need the residue representation of encrypted values already reduced modulo N).

Theorem 2. If z, N are positive integers with z dividing N, then for any non-negative integer X we have:

$$(X \bmod N) \bmod z = X \bmod z. \tag{3}$$

Proof: Let $a = X \bmod N$ and $b = a \bmod z = (X \bmod N) \bmod z$. Then we have $X = k \cdot N + a$ for some integer k and $a = j \cdot z + b$ for some integer j. Combining these equalities, we have $X = k \cdot N + j \cdot z + b$. Since z divides N, there exists integer m so that $N = m \cdot z$; thus:

$$X = k \cdot m \cdot z + j \cdot z + b = z \cdot (k \cdot m + j) + b. \tag{4}$$

If we divide this quantity by z, we get b as the residue:

$$X \bmod z = (z \cdot (k \cdot m + j) + b) \bmod z = b. \tag{5}$$

Since $X \bmod z = b$ and $(X \bmod N) \bmod z = b$ by definition of b, we have $(X \bmod N) \bmod z = X \bmod z$, for $X \in \mathbb{Z}_{\geq 0}$, and $z, N \in \mathbb{Z}_+$, when z divides N. □

Modular Multiplication: Theorem 2 is essential in this work, as we can use it to demonstrate the following Corollary for ALUs performing modular multiplication on *values already reduced to a modulus*. It should be noted that when the divisibility conditions of the Corollary do not hold, multiplication of the operand residues yields an incorrect residue representation of the result.

Corollary 3. If z, N are positive integers and z divides N, then for any non-negative integers X and Y we have:

$$(X \cdot Y \bmod N) \bmod z = ((X \bmod z) \cdot (Y \bmod z)) \bmod z.$$

Proof: Using the result of Theorem 2, we have that:

$$(X \cdot Y \bmod N) \bmod z = X \cdot Y \bmod z. \tag{6}$$

Let $a = (X \bmod z)$, $b = (Y \bmod z)$ and $c = (X \cdot Y \bmod z)$. Then, there exist integers x and y so that $X = (x \cdot z + a)$ and $Y = (y \cdot z + b)$. Using the equalities for X and Y, we have:

$$X \cdot Y = (x \cdot z + a) \cdot (y \cdot z + b) = x \cdot y \cdot z^2 + x \cdot z \cdot b + y \cdot z \cdot a + a \cdot b, \tag{7}$$

$$X \cdot Y = z \cdot (x \cdot y \cdot z + x \cdot b + y \cdot a) + a \cdot b. \tag{8}$$

If we define integer $w = x \cdot y \cdot z + x \cdot b + y \cdot a$, the last equation becomes $X \cdot Y = w \cdot z + a \cdot b$, so:

$$X \cdot Y \bmod z = (w \cdot z + a \cdot b) \bmod z = a \cdot b \bmod z, \tag{9}$$

using the fact that z divides $z \cdot w$ (i.e. $z \cdot w + a \cdot b$ and $a \cdot b$ are *congruent* modulo z). Combining Eq. 6 and Eq. 9, we get the equality stated in the Corollary. □

Algorithm 1 Fast Reduction Modulo a Mersenne Prime M_p

Input: X, p, where p is prime so that $2^p - 1$ is also prime
Output: $Residue$

```
1:  procedure FastMod(X, p)
2:      M_p ← 2^p − 1
3:      while X > 2 · M_p − 1 do
4:          Sum ← 0
5:          while X ≠ 0 do
6:              Sum ← Sum + X&M_p        ▷ Add masked digit
7:              X ← X ≫ p                ▷ Shift right by p bits
8:          X ← Sum                      ▷ Recur for digits of Sum
9:      if X ≥ M_p then
10:         X ← X − M_p
11:     return Residue = X
```

4 Fast Reduction Modulo Mersenne Primes

As it is evident from the discussion in the previous section, a prerequisite for using residue numbering is the ability to perform modular reductions of integers. Typically, modular reductions using an arbitrary modulus may incur significant overheads and undermine the overall system performance (especially when 2048-bit long Paillier ciphertexts are provided), because integer division of the dividend with a given divisor is typically required. Even though there exist fast modulo implementations (e.g. [42]), their overhead is still linear to the size of the dividend.

Our observation with regards to residue numbering for modular multiplication is that the RNS modulus z *does not have to be random*, other than the security requirement to be a prime factor of the multiplication modulus N (as elaborated in Section 3). Thus, judiciously selecting z to differ by 1 from the next power of 2, would allow very efficient reductions modulo z. Integers with this property are called *Mersenne primes* [29], and are formally defined as $M_p = 2^p - 1$, where both M_p and p are primes.

As part of our contribution, we provide an algorithm for fast modular reduction, when the reduction modulus is a Mersenne prime $2^p - 1$. In this case, we can represent the input using radix-2^p digits and calculate the modular reduction by summing those digits together, before reducing the sum using modulus $2^p - 1$. This reduction (presented in Alg. 1) is very efficient, since the execution overhead is linear to the number of radix-2^p digits in the input, rather than the input bit length. The mathematical principle of the algorithm is presented in the following theorem.

Theorem 4. Let p be a prime and $M_p = 2^p - 1$ a Mersenne prime. Then, for any positive radix-2^p integer X in the form:

$$X = \sum_{i=0}^{\lfloor \log_{2^p} X \rfloor} a_i \cdot (2^p)^i, \tag{10}$$

where 2^p is the *numbering system base* of the representation and a_i are the digits of the number, the reduction of X modulo M_p is:

$$X \bmod M_p = \left(\sum_{i=0}^{\lfloor \log_{2^p} X \rfloor} a_i \right) \bmod M_p. \tag{11}$$

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2017.2722440, IEEE Transactions on Computers

Proof: Let $a'_i = a_i \bmod M_p$ and $b'_i = (2^p)^i \bmod M_p$ be the reductions modulo M_p of each digit a_i and each base power $(2^p)^i$ respectively. Then, by definition of *congruence* [25], a'_i and a_i are congruent modulo M_p for each i, and the same holds for each b'_i and $(2^p)^i$ as well. Due to congruence properties, we can apply the standard rules of arithmetic with respect to addition and multiplication over congruent numbers. Thus, we can apply the reductions modulo M_p *distributively* to each digit and base power, for each $i \geq 0$:

$$(a_i \cdot (2^p)^i) \bmod M_p = (a'_i \cdot b'_i) \bmod M_p. \tag{12}$$

The same holds for each product in the sum of Eq. 10:

$$\begin{aligned} X \bmod M_p = \left(\sum_{i=0}^{\lfloor \log_{2^p} X \rfloor} a_i \cdot (2^p)^i \right) \bmod M_p = \\ \left(\sum_{i=0}^{\lfloor \log_{2^p} X \rfloor} \left(a_i \cdot (2^p)^i\right) \bmod M_p \right) \bmod M_p. \end{aligned} \tag{13}$$

Likewise, we can distribute the reductions modulo M_p to each (2^p) factor comprising the base powers $(2^p)^i$, for each $i \geq 0$ so that the *empty product* for $i = 0$ is equal to 1:

$$(2^p)^i \bmod M_p = \left(\prod_1^i (2^p \bmod M_p) \right) \bmod M_p. \tag{14}$$

Combining Eq. 12, 13 and 14, as well as the definitions of a'_i and b'_i, we get the following expression for $X \bmod M_p$:

$$\left(\sum_{i=0}^{\lfloor \log_{2^p} X \rfloor} \left(a'_i \cdot \prod_1^i (2^p \bmod M_p) \right) \right) \bmod M_p. \tag{15}$$

In the last expression, we can simplify the product term considering that $2^p \bmod M_p = (M_p + 1) \bmod M_p = 1$, by definition of M_p. Finally, using the definition of a'_i, we have $(\sum a'_i) \bmod M_p = (\sum (a_i \bmod M_p)) \bmod M_p$, which equals $(\sum a_i) \bmod M_p$ for each i, after applying standard arithmetic rules. Thus, Eq. 15 can be simplified to Eq. 11. □

5 Error Detection for Random Faults

The first aspect in protecting the integrity of encrypted computation, is ensuring the detection of errors caused by random faults. Common causes for such faults include high-energy neutrons from cosmic radiation, low-energy cosmic neutron interactions with IC insulator layers, as well as emission of alpha particles from IC packaging impurities [43]. In the next paragraphs, we present our methodology for efficiently detecting these random errors in ALUs used for additive encrypted computation, as well as in encrypted data memories. A high-level block diagram of our random error detection method is also illustrated in Fig. 1.

5.1 Error Detection for Modular Multiplication ALU

In order to detect random errors at the output X of a modular multiplication ALU operation, we will leverage the residue numbering extensions presented in Section 3 and build a homomorphic *syndrome*. We define our syndrome to be the RNS residue $sndr = X \bmod M_p$, where M_p is a predefined Mersenne prime modulus, and this reduction can be computed efficiently using Alg. 1. As also mentioned

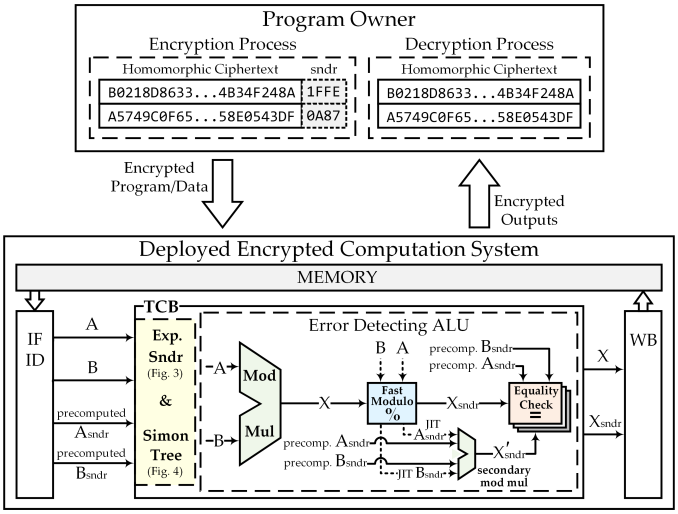

Fig. 1. Abstract view of an encrypted computation pipeline that features a modular multiplication ALU with error detection and redundant equality checks (along with additional TCB modules presented in Section 6). In typical encrypted computation scenarios (e.g., private outsourcing to a potentially untrusted party), users encrypt their programs and data and upload them to the remote processor for homomorphic evaluation (e.g., [11]); as soon as the encrypted outputs are computed, the user downloads the results for decryption.

in Section 3, one important requirement for both security and correctness is that the Paillier modulus N (where N^2 is used to reduce the multiplication of two ciphertexts [23]), is the product of $k > 2$ prime numbers, where one of them is Mersenne prime M_p.[4] Without loss of generality, we assume that N is the product of $k = 3$ primes, namely u, v and M_p, and since the third factor of N is a Mersenne prime, we are able to combine Alg. 1 and Eq. 3 to efficiently compute the syndrome of the ALU output X. Recall that if the third prime factor was not present, we would have to reveal either u or v as the RNS modulus, which would break encryption security; to prevent factorization of N, at least two primes of adequate length should always remain secret.

