Efficient Fault-Detection Architectures for Barrett Reduction and Multiplication in Classical and Post-Quantum Cryptographic Systems

Saeed Aghapour, Kiarash Sedghighadikolaei, *Graduate Student Member, IEEE*, Attila A. Yavuz, *Senior Member, IEEE*, Bechir Hamdaoui, and Mehran Mozaffari-Kermani, *Senior Member, IEEE*

***Abstract*—Barrett modular reduction and multiplication are essential primitives for efficient modular computation in cryptographic schemes, including post-quantum standards such as module lattice-based (ML) key encapsulation mechanism (KEM) and ML-digital signature algorithm (DSA). To protect against faults that compromise correctness and security, we introduce the first efficient fault-detection mechanisms tailored to these operations. For modular reduction, we leverage word-based representations with compact word-sum checks that exploit algebraic input-output relations to ensure computational integrity. For modular multiplication, we adopt a tunable hybrid strategy: early stages apply word-sum checks, while later stages use partial recomputation with encoded inputs, providing robust protection against injected faults. Formal analysis, fault-injection simulations, and hardware/software implementations show that our methods detect a wide range of faults with minimal performance and area overhead. Evaluation results demonstrate overheads of 3.43% and 7.15% for 512-bit and 1024-bit inputs in modular reduction, and 26.47% and 27.22% for 2048-bit inputs in modular multiplication in the number of clock cycles in software. Moreover, in hardware, we observed reasonable overheads: less than 27.5% in area and 2.1% in delay for modular reduction, and less than 23.5% in area and 16.2% in delay for modular multiplication. These results confirm the practicality of our methods for secure yet efficient integration.**

***Index Terms*—Advanced RISC machines (ARMs) processor, Barrett modular multiplication (BMM), Barrett modular reduction, cryptography, fault detection, FPGA.**

I. Introduction

In TODAY's digital landscape, cryptography, particularly public-key cryptography (PKC), is essential for securing data through confidentiality, integrity, and authentication. Traditional PKC systems are based on well-established computational hardness assumptions, such as integer factorization (e.g., Rivest–Shamir–Adleman (RSA) [1]), the discrete logarithm problem (e.g., ElGamal [2] and Schnorr), and problems defined over elliptic curves (e.g., elliptic curve digital signature algorithm (ECDSA) [3], EdDSA [4], and Schnorr [5]). Moreover, the advent of quantum computing threatens these schemes, as quantum algorithms (e.g., Shor's [6]) can solve these problems efficiently. In response, extensive research, primarily through the NIST-led post-quantum cryptography standardization process [7], has led to the selection of module lattice-based (ML) digital signature algorithm (DSA) [8], Falcon [9], and SLH-DSA [10] as standard quantum-resistant digital signatures and ML-key encapsulation mechanism (KEM) [11] as the standard KEM. The standardization is still ongoing to select additional PQ-secure primitives [12].

Classical PKC schemes (e.g., RSA, ECDSA, and Schnorr) depend extensively on modular arithmetic over large integers to achieve strong security guarantees. Core operations, notably modular multiplication followed by reduction, are computationally intensive over the large numbers and present significant implementation challenges. Similarly, PQC schemes like ML-KEM and ML-DSA utilize polynomial multiplication through the number-theoretic transform (NTT) [13], necessitating efficient modular arithmetic. Therefore, developing optimized techniques for modular multiplication and reduction is crucial to the performance of both conventional and PQ cryptographic systems.

Barrett modular reduction (`BMR`) [14] and Montgomery reduction [15] are widely adopted techniques for efficient modular reduction, replacing expensive division operations with lower-cost shifts and multiplications. `BMR` requires only a single precomputed constant and operates directly on standard integer representations, eliminating the need for conversions into and out of a special domain, unlike Montgomery reduction. In contrast, Montgomery involves at least three steps, two domain conversions and the reduction itself, which introduces overhead that is only justified when performing many modular reductions (e.g., in exponentiation loops). Hence, Barrett is often preferred in hardware-constrained or latency-sensitive scenarios like ASIC/FPGA designs with limited area, cryptographic applications involving few modular operations (e.g., hashing or ECC verification), and quantum or edge computing, where conversion costs are significant [16].

Modular multiplication can involve either full multiplication followed by reduction or Barrett modular multiplication (`BMM`) [17], which integrates quotient approximation directly. Due to the transformation overhead of the Montgomery reduction,

Received 26 May 2025; revised 3 August 2025; accepted 7 September 2025. Date of publication 18 September 2025; date of current version 27 November 2025. This work was supported by USA National Science Foundation (NSF) under Award SaTC-2350213 and Award SaTC-2350214. *(Saeed Aghapour and Kiarash Sedghighadikolaei contributed equally to this work.) (Corresponding author: Mehran Mozaffari-Kermani.)*

Saeed Aghapour, Kiarash Sedghighadikolaei, Attila A. Yavuz, and Mehran Mozaffari-Kermani are with the College of AI, Cybersecurity, and Computing, University of South Florida, Tampa, FL 33620 USA (e-mail: aghapour@usf.edu; kiarashs@usf.edu; attilaayavuz@usf.edu; mehran2@usf.edu).

Bechir Hamdaoui is with the Department of Electrical Engineering and Computer Science, Oregon State University, Corvallis, OR 97331 USA (e-mail: hamdaoui@eecs.oregonstate.edu).

Digital Object Identifier 10.1109/TVLSI.2025.3609666

Barrett-based approaches BMR and BMM are generally preferred for resource-sensitive settings.

Although BMR and BMM offer significant computational efficiency, they remain vulnerable to fault attacks, a class of side-channel attacks that target the integrity of cryptographic computations [18]. In such attacks, adversaries deliberately inject faults into cryptographic computations to exploit algorithmic vulnerabilities and potentially extract sensitive information. Even when no direct data leakage occurs, induced faults can compromise system integrity by producing incorrect results or triggering malfunctions. To mitigate these risks, fault-detection [19], [20], [21], [22], [23] has become an essential defense mechanism, capable of identifying both naturally occurring errors (e.g., single-event upsets) and maliciously induced faults, thereby enhancing system robustness and, in some cases, enabling error recovery. Fault detection is also critically important in distributed cryptographic computations (e.g., threshold cryptography [24]) where computations are performed across multiple parties that may not fully trust one another (malicious security models).

A. Related Work

BMM and BMR are foundational techniques with broad applicability across both classical and post-quantum cryptographic systems. Recent efforts have focused on enhancing their efficiency for core arithmetic operations. Abdulrahman et al. [25] introduced a signed BMR for ML-KEM and a reduced prime modulus for ML-DSA, facilitating efficient use of the Fermat Number Transform. Their optimizations yielded 5%–6% speedups in NTT operations, up to 37.6% faster signing in ML-DSA, and a 15.9%–17.8% improvement in matrix-vector multiplication for ML-KEM. Complementing this, Becker et al. [26] established new performance benchmarks on the Armv8-A architecture by exploiting BMR and leveraging NEON vector instructions and advanced butterfly operations for further acceleration. In addition, Chiu and Parhi [27] proposed a flexible BMR architecture handling input word lengths exceeding conventional limits, enabling efficient computation of residual polynomials in homomorphic encryption while achieving up to 31% improvements in DSP area-delay efficiency.

Fault attacks [18], which manipulate inputs or execution to compromise cryptographic operations, pose serious threats even through minor disruptions. In response, various fault-detection mechanisms have been developed for both classical and post-quantum cryptographic systems. Studies have addressed fault resilience in ECC and RSA [21], [22], [28], [29], [30], [31], [32], as well as AES implementations [33], [34]. In the PQC domain, countermeasures for lattice-based signatures have been evaluated for their effectiveness and overhead [35]. Notably, Sarker et al. [36] and Howe et al. [37] proposed detection methods targeting NTT computations and error samplers, respectively.

To the best of our knowledge, no prior work has proposed an efficient fault-detection mechanism explicitly designed for BMR and BMM. Considering their pervasive role in both classical and post-quantum cryptographic constructions, and the well-established threat posed by fault-induced vulnerabilities, it is remarkable that these foundational operations have not yet been the focus of dedicated fault-detection research. This oversight underscores a critical gap in the literature, one that calls for the integration of robust, lightweight detection techniques tailored specifically to BMR and BMM computations.

B. Contribution

In this work, we propose two constructions, BMR and BMM, both integrated with efficient fault-detection mechanisms. The primary contributions of this study are outlined as follows.

1) *Efficient Fault Detection for* BMR: We propose two countermeasures to detect fault injections. The first is a simple counter, while the second is a function we call Sum(.). This function efficiently sums the words of a parameter and subsequently, through the utilization of a relation between the inputs and outputs, verifies two equations to detect injected faults.
2) *Efficient Fault Detection for* BMM: For this algorithm, in addition to the Sum(.) countermeasure from BMR, we have employed recomputation techniques to further enhance the security of the design against a wider range of fault models. The recomputation module not only improves the detection rate for previously considered fault injections, but also adds coverage for scenarios where the prior countermeasure alone might fail. Moreover, to address the high overhead associated with full recomputation, we propose efficient encoding and partial recomputation techniques.
3) *Comprehensive Simulation of Fault Attacks*: We evaluate our proposed detection method for both BMR and BMM under a wide range of simulated fault attack scenarios, including random and Burst types and bit-flipping and stuck-at faults models. Our theoretical and simulation results indicate that the proposed fault-detection methods for BMR and BMM algorithms provide high error coverage, especially when the number of injections increases.
4) *Hardware and Software Benchmarking*: Our software-based implementation, conducted on a 64-bit Raspberry Pi 4 platform, exhibits an overhead ranging from 7% to 10% for input sizes of 1024 and 2048 bits, and between 3% and 4% for input sizes of 512 and 4096 bits in the context of BMR in terms of number of clock cycles. Furthermore, empirical evaluations indicate that the proposed methodologies introduce an average overhead of approximately 27% in terms of number of clock cycles for BMM compared with the baseline implementation. For hardware-based evaluation, we target AMD/Xilinx FPGAs, specifically the Artix UltraScale+ xcau15p-ffvb676-1LV-iand the Zynq[1]-7000 xc7z030fbg676-2. In general, similar results were observed across both platforms. For the BMR algorithm, we measured an average overhead of 25.9% and 10.6% in the number of LUTs and FFs, and 2.02% in total

[1]Registered trademark.