Based on the definition of our homomorphic syndrome, if we are given an ALU result X, as well as the corresponding syndrome $X \bmod M_p$, we are able to verify if the result correctly matches its syndrome, and assert that no random error has happened (with high probability). This, however, would have not been feasible, if we couldn't compute the syndrome of the result given the syndromes of the ALU inputs. Hence, if A and B are the two ALU inputs and X is the ALU output, we can use Corollary 3 to verify the following condition for fault-free results:

$$X \bmod M_p = (A_{sndr} \cdot B_{sndr}) \bmod M_p, \tag{16}$$

where $A_{sndr} = A \bmod M_p$ and $B_{sndr} = B \bmod M_p$. In this case, if we know A, B, A_{sndr} and B_{sndr} before each ALU operation, we can use Eq. 16 to verify the correctness of X; this error detection procedure is also summarized in Alg. 2. Typically, the initial values for all syndromes can be pre-computed offline, stored in the encrypted memory along with each corresponding ciphertext, and updated after each

4. In this case, the encryption scheme is naturally extended using the product of all primes as the public parameter N, and *Carmichael's λ function* of N as the private parameter.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2017.2722440, IEEE Transactions on Computers

Algorithm 2 Modular Multiplication with Error Detection

Input: A, B, N, M_p so that M_p divides N
Output: (Multiplication Result, Fault Status)
1: **procedure** MODMUL-ED-ALU(A, B, N, M_p)
2: $X \leftarrow A \cdot B \bmod N^2$
3: $X_{sndr} \leftarrow X \bmod M_p$
4: $A_{sndr} \leftarrow A \bmod M_p$ ▷ For JIT syndromes
5: $B_{sndr} \leftarrow B \bmod M_p$ ▷ For JIT syndromes
6: $tmp \leftarrow A_{sndr} \cdot B_{sndr} \bmod M_p$ ▷ Parallelizable step
7: **if** $tmp = X_{sndr}$ **then** ▷ Redundant equality check
8: **return** $(X, Correct)$
9: **else**
10: **return** $(\perp, Faulty)$

homomorphic operation. As it will become evident in the next subsection, storing syndromes in memory would also enable us perform efficient memory error detection. Otherwise, in case we require error detection only for the ALU, syndromes can be computed Just-In-Time (JIT), immediately before the arguments enter the ALU for processing (i.e., no extra memory storage is necessary).

Notably, Eq. 16 requires that a second ALU operation is necessary to compute the homomorphic operation on the syndromes themselves. For that matter, a separate ALU module is necessary, since the primary ALU hardware is used to multiply inputs A and B (which have different sizes from A_{sndr} and B_{sndr}), and reduce the result using the Paillier modulus (i.e., not M_p). Having different modules also allows both ALU operations (Steps 2 and 6 in Alg. 2) to be performed in parallel, and the cost of syndrome multiplication can be masked by the homomorphic operation on A and B. For syndrome verification, however, the computation of X_{sndr} in Step 3 of Alg. 2 depends on the calculation of X in Step 2. Alternatively, if the primary ALU is part of a pipeline, it is also possible to start performing the reduction of Step 3 in parallel with *the next* ALU operation, and "flush" the pipeline in case a fault is ultimately detected.

In general, if the equality in Step 7 of Alg. 2 does not hold, then an error may have occurred either during syndrome multiplication (i.e., in the primary or the secondary ALU or both), or during equality checking, or during syndrome computation (i.e., in the Mersenne reduction unit). Assuming that the error was caused by transient faults, the exception can be handled by repeating the homomorphic operation using freshly-computed syndromes; likewise, to prevent *undetected errors* during equality checking, redundant checks should be employed. Moreover, even though this methodology is capable of detecting an arbitrary number of random errors in the primary ALU output, it is still possible that the syndrome X_{sndr} of a faulty output X' is equal to the expected syndrome $A_{sndr} \cdot B_{sndr} \bmod M_p$ (e.g., when $X' = X + k \cdot M_p$ for some integer $k \neq 0$). In this case, the incorrect ALU output *collides* with the correct one, and the fault *escapes*.

In order for our error detection method to be efficient, the bit length of M_p needs to be much smaller compared to the bit length of N, typically by one or two orders of magnitude. Since the selection of prime modulus M_p determines the bit length of each homomorphic syndrome, and the secondary ALU performs a modular multiplication of two syndromes (Eq. 16), that bit length determines the size and computation overhead for the second ALU. As it will become evident in our experimental evaluation (Section 7), it is sufficient to select Mersenne primes M_{19} or M_{31}, to achieve fault coverage above 99.999%. In general, the probability of an escaped fault due to collisions is ${M_p}^{-1}$. If M_p is too small, this probability increases, while increasing the size of M_p would require extra memory storage when syndromes are pre-computed and stored along with the operands, as previously mentioned. It's worth noticing that an ALU performing modular multiplication of two 19-bit or 31-bit syndromes, would require 22 or 34 clock cycles respectively, while the multiplication of two 2048-bit ciphertexts requires 2051 cycles [20], [44]; in general, this overhead is a linear function of the bit length of the arguments.

5.2 Error Detection for Encrypted Memories

In encrypted computation applications, instruction arguments and encrypted data are typically stored inside memory modules during processing. As these modules are continuously vulnerable to soft errors, it is important to be able to detect when such errors occur, so that further corrective actions can be taken (e.g., reloading the affected values from permanent storage or redundant memories). One popular solution to the problem is to use *Error Correcting Code (ECC)* memories (e.g., [45]), that can tolerate a limited number of faults in each data word. One drawback of such solutions, however, is the fact that they incur non-negligible area and delay overheads, as dedicated ECC circuits are required. Moreover, the number of required ECC parity bits increases logarithmically with the argument bit size, so, for encrypted arguments sizes of up to 2048 bits each, these approaches do not scale adequately.

As discussed in the previous section, one option to obtain the syndrome of an ALU argument is to pre-compute and store it in memory along with the corresponding ciphertext, forming a (ciphertext, syndrome) pair. In our framework, this also enables the detection of random memory errors *without any additional cost*: At runtime, as soon as each ciphertext is loaded from memory, we also load its pre-computed syndrome; then, we can verify the ciphertext's integrity by computing its residue modulo M_p (using Alg. 1), and comparing the result with the existing syndrome. If a random error has occurred either in the syndrome or in the ciphertext, it can be correctly detected with probability $1 - {M_p}^{-1}$, by checking that the residue and the syndrome do not match. This boundary considers reduction errors as well as escapes due to collisions, and assumes redundancy in all comparisons, to avoid equality errors. Conversely, if no error has occurred in memory, any mismatch would be a false positive, which is attributed to residue computation errors, and is treated like a regular error, as it is indistinguishable from a true positive. In addition, since modular multiplications in the context of encrypted computation are at least 18 times slower compared to the fast Mersenne reductions (as presented in Section 7.2), these memory integrity checks can be performed by time-multiplexing existing reduction modules, and mask their overhead by running the reductions in parallel to the primary ALU operation (i.e. achieve *delay-free* error detection).

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2017.2722440, IEEE Transactions on Computers

5.3 Complete Coverage for Single Memory Bit-Flips

One important observation is that, depending on the nature and number of memory errors, our residue based detection method can offer a fault coverage higher than the aforementioned probabilistic estimates of $1 - {M_p}^{-1}$. Specifically, if we consider single bit-flips, the coverage would always be 100%, as it is algebraically impossible to have two congruent values (modulo M_p) with Hamming distance equal to 1. Assuming that X is an original memory value and $Y = X \oplus 2^j$ is a faulty value after a single bit-flip on X at bit position j, the corresponding single bit error should be masked if and only if:

$$|X - Y| = k \cdot M_p, \quad \text{for some } k \in \mathbb{Z}_{>0}, \tag{17}$$

which is a congruence relation between X and Y. In our case, however, we claim that Eq. 17 cannot be true, since it holds for $x_i \in [0, 1]$ and $j \geq 0$:

$$X = \sum_{i=0}^{\lfloor \log_2 X \rfloor} x_i \cdot 2^i \equiv x_j \cdot 2^j + \sum_{i=0, i \neq j}^{\lfloor \log_2 X \rfloor} x_i \cdot 2^i, \tag{18}$$

$$Y = X \oplus 2^j = (1 - x_j) \cdot 2^j + \sum_{i=0, i \neq j}^{\lfloor \log_2 X \rfloor} x_i \cdot 2^i, \tag{19}$$

$$|X - Y| = \left| x_j \cdot 2^j - (1 - x_j) \cdot 2^j \right| = 2^j. \tag{20}$$

Therefore, in order to have a masked single bit flip, there should exist integers $k > 0$ and $j \geq 0$ so that $2^j = k \cdot M_p$. In the trivial case where $j = 0$, the equation does not hold for any $k > 0$, as 1 cannot have M_p as a factor. Similarly, if $j > 0$, by the fundamental theorem of arithmetic [46], 2^j should have a *unique prime factorization*; since the latter is a power of 2, it cannot have prime M_p as a factor. As a result, the fault coverage for single bit flips is exactly 100%.

Clustered faults: The same coverage can also be shown algebraically for clustered faults of up to 4 bits (as in [32]), where the unique factors of the absolute difference in Eq. 20 are primes 2, 3, 5 and/or 7, and thus any M_p with $p > 3$ (i.e., any M_p other than $M_2 = 3$ and $M_3 = 7$) is not a factor either.

6 Error Detection Against Active Attacks

As elaborated in our threat model, besides random errors in the ALU, our objective is to also detect malicious modifications in encrypted memory contents. Following the discussion of Section 2, it is necessary to use secret keys to authorize generation and verification of message authentication tags for each memory value. These operations should be performed inside a tamper-proof TCB, inaccessible by adversaries, where the secret keys are stored as well. Moreover, our objective is to maintain compatibility with the aforementioned ALU error detection method (Section 5.1) and avoid using an entirely different scheme on top of our RNS-based approach, as this allows reuse of hardware modules and conserves memory. Thus, the proposed tags should be homomorphic and be able to propagate correctly through modular multiplication operations. Overall, we introduce:

1) **Exponential Syndromes:** a keyed construction that extends the original (RNS-based) syndromes and allows detecting malicious and random errors in the encrypted memory and the ALU respectively; the construction is selectively homomorphic through the use of pseudorandom "blinding" (discussed in Section 6.1.2), and enforces knowledge of secret keys for syndrome generation and verification.
2) **Simon Trees:** a non-malleable tree construction, which leverages the `Simon` lightweight block cipher [28], and can reduce the integrity of all exponential syndromes to the integrity of a single value (i.e., the tree root), effectively creating an "integrity tree"; this root is stored within the TCB, where it is protected against modification and eavesdropping.

6.1 Exponential Syndromes Against Memory Attacks

Our starting point for the design of exponential syndromes $(esndr)$, is the original $sndr$ construction described in Section 5.1. There, we defined $X_{sndr} = X \bmod M_p$, where X is a Paillier ciphertext and M_p is a Mersenne prime, and highlighted the homomorphic property of that construction when it propagates through a modular multiplication ALU. In order to maintain this property, while adding the requirement of a secret key K_{esndr} in the generation and verification of the syndrome, we define $X_{esndr} = (X_{sndr})^{K_{esndr}} \bmod M_p$. Due to the multiplicative property of the powers of products, we are then able to multiply exponential syndromes homomorphically. Specifically, if A and B are ciphertexts, and $X = A \cdot B \bmod N$ is the modular multiplication ALU output for these inputs, we have:

$$A_{esndr} \cdot B_{esndr} = (X \bmod M_p)^{K_{esndr}} \bmod M_p, \tag{21}$$

where we applied the result of Corollary 3 as in Eq. 16, and raised each side of the equation to power K_{esndr} before reducing to modulus M_p.

By construction, exponential syndromes remain backwards compatible with the ALU random error detection procedure described in Alg. 2, while due to the modular exponentiation step, it is intractable to generate or verify them without access to the secret exponent K_{esndr}. More importantly, exponential syndromes can detect malicious errors in encrypted memories, which is a limitation of the original syndromes. Specifically, if exponential syndromes are stored in memory alongside ciphertexts (similar to our detection method in Section 5.2), only the TCB can verify the integrity of a ciphertext X, by raising the M_p residue of X to secret exponent K_{esndr}, before reducing it to M_p and comparing the result with the stored syndrome.

Without invoking the TCB and the secret exponent (as was the case with the original syndromes), any adversary could maliciously modify a ciphertext X into X' and simply replace the stored syndrome with $X' \bmod M_p$. The latter is prevented in this construction, as adversaries do not know which exponent to use while forging the syndrome for their chosen X'. In addition, to prevent guessing the secret exponent, K_{esndr} should be randomly selected from an adequately large key space, and different exponents must be shared between the TCB and the user across different sessions.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2017.2722440, IEEE Transactions on Computers

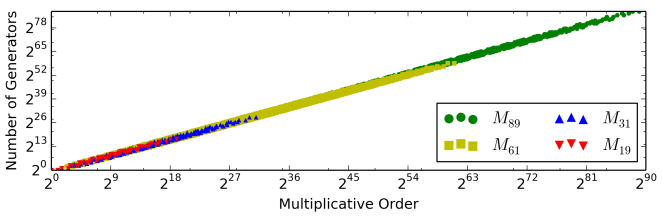

Fig. 2. Distribution of the generators for each cyclic subgroup of $\mathbb{Z}^*_{M_p}$, for $p \in [19, 31, 61, 89]$. For each divisor d of the order of $\mathbb{Z}^*_{M_p}$, there exists a unique subgroup of d elements, and $\phi(d)$ of them are its generators. Since $\phi(M_p)$ is *smooth* and has many divisors, solving the GDLP can be easy: even though there are many discrete logarithm bases X_{sndr} (i.e., generators) for which the search space for exponents K_{esndr} (i.e., their order) is large, we also observe many generators with small orders. For the GDLP to be hard, *all generators should have maximum order*.

6.1.1 Attack Vectors against Exponential Syndromes

Even though adversaries can no longer selectively forge a syndrome for a maliciously modified ciphertext X', our baseline construction requires additional protections to thwart other known attack vectors, as elaborated in the following paragraphs.

Exponent Recovery: In an effort to recover the secret exponent K_{esndr}, adversaries may attempt to analyze the relationship between a ciphertext and its exponential syndrome. In effect, an adversary may exploit the knowledge of X, M_p and $(X \bmod M_p)^{K_{esndr}} \bmod M_p$, to predict K_{esndr}. Nevertheless, this attack requires solving a generalized discrete logarithm problem (GDLP) [47], which recovers the exponent to which X_{sndr} should be raised to equal X_{esndr}. In general, depending on the factorization of the order of a cyclic group, solving the GDLP can be computationally intractable[5]; however, if the group order is a *smooth integer* rather than a prime, solving the GDLP is not intractable (e.g., using the Pohlig-Hellman algorithm) [47].

An important observation is that, for relatively small Mersenne primes M_p, the cyclic group $\mathbb{Z}^*_{M_p}$ has order $\phi(M_p) = M_p - 1$, which could be a smooth integer, depending on the selection of M_p. Specifically, there are cases where the group order $M_p - 1$ can be factorized to the e_i powers of several small primes q_i (i.e., $M_p - 1 = \prod_i q_i^{e_i}$), where the square root $\sqrt{q_j}$ of the largest prime factor q_j sets an upper bound to the discrete logarithm computation cost using the Pohlig-Hellman algorithm [47].[6] As illustrated in Fig. 2, there exist Mersenne primes M_p so that for several discrete logarithm bases X_{sndr}, the search space for potential secret exponents K_{esndr} is reduced.

Evidently, if relatively small Mersenne prime moduli are chosen (such as M_{31} or M_{61}, which helps reducing the memory overhead of storing exponential syndromes), adversaries may be able to solve the GDLP, recover K_{esndr} and forge syndromes for maliciously crafted ciphertexts. To launch this attack, however, adversaries need to know the base X_{sndr} of the discrete logarithm, as well as the exponential syndrome itself (i.e., X_{esndr}); if either one of these is not available to the adversary, it would no longer be possible to solve the GDLP. Our observation is that exponential syndromes can be *blinded* with a pseudorandom value (making them indistinguishable from random values), before being stored alongside ciphertexts. In this case, even if adversaries attempt to analyze them, the blinding step would render them unusable. Moreover, it is possible to construct an efficient and reversible blinding transformation upon each X_{esndr}, so that only the TCB can recover the actual syndrome from the blinded value X_{bsndr}.

Syndrome Homomorphism: Another potential attack vector that justifies blinding is the malleability of exponential syndromes (Eq. 21). Even though homomorphism is necessary to enable correct propagation of syndromes while ciphertexts are processed in the ALU, it can also be exploited by active attackers. Specifically, if exponential syndromes remain unblinded while at rest in encrypted memories, it is possible to generate valid tags for new ciphertexts (related to those already stored in memory).[7] Hence, we require that homomorphism is respectively "disabled" or "enabled", when the syndromes are stored in memory (outside the TCB) or processed by the secondary ALU (inside the TCB). This selective operation can be implemented by applying an efficient exclusive-or function between the syndrome and a pseudorandom value C; in this case, we define a blinded syndrome as $X_{bsndr} = X_{esndr} \oplus C$, which can be reversed.

Reduction Commutativity: Our RNS-based methodology leverages Mersenne prime moduli to efficiently digest long ciphertexts into compact syndromes (Alg. 1). In effect, reducing an input ciphertext X to modulus $M_p = 2^p - 1$ requires adding together all radix-2^p digits of X; however, since addition is a commutative operation, the relative order of these digits is not preserved and any permutation would eventually yield the same residue X_{esndr}. Thus, even without knowledge of K_{esndr}, an active adversary may find a ciphertext X' congruent to X (i.e., find a "second preimage" [47]) and existentially forge the pair (X', X_{esndr}). Moreover, we cannot simply replace our Mersenne reduction with a non-commutative, collision resistant function (e.g., SHA-256), as the homomorphism of Eq. 21 will no longer hold, invalidating our ALU error detection method.

In contrast, if we blind X_{esndr} with a pseudorandom value C that is also a *collision resistant digest* of the input ciphertext X, then, even if X' is congruent to X, the corresponding blinded syndromes $X_{bsndr} = X_{esndr} \oplus C$ and $X'_{bsndr} = X_{esndr} \oplus C'$ will be different (except with negligible probability, attributed to the *birthday problem* [25]). Notably, collision resistance is a vital requirement for C digests to prevent forgeries: if adversaries can find X and X' so that $X_{esndr} = X'_{esndr}$ and $C = C'$, the corresponding blinded syndromes would also collide. Furthermore, we require that C offers *computation resistance* [47], so it should be infeasible to compute C from X without access to a secret key. The latter is necessary to prevent adversaries from extracting X_{esndr} from the blinded value using the expression $X_{bsndr} \oplus C$, as well as finding a valid C' to forge

5. Since M_p is a prime, the group $\mathbb{Z}^*_{M_p}$ and the subgroups generated by different $\langle X_{sndr} \rangle$ are *cyclic*. For each divisor d of the group's order $\phi(M_p)$, there is exactly one subgroup of order d that has exactly $\phi(d)$ different generators, where ϕ is *Euler's Totient function* [47]. If all generators have maximum order, the GDLP is hard.

6. E.g., $\phi(M_{31}) = 2 \cdot 3^2 \cdot 7 \cdot 11 \cdot 31 \cdot 151 \cdot 331$, while $\phi(M_{61}) = 2 \cdot 3^2 \cdot 5^2 \cdot 7 \cdot 11 \cdot 13 \cdot 31 \cdot 41 \cdot 61 \cdot 151 \cdot 331 \cdot 1321$. The factorization of the order d of each cyclic subgroup with $\phi(d)$ elements, is a subset of these prime powers.