Algorithm 1 BMR

Input: X, $n = \log_2 N$
Output: r, such that $r = X \bmod N$
Precomputation
1: $\mu = \left\lfloor \frac{b^{2d}}{N} \right\rfloor$, $d = \lceil n/w \rceil$, and $b = 2^w$
Main Body
2: $\hat{q} \leftarrow \left\lfloor \left\lfloor \frac{X}{b^{d-1}} \right\rfloor \frac{\mu}{b^{d+1}} \right\rfloor$
3: $r_1 \leftarrow X \bmod b^{d+1}$ and $r_2 \leftarrow (\hat{q}N) \bmod b^{d+1}$
4: $r \leftarrow r_1 - r_2$
5: **if** $r < 0$
6: $\quad r \leftarrow r + b^{d+1}$
7: **While** $r \geqslant N$
8: $\quad r \leftarrow r - N$
9: **return** r

Algorithm 2 Barrett Modular Multiplication

Input: X, Y, $n = \log_2 N$
Output: $Z = XY \bmod N$
Precomputation
1: $\mu = \lfloor 2^{n+\alpha}/N \rfloor$, $d = \lceil n/w \rceil$, and $b = 2^w$
Main Body
2: $Z^{(d)} = 0$
3: **for** $i = d-1$ **to** 0 **do**
4: $\quad Z^{(i)} \leftarrow Z^{(i+1)}b + XY_i$
5: $\quad q^{(i)} \leftarrow \left\lfloor \left\lfloor \frac{Z^{(i)}}{2^{n+\beta}} \right\rfloor \frac{\mu}{2^{\alpha-\beta}} \right\rfloor$
6: $\quad Z^{(i)} \leftarrow Z^{(i)} - q^{(i)}N$
7: **if** $Z^{(0)} \geqslant N$
8: $\quad Z^{(0)} \leftarrow Z^{(0)} - N$
9: **return** $Z^{(0)}$

Algorithm 3 Proposed BMR

Input: X, N, $n = \log_2 N$
Output: r and *flag* (0 and 1 for *flag* mean erroneous and valid output, respectively).
Precomputation
1: $\mu = \left\lfloor \frac{b^{2d}}{N} \right\rfloor$, $d = \lceil n/w \rceil$, and $b = 2^w$
2: $(S_1^N, S_2^N) = \mathsf{Sum}(N)$
Main Body
3: $(S_1^x, S_2^x) = \mathsf{Sum}(X)$
4: $\hat{q} \leftarrow \left\lfloor \left\lfloor \frac{X}{b^{d-1}} \right\rfloor \frac{\mu}{b^{d+1}} \right\rfloor$
5: $r_1 \leftarrow X \bmod b^{d+1}$ and $r_2 \leftarrow (\hat{q}N) \bmod b^{d+1}$
6: $r \leftarrow r_1 - r_2$
7: **if** $r < 0$
8: $\quad r \leftarrow r + b^{d+1}$
9: **While** $r \geqslant N$
10: $\quad r \leftarrow r - N$
11: $\quad \mathsf{Sub}_{\mathsf{count}} \leftarrow \mathsf{Sub}_{\mathsf{count}} + 1$
12: $\quad$ **if** $\mathsf{Sub}_{\mathsf{count}} > 2$
13: $\quad\quad flag = 0$ (faulty output)
14: $\quad\quad$ **break**
15: $q = \hat{q} + \mathsf{Sub}_{\mathsf{count}}$
16: $(S_1^q, S_2^q) = \mathsf{Sum}(q)$
17: $(S_1^{xr}, S_2^{xr}) = \mathsf{Sum}(X - r)$
Equation Checks
18: $E_1 = (S_1^{xr} - S_1^N S_1^q) \bmod (2^w - 1)$
19: $E_2 = (S_2^{xr} - S_2^N S_2^q) \bmod (2^w - 2)$
20: **if** $(E_1 == 0)$ and $(E_2 == 0)$
21: $\quad flag \leftarrow 1$ (valid output)
22: **return** $r, q, flag$

execution time. For the BMM algorithm, the overhead was 19.5% and 23.3% in the number of LUTs and FFs, and 16.1% in total execution time.

II. Preliminaries

Notation: Let X and Y denote input integers, and let N represent the modulus. The parameter w corresponds to the processor word size. The bit-length of a value x is denoted by $|x|$, which is equivalent to $\log_2 x$, the base-2 logarithm of x. The notation $\lfloor \cdot \rfloor$ denotes the floor function, which returns the greatest integer less than or equal to its input, while $\lceil \cdot \rceil$ denotes the ceiling function, which returns the smallest integer greater than or equal to its input.

A. Barrett Modular Reduction

Given an input integer X and a modulus N, BMR [16], [38] computes the remainder $r = X \bmod N$ efficiently, avoiding explicit division operations. The algorithm operates with respect to a base b, typically chosen as $b = 2^w$. A detailed description of the BMR is provided in Algorithm 1.

B. Barrett Modular Multiplication

Given a modulus $N$and two input values $X$and Y, BMM[17], [39]efficiently computes the product $Z = XY \bmod N$without explicitly computing the full product XYprior to modular reduction. An n-bit integer Ycan be represented in radix-2 as $Y = \sum_{i=0}^{n-1} Y[i] \cdot 2^i$, or alternatively in radix-2^was $Y = \sum_{i=0}^{d-1} Y_i \cdot 2^{iw}$, where $d = \lceil n/w \rceil$, and $Y_i = Y[iw + w - 1 : iw]$denotes the group of bits in Ycorresponding to word i. The BMMalgorithm is outlined in Algorithm 2.

III. Proposed Fault-Detection Architectures

In this section, we propose our countermeasures for these BMR and BMM algorithms.

A. Barrett Modular Reduction

For the BMR algorithm, we propose two countermeasures as follows.

1) Countermeasure 1—Counter in the Subtraction: The first is a counter to be implemented in line 8 of Algorithm 1. An interesting property of BMR is that, in the absence of faults, the while loop in line 7 is executed at most twice (i.e., 0, 1, or 2 times). However, if faults are injected into r_1, r_2, or r before step 7, the values may change in a way that requires significantly more subtractions in line 8. More specifically, the values r_1 and r_2 are both of size $b^{d+1} = 2^{n+w}$, which is larger than N, whose length is n bits. On the other hand, due to precise precomputation, q in BMR is designed to be very close to the actual quotient of X/N. As a result, in the absence of faults, the value $r_1 - r_2$ is very close in size to the modulus N.

However, if faults are injected into the higher bits of r_1 or r_2, the previously computed q becomes inaccurate, and the resulting r can be significantly larger than N. The exact

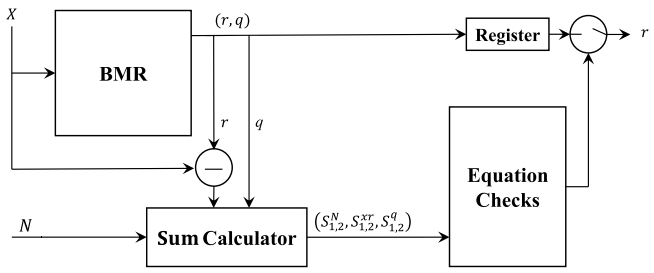

Fig. 1. Architecture of the proposed BMR module.

magnitude depends on the parameter sizes and the specific fault location. In the worst case scenario, the value of r could be up to $n + w$ bits, while N is only n bits, which would require up to 2^w subtractions to exit the while loop in step 9 of Algorithm 3. As a result, a simple counter that tracks the number of subtractions in line 8 can efficiently detect such injected faults inside the algorithm.

2) Countermeasure 2—Sum of Words: The second countermeasure we propose is a function that adds the word components of the values involved in the algorithm and uses two checks to verify whether an error has occurred. To elaborate, any number can be represented as $X = \sum_{i=0}^{n-1} x_i 2^{wi}$, where w is the word size, x_i is the ith word of X, and n is the number of words comprising X. From this, we know that $\sum_{i=0}^{n-1} x_i \equiv X \bmod (2^w - 1)$ and $\sum_{i=0}^{n-1} x_i 2^i \equiv X \bmod (2^w - 2)$. We propose a function named $\mathsf{Sum}(X) = (S_1^X, S_2^X)$ that takes X as input and efficiently outputs two values: $S_1^X = (x_0 + x_1 + \cdots + x_{n-1}) \bmod (2^w - 1)$ and $S_2^X = (x_0 + 2x_1 + \cdots + 2^{n-1}x_{n-1}) \bmod (2^w - 2)$.

Moreover, since in Barrett reduction the output satisfies the relation $X = q \times N + r$, the following equations hold:

$$S_1^X \equiv S_1^q \cdot S_1^N + S_1^r \bmod (2^w - 1)$$
$$S_2^X \equiv S_2^q \cdot S_2^N + S_2^r \bmod (2^w - 2).$$

Since N is known in advance, both S_1^N and S_2^N can be precomputed. Furthermore, to improve efficiency, we exploit the relation $X - r = q \times N$ and compute $\mathsf{Sum}(X - r)$, rather than computing $\mathsf{Sum}(X)$ and $\mathsf{Sum}(r)$ separately. These two proposed equation checks add an additional layer of protection against fault-injection attacks. Most attempts to alter internal values within the algorithm will disrupt the expected functionality, causing these equations to no longer hold. As a result, we verify both equations, and only if they are satisfied do we conclude that no fault has occurred.