7. For example, after seeing an existing pair (X, X_{esndr}), adversaries can forge a valid pair (X^d, X^d_{esndr}) for some integer $d \in \mathbb{Z}_N$.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2017.2722440, IEEE Transactions on Computers

Algorithm 3 Syndrome Protection with Blinding

Input: $X = (x_\ell, \ldots, x_2, x_1), K_{msk}, K_{esndr}, K_{Simon}, M_p, ctr$
Output: Blinded Syndrome X_{bsndr}
1: **procedure** BLINDING($X, K_{msk}, K_{esndr}, K_{Simon}, M_p$)
2: $X_{sndr} \leftarrow X \bmod M_p$
3: $X_{msk} \leftarrow \mathcal{U}_{K_{msk}}(x_\ell, \ldots, x_2, x_1)$ ▷ Parallelizable step
4: $X_{esndr} \leftarrow (X_{sndr})^{K_{esndr}} \bmod M_p$
5: $C \leftarrow \mathcal{E}(K_{Simon}, ctr) \oplus X_{msk}$ ▷ Parallelizable step
6: **return** $X_{bsndr} = X_{esndr} \oplus (C \ \& \ M_p)$ ▷ Keep p LSBs

Algorithm 4 Parallel Modular Exponentiation Ladder

Input: X_{sndr}, K_{esndr}, M_p
Output: Exponential Syndrome X_{esndr}
1: **procedure** MODEXP(X_{sndr}, K_{esndr}, M_p)
2: $W_1 \leftarrow X_{sndr}, \quad W_0 \leftarrow 1$ ▷ Initialize temp variables
3: **for each** b **in** K_{esndr} **do** ▷ Left to right bit parsing
4: $W_{(1-b)} \leftarrow W_1 \cdot W_0 \bmod M_p$
5: $W_b \leftarrow W_b \cdot W_b \bmod M_p$ ▷ Parallelizable step
6: **return** $X_{esndr} = W_0$

Algorithm 5 Universal Digest using a Polynomial $\bmod M_p$

Input: K_{msk}, X, p so that p and $2^p - 1$ are primes
Output: Universal Digest U
1: **procedure** POLYDIGEST(K_{msk}, X, p)
2: $M_p \leftarrow 2^p - 1, \quad U \leftarrow X \ \& \ (M_p \gg 1)$
3: **while** $X \neq 0$ **do** ▷ Loop ℓ times
4: $X \leftarrow X \gg (p-1)$ ▷ Shift right by $(p-1)$ bits
5: $blk \leftarrow X \ \& \ (M_p \gg 1)$ ▷ Mask $p-1$ LSBs
6: $U \leftarrow blk + U \cdot K_{msk} \bmod M_p$ ▷ Horner's rule
7: **return** U

X'_{bsndr} when X' is congruent to X. Besides, since computing X_{esndr} requires knowledge of K_{esndr}, adversaries cannot extract C from X_{bsndr} either.

6.1.2 Protecting Exponential Syndromes with Blinding

Based on the previous analysis, to protect our exponential syndromes against potential attack vectors (such as one-wayness, malleability and second-preimage threats), we blind each X_{esndr} with a pseudorandom value C that is bound to the corresponding ciphertext X via a collision resistant digest. Specifically, given a computation resistant function $\mathcal{U}_k(X)$ that requires k to generate a collision resistant digest of X, we can compute C as the encryption of the output of $\mathcal{U}_k$. Encrypting each digest using a cipher that is *secure against chosen plaintext attacks (CPA-secure)* [25] is beneficial, since it transforms the digest to a pseudorandom value (a requirement for C), and prevents adversaries from detecting if a collision has occurred in the digests (due to CPA-security). Preventing collision detection is essential in case finding a collision allows bypassing the computation resistance of $\mathcal{U}_k$ (e.g., recovering the secret key).

In this work, we encrypt each $\mathcal{U}_k$ output using the `Simon` block cipher in *counter mode*, which is a CPA-secure mode of operation [25]. Moreover, if X is represented as coefficients vector $(x_\ell, \ldots, x_2, x_1) \in \mathbb{Z}^\ell_{M_p}$ and $k \in \mathbb{Z}_{M_p}$, we can define a family of functions $\mathcal{U}_k(X)$ as a degree-ℓ polynomial modulo M_p evaluated at point k:

$$\mathcal{U}_k(X) = x_1 k^\ell + x_2 k^{\ell-1} + \cdots + x_\ell k \bmod M_p. \qquad (22)$$

This family extends the original Carter and Wegman construction to ℓ-block inputs [48], and offers both collision and computation resistance. Specifically, $\mathcal{U}_k$ is a ε-*almost*-Δ-*universal* (ε-AΔU) family, with collision probability less than $\varepsilon = \ell/M_p$ for any pair of distinct inputs [49]. In addition, since a secret key k is used to select a random member of the family, and $\mathcal{U}_k$ outputs are encrypted with a CPA-secure cipher (to prevent finding the polynomial roots), the construction also offers computation resistance. Notably, the encrypted $\mathcal{U}_k$ construction is computationally unforgeable, since it follows the common *digest-then-mask* paradigm introduced in [50].

As presented in Alg. 3, we compute the blinded syndrome of X as the exclusive-or between X_{esndr} and the (ctr-mode) `Simon` encryption of digest $X_{msk} = \mathcal{U}_{K_{msk}}(x_\ell, \ldots, x_2, x_1)$, where X is split in ℓ blocks and K_{msk} is a secret key. Each such block is $p-1$ bits long (recall that $x_i \in \mathbb{Z}_{M_p}$), while, to ensure CPA security, each encryption of X_{msk} should use a unique ctr value.[8] The computed blinded syndrome and the associated ctr value can be safely stored in memory, as adversaries cannot use X_{bsndr} to forge new syndromes of chosen messages, or recover X_{esndr}. The TCB is responsible for protecting the corresponding secret keys, as well as executing Alg. 3 to generate or verify syndromes. The TCB may also compute C to selectively unblind X_{bsndr} and recover $X_{esndr} = X_{bsndr} \oplus C$, using the corresponding ctr value and Steps 3 & 5 of Alg. 3.

6.1.3 Implementation Remarks

The hardware implementation of blinded exponential syndromes requires careful design, in order to optimize their computation as much as possible. In addition to the requirement for fast Mersenne reduction modules (which was the only requirement for our original syndromes X_{sndr}), in this case we also require modular exponentiation and blinding modules. Since the computation of blinded syndromes is a sensitive operation, and secret keys are involved, the aforementioned modules should be located exclusively within the boundaries of our TCB. Even though adversaries cannot tamper with the TCB in our threat model, our goal is to *minimize the risk of side channels*, especially in the case of modular exponentiation, where simple square and multiply algorithms can reveal the bits of the secret exponent.

With respect to exponentiation, we adapt the Montgomery *powering ladder* algorithm, which is designed to eliminate conditional branches based on the bits of the exponent, and allows natural parallelization of the two modular multiplications in each step [51]. In more details, our adapted algorithm iterates over all the bits of the exponent K_{esndr}, and in each iteration employs two regular multipliers in parallel, before reducing the computed products to Mersenne modulus M_p. Trading hardware redundancy for performance, this parallelism translates to a time overhead

8. Since our construction also uses X_{esndr} (which is pseudorandom), our blinded syndromes remain robust even in light of ctr reuse, as long as X_{esndr} does not simultaneously collide as well.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2017.2722440, IEEE Transactions on Computers

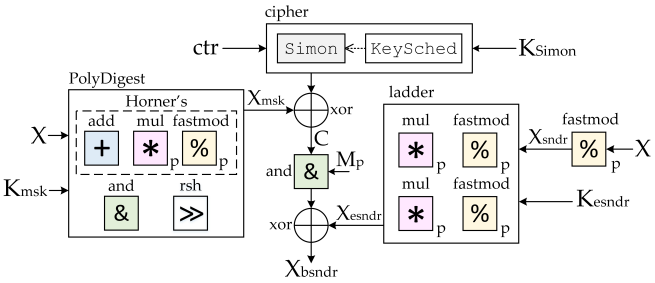

Fig. 3. Block diagram of our blinding process for exponential syndromes.

of one p-by-p bit multiplication, as well as up to three p-by-p bit additions (needed to reduce a $2p$-bit product using Alg. 1), for each bit of the exponent K_{esndr}; Alg. 4 presents our adapted exponentiation procedure. Likewise, as described in Alg. 5, our ε-AΔU polynomial digest can be implemented using only one p-by-p bit multiplication, and two to five p-by-p bit additions (subject to M_p reductions) for each $(p-1)$-bit block, by leveraging *Horner's rule* for computing polynomials. This observation enables significantly fast digesting, considering that argument sizes are in the order of thousands of bits.

Another observation is that we can reduce the memory storage requirements for ctr values, given that ctr-mode encryption requires storing each used counter value along with the corresponding CPA-secure ciphertext [25]. Specifically, instead of having a global counter state (where the most recent ctr value is maintained inside the TCB), we can *decentralize* the counter state to many independent counter values, each paired with the physical address $addr$ of a blinded syndrome. The latter allows to store only the local ctr_{addr} values, which remain unique for each $addr$ and are shorter than global ctr values. Indeed, if we set $ctr = (addr||00\ldots0||ctr_{addr})$ in Step 5 of Alg. 3, it suffices to store only the ctr_{addr} portion in encrypted memory (along with X and X_{bsndr}), as the corresponding $addr$ prefix is available to the TCB.[9] Each ctr_{addr} represents the most recent counter value for the corresponding X_{bsndr}, and is incremented when the syndrome is updated; any unauthorized modification or replaying is immediately detected during X_{bsndr} verification.

Additionally, by construction of the blinding procedure in Alg. 3, the inputs to the `Simon` encryption in Step 5, are independent of the outputs in the previous Step 4. Hence, the `Simon` encryption module can run in parallel to the exponentiation ladder, and their respective results are eventually combined using an XOR operation.[10] Parallelism can also be leveraged in Steps 2 and 3 of Alg. 3, which are also independent, assuming that redundant fast Mersenne reduction modules are available. A high level block diagram of our blinding process is presented in Fig. 3.