The reason we chose the moduli $2^w - 1$ and $2^w - 2$ is that computations with these values can be performed efficiently. First, the expression $(x_0 + x_1 + \ldots + x_{n-1})$ requires no multiplications, and the multiplications in $(x_0 + 2x_1 + \cdots + 2^{n-1}x_{n-1})$ can be implemented using simple shifts, meaning the entire expression can be computed through additions alone. Second, using $2^w - 1$ and $2^w - 2$ ensures that both S_1^A and S_2^A are smaller than the word size, which can be stored in one register. This also enhances the efficiency of operations such as $S_1^q \cdot S_1^N$ and $S_2^q \cdot S_2^N$, because the resulting multiplications can be performed in one single word by word multiplication step.

Furthermore, the reason we employ two checks instead of one is to strike a balance between system overhead and detection rate. Although a single check is sufficient to detect most naturally occurring faults, the second check can identify certain fault-injection cases that the first check alone cannot. One such scenario arises when an attacker injects the same fault into different chunks of a register. With only one check, the adversary could add the fault to one chunk and subtract it from another, leaving the overall summation unchanged. However, with the second check, to pass the equation checks, the adversary must inject matching faults specifically into either the odd- or even-indexed chunks of one parameter. This significantly increases the attack's complexity, especially in the burst fault model, where consecutive bits are affected. Overall, the dual-check approach increases the robustness of the system with only a moderate increase in system overhead compared with a single check. The proposed BMR algorithm is detailed in Algorithm 3 and Fig. 1.

B. Barrett Modular Multiplication

Similar to BMR algorithm, we propose two yet different countermeasures to enhance the security of the BMM algorithm against fault injections.

1) Countermeasure 1—Sum of Words: This countermeasure is similar to the second countermeasure in BMR algorithm. In here, we apply the same idea on line 4 of Algorithm 2 to check if the two following equations hold:

$$S_1^{Z^i} = S_1^{Z^{i+1}} + S_1^X \cdot Y_i \bmod (2^w - 1)$$
$$S_2^{Z^i} = S_2^{Z^{i+1}} + S_2^X \cdot Y_i \bmod (2^w - 2).$$

By linking the results of the current iteration to those of the previous iteration, these equations can detect many faults injected into intermediate registers during the execution of the algorithm. However, if faults are injected into the inputs before the algorithm begins, this countermeasure cannot detect them, as the faulty input is treated as a legitimate input by the algorithm. This limitation is common to all fault-detection mechanisms where the input to the module itself may be compromised. To address such cases, we propose an additional countermeasure.

2) Countermeasure 2—Partial Recomputation With Encoded Inputs: As mentioned earlier, if faults are injected into the inputs before the algorithm starts, the algorithm treats the faulty input as legitimate, and therefore, the equations used in the first countermeasure still hold. To detect such faults, we employ recomputation-based methods.

In recomputation-based methods, the output is computed multiple times (typically twice), and it is accepted only if the results match; otherwise, a fault is assumed to have occurred. To bypass this countermeasure, an injector would need to inject faults at the exact same location in the same inputs across both computation branches, which is already a difficult task. To further increase the difficulty of a successful injection, the inputs are encoded so that even identical faults in the two branches produce different effects on the outputs, making them easily detectable. Finally, the output generated from the

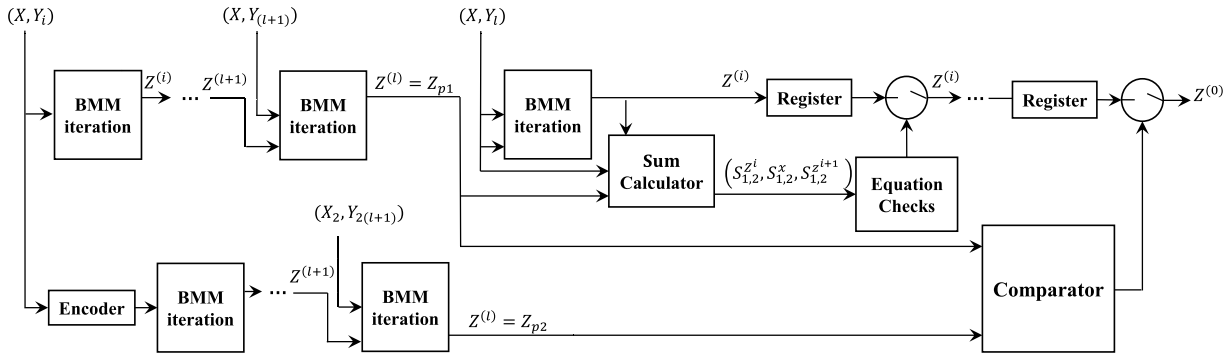

Fig. 2. Architecture of the proposed BMM module.

encoded inputs is decoded and compared with the output from the first computation stage.

In the case of the BMM algorithm, input encoding can be efficiently implemented by simply altering the order of operand execution. Specifically, in the main computation, the algorithm is executed on inputs X and Y to compute $Z_1 = X \times Y \bmod N$, whereas in the recomputation stage, the operation is performed as $Z_1 = Y \times X \bmod N$. Although operand reordering may appear trivial at first glance, it provides a robust mechanism for fault detection, particularly for faults injected into registers or inputs during the for loop. According to Algorithm 2, BMM fixes one operand and iteratively multiplies it with shifted portions of the other operand. For example, with $X \times Y$, the algorithm fixes X and computes $X \times Y_i$ in each iteration. By switching the operands to compute $Y \times X$, the algorithm instead fixes Y and computes $Y \times X_i$ at each step. Consequently, the values held in the intermediate registers differ significantly between the two executions, making fault injections that affect both computations in the same way highly unlikely.

To further enhance protection, we can mask the inputs by adding multiples of the modulus to them. Specifically, the new inputs are encoded as $X_2 = X + k_1 N$ and $Y_2 = Y + k_2 N$, where k_1 and k_2 are random values. This addition randomizes and masks both the original inputs and the intermediate values throughout the execution. Accordingly, while the main computation is performed on (X, Y), the recomputation is carried out on (Y_2, X_2), both masked and with swapped operands. This encoding strategy requires no decoding, since

$$XY \equiv (Y + k_2 N)(X + k_1 N) \bmod N.$$

Recomputation-based designs offer a high fault-detection rate; however, they introduce significant overhead to the system, as they essentially duplicate the entire computation process. For instance, if the two computation branches are executed sequentially, the delay overhead can exceed 100%, especially when accounting for additional steps such as input encoding and output decoding. Alternatively, implementing the two branches in parallel leads to a substantial increase in area consumption, which may not be acceptable in resource-constrained environments.

To mitigate this, partial recomputation is employed, offering a balance between efficiency and fault coverage by introducing only modest overhead while maintaining high detection capability against both natural and malicious fault-injection scenarios. In this approach, during the main computation phase, a partial result, an intermediate value naturally produced during the computation of the final output, is stored alongside the final result. As this value is already part of the normal execution flow, storing it incurs no additional computational cost. During the recomputation phase, rather than recomputing the entire output, the computation is carried out only up to the generation of the partial result. This recomputed value is then compared with the original partial result from the main computation. Any mismatch between the two indicates the occurrence of a fault.

Partial recomputation can provide high fault coverage with low overhead when combined with efficient algorithmic countermeasures (e.g., Countermeasure 1). In the case of the BMM algorithm, as described earlier, one operand is fixed entirely, and the other one is processed word by word in each iteration, starting from the most significant word. The intermediate result is updated accordingly until the algorithm reaches the least significant word. In this approach, during the main computation, we store the intermediate result $Z^{(i)}$ at iteration $d = l$ in a register, and proceed normally to compute the final result $Z^{(0)}$.

As discussed previously, to ensure that injected faults have different effects in the main and recomputation stages, input encoding must be applied. However, the input encoding methods discussed previously, including operand switching and summation with a multiple of the modulus, cannot be applied in partial recomputation. First, changing the order of operand execution, although it produces the same final output, it yields completely different intermediate results, making partial result comparison invalid. Second, using additive encoding such as $Y_2 = Y + kN$ also fails in this context. This is because the influence of adding a multiple of the modulus is progressively canceled during each iteration, and by the last iteration, it is entirely removed. However, during partial recomputation up to iteration l, these encoding effects still persist, resulting in a mismatch between the stored and recomputed partial results.

For the partial recomputation design, we propose encoding the inputs as $X_2 = X + k_1 N$ and $Y_2 = Y \ll k_2$. Note that since the entire operand X is used in each iteration of the BMM algorithm, adding a multiple of the modulus to X does not affect either the final result or any intermediate values of $Z^{(i)}$,

including the partial result. To prove that, first, we add the indices M and R to the computed values when the inputs are (X, Y) and $(X + kN, Y)$, respectively. Then we show that, in Algorithm 2, these two sets of inputs will result in a similar value for $Z^{(i)}$ in step 6.