Key Management & Derivation: As presented in Alg. 3 and elaborated in Section 6.1.2, our blinding construction comprises an exponentiation ladder (Alg. 4), a universal digest (Alg. 5) and the `Simon` encryption algorithm $\mathcal{E}$. Following the basic principle of security and cryptography that emphasizes the need for different keys across instances of different primitives [25], our construction assumes three independent secret keys (summarized in Table 1): K_{esndr}, K_{msk} and K_{Simon} . Different instantiations of the system across different users, as well as different encrypted computation sessions of the same user, should use different keys. These unique session keys are only shared between the program owner (i.e., the user) and the respective TCB of the system hosting the encrypted computation session, and are established before the encrypted program is uploaded (for example, using a Diffie-Hellman key agreement). Alternatively, users may opt to exchange only a master key that is then securely expanded to the three aforementioned keys through a key derivation function [47].

TABLE 1
Summary of independent secret keys for different TCB primitives

Key	Size (bits)	Finite Group	Associated Primitive
K_{esndr}	61, 107	$\mathbb{Z}_{M_{61}}, \mathbb{Z}_{M_{107}}$	Modular exponentiation
K_{msk}	61, 107	$\mathbb{Z}_{M_{61}}, \mathbb{Z}_{M_{107}}$	Universal digest
K_{Simon}	96, 128	$\mathbb{F}_{2^{96}}, \mathbb{F}_{2^{128}}$	`Simon` encryption

6.2 Simon Trees Against Advanced Replay Attacks

As elaborated in the previous paragraphs, our blinded syndrome construction provides integrity protection against forgery, since adversaries are unable to generate syndromes for chosen ciphertexts, without knowledge of the secret keys. Moreover, if ctr values include immutable physical addresses, adversaries will not be able falsify a $(X, X_{bsndr}, ctr_{addr})$ tuple by copying another valid 3-tuple from a neighboring location, as the applied $addr$ value is not under their control. Nonetheless, binding counter values to physical addresses only prevents duplication, but does not prevent adversaries from *replaying an outdated instance* of a 3-tuple from that same address. Hence, we need to ensure that the stored syndromes are the most recent ones, which consequently guarantees (after verifying the syndromes) that the corresponding ciphertexts and counters are also the most recent. In effect, we must verify both the integrity and freshness of all syndromes as a group.

One simple approach to verify the integrity of all syndromes could be to treat their concatenation as a very long ciphertext, generate the blinded syndrome for it, and store that syndrome in the TCB. That way, verification of all syndromes is reduced to verifying the integrity of a single syndrome. Nevertheless, this approach incurs prohibitive overheads, as the associated cost is linear to the size of the entire memory; in fact, we would have to read all syndromes just to verify one of them. To overcome this limitation, we propose to use a hierarchical structure in the form of a tree, which makes the verification cost logarithmic to the size of the memory.

6.2.1 Description of our Replay Resistant Construction

As illustrated in Alg. 6, our "`Simon` Tree" construction employs the encryption algorithm $\mathcal{E}$ of the homonymous block

9. This compound ctr is globally unique, as each syndrome is stored at an immutable $addr$, for which ctr_{addr} is unique. The maximum ctr_{addr}, which controls how many times each X_{bsndr} can be updated at runtime, depends on the bitsize of $addr$ and the `Simon` block size.

10. The block size of the selected `Simon` variant should be larger than the bitsize of modulus M_p; since this cipher family supports block sizes up to 128 bits, the largest compatible modulus is M_{127}.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2017.2722440, IEEE Transactions on Computers

cipher family, as well as fast Mersenne reductions (Alg. 1), to securely digest multiple blinded syndromes into a parent syndrome at each tree level (e.g., compress 8 syndromes into 1). For higher tree levels, the same syndrome digestion algorithm is applied recursively over the parent values of the previous level, until one final digest is generated (i.e., the root of the tree). This root is then stored within the TCB, and this is sufficient to assert the integrity of the leaves in each path traversal of the tree. An important benefit of our construction is that it can leverage redundant encryption cores, as it is inherently parallelizable, while at the same time can enforce the relative order of tree leaves.

Rationale: Without loss of generality, we first assume that in each tree level we want to digest L blinded syndromes (referred to as $S_1, S_2, \ldots, S_L$) into a single digest D. Our primary goal at this point is to prevent adversaries from controlling the input values in a meaningful way for them (i.e., be able to predict how the digest D will be affected by changing an input syndrome S_i). This objective is accomplished by encrypting each input syndrome using a secure block cipher under a secret key: since block ciphers are pseudo-random permutations, adversaries will be unable to predict how input syndromes (under their control) are transformed before being digested into D. At this point, since encryption transforms the inputs in an unpredictable way, we can accumulate them as $\mathcal{E}(S_1)+\mathcal{E}(S_2)+\cdots+\mathcal{E}(S_L)$ and efficiently digest the latter using a fast Mersenne reduction.

The resulting residue is unpredictable and its computation parallelizable, but it cannot be securely stored outside the TCB yet, as it is easily extendable if an adversary has direct access to it.[11] Hence, adapting the *digest-then-mask* paradigm as in the construction of Section 6.1.2, we further encrypt this intermediate residue (Alg. 6, step 7), before returning the final digest D to untrusted or eavesdroppable memory. In effect, enclosing the digest operation (i.e., addition modulo M_p) between two `Simon` encryptions inside the TCB creates a virtual black box for adversaries, at it prevents predicting how D is affected by altering the inputs S_i, as well as predicting the intermediate residue by observing both inputs and outputs.

Commutativity Protection: A desired feature of our construction is the ability to encrypt inputs independently (i.e., there is no cascading requirement), which allows parallelization during computation of the intermediate residue. Without additional care, however, adversaries may cause collisions to that residue (and eventually to the resulting digest D) by judiciously permuting the order of the input syndromes S_i. Specifically, since the inputs are digested using addition modulo M_p, which is a commutative operation, their relative order is not enforced in the residue. To prevent these unwanted collisions, it is necessary to ensure that a different intermediate residue is computed if this relative order is modified. Thus, whether we are processing the lowest level of the `Simon` Tree (i.e., the tree leaves), or intermediate levels, we must concatenate *unique padding bits* to each input syndrome S_i, before encrypting it as an input block (Alg. 6, steps 3 & 4). Provided that the padding bits added to each S_i differ by at least 1 bit from those added to the other $L-1$ syndromes, permuting the order of the inputs would match each syndrome with different padding bits; as a result, due to the *diffusion* properties of the `Simon` block cipher, significantly different encrypted syndrome blocks would be accumulated and the final digest D would also be different. Using this improvement, the output of SIMONDIGEST process is effectively a (multi-query) universal digest of L syndromes, based on the `Simon` cipher.

11. This intermediate residue is only *prefix-free secure*, so without protection, it is theoretically vulnerable to extension attacks (e.g., [52]).

Algorithm 6 Simon Tree Digestion and Update (one level)

Input: $S = (S_1, S_2, \ldots, S_L), D_{old}, idx, S_{new}, K_{Simon}, M_p$
Output: (Updated) Digest D

1: **procedure** SIMONDIGEST(S, K_{Simon}, M_p)
2: **for each** S_i **in** S **do**
3: $E_i \leftarrow \mathcal{E}(K_{Simon}, \text{Pad}(i)||S_i)$ ▷ All in parallel
4: **return** $D_{tmp} = E_1 + E_2 + \cdots + E_L \bmod M_p$
5: **procedure** GENERATE(S, K_{Simon}, M_p)
6: $D_{tmp} \leftarrow$ SIMONDIGEST(S, K_{Simon}, M_p)
7: **return** $D = \mathcal{E}(K_{Simon}, D_{tmp})$ & M_p ▷ Keep p LSBs
8: **procedure** UPDATE($S, D_{old}, idx, S_{new}, K_{Simon}, M_p$)
9: $D_{tmp} \leftarrow$ SIMONDIGEST(S, K_{Simon}, M_p)
10: $E_{old} \leftarrow \mathcal{E}(K_{Simon}, \text{Pad}(idx)||S_{idx})$ ▷ Done in 3:
11: $E_{new} \leftarrow \mathcal{E}(K_{Simon}, \text{Pad}(idx)||S_{new})$ ▷ Parallel to 3:
12: $D_{new} \leftarrow D_{tmp} - E_{old} + E_{new} \bmod M_p$
13: **if** $D_{old} \neq \mathcal{E}(K_{Simon}, D_{tmp})$ **then return** $\perp$ ▷ Error
14: **else return** $D_{upd} = \mathcal{E}(K_{Simon}, D_{new})$ & M_p

Verification & Update: For each tree level, we can apply the aforementioned digestion process to groups of L syndromes, and recursively to higher levels, in order to verify the integrity and freshness of all syndromes up to the tree root. All digestion operations take place within the TCB, where the tree root is also stored; however, all digests at any level between the leaves and the root can be stored in regular memory without risk, as any malicious modification will be detected with high probability by verifying the corresponding path up to the root. When a single leaf syndrome S_i needs to be verified, the TCB loads from memory all leaf syndromes that share the same immediate parent and verifies that the parent digest is correct. The same step is then repeated to the L parents with a common grandparent, which ultimately reduces the verification of a leaf to the verification of the root.

Likewise, updating a leaf syndrome requires verifying first, and then overwriting, each parent digest across the path connecting that leaf to the root. Our construction is beneficial as it allows updating and verifying a tree level at the same time, minimizing the cost of updates. In particular, since the intermediate residue is a modular addition, it can be easily updated with just one modular subtraction of the outdated block and one modular addition of the updated block. Fig. 4 presents a block diagram of our construction, while the digest generation and updating procedures (for one tree level) are summarized in Alg. 6.

6.2.2 Implementation Remarks

One important aspect in the implementation of `Simon` Trees is the time overhead associated with digest generation and updating, as these operations are recursive over multiple tree levels. Traversing all levels of our tree construction

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2017.2722440, IEEE Transactions on Computers

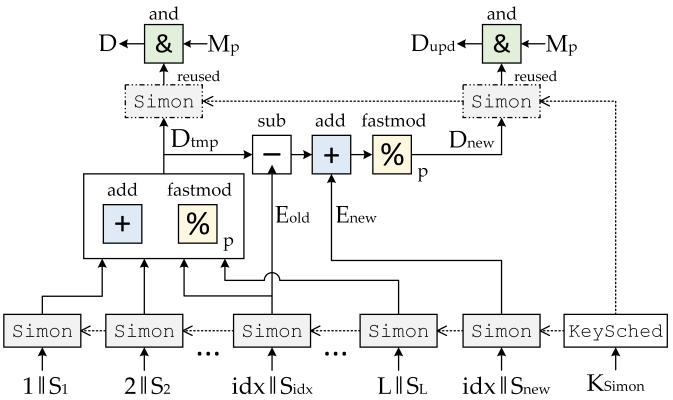

Fig. 4. Block diagram of one level in our `Simon` Tree construction.

incurs a logarithmic overhead on the number of leaves (each corresponding to a blinded syndrome protecting one Paillier ciphertext), and this is consistent with our goal to avoid linear overheads. For example, if $L = 2$ syndromes are used as inputs for each digest D, the recursions required to traverse all tree levels are equal to the number of bits needed for addressing each $(X, X_{bsndr}, ctr_{addr})$ tuple. Moreover, leveraging the inherent parallelization in Step 3 of Alg. 6, we can digest $L = 2^B$ syndromes at once (using redundant encryption cores), while the required number of recursions is $\lceil \log_L V \rceil$, if V is the total number of addressable 3-tuples. In effect, by increasing B we can reduce the total number of non-parallelizable encryptions across all tree levels, trading hardware resources for parallelization. For instance, if $V = 2^{32}$ and $L = 2^3$ parallel encryption cores are available, we can update a tree leaf within $\lceil 32/3 \rceil = 11$ recursions.

Another vital concern is the selection of a Mersenne prime $M_p = 2^p - 1$, which should be compatible with the available block sizes in the `Simon` cipher family, and allow adequate room for unique padding bits. For example, if the number of input syndromes is $L = 8 = 2^3$, at least 8 unique padding values are required, or a minimum of 3 padding bits. Then, in case the block length is 64 bits and we use $M_p = M_{61}$ (i.e., the blinded syndromes are 61 bits long), the concatenation of the syndrome with its padding matches the block size exactly. In general, we require that $p + B \leq$ BlkSize to ensure each padding is unique; if this constraint is violated, the next available block size should be used. Overall, the block size determines the time overhead for digesting L syndromes, as the latter corresponds to the cost of two consecutive `Simon` encryptions in addition to $L+1$ or $L+3$ modular additions for generating or updating digests.[12]

7 Experimental Evaluation

We evaluated the effectiveness and efficiency of our error detection framework using Monte Carlo simulations in Python 2.7, as well as by synthesizing Verilog implementations of the proposed primitives with Xilinx XST 14.7. All simulations were executed on two 8-core Xeon E5-2650 servers, with 64GB RAM, running 16 Python threads at 2.00GHz each, while the XST target was a Kintex xc7k160t-3. As already mentioned in our threat model (Section 2), we focus on detecting (a) random errors at any round of a homomorphic ALU, (b) soft errors in encrypted memory (SEUs and MBUs as in [32]), and (c) malicious modifications in encrypted memory (assuming a tamper-proof TCB). The basic primitives for detecting random errors in memories and ALUs are a fast Mersenne reduction module and a small modular multiplier for syndromes. To add protection against malicious memory modifications (Section 6), our required TCB comprises the blinded exponential syndrome unit (Fig. 3), the `Simon` tree unit (Fig. 4), protected storage for secret keys (Table 1) and the `Simon` tree root, and well the error-detecting ALU (illustrated in Fig. 1). In the following paragraphs, we present the corresponding detection probabilities and RTL implementation overheads.

12. We assume $L + 1$ parallel encryption cores are available for Steps 3 & 11 in Alg. 6 (equivalent to one encryption), and two of those $L + 1$ cores are reused in parallel for Steps 13 & 14 (equivalent to a second encryption). All $L + 1$ cores can share the same key schedule hardware.

TABLE 2
Error Detection Probabilities for Different Mersenne Primes

Mersenne Prime	Error Sources			
	SEU	MBU	ALU	Malicious
M_{19}	100%	100%	99.999809700%	—
M_{31}	100%	100%	99.999999951%	—
M_{61}	100%	100%	$1 - 2^{-61}$	$1 - 2^{-61}$
M_{107}	100%	100%	$1 - 2^{-107}$	$1 - 2^{-107}$

7.1 Error Detection Probabilities

Random Errors (ALU & Memory): In the ALU column of Table 2, we present the detection probabilities for random errors on any round of a homomorphic ALU operation or during syndrome verification (Alg. 2), using different Mersenne primes (which define the syndrome bit length). For M_{19} and M_{31}, we report the detection probabilities after about 10^{10} Monte Carlo simulations in Python, which is consistent with the theoretically expected value of $(1 - M_p^{-1})$. For M_{61} and M_{107} we report the theoretically expected values, as more than 10^{18} simulations would have been required to generate even one escaped fault. Regarding memory soft errors, the SEU and MBU columns of the same Table illustrate the complete coverage achieved by our methodology for single bit flips and clustered faults of up to 4 bits (as in [32]), following the analysis in Section 5.3.

Malicious Modifications (Memory): Our memory error detection methodology against active adversaries requires Mersenne primes of adequate bit length to render brute force attacks infeasible. For that matter, the protection offered by M_{19} and M_{31} is insufficient for the computational power of modern computers. Conversely, for M_{61} and M_{107}, the corresponding exponential syndrome keys are 61 and 107 bits (and the matching `Simon` encryption keys are 96 and 128 bits respectively), so it is intractable for adversaries to brute force these keys and generate malicious syndromes. For these Mersenne primes, the theoretical error detection probability is $(1 - 2^{-61})$ and $(1 - 2^{-107})$, as shown in the last column of Table 2.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2017.2722440, IEEE Transactions on Computers

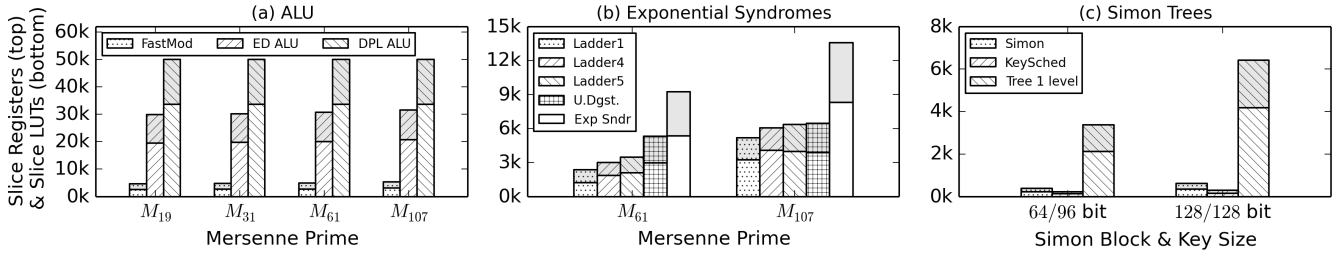

Fig. 5. Area overheads in terms of FPGA Slice LUTs (bottom bars) and Slice Registers (top stacked bars) for our implemented primitives.

7.2 Area & Runtime Overheads

To evaluate the runtime and area overheads of our proposed methodology, we implemented all associated primitives in Verilog HDL, and compared them to a typical fault detection approach of resource duplication. Specifically, we implemented (a) fast modular reduction cores for different input sizes and Mersenne primes (Alg. 1), (b) primary and secondary modular multiplication units (Step 2 & 6 of Alg. 2), (c) `Simon` encryption cores variants (with key schedules) used for syndrome blinding and `Simon` trees (Alg. 3 & 6), (d) a parallel modular exponentiation ladder (Alg. 4), (e) a universal digest module based on Horner's rule for evaluating polynomials (Alg. 5), and (f) high-radix multicycle multiplication variants to handle each exponentiation step, while reducing the critical paths of Alg. 4. Without loss of generality, we focus on Mersenne primes M_{19} and M_{31} (which are adequate for random error detection in the ALU and memory), as well as M_{61} and M_{107}, which are good matches for both random and malicious error detection.

Area: In Fig. 5, we report the area overheads of our implementation in terms of FPGA Slice registers (darker bars stacked on top of lighter bars) and Slice LUTs (lighter bars below the darker ones). In the left subplot of Fig. 5, we compare the area of the fast Mersenne reduction module for 2048 bit arguments and four different primes M_p, against the area of a duplicated modular multiplication ALU for 2048 bit arguments ("DPL ALU") and our proposed error-detecting ALU (Fig. 1). Recall that error detection increases the area of the primary ALU with one fast reduction unit and one small syndrome multiplier; as our results indicate for M_{19} and M_{31}, the ALU area is increased about 20% to enable random error detection (attributed to the cost of fast reductions and syndrome multiplication), which is five times smaller compared to the 100% increase in case of resource duplication.

In the center subplot of Fig. 5, we report the total area overhead of our blinded exponential syndrome unit (Fig. 3) for primes M_{61} and M_{107}, and the individual overheads of our universal digest module and three different instantiations of our exponentiation ladder with varying multiplication steps (traditional 1-step, and high-radix 4- or 5-step). Since exponentiation requires multiplications of $61 \cdot 61$ or $107 \cdot 107$ bit numbers, multiple steps help reducing the critical paths. Our results indicate that increasing the multiplication steps modestly increases area overheads (in an effort to trade area and execution steps for shorter clock periods), while the universal digest and exponentiation ladder modules dominate the area cost of the blinding unit. Moreover, in the right subplot, we compare the area cost of a single `Simon` encryption module and a single key schedule,

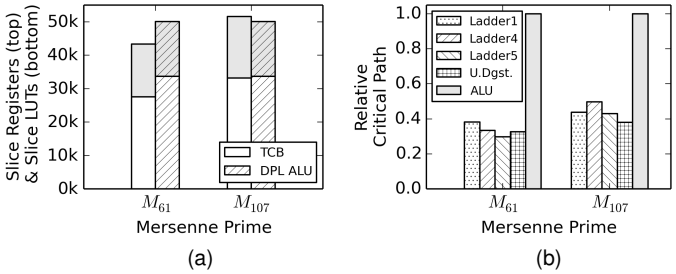

Fig. 6. (a) Total area overhead of our TCB for different M_p primes. (b) Comparison of relative critical paths for our primary ALU, universal digest, and ladder exponentiation units (for varying multiplication steps).

to the cost of our `Simon` Tree construction for digesting $L = 8$ syndromes in parallel using redundant encryption cores and a common key schedule (as in Fig. 4), for the $64/96$ and $128/128$ block cipher variants.