By the floor function property $\lfloor x \rfloor = m$ if and only if $x-1 < m \leq x$. We can express

$$\mu = \left\lfloor \frac{2^{n+\alpha}}{N} \right\rfloor = \frac{2^{n+\alpha}}{N} - \sigma_1, \qquad \text{where } 0 \leq \sigma_1 < 1.$$

Multiplying both sides by $(1/2^{\alpha-\beta})$, we obtain

$$\frac{\mu}{2^{\alpha-\beta}} = \frac{2^{n+\beta}}{N} - \frac{\sigma_1}{2^{\alpha-\beta}}. \tag{1}$$

Similarly, for the term $\left\lfloor (Z_M^{(i)}/2^{n+\beta}) \right\rfloor$, we can write

$$\left\lfloor \frac{Z_M^{(i)}}{2^{n+\beta}} \right\rfloor = \frac{Z_M^{(i)}}{2^{n+\beta}} - \sigma_2, \qquad \text{with } 0 \leq \sigma_2 < 1. \tag{2}$$

Multiplying (1) and (2), expanding, and simplifying, we obtain

$$\left\lfloor \frac{Z_M^{(i)}}{2^{n+\beta}} \right\rfloor \left(\frac{\mu}{2^{\alpha-\beta}} \right) = \frac{Z_M^{(i)}}{N} - \sigma_1 \left(\frac{Z_M^{(i)}}{2^{n+\alpha}} \right) - \sigma_2 \left(\frac{2^{n+\beta}}{N} \right) + \frac{\sigma_1\sigma_2}{2^{\alpha-\beta}}.$$

We name $\sigma_3 = \sigma_2((2^{n+\beta}/N)) - (\sigma_1\sigma_2/2^{\alpha-\beta})$. Since $N \leq 2^n - 1$, β is a negative number (in our settings it is -20), and α is a positive number (104 in our settings), we have $0 \leq \sigma_3 < 1$. Therefore,

$$q_M = \left\lfloor \left\lfloor \frac{Z_M^{(i)}}{2^{n+\beta}} \right\rfloor \left(\frac{\mu}{2^{\alpha-\beta}} \right) \right\rfloor = \left\lfloor \frac{Z_M^{(i)}}{N} - \sigma_1 \left(\frac{Z_M^{(i)}}{2^{n+\alpha}} \right) - \sigma_3 \right\rfloor. \tag{3}$$

Thus, in step 6 we obtain $Z_M^{(i)} \leftarrow Z_M^{(i)} - q_M N$. Now we proceed with the recomputation where the inputs are $(X + kN, Y)$. This results in $Z_R^{(i)} = Z_R^{(i+1)} b + (X + kN)Y_i = Z_M^{(i)} + kNY_i$ in step 4. Therefore, we have

$$q_R = \left\lfloor \left(\frac{Z_M^{(i)} + kNY_i}{2^{n+\beta}} - \sigma_2 \right) \left(\frac{2^{n+\beta}}{N} - \frac{\sigma_1}{2^{\alpha-\beta}} \right) \right\rfloor.$$

By naming $\sigma_4 = \sigma_1((NY_i/2^{n+\alpha})) + \sigma_2((2^{n+\beta}/N)) - (\sigma_1\sigma_2/2^{\alpha-\beta})$ with similar logic we realize $0 \leq \sigma_4 < 1$. After expanding, simplifying, and given the fact that kY_i is an integer, we obtain

$$q_R = kY_i + \left\lfloor \frac{Z_M^{(i)}}{N} - \sigma_1 \left(\frac{Z_M^{(i)}}{2^{n+\alpha}} \right) - \sigma_4 \right\rfloor.$$

Hence, $q_R = q_M + kY_i$. Substituting it in step 6 leads to $Z_R^{(i)} \leftarrow Z_R^{(i)} - q_R N = Z_M^{(i)} + kNY_i - (q_M + kY_i)N$. After simplifying, $Z_R^{(i)} \leftarrow Z_M^{(i)} - q_M N$, which is equal to what we obtained in the main computation stage. As a result, inputs (X, Y) and $(X + kN, Y)$ yields same intermediate results for $Z^{(i)}$ in step 6 of Algorithm 2.

However, as previously discussed, encoding Y by adding a multiple of the modulus does not work, as the partial result only depends on a subset of Y's bits, which would be affected disproportionately. Instead, we encode Y by applying a left shift. This transformation modifies Y for outsiders while preserving its most significant words. Since only the l most significant words are responsible for computing the partial result, they remain unchanged, ensuring their comparability across the main and recomputation stages. Similar to the full recomputation design, this scheme does not require a decoding stage. If no faults are injected, the partial results from both computation paths will match.

Algorithm 4 Proposed BMM

Input: $X, Y \in [0, N)$, $2^{n-1} \leq N < 2^n$
Output: r and $flag$ ($flag = 1$ means valid output).

Precomputation
1: $\mu = \lfloor 2^{n+\alpha}/N \rfloor$, $d = \lceil n/w \rceil$, and $b = 2^w$
Main Body
2: $flag = 0$
3: $Z^{(d)} = 0$
4: $(S_1^x, S_2^x) = \mathsf{Sum}(X)$
5: **for** $i = d - 1$ **to** 0 **do**
6: $\quad Z^{(i)} \leftarrow Z^{(i+1)} b + XY_i$
7: $\quad$ **If** $i < l$
8: $\quad\quad (S_1^{Z^i}, S_2^{Z^i}) = \mathsf{Sum}(Z^i)$
9: $\quad\quad E_1 = (S_1^{Z^i} - S_1^{Z^{i+1}} - S_1^x Y_i) \bmod (2^w - 1)$
10: $\quad\quad E_2 = (S_2^{Z^i} - S_2^{Z^{i+1}} - S_2^x Y_i) \bmod (2^w - 2)$
11: $\quad q^{(i)} \leftarrow \left\lfloor \left\lfloor \frac{Z^{(i)}}{2^{n+\beta}} \right\rfloor \frac{\mu}{2^{\alpha-\beta}} \right\rfloor$
12: $\quad Z^{(i)} \leftarrow Z^{(i)} - q^{(i)} N$
13: $\quad$ **if** ($E_1 == 0$ and $E_2 == 0$)
14: $\quad\quad (S_1^{Z^{i+1}}, S_2^{Z^{i+1}}) = (S_1^{Z^i}, S_2^{Z^i})$
15: $\quad$ **if** $i == l$
16: $\quad\quad Z_{part1} \leftarrow Z^{(i)}$
17: **if** $Z^{(0)} \geqslant N$
18: $\quad Z^{(0)} \leftarrow Z^{(0)} - N$
Encoding
19: $X_2 = X + k_1 N$
20: $Y_2 = Y \ll k_2$
Recomputation
21: $Z_{part2}^{(d)} = 0$
22: **for** $i = d - 1$ **to** l **do**
23: $\quad Z_{part2}^{(i)} \leftarrow Z_{part2}^{(i+1)} b + X_2 Y_{2i}$
24: $\quad q^{(i)} \leftarrow \left\lfloor \left\lfloor \frac{Z_{part2}^{(i)}}{2^{n+\beta}} \right\rfloor \frac{\mu}{2^{\alpha-\beta}} \right\rfloor$
25: $\quad Z_{part2}^{(i)} \leftarrow Z_{part2}^{(i)} - q^{(i)} N$
Comparison
26: **if** $Z_{part2} == Z_{part1}$
27: $\quad flag \leftarrow 1$ (valid output)
28: **return** $Z^{(0)}, flag$

Algorithm 4 and Fig. 2 show the details of the proposed `BMM` algorithm. In both Algorithms 3 and 4, flag determines if the output is reliable (flag = 1) or needs to be discarded (flag ≠ 1).

IV. Error Coverage and Simulation Results

In this section, we first explain the theoretical detection ratio of our design against an intelligent fault injection. Then, we argue that the complexity of such targeted injections is high and nearly impractical. Instead, we simulate various practical fault models and fault types to evaluate the real-world fault-detection ratio of our design.

A. Detection Ratio Against Intelligent Injection

We divide this section into two parts, discussing the success rate of an intelligent injector against BMR and BMM independently.

1) Intelligent Injection at BMR: The idea of this injection is to exploit a weakness in $\mathsf{Sum}(X - r)$ by causing the same effect on both X and r simultaneously. In this attack, the injector attempts to modify both X and r in such a way that their effects cancel out within the $\mathsf{Sum}(\cdot)$ function, causing the two equation checks to fail to detect the fault. One possible method is for the adversary to add or subtract the same amount to corresponding words of both X and r. Adding or subtracting an identical value without knowing the values is hard to achieve. To do this, first, the injector must be capable of injecting faults at the exact same bits of the two variables. Then, he must make sure that the injection into both registers of X and r will result in the same effect.

To elaborate further, assume the injector has the capability to inject faults into any bits of any variable they choose (a challenging task). Now, suppose they inject n faults at the same word positions of X and r without knowing their actual values. In this case, the injector has no control over the precise effect their injection will have on the final value of the corrupted variables. For example, flipping the least significant bit (LSB) of a word may result in an increase or decrease of the value by 1, depending on the original state of the LSB. This uncertainty applies to both X and r. Therefore, for the effects of the faults to cancel out in both variables, the actual corresponding bit values at those n positions in X and r must be identical before the injection.

As a result, the more faults the adversary injects, the lower the probability of success, making the attack increasingly impractical. Moreover, it is well known that injecting a single-bit fault is nearly impossible, as nearby bits are typically affected during the fault-injection process. This is known as burst injection and will be explained in detail later.

In general, assume $X = x_n, \ldots, x_1 x_0$, where x_i refers to the ith bit of X. Then, assume we randomly select n bits and store their indices in a set named Fault_x. Similarly, we select n bits and store their indices in a set named Fault_r. Now, if the injection model is bit-flipping, the effect of the injection on the new values of X and r is given by $X_{\mathrm{faulty}} = X + \sum_{i \in \mathrm{Fault}_x} (-1)^{x_i} 2^i$ and $r_{\mathrm{faulty}} = r + \sum_{i \in \mathrm{Fault}_r} (-1)^{r_i} 2^i$.

Therefore, in order for the faults to go undetected, we must have $\mathsf{Sum}(X_{\mathrm{faulty}} - r_{\mathrm{faulty}}) = \mathsf{Sum}(X - r)$, or equivalently

$$\sum_{i \in \mathrm{Fault}_x} (-1)^{x_i} 2^i = \sum_{i \in \mathrm{Fault}_r} (-1)^{r_i} 2^i.$$

In this case, the best strategy for the injector is to select faulty bits from the same set for both Fault_x and Fault_r. But, even if the adversary is able to inject faults at exactly the same positions in both X and r, the actual values of X and r remain unknown to them. Hence, based on the original values and the injection model, such injections may or may not result in the same change in the values of these two parameters. If the injections result in identical changes, they cannot be detected by our design. We show that the probability of such undetectable cases decreases as the number of injected faults increases.

TABLE I
INTELLIGENT INJECTION AT THE SAME PLACE OF BOTH (X, r) PARAMETERS IN BMR ALGORITHM

Model	Fault Injection at (X, r)		
	Number	Detection rate	
	n	Ratio (%)	Formulation
Bit flipping	1	50	$1 - (\frac{1}{2})^n$
	2	75	
	3	87.5	
	5	96.87	
	7	99.21	
	10	99.9	
Stuck at 0 or 1	1	75	$1 - (\frac{1}{2})^n + (\frac{1}{2})^{2n}$
	2	81.25	
	3	89.06	
	5	96.97	
	7	99.22	
	10	99.9	

First, assume the injector aims to inject n faults into both X and r at the same bit positions, based on the bit-flipping model. If the actual values of the selected n bits in X and r are identical, flipping them will produce the same effect. As a result, in the computation of $(X - r)$, these faults will cancel each other out and remain undetected, even though the output becomes erroneous. The probability that the selected n bits of X and r are equal is $((1/2))^n$; therefore, the detection ratio is $1 - ((1/2))^n$.