The total area overhead of our entire TCB is summarized in Fig. 6a. For M_{61}, our results indicate that the TCB cost is about 13.5% less than the cost of a duplicated ALU, which does not offer any memory protection against active adversaries; in case of M_{107}, this cost is merely 3.0% more than the duplicated ALU cost. Hence, the additional area to defend against our extended threat model is about the same as one unprotected 2048-bit modular multiplier.

Critical Path: An important observation is that the modular exponentiation ladder could benefit from high-radix multiple step multiplication, in order to decrease its critical path. For that matter, in Fig. 6b we perform a relative comparison of the critical path of our universal digest and our ladder exponentiation (for a traditional single step, as well as 4- and 5-step high-radix multiplication cores), against the critical path of our primary ALU. Our results for M_{61} show that the shortest path is achieved using a 5-step multiplier (which, however, incurs larger area and runtime overheads). Yet, our multi-step optimization is less effective for M_{107}, where the critical path using a single step multiplication is almost the same as using a 5-step multiplication. In all cases, both our 61-bit and 107-bit ladder exponentiations, as well as our Horner's rule universal digests, have shorter critical paths compared to our primary ALU operation.

Clock Cycles: Based on our HDL simulations, we also compare the runtime overhead of our primitive, depending on the selection of Mersenne prime M_p. In Fig. 7, we report the required cycles for digesting one level of a `Simon` tree and computing one blinded exponential syndrome (with 1-step, 4-step and 5-step multiplications respectively), compared to one error detecting ALU operation and one duplicated ALU operation. For M_{19} and M_{31}, we only report the ALU runtime overheads, as these primes are not adequately large to support our malicious error detection primitives. In case

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2017.2722440, IEEE Transactions on Computers

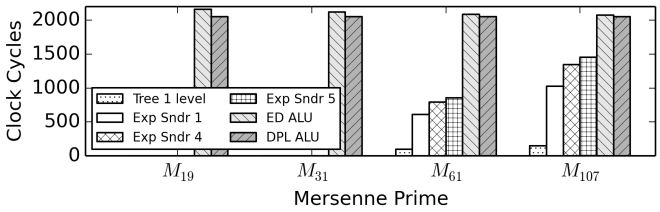

Fig. 7. Runtime overhead (clock cycles) for digesting 1 Simon Tree level, computing a blinded syndrome (with 1-step, 4-step or 5-step multiplication) and completing a 2048-bit error detecting ALU operation, compared against a duplicated ALU operation of the same bitsize.

of malicious memory errors, our simulations show that the cost of computing a blinded exponential syndrome is completely masked by one primary ALU operation, while the small overhead of digesting one level of a Simon Tree allows pipelining the evaluation of multiple levels in parallel to ALU operations. Moreover, compared to a duplicated ALU, our results indicate that one error detecting ALU operation requires between 1.2% and 5.5% more cycles (depending on M_p), due to the additional reduction in Step 3 of Alg. 2.

8 Related Work

In the area of reliable computation, several error detection methods have been proposed in the past. Traditional approaches, such as resource duplication, M out-of N majority vote (e.g. triple modular redundancy) or time redundancy, can be effective, but incur undesirable area or delay overheads. In addition, error codes, such as Berger, Bose-Lin, BCH, Reed-Muller, Hamming, or Cyclic Redundancy may provide robust detection of random memory errors [53], [54], but are incompatible with encrypted computation by their construction: they either cannot support homomorphic operations like modular multiplication, or can only be ported to multiplication over specific fields like *binary extension fields* (e.g. [55]), which is not generally the case in encrypted computation. Moreover, random memory errors can be mitigated using ECC memory modules, where protections are implemented within memory ICs. Still, considering that such memories typically use 8 parity bits for every 64 data bits [45], storage efficiency can be lower, compared to the practical syndrome sizes in this work.

In cryptographic applications, error detection is possible using integrity primitives, such as message authentication codes (MACs) or digital signatures [25], and encrypted values can be protected by performing either *MAC-then-Encrypt* or *Encrypt-then-MAC* operations [56]. Notably, these methods rely on storage hungry and time-consuming hashing operations, especially when *hash chaining* is required for values in the order of thousand bits. Our methodology, on the other hand, leverages efficient and parallelizable digests. Furthermore, since hashing is not homomorphic, directly porting these methods to encrypted computation would require either decrypting and re-encrypting arguments after HE operations (for MAC-then-Encrypt) or expensive rehashing (for Encrypt-then-MAC).

Memory integrity against malicious modifications (e.g., replay attacks) can be achieved efficiently using hash tree constructions. In [57], the authors introduce "Bonsai" Merkle trees, which have compressed sizes, as it is sufficient to protect short counter values instead of longer MACs. Nevertheless, this approach leverages symmetric encryption for protecting memory values, and both the ciphertexts and the MACs are not homomorphic by construction. Likewise, "skewed" Merkle trees are proposed in [58], in an effort to reduce the paths between tree nodes for frequently used memory blocks and the tree root; in encrypted computation, however, memory is randomly permuted (e.g., [44]) and it cannot be predicted which memory locations will be used more or less frequently.

With respect to efficient modulo operations, the authors of [42], propose an improved hardware implementation of modular reductions that requires $\mathcal{O}(n-m)$ steps, where n is the argument size and m is the modulus size in bits. Their approach is more attractive when the modulus size in very big, while in this work we purposefully select relatively small Mersenne primes, to reduce the storage requirements for syndromes. Furthermore, the authors of [59] discuss improved reduction algorithms for arbitrary moduli, but their approach uses lookup tables that require continuous memory accesses and may leak side channel information in addition to affecting performance. In this work, our reductions are optimized, since we do not require arbitrary moduli (except in the primary HE ALU).

Protecting modular multiplication ALUs using residue numbering has also been explored in the literature, in the context of fault injection attacks. In [60], the authors instantiate an algorithm for RNS-based Montgomery multiplication that enables detection of single faults, while preventing private key extraction attacks in cryptosystem instantiations through leak resistant arithmetic. Their main security objective is to prevent side channel leakage, while our primary goal in this work is detecting active integrity attacks in encrypted memory, as well as random errors in the ALU.

9 Concluding Remarks

In this work, we introduce an efficient framework that provides error detection in additive encrypted computation ALUs and memories. Our contribution leverages residue numbering properties and fast reductions modulo Mersenne primes, to provide random error detection rates of at least 99.999% in homomorphic ALUs, and 100% coverage for single bit-flips and up to four clustered faults in encrypted memories. Moreover, we introduce two memory integrity primitives, namely blinded exponential syndromes and Simon Trees, which support parallelization and enable the detection of a wide range of malicious modifications in encrypted memory, provided that a TCB is present for storing secret information and computing integrity metadata. In our experiments, we measured the area and runtime overheads of HDL implementations of our primitives for an FPGA target, and compared our approach against a resource duplication strategy. Our results indicate a runtime overhead between 1.2% and 5.5% for ALU operations, which renders this approach a suitable alternative for generic VLSI fault detection methods ported to encrypted computation.

Appendix A
The Paillier Cryptosystem

The Paillier cryptosystem is one of the first efficient homomorphic encryption schemes that supports the addition

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2017.2722440, IEEE Transactions on Computers

operation [23]. The HE scheme is mathematically based on the *decisional composite residuosity assumption*, which states that given a composite number n and an integer z, it is *hard* to decide whether there exists y such that:

$$z \equiv y^n \pmod{n^2}. \tag{23}$$

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 primes and $\lambda = lcm(p-1, q-1)$; the bit size of n is the *security parameter* of the cryptosystem. Let g be a random integer in $\mathbb{Z}^*_{n^2}$ so that $\mu = (L(g^\lambda \pmod{n^2}))^{-1} \pmod{n}$ exists, where -1 power refers to the modular multiplicative inverse and $L(x) = \frac{x-1}{n}$. The public key is (n, g) and the private key is (λ, μ).

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 function of a message m is:

$$Enc[m] = g^m r^n \pmod{n^2} \tag{24}$$

and the decryption function of a ciphertext c is:

$$Dec[c] = L(c^\lambda \pmod{n^2}) * \mu \pmod{n}. \tag{25}$$

The homomorphism of this scheme is defined as $Dec[Enc[m_1] * Enc[m_2] \pmod{n^2}] = m_1 + m_2 \pmod{n}$, 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 can be the basis for encrypted computation (e.g. [11], [22]).

Paillier Cryptosystem with Multiple Primes

In case more than two prime numbers are used in the calculation of modulus n, the key generation is modified as follows:

Let $\{p_1, p_2, \cdots, p_k\}$ a set of k prime numbers, where at least two of them are sufficiently large (of equivalent length), chosen randomly and independently of each other. Then the modulus would be $n = \prod p_i$ with $1 \leq i \leq k$, and λ would be equal to $lcm(p_1, p_2, \cdots, p_k)$.

The security of the construction holds, since it is computationally infeasible to calculate λ without knowledge of the factorization of n. Indeed, since at least two of the prime factors of n are large and random, factorization of n is assumed to be an intractable problem.

ACKNOWLEDGMENTS

This work was partially sponsored by the NYU Abu Dhabi Global Ph.D. Student Fellowship program.

REFERENCES

[1] R. L. Rivest, L. Adleman, and M. L. Dertouzos, "On data banks and privacy homomorphisms," *Foundations of secure computation*, vol. 4, no. 11, pp. 169–180, 1978.

[2] C. Gentry, "Computing arbitrary functions of encrypted data," *Communications of the ACM*, vol. 53, no. 3, pp. 97–105, 2010.

[3] 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.

[4] T. ElGamal, "A public key cryptosystem and a signature scheme based on discrete logarithms," *IEEE Transactions on Information Theory*, vol. 31, no. 4, pp. 469–472, 1985.

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

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

[7] I. Damgård, M. Jurik, and J. B. Nielsen, "A generalization of paillier's public-key system with applications to electronic voting," *International Journal of Information Security*, vol. 9, no. 6, pp. 371–385, 2010.