On the other hand, for stuck-at-0 or stuck-at-1 models, there is a difference. In these models, there exists a scenario where the injection does not introduce an error in the output. For instance, if the selected n bits are all 1 and the fault model is stuck-at-1, the injection will have no effect on the output. The probability of such a case is $(1/2)^{2n}$. Excluding this case, the remaining behavior is similar to the bit-flipping model. Thus, the detection ratio of the proposed method in this model is $1 - ((1/2))^n + ((1/2))^{2n}$.

Table I shows the detection ratio of the proposed algorithm against this injection strategy. Based on the formulation, the detection ratio increases significantly as the number of injected faults increases.

2) Intelligent Injection at BMM: Apart from the previously mentioned injection strategy, an intelligent injector can also exploit the partial recomputation module in the proposed BMM algorithm. As discussed, the recomputation module primarily aims to detect faults in the inputs. However, since we employ partial recomputation, not all portions of the inputs are involved in the verification process.

More specifically, in Algorithm 1, the operand X is fixed and used entirely in each iteration. As a result, any fault injected into any part of X is detectable. On the other hand, operand Y is shifted word by word in each iteration. In partial recomputation, we halt at iteration l, meaning only the l most significant words of Y are used in the recomputation process. Therefore, if an injector manages to inject faults into the lower (least significant) words of Y during the main

computation phase before the algorithm starts, those faults will go undetected.

The detection capability of our design against this injection is directly influenced by the value of l. However, increasing l leads to higher delay and computational overhead. To address this tradeoff, the encoding mechanism can be modified to $Y_i + m_i$ to link each lower word of Y_i to the l most significant words that are used during the recomputation stage, through a parameter such as m_i. For instance, these m_is can be generated by adding several of the uninvolved lower words of the Y_is. However, this change would interfere with the decoding process. To address this, since the input in each iteration becomes $Y_i + m_i$, we subtract Xm_i from $Z^{(i)}$ in step 4 of Algorithm 2.

In addition, the most resource-intensive operations in the BMM algorithm are its large multiplications. Based on Algorithm 2, there are three main multiplications: the computation of XY_i in step 4, $\lfloor (Z^{(i)}/2^{n+\beta}) \rfloor \mu$ as part of computing q in step 5, and $q^{(i)}N$ in step 6. Therefore, excluding other steps, our method introduces one additional multiplication on top of these three. However, this additional multiplication is only added in the recomputation module, which runs for l iterations.

Now, in our architecture, in the previous encoding design, Algorithm 4 ran for $d = (n/w)$ iterations, each containing three large multiplications as the resource-dominating steps. As a result, the algorithm performs $3d$ large multiplications, in addition to other overheads such as summation units, comparisons, etc., which we denote as U. Conversely, the newly discussed encoding design requires four multiplications for the first l iterations and 3 for the remaining ones, plus the unchanged overhead U. Thus, the total overhead will be $4l+3(d-l)+U = 3d + l + U$. Excluding other insignificant components, the imposed overhead is approximately $(l/3d+U) = (lw/3n+Uw)$. For a chosen setting of $(n = 512, w = 64, l = 2)$, we observed an overhead of around 8.7%, which became less noticeable as n increased.

B. Simulation Methodologies, Details, and Definitions

As mentioned, the previously discussed malicious injections are often difficult to implement in practice. For instance, injecting a specific number of faults precisely at targeted register locations requires sophisticated tools and is generally considered impractical. Instead, for real-world fault-injection scenarios, several well-known types and models have been proposed and are commonly considered.

1) Fault-Injection Types and Models: In general, we consider two types of fault injections: Random and Burst. In the Random type, based on the number of bits injections, the faulty bits are selected randomly but uniformly among all bits of the target. In contrast, in the Burst type, after selecting the first faulty bit randomly and uniformly among all bits of the target, the subsequent consecutive bits will become faulty. After determining the faulty bits, fault models decide how these selected bits are impacted.

Accordingly, we consider three fault-injection models: Bit Flipping, stuck-at-1, and stuck-at-0. In the Bit Flipping model, the selected bits are flipped. In the stuck-at-1 and stuck-at-0 models, the faulty bits are forcibly set to 1 or 0, respectively, regardless of their original values. These behaviors can be implemented using bitwise operations: XOR with 1 for bit Flipping, OR with 1 for stuck-at-1, and AND with 0 for stuck-at-0.

TABLE II
SIMULATION SETTINGS AND THEIR DESCRIPTIONS

Parameters	Description	Settings
X	Algorithm input	2048 bit
Y	Algorithm input	2048 bit
N	Modulus	2048 bit
l	Recomputation index	5
(k_1, k_2)	Encoding indices	$(2, 4)$
n	Size of modulus in bits	2048
w	Word size of the processor	64
m_1	Number of faulty iterations	$[1, 3]$
n_1	Number of faulty bits	[1, 25]

2) Fault-Injection Placements and Targets: In addition, we consider two independent fault-injection placements for the BMM algorithm and one for the BMR algorithm. These placements determine where and how the faults are injected. For BMR algorithm, we assume that faults are injected during the execution of the algorithm. However, for BMM algorithm, since we employ recomputation, we also consider a case where faults are injected into the inputs before the algorithm begins. In this case, the injector's goal is to manipulate the inputs in such a way that the faults produce the same effect in both computation branches, thereby bypassing the countermeasures.

For each placement, we define different targets. These targets may be either the intermediate values stored in registers or the algorithm's inputs. Furthermore, we consider whether the faults are injected solely into the main computation or into both the main and recomputation stages.

3) Simulation Settings: The simulation settings along with notations are provided in Table II. For each fault-injection case, we simulated the scenario for 10 000 runs and reported the average detection rate. In each simulation, new random values for X, Y, and N are selected. Then, based on the chosen injection types and models, placement, targets, and number of injections, new faulty bits are selected randomly in each run.

Moreover, in the BMM algorithm, the parameters α and β control the range and size of the intermediate results. In all our simulations and implementations, both in our design and in the baseline implementation, we used the general settings $(\alpha, \beta) = (104, -20)$. This choice ensures the correctness of the encoding algorithm, since smaller values of α and β may cause variations in (3).

C. Simulation Results and Error Coverage Rate

For the BMR algorithm, we have simulated the following cases: random and Burst types, each including bit flipping, stuck-at-0, and stuck-at-1 models. We considered X, $\hat{q}$, r_1, r_2, and r as the targets of the injection, and varied the number of injected faults in the range $[1, 20]$. Our algorithm successfully detected all of these natural faults with 100% coverage.

For the BMM algorithm, the detection rates for random and Burst type are provided in Tables III and IV, respectively. These injection scenarios are performed based on the settings

TABLE III
DETECTION RATIO OF THE PROPOSED BMM ALGORITHM AGAINST RANDOM INJECTION TYPE

Model	Fault Injection Placements and Numbers										
	#		Fault targets on registers				#	Fault targets on registers on Inputs			
	m_1	n_1	$Z^{(i)}$	$q^{(i)}$	$(Z^{(i)}, Z_2^{(i)})$	$(q^{(i)}, q_2^{(i)})$	n_1	X	Y	(X, X_2)	(Y, Y_2)
Bit flipping	1	1	100	96.09	100	97.63	1	100	15.9	100	28.52
	1	3	100	96.29	100	97.19	3	100	39.52	100	64.11
	1	5	100	95.6	100	97.04	5	100	57.33	100	81.93
	2	6	100	94.88	100	99.01	6	100	64.01	100	86.64
	2	7	100	95	100	99.04	7	100	69.44	100	90.78
	2	8	100	94.49	100	98.97	8	100	73.97	100	92.98
	3	10	100	94.84	100	100	10	100	81.57	100	96.45
	3	15	100	94.69	100	100	15	100	91.89	100	99.40
	3	25	100	93.06	100	100	25	100	98.68	100	100
Stuck at 1	1	1	100	98.3	100	98.83	1	100	58.75	100	61.71
	1	3	100	97.37	100	97.86	3	100	33.89	100	47.70
	1	5	100	97.16	100	97.94	5	100	36.48	100	58.28
	2	6	100	99.12	100	99.86	6	100	40.12	100	63.16
	2	7	100	99.18	100	99.75	7	100	43.48	100	67.89
	2	8	100	99.27	100	99.75	8	100	47.98	100	72.36
	3	10	100	99.98	100	99.99	10	100	54.76	100	79.91
	3	15	100	99.96	100	100	15	100	70.70	100	91.34
	3	25	100	99.98	100	100	25	100	86.53	100	98.21
Stuck at 0	1	1	100	98.19	100	98.15	1	100	58.47	100	61.94
	1	3	100	96.14	100	97	3	100	34.06	100	48.17
	1	5	100	95.3	100	96.76	5	100	36.79	100	57.73
	2	6	100	90.39	100	96.51	6	100	40.74	100	62.3
	2	7	100	90.35	100	96.16	7	100	43.67	100	68.35
	2	8	100	90.09	100	96.3	8	100	48.83	100	73.47
	3	10	100	85.48	100	98.4	10	100	54.75	100	79.48
	3	15	100	84.4	100	98.9	15	100	70.01	100	90.83
	3	25	100	84.28	100	99.43	25	100	86.80	100	98.32

discussed in Section IV-B1. Fault coverage for Burst type when the injection is on Y is mentioned in Section IV-A2, thus it is not included here.

V. IMPLEMENTATION RESULTS

We implement fault-detection methods for BMR and BMM across two distinct platforms. The first is a software-based implementation executed on a Raspberry Pi 4, equipped with a 64-bit Quad-core Cortex-A72 processor and 8 GB of LPDDR4 RAM. Numerous side-channel attacks have been demonstrated against advanced RISC machines (ARMs) processors across various platforms, including smartphones and development boards. Raspberry Pi devices, in particular, have been widely used due to their accessibility and representative ARM core architectures. Successful attacks have been reported on the Raspberry Pi 2 [40] and 2B [41], targeting Cortex-A7 and Cortex-A53 cores, respectively. More recent work has extended these attacks to the Raspberry Pi 3 and 5 [42], as well as the Raspberry Pi 4 [43], demonstrating the practicality of side-channel analysis even on more advanced platforms. The second is a hardware-based implementation realized on AMD/Xilinx FPGAs, specifically the Artix UltraScale+ `xcau15p-ffvb676-1LV-i`and the Zynq-7000 `xc7z030fbg676-2`platforms.

A. Software Implementation Approach

The software implementation was developed in the C programming language to exploit its efficiency and fine-grained control over system resources. For operations involving large integers, we employed the GNU multiple-precision arithmetic library (GMP),[2] which is well-suited for high-performance arithmetic commonly required in cryptographic computations.

1) GMP Number Representation: In the GMP library, a multiprecision integer X is internally represented as an array of *limbs*, where each limb corresponds to a fixed-size chunk of bits. This representation can be expressed by the following radix decomposition:

$$X = \sum_{i=0}^{k-1} L_i \cdot B^i.$$

Here, L_i denotes the ith limb, B is the numerical base (referred to as the *limb base*), and k is the total number of limbs required to represent X. The limb base B is architecture-dependent: it is typically 2^{32} on 32-bit systems and 2^{64} on 64-bit systems. On our 64-bit target platform, we adopt $B = 2^{64}$, with each limb L_i represented as a 64-bit unsigned integer.

2) Efficient Remainder Computation: As outlined in Section III, our countermeasure implementation relies on the function Sum(.), which computes remainders with respect to two special moduli: $(2^w - 1)$ and $(2^w - 2)$. On our 64-bit target architecture, these correspond to $(2^{64} - 1)$ and $(2^{64} - 2)$, respectively. The internal limb-based representation of integers in the GMP library is particularly well-suited for such moduli,

[2]https://gmplib.org/

TABLE IV
DETECTION RATIO OF THE PROPOSED BMM ALGORITHM AGAINST BURST INJECTION TYPE

Model	Injection Placement					
	On registers (inside the loop)					
	m_1	n_1	$Z^{(i)}$	$q^{(i)}$	$(Z^{(i)}, Z_2^{(i)})$	$(q^{(i)}, q_2^{(i)})$
Bit flipping	1	2	100	96.19	100	97.22
	1	5	100	96.5	100	97.41
	1	8	100	96.16	100	97.38
	3	12	100	96.59	100	99.4
	3	17	100	96.67	100	99.52
	3	25	100	96.28	100	99.61
Stuck at 1	1	2	100	97.87	100	97.92
	1	5	100	97.46	100	98.18
	1	8	100	96.91	100	97.58
	3	12	100	99.94	100	99.99
	3	17	100	99.94	100	100
	3	25	100	99.95	100	100
Stuck at 0	1	2	100	96.48	100	97.73
	1	5	100	96.26	100	97.37
	1	8	100	95.82	100	96.95
	3	12	100	87.81	100	97.15
	3	17	100	87.36	100	97.66
	3	25	100	86.42	100	97.89

as it enables efficient remainder computation without resorting to expensive division-based modular operations

$$X \bmod \left(2^{64}-1\right) = \sum_{i=0}^{k-1} L_i \bmod \left(2^{64}-1\right)$$

$$X \bmod \left(2^{64}-2\right) = \sum_{i=0}^{k-1} (L_i \ll i) \bmod \left(2^{64}-2\right).$$

Since the accumulation of limb values can exceed 64 bits, we employed the compiler-specific 128-bit unsigned integer type `__uint128_t`(available in GCC[3]), which represents a 128-bit value as a pair of 64-bit components. This strategy allows for low-cost modulo operations tailored to our specific architectural constraints, supporting the overall efficiency of the proposed fault-detection scheme.

3) Evaluation Results of BMR: In the performance evaluation of our proposed BMR scheme, we selected input values X with bit-lengths $\{512, 1024, 2048, 4096\}$, and modulus N with bit-length $\{256, 512, 1024, 2048, 4096\}$. In all test configurations, we ensured that the operand size exceeds that of the modulus to accurately reflect real-world cryptographic scenarios, where intermediate values commonly exceed the modulus during modular arithmetic operations. The performance results for both the baseline implementation and our proposed BMR scheme are presented in Table V. Our results indicate that the proposed fault-detection method for BMR is highly efficient, incurring an average overhead of only 8% for 512–1024-bit moduli and 4% for 256 and 2048-bit moduli. Furthermore, although our evaluations are conducted on a 64-bit ARM Cortex-A72 processor, the algorithm supports flexibility in word size selection, allowing configuration as either 32 or 64 bits, which are shown in Table V.

[3]https://gcc.gnu.org/

TABLE V
COMPARISON OF TOTAL COMPUTATION CYCLES BETWEEN BASELINE AND PROPOSED BMR IMPLEMENTATIONS

Setting	Implementation on Raspberry Pi 4		
$(\lvert X\rvert, n, w^1)$	Baseline	Proposed	Overhead (%)
$(512, 256, 32)$	3147	3299	4.82
$(512, 256, 64)$	3083	3189	3.43
$(1024, 512, 32)$	3858	4134	7.15
$(1024, 512, 64)$	3784	4163	10.01
$(2048, 1024, 32)$	6977	7582	8.67
$(2048, 1024, 64)$	6859	7393	7.78
$(4096, 2048, 32)$	15006	15677	4.47
$(4096, 2048, 64)$	15089	15617	3.49

[1] All results are averaged over one million iterations to ensure robustness.

TABLE VI
COMPARISON OF TOTAL COMPUTATION CYCLES BETWEEN BASELINE AND PROPOSED BMM IMPLEMENTATIONS

Setting	Implementation on Raspberry Pi 4		
$((\lvert X\rvert \text{ and } \lvert Y\rvert), n)$	Baseline	Proposed	Overhead (%)
$(256, 256)$	11559	14619	26.47
$(512, 512)$	16358	21041	28.62
$(1024, 1024)$	29785	37814	26.95
$(2048, 2048)$	71442	90895	27.22
$(4096, 4096)$	195640	249401	27.47

[1] In all experimental configurations, we used fixed parameters: $k_1 = 4$, $l = 2$, $w = 64$, $\alpha = 104$, and $\beta = -20$.
[2] All results are averaged over one million iterations to ensure robustness.

4) Barret Modular Multiplication: For the performance evaluation of our proposed BMM scheme, we selected input values X and Y with bit-lengths $\{256, 512, 1024, 2048\}$, and modulus N with bit-lengths $\{256, 512, 1024, 2048, 4096\}$. These parameter choices reflect cryptographic scenarios, where operands are drawn from a finite field defined by the modulus and must undergo multiplication followed by modular reduction. We note that the inputs are not first multiplied in full and then reduced. The performance results for both the baseline implementation and our proposed BMM scheme are presented in Table VI. Our results indicate that the proposed fault-detection method for BMM introduces a nearly constant overhead across various configurations, with an average overhead of 27%.

B. Hardware Implementation Methodologies

For hardware implementation, we mounted the design on two FPGA families: AMD/Xilinx Artix UltraScale+ `xcau15p-ffvb676-1LV-i`and the Zynq-7000 `xc7z030fbg676-2`platforms as our benchmarking platforms. However, the choice of FPGA is not limited to these two platforms. We obtained similar percentage results on other families as well, though they are omitted here for brevity.

In addition, in the BMR algorithm, the input size is set to 2048 bits, with a modulus size of 1024 bits. For the BMM

TABLE VII
OUR HARDWARE IMPLEMENTATION RESULTS FOR BOTH BMR AND BMM ALGORITHMS ON TWO FPGA PLATFORMS

Algorithm		Barrett Modular Reduction (Period of 20 ns)						Barrett Modular Multiplication (Period of 25 ns)					
Family		Artix UltraScale			Zynq-7000			Artix UltraScale			Zynq-7000		
Platform		+ xcau15p-ffvb676-1LV-i			xc7z030fbg676-2			+ xcau15p-ffvb676-1LV-i			xc7z030fbg676-2		
Scheme		Baseline	Ours	(%)	Baseline	Ours	(%)	Baseline	Ours	(%)	Baseline	Ours	(%)
Area	LUT	15105	18801	24.46	14072	17922	27.35	11708	13720	17.18	11794	14382	21.94
	FF	10519	11646	10.71	10521	11634	10.57	8540	10522	23.2	8534	10533	23.42
	DSP	4	4	—	4	4	—	4	4	—	4	4	—
Clock Cycles		2219	2264	2.02	2219	2264	2.02	6340	7365	16.16	6340	7365	16.16
Max Freq (MHz)		56	55.7	-0.53	54.8	53.7	-2.04	43.6	43.1	-1.16	40.4	40.1	-0.74
Total Time (μs)		44.38	45.28	2.02	44.38	45.28	2.02	158.5	184.12	16.16	158.5	184.12	16.16
ADP [1] (μs)	$(*)$	269.62	337.28	25.09	256.53	333.7	30.08	268.11	318.02	18.61	291.66	357.96	22.73
	$(**)$	187.76	208.92	11.26	191.79	216.62	12.94	195.56	243.9	24.71	211.04	262.16	24.22
Power (W)		0.052	0.067	28.8	0.098	0.11	12.24	0.041	0.043	4.87	0.057	0.066	15.78
Energy [2] (μJ)		2.3	3.03	31.73	4.34	4.98	14.74	6.49	7.91	21.87	9.03	12.15	34.55

[1] $(*)$ and $(**)$ refer to $(LUT)\times$Total Delay and $(FF)\times$Total Delay, respectively.
[2] Energy $= Power\times$Total Time

algorithm, both the two inputs and modulus sizes are 1024 bits. A word size of 32 bits is used, and to minimize the number of utilized DSP blocks. Thus, each large multiplication is divided into 32-bit slices and performed using the Schoolbook method. To avoid exceeding the allowable number of I/O pins, a 64-bit input bus is employed to feed inputs to the algorithms.