[8] D. Catalano and D. Fiore, "Using linearly-homomorphic encryption to evaluate degree-2 functions on encrypted data," in *2015 ACM Computer and Communications Security (CCS)*. ACM, 2015, pp. 1518–1529.

[9] I. Damgård, V. Pastro, N. Smart, and S. Zakarias, "Multiparty computation from somewhat homomorphic encryption," in *Advances in Cryptology–CRYPTO 2012*. Springer, 2012, pp. 643–662.

[10] M. Brenner *et al.*, "Secret program execution in the cloud applying homomorphic encryption," in *Digital Ecosystems and Technologies Conference (DEST)*, 2011, pp. 114–119.

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

[12] S. Goldwasser, Y. Kalai, R. A. Popa, V. Vaikuntanathan, and N. Zeldovich, "Reusable garbled circuits and succinct functional encryption," in *ACM symposium on Theory of computing*. ACM, 2013, pp. 555–564.

[13] F. Baldimtsi and O. Ohrimenko, "Sorting and searching behind the curtain," in *Financial Cryptography and Data Security*. Springer, 2015, pp. 127–146.

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

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

[16] X. Chen, X. Huang, J. Li, J. Ma, W. Lou, and D. S. Wong, "New algorithms for secure outsourcing of large-scale systems of linear equations," *IEEE Transactions on Information Forensics and Security*, vol. 10, no. 1, pp. 69–78, 2015.

[17] R. A. Popa, C. Redfield, N. Zeldovich, and H. Balakrishnan, "CryptDB: protecting confidentiality with encrypted query processing," in *ACM Symposium on Operating Systems Principles (SOSP)*. ACM, 2011, pp. 85–100.

[18] M. Naehrig, K. Lauter, and V. Vaikuntanathan, "Can Homomorphic Encryption Be Practical?" in *ACM Cloud Computing Security Workshop (CCSW)*. ACM, 2011, pp. 113–124.

[19] K. Peng *et al.*, "Multiplicative homomorphic e-voting," in *Progress in Cryptology-INDOCRYPT 2004*. Springer, 2005, pp. 61–72.

[20] 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.

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

[22] 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.

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

[24] D. A. Barrington, "Bounded-width polynomial-size branching programs recognize exactly those languages in NC^1," in *ACM Symposium on Theory of Computing*. ACM, 1986, pp. 1–5.

[25] J. Katz and Y. Lindell, *Introduction to modern cryptography*. CRC Press, 2008.

[26] P. Paillier and D. Pointcheval, "Efficient public-key cryptosystems provably secure against active adversaries," in *Advances in Cryptology-ASIACRYPT'99*. Springer, 1999, pp. 165–179.

[27] H. L. Garner, "The residue number system," *IRE Transactions on Electronic Computers*, vol. EC-8, no. 2, pp. 140–147, 1959.

[28] 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.

[29] R. M. Robinson, "Mersenne and fermat numbers," *Proceedings of the American Mathematical Society*, vol. 5, no. 5, pp. 842–846, 1954.

[30] R. Cramer, R. Gennaro, and B. Schoenmakers, "A secure and optimally efficient multi-authority election scheme," *European transactions on Telecommunications*, vol. 8, no. 5, pp. 481–490, 1997.

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

[32] M. Maniatakos, M. K. Michael, and Y. Makris, "Multiple-bit upset protection in microprocessor memory arrays using vulnerability-based parity optimization and interleaving," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 23, no. 11, pp. 2447–2460, 2015.

[33] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten, "Lest we remember: cold-boot attacks on encryption keys," *Communications of the ACM*, vol. 52, no. 5, pp. 91–98, 2009.

[34] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu, "Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors," in *ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, 2014, pp. 361–372.

[35] R. W. Hamming, "Error detecting and error correcting codes," *Bell System technical journal*, vol. 29, no. 2, pp. 147–160, 1950.

[36] R. T. Chien, "Cyclic decoding procedures for Bose-Chaudhuri-Hocquenghem codes," *IEEE Transactions on Information Theory*, vol. 10, no. 4, pp. 357–363, 1964.

[37] Apple Inc., "iOS Security (whitepaper)," [Online]. Available: https://www.apple.com/business/docs/iOS_Security_Guide.pdf, 2017, (Accessed: March 2017).

[38] F. McKeen *et al.*, "Innovative instructions and software model for isolated execution," in *Hardware and Architectural Support for Security and Privacy (HASP)*, 2013, pp. 1–8.

[39] W. Jenkins, "The design of error checkers for self-checking residue number arithmetic," *IEEE Transactions on Computers*, vol. 100, no. 4, pp. 388–396, 1983.

[40] J.-C. Bajard, L.-S. Didier, and P. Kornerup, "An RNS montgomery modular multiplication algorithm," *IEEE Transactions on Computers*, vol. 47, no. 7, pp. 766–776, 1998.

[41] D. Boneh and H. Shacham, "Fast variants of RSA," *CryptoBytes*, vol. 5, no. 1, pp. 1–9, 2002.

[42] J. T. Butler and T. Sasao, "Fast hardware computation of x mod z," in *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW)*. IEEE, 2011, pp. 294–297.

[43] R. Baumann, "Soft errors in advanced computer systems," *IEEE Design & Test of Computers*, vol. 22, no. 3, pp. 258–266, 2005.

[44] N. G. Tsoutsos and M. Maniatakos, "HEROIC: Homomorphically EncRypted One Instruction Computer," in *Design, Automation and Test in Europe Conference and Exhibition (DATE)*, 2014, pp. 1–6.

[45] D. H. Yoon and M. Erez, "Memory mapped ECC: Low-cost error protection for last level caches," in *International Symposium on Computer Architecture (ISCA)*. ACM, 2009.

[46] G. H. Hardy and E. M. Wright, *An introduction to the theory of numbers*. Oxford University Press, 1979.

[47] A. J. Menezes, P. C. Van Oorschot, and S. A. Vanstone, *Handbook of applied cryptography*. CRC press, 1996.

[48] J. L. Carter and M. N. Wegman, "Universal classes of hash functions," *Journal of computer and system sciences*, vol. 18, no. 2, pp. 143–154, 1979.

[49] T. Krovetz, "Message authentication on 64-bit architectures," in *International Workshop on Selected Areas in Cryptography*. Springer, 2006, pp. 327–341.

[50] M. N. Wegman and J. L. Carter, "New hash functions and their use in authentication and set equality," *Journal of computer and system sciences*, vol. 22, no. 3, pp. 265–279, 1981.

[51] M. Joye and S.-M. Yen, "The montgomery powering ladder," in *Cryptographic Hardware and Embedded Systems (CHES)*. Springer, 2002, pp. 291–302.

[52] N. G. Tsoutsos and M. Maniatakos, "Cryptographic vote-stealing attacks against a partially homomorphic e-voting architecture," in *International Conference on Computer Design (ICCD)*. IEEE, 2016, pp. 157–160.

[53] S. S. Gorshe, "Concurrent error detection," Ph.D. dissertation, Oregon State University, 2002.

[54] S. Mitra and E. J. McCluskey, "Which concurrent error detection scheme to choose?" in *International Test Conference (ITC)*. IEEE, 2000, pp. 985–994.

[55] C.-W. Chiou *et al.*, "Concurrent error detection in montgomery multiplication over $GF(2^m)$," *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, vol. 89, no. 2, pp. 566–574, 2006.

[56] M. Bellare and C. Namprempre, "Authenticated encryption: Relations among notions and analysis of the generic composition paradigm," in *Advances in Cryptology-ASIACRYPT 2000*. Springer, 2000, pp. 531–545.

[57] B. Rogers, S. Chhabra, M. Prvulovic, and Y. Solihin, "Using Address Independent Seed Encryption and Bonsai Merkle Trees to Make Secure Processors OS- and Performance-Friendly," in *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2007, pp. 183–196.

[58] J. Szefer and S. Biedermann, "Towards Fast Hardware Memory Integrity Checking with Skewed Merkle Trees," in *Hardware and Architectural Support for Security and Privacy (HASP)*, 2014, pp. 1–8.

[59] M. A. Will and R. K. L. Ko, "Computing mod without mod," Cryptology ePrint Archive, Report 2014/755, 2014.

[60] J.-C. Bajard, J. Eynard, and F. Gandino, "Fault detection in RNS Montgomery modular multiplication," in *Symposium on Computer Arithmetic (ARITH)*. IEEE, 2013, pp. 119–126.

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 Tandon School of Engineering, New York University, Brooklyn, NY, USA.

Mr. Tsoutsos has been the Challenge Leader of the Embedded Security Competition held during the Cyber Security Awareness Week in Brooklyn, NY, USA, since 2015. He holds a patent on encrypted computation using homomorphic encryption, and has authored several articles in IEEE Transactions and conference proceedings. His current research interests include computer security, computer architecture, encrypted computation and embedded systems.

Michail Maniatakos (S'08–M'12) received the B.Sc. degree in computer science and the M.Sc. degree in embedded systems from the University of Piraeus, Piraeus, Greece, in 2006 and 2007, respectively, and the M.Sc., M.Phil., and Ph.D. degrees from Yale University, New Haven, CT, USA, in 2009, 2010, and 2012, respectively, all in electrical engineering. He is currently an Assistant Professor of Electrical and Computer Engineering with New York University (NYU) Abu Dhabi, Abu Dhabi, UAE, and a Research Assistant Professor with the NYU Tandon School of Engineering, Brooklyn, NY, USA. He is also the Director of the MoMA Laboratory, NYU Abu Dhabi. His research interests, funded by industrial partners and the U.S. Government, include robust microprocessor architectures, privacy preserving computation, and industrial control systems security. He has authored several publications in IEEE Transactions and conferences, and holds patents on privacy-preserving data processing.

Dr. Maniatakos is currently the Co-Chair of the Security track at the IEEE International Conference on Computer Design and the IEEE International Conference on Very Large Scale Integration. He also serves on the Technical Program Committee for various conferences, including the IEEE/ACM Design Automation Conference, the International Conference on Computer Aided Design, the International Test Conference, and the International Conference on Compilers, Architectures and Synthesis for Embedded Systems. He has organized several workshops on security, and he currently is the Faculty Lead of the Embedded Security challenge held annually at Cyber Security Awareness Week, Brooklyn, NY, USA.