Furthermore, Verilog was used as the hardware description language, and the AMD/Xilinx Vivado toolchain was utilized to assess the performance of the designs. To ensure a fair comparison between our design and the baseline, all shared parameters and sizes were kept identical. The reported overheads for both designs were obtained using the default synthesis and implementation settings in Vivado. In addition, the clock delay period was set to 20 ns for the BMR algorithm and 25 ns for the BMM algorithm.

Moreover, in the BMM algorithm, to optimize area utilization, the recomputation module is implemented sequentially. However, it can also be implemented in parallel with the main computation module, which would reduce delay at the cost of increased area. Table VII presents the hardware overhead of our proposed design compared with the baseline on the two aforementioned FPGA platforms.

According to this table, for the BMR algorithm, we observe an average overhead of 25.9% in the number of LUTs, 10.6% in the number of FFs, and 2.02% in total execution time across the two evaluated platforms. In the case of the BMM algorithm, the sequential implementation of the recomputation module contributed to higher overhead in execution time. On average, we observed an overhead of 19.5% in the number of LUTs, 23.3% in the number of FFs, and 16.1% in total execution time.

It is important to note that our designs do not utilize optimization techniques such as pipelining to boost the operating frequency. For instance, without pipelining, the works in [27], [44], and [45] reported frequencies of 46, 53, and 56 MHz, respectively, for the parameter setting ($|X| = 2^{13}, |N| = 360$), and 39, 46, and 50 MHz for ($|X| = 2^{15}, |N| = 1440$) in the BMR algorithm. However, by introducing four stages of pipelining, these works improved the frequency to 135, 176, and 180 MHz for ($|X| = 2^{13}, |N| = 360$), and to 128, 152, and 168 MHz for ($|X| = 2^{15}, |N| = 1440$).

Based on Table VII, on a similar platform, we achieved frequencies of 56 and 55.7 MHz for the protected and unprotected BMR schemes, respectively, under the parameter setting ($|X| = 2^{11}, |N| = 1024$). It is important to note that the purpose of our implementations was not to optimize the architectures, but rather to demonstrate a proof of concept for the imposed overhead of our design under fair and consistent conditions for both protected and unprotected designs. However, our countermeasures are applicable to other implementations of these algorithms, regardless of implementation techniques such as pipelining, and we expect to observe similar percentage overheads.

VI. CONCLUSION

The reliability of cryptographic implementations is essential, as even single-bit faults could cause severe outcomes (e.g., a single-bit change in the inputs of hash functions could cause their output to completely change). Whether caused by natural phenomena such as aging, radiation, or voltage fluctuations, or by deliberate fault-injection attacks through lasers or voltage glitching, these faults, can undermine correctness or lead to the exposure of sensitive information like secret keys.

Addressing this challenge, our work introduces efficient fault-detection mechanisms for the BMR and BMM algorithms, which are essential components in both classical and post-quantum cryptographic systems. We provide comprehensive theoretical analysis supported by extensive real-world simulations and practical implementations in both software and hardware. The results demonstrate that our techniques can reliably detect a wide range of fault-injection scenarios while maintaining low overhead in terms of area and execution time. These contributions offer a novel, practical, and scalable

solution to improving the fault tolerance of cryptographic systems, particularly in environments with limited resources.

REFERENCES

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

[2] T. Elgamal, "A public key cryptosystem and a signature scheme based on discrete logarithms," *IEEE Trans. Inf. Theory*, vol. IT-31, no. 4, pp. 469–472, Jul. 1985.

[3] G. M. Raimondo and L. E. Locascio, *FIPS 186-5, Digital Signature Standard (DSS)*, Gaithersburg, MD, USA: National Institute of Standards and Technology, 2023.

[4] D. J. Bernstein, N. Duif, T. Lange, P. Schwabe, and B.-Y. Yang, "High-speed high-security signatures," *J. Cryptograph. Eng.*, vol. 2, no. 2, pp. 77–89, Sep. 2012.

[5] C. P. Schnorr, "Efficient identification and signatures for smart cards," in *Proc. Conf. Theory Appl. Cryptol.*Cham, Switzerland: Springer, 2007, pp. 239–252.

[6] P. W. Shor, "Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer," *SIAM J. Comput.*, vol. 26, no. 5, pp. 1484–1509, Oct. 1997.

[7] S. Darzi, K. Ahmadi, S. Aghapour, A. A. Yavuz, and M. M. Kermani, "Envisioning the future of cyber security in post-quantum era: A survey on PQ standardization, applications, challenges and opportunities," 2023, *arXiv:2310.12037*.

[8] T. Dang et al., *Module-Lattice-Based Digital Signature Standard*. Gaithersburg, MD, USA: National Institute of Standards and Technology (NIST), Thinh Dang, Jacob, 2024.

[9] P.-A. Fouque et al., "FALCON: Fast-Fourier lattice-based compact signatures over NTRU," *NIST's Post-Quantum Cryptogr. Standardization Process*, vol. 36, no. 5, pp. 1–75, 2018.

[10] N. I. of Standards, T. (NIST), and D. Cooper. (Aug. 13, 2024).*Stateless Hash-Based Digital Signature Standard*. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.205.pdf

[11] G. M. Raimondo and L. E. Locascio, *FIPS 203 Federal Information Processing Standards Publication Module-Lattice-Based Key-Encapsulation Mechanism Standard*, vol. 47. Gaithersburg, MD, USA: National Institute of Standards and Technology (NIST), 2024.

[12] G. Alagic et al., *Status Report on the Fourth Round of the NIST Post-Quantum Cryptography Standardization Process*, Gaithersburg, MD, USA: National Institute of Standards and Technology (NIST), 2025.

[13] B. Li, Y. Yan, Y. Wei, and H. Han, "Scalable and parallel optimization of the number theoretic transform based on FPGA," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 32, no. 2, pp. 291–304, Feb. 2024.

[14] M. Knezevic, F. Vercauteren, and I. Verbauwhede, "Faster interleaved modular multiplication based on Barrett and Montgomery reduction methods," *IEEE Trans. Comput.*, vol. 59, no. 12, pp. 1715–1721, Dec. 2010.

[15] K. H. Kim, S. Mesnager, and K. I. Pak, "Montgomery curve arithmetic revisited," *J. Cryptograph. Eng.*, vol. 14, no. 2, pp. 343–362, Jun. 2024.

[16] J. Zhang, S.-M. Cho, C. Lee, and S.-H. Seo, "Optimized quantum folding Barrett reduction for quantum modular multipliers," *Sci. Rep.*, vol. 15, no. 1, p. 22808, Jul. 2025.

[17] B. Zhang, Z. Cheng, and M. Pedram, "Design of a high-performance iterative Barrett modular multiplier for crypto systems," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 32, no. 5, pp. 897–910, May 2024.

[18] X. Lou, T. Zhang, J. Jiang, and Y. Zhang, "A survey of microarchitectural side-channel vulnerabilities, attacks, and defenses in cryptography," *ACM Comput. Surv.*, vol. 54, no. 6, pp. 1–37, Jul. 2022.

[19] K. Ahmadi, S. Aghapour, M. M. Kermani, and R. Azarderakhsh, "Efficient error detection schemes for ECSM window method benchmarked on FPGAs," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 32, no. 3, pp. 592–596, Mar. 2024.

[20] S. Aghapour, K. Ahmadi, M. Mozaffari Kermani, and R. Azarderakhsh, "Efficient partial recomputation-based fault detection approaches for Z-transform," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 33, no. 7, pp. 1–11, Jul. 2025.

[21] A. Cintas-Canto, M. Mozaffari-Kermani, R. Azarderakhsh, and K. Gaj, "CRC-oriented error detection architectures of post-quantum cryptography niederreiter key generator on FPGA," in *Proc. IEEE Nordic Circuits Syst. Conf. (NorCAS)*, Oct. 2022, pp. 1–7.

[22] S. Saha, D. Jap, D. Basu Roy, A. Chakraborty, S. Bhasin, and D. Mukhopadhyay, "A framework to counter statistical ineffective fault analysis of block ciphers using domain transformation and error correction," *IEEE Trans. Inf. Forensics Security*, vol. 15, pp. 1905–1919, 2020.

[23] S. Aghapour, K. Ahmadi, M. M. Kermani, and R. Azarderakhsh, "Partial recomputation fault detection architecture for multiple-precision Montgomery modular multiplication," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, early access, Jul. 24, 2025, doi: 10.1109/TCAD.2025.3592590.

[24] K. Sedghighadikolaei and A. Altay Yavuz, "A comprehensive survey of threshold signatures: NIST standards, post-quantum cryptography, exotic techniques, and real-world applications," 2023, *arXiv:2311.05514*.

[25] A. Abdulrahman, V. Hwang, M. J. Kannwischer, and A. Sprenkels, "Faster Kyber and Dilithium on the Cortex-M4," in *Proc. Int. Conf. Appl. Cryptogr. Netw. Secur.*Rome, Italy: Springer, Jun. 2022, pp. 853–871.

[26] H. Becker, V. Hwang, M. J. Kannwischer, B.-Y. Yang, and S.-Y. Yang, "Neon NTT: Faster Dilithium, Kyber, and saber on cortex-A72 and apple M1," *IACR Trans. Cryptograph. Hardw. Embedded Syst.*, vol. 2022, no. 1, pp. 221–244, Nov. 2021.

[27] S.-W. Chiu and K. K. Parhi, "Low-latency preprocessing architecture for residue number system via flexible Barrett reduction for homomorphic encryption," *IEEE Trans. Circuits Syst. II, Exp. Briefs*, vol. 71, no. 5, pp. 2784–2788, May 2024.

[28] B. Lac, A. Canteaut, J. J. A. Fournier, and R. Sirdey, "Thwarting fault attacks against lightweight cryptography using SIMD instructions," in *Proc. IEEE Int. Symp. Circuits Syst. (ISCAS)*, May 2018, pp. 1–5.

[29] A. Aghaie, M. M. Kermani, and R. Azarderakhsh, "Fault diagnosis schemes for secure lightweight cryptographic block cipher RECTANGLE benchmarked on FPGA," in *Proc. IEEE Int. Conf. Electron., Circuits Syst. (ICECS)*, Dec. 2016, pp. 768–771.

[30] M. Ciet and M. Joye, "Practical fault countermeasures for Chinese remaindering based RSA," in *Proc. Workshop Fault Diagnosis Tolerance Cryptography (FDTC)*, vol. 5, 2005, pp. 124–132.

[31] T. Ç. Köylü, C. R. W. Reinbrecht, S. Hamdioui, and M. Taouil, "RNN-based detection of fault attacks on RSA," in *Proc. IEEE Int. Symp. Circuits Syst. (ISCAS)*, Oct. 2020, pp. 1–5.

[32] A. Dominguez-Oviedo and M. A. Hasan, "Algorithm-level error detection for ECSM," Centre Appl. Crypto. Res., Univ. Waterloo, ON, Canada, Tech. Rep. TR-2009-05, 2009.

[33] M. Mozaffari-Kermani and A. Reyhani-Masoleh, "Concurrent structure-independent fault detection schemes for the advanced encryption standard," *IEEE Trans. Comput.*, vol. 59, no. 5, pp. 608–622, May 2010.

[34] M. Bedoui, H. Mestiri, B. Bouallegue, and M. Machhout, "A reliable fault detection scheme for the AES hardware implementation," in *Proc. Int. Symp. Signal, Image, Video Commun. (ISIVC)*, Nov. 2016, pp. 47–52.

[35] N. Bindel, J. Krämer, and J. Schreiber, "Hampering fault attacks against lattice-based signature schemes: Countermeasures and their efficiency (special session)," in *Proc. 12th IEEE/ACM/IFIP Int. Conf. Hardw./Softw. Codesign Syst. Synth. Companion*, Oct. 2017, pp. 1–3.

[36] A. Sarker, A. C. Canto, M. Mozaffari Kermani, and R. Azarderakhsh, "Error detection architectures for hardware/software co-design approaches of number-theoretic transform," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 42, no. 7, pp. 2418–2422, Jul. 2023.

[37] J. Howe, A. Khalid, M. Martinoli, F. Regazzoni, and E. Oswald, "Fault attack countermeasures for error samplers in lattice-based cryptography," in *Proc. IEEE Int. Symp. Circuits Syst. (ISCAS)*, May 2019, pp. 1–5.

[38] M. Safieh, A. Furch, and F. De Santis, "An efficient Barrett reduction algorithm for Gaussian integer moduli," in *Proc. IEEE 30th Symp. Comput. Arithmetic (ARITH)*, Sep. 2023, pp. 76–83.

[39] H. Huang et al., "Review of modular multiplication algorithms over prime fields for public-key cryptosystems," *Cryptography*, vol. 9, no. 2, p. 46, Jun. 2025.

[40] I. Frieslaar and B. Irwin, "Recovering AES-128 encryption keys from a raspberry pi," in *Proc. Southern Afr. Telecommun. Netw. Appl. Conf. (SATNAC)*, 2017, pp. 228–235.

[41] D. Wang, Y. Gao, Y. Zhou, and X. Huang, "Revisiting a realistic EM side-channel attack on a complex modern SoC," *Cryptol. ePrint Arch.*, vol. 2024, no. 1322, Jan. 2024.

[42] Z. Liu, S. van Hoek, P. Horvath, D. Lauret, X. Xu, and L. Batina, "Real-world edge neural network implementations leak private interactions through physical side channel," 2025, *arXiv:2501.14512*.

[43] S. Bhasin, H. Boyapally, and D. Jap, "Reality check on side-channels: Lessons learnt from breaking aes on an ARM Cortex-A processor," *Cryptol. ePrint Arch.*, vol. 2024, p. 1381, Jan. 2024.

[44] W. Tan, S.-W. Chiu, A. Wang, Y. Lao, and K. K. Parhi, "PaReNTT: Low-latency parallel residue number system and NTT-based long polynomial modular multiplication for homomorphic encryption," *IEEE Trans. Inf. Forensics Security*, vol. 19, pp. 1646–1659, 2024.

[45] S. Sinha Roy, K. Järvinen, J. Vliegen, F. Vercauteren, and I. Verbauwhede, "HEPCloud: An FPGA-based multicore processor for FV somewhat homomorphic function evaluation," *IEEE Trans. Comput.*, vol. 67, no. 11, pp. 1637–1650, Nov. 2018.

Saeed Aghapour received the B.Sc. degree in electrical engineering from Babol Noshirvani University of Technology, Babol, Iran, in 2014, and the M.Sc. degree in communication cryptology from the Electrical Engineering Department, Sharif University of Technology, Tehran, Iran, in 2016. He is currently working toward the Ph.D. degree at the College of AI, Cybersecurity, and Computing, University of South Florida (USF), Tampa, FL, USA.

His current research interests include applied cryptography, post-quantum cryptography, hardware security, and fault detection.

Kiarash Sedghighadikolaei (Graduate Student Member, IEEE) received the B.Sc. degree in computer science and engineering from the University of Isfahan, Isfahan, Iran, in 2021. He is currently working toward the Ph.D. degree at the Bellini College of Artificial Intelligence, Cybersecurity, and Computing, University of South Florida (USF), Tampa, FL, USA.

His research focuses on post-quantum cryptography for the Internet of Things (IoT) and threshold cryptography. His research interests include applied cryptography, post-quantum cryptography, and secure multiparty computation.

Attila A. Yavuz (Senior Member, IEEE) received the M.S. degree in computer science from Bogazici University, Istanbul, Turkey, in 2006, and the Ph.D. degree in computer science from North Carolina State University, Raleigh, NC, USA, in 2011.

He is an Associate Professor at the Bellini College of Artificial Intelligence, Cybersecurity, and Computing, University of South Florida (USF); the Director of the Applied Cryptography Research Laboratory; and the Co-Director Center of Cryptologic Research, USF, Tampa, FL, USA. He was an Assistant Professor at the School of Electrical Engineering and Computer Science, Oregon State University, Corvallis, OR, USA, from 2014 to 2018, and at the Department of Computer Science and Engineering, USF, from 2018 to May 2025. He is broadly interested in the design, analysis, and application of cryptographic tools and protocols to enhance the security of computer systems. He has authored more than 110 products, including research articles in top conferences, journals, and patents. His work resulted in technology transfers (e.g., intravehicular network and searchable encryption), positively impacting tens of millions of users across the world.

Dr. Yavuz was a member of the security and privacy research group at the Robert Bosch Research and Technology Center North America from 2011 to 2014. He is a recipient of the NSF CAREER Award, Cisco Research Award (four times), unrestricted research gifts from Robert Bosch (five times), USF Faculty Outstanding Research Achievement Award, USF Excellence in Innovation Award, and USF College of Engineering Outstanding Research Achievement Award.

Bechir Hamdaoui received the M.S. degree in electrical and computer engineering, the M.S. degree in computer science, and the Ph.D. degree in electrical and computer engineering from the University of Wisconsin at Madison, WI, USA, in 2002, 2004, and 2005, respectively.

He is a Professor at the School of Electrical Engineering and Computer Science, Oregon State University, Corvallis, OR, USA. He is the Founding Director of the NetSTAR Laboratory, Oregon State University. His general interests include theoretical and experimental research that enhances the cybersecurity and resiliency of future intelligent networked systems.

Dr. Hamdaoui and his team have won several awards, including the ACM WiSec 2024 Runner-Up Paper Award, the ISSIP 2020 Distinguished Recognition Award, the ICC 2017 Best Paper Award, the IWCMC 2017 Best Paper Award, the 2016 EECS Outstanding Research Award, the COMMANTEL 2014 Best Paper Award, and the 2009 NSF CAREER Award. He serves/served as an associate editor for several IEEE journals and magazines and chaired and organized many IEEE/ACM conference symposia and workshop programs. He served as a Distinguished Lecturer for the IEEE Communications Society in 2016 and 2017 and served as the Chair and Co-Chair of the IEEE Communications Society's Wireless Technical Committee (WTC) from January 2019 until December 2022.

Mehran Mozaffari-Kermani (Senior Member, IEEE) received the B.Sc. degree from the University of Tehran, Tehran, Iran, in 2005, and the M.E.Sc. and Ph.D. degrees from the University of Western Ontario, London, ON, Canada, in 2007 and 2011, respectively.

From 2013 to 2017, he was a Faculty Member with Rochester Institute of Technology, Rochester, NY, USA, and starting 2017, he is an Associate Professor with the College of AI, Cybersecurity, and Computing, University of South Florida (USF), Tampa, FL, USA.

Dr. Mozaffari-Kermani has served as the technical committee member for a number of related conferences on embedded systems security and reliability, including HOST (Publications Chair), CCS (Publications Chair), DAC, DATE, INFOCOM, FDTC, GLSVLSI, DFT, ISCAS, ISQED, RFIDSec, LightSec, and WAIFI. He was a recipient of Texas Instruments Faculty Award (Douglas Harvey) in 2014, the Outstanding Research Award at the College of Engineering, USF, in 2018, the Nexus Initiative Global Award, in 2019, and the USF University-Wide Faculty Outstanding Research Achievement Award in 2021. He has served and is serving as an Associate Editor for IEEE TRANSACTIONS ON VERY LARGE SCALE INTEGRATION (TVLSI) from 2016 to 2025, IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMS—I: REGULAR PAPERS (TCAS I) from 2016 to 2023, the ACM TECS since 2015, IEEE TRANSACTIONS ON INDUSTRIAL INFORMATICS (TII) since 2025, and IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMS—II: EXPRESS BRIEFS (TCAS II) (2026 onward).