Partial Recomputation Fault Detection Architecture for Multiple-Precision Montgomery Modular Multiplication

Saeed Aghapour, Kasra Ahmadi, *Graduate Student Member, IEEE*, Mehran Mozaffari Kermani, *Senior Member, IEEE*, and Reza Azarderakhsh, *Member, IEEE*

***Abstract*—Detection of soft errors and faults are one of the most critical factors in ensuring the reliability of algorithm implementations. Multiplication, as a fundamental and computationally intensive operation, is particularly vulnerable to such errors. Given its widespread use in cryptography and coding applications, detecting these errors is crucial. For example, in hash functions, even a single-bit change in the input can completely alter the output ([ideally, each bit of the output changes with a probability of (1/2)]). Montgomery multiplication as an efficient multiplication method is an integral part of numerous cryptographic applications expanding both classical and post quantum cryptography. For that reason, this brief introduces a fault detection method for the multiple-precision Montgomery modular multiplication algorithm based on partial recomputation. Through extensive simulations and implementations, we demonstrate that our approach efficiently detects both permanent and transient errors with a high-success rate, while imposing modest area and time overhead on the system.**

***Index Terms*—Cryptography, fault detection, FPGA, Montgomery modular multiplication.**

I. Introduction

In addition to the theoretical and mathematical analysis of algorithms, implementations without care can lead to the exposure of sensitive information through side-channel attacks. One particular type of side-channel attacks is called fault analysis, introduced in [1]. In these attacks, adversaries inject intentional faults into the algorithm to uncover secret values within the system. Furthermore, errors can occur in any system due to factors, such as aging or environmental changes. These errors, if not addressed, can degrade the scheme's reliability, leading to incorrect outputs, even if they do not expose sensitive information. The study in [2] provides a comprehensive overview of various easy to implement fault injection methods.

To mitigate the potential threat of fault attacks, various fault detection schemes have been developed. Fault detection plays a critical role in cryptosystems, and over the years, numerous methods have been proposed for detecting faults in cryptographic algorithms. For instance, the work in [3] introduced highly effective fault detection schemes for the AES, while the work in [4] proposed a method for RSA. A novel recomputation-based fault detection scheme for elliptic curve scalar multiplication (ECSM) module is detailed in [5]. Additionally, [6] presented efficient methods for detecting transient and permanent errors in ECSM. Moreover, developing a general approach to identify vulnerabilities in ciphers through deep learning is explored in [7].

Received 31 October 2024; revised 14 May 2025; accepted 17 July 2025. Date of publication 24 July 2025; date of current version 22 January 2026. This work was supported by the U.S. National Science Foundation under Award SaTC-2350213. This article was recommended by Associate Editor J. L. Dworak. *(Corresponding author: Mehran Mozaffari Kermani.)*

Saeed Aghapour, Kasra Ahmadi, and Mehran Mozaffari Kermani are with the Department of Computer Science and Engineering, University of South Florida, Tampa, FL 33620 USA (e-mail: aghapour@usf.edu; ahmadi1@usf.edu; mehran2@usf.edu).

Reza Azarderakhsh is with the Department of Computer and Electrical Engineering and Computer Science, Florida Atlantic University, Boca Raton, FL 33431 USA (e-mail: razarderakhsh@fau.edu).

Digital Object Identifier 10.1109/TCAD.2025.3592590

Montgomery modular multiplication, introduced in [8], eliminates the need for the costly division operation by converting numbers into Montgomery form. Due to its compatibility with both hardware and software, this algorithm is widely used in various cryptographic applications. There has been a number of research works to optimize this algorithm for specific applications [9], [10], [11]. Additionally, various fault detection schemes have been proposed to ensure the reliability of Montgomery modular multiplication. The work in [12] introduces a parity-based fault detection scheme for Montgomery modular multiplication in $GF(2^m)$, offering high error coverage. However, in some cases, the area and delay overheads of their design exceed 25% and 65%, respectively. Furthermore, the studies in [13] and [14] focus on fault detection specifically for Montgomery-based ECSM operations.

Moreover, [15] presents a redundancy-based fault detection method with minimal area overhead, applicable to Montgomery-based RSA and ECC. Although this method adds only a small area overhead, it is designed for single fault detection and to detect multiple faults, the overhead increases proportionally. While the work in [16] proposed a highly efficient error detection method for systolic Montgomery multiplication over $GF(2^m)$, their design is limited to pipelined implementations of Montgomery multiplication. Moreover, the work in [17] introduced a fault detection algorithm for bit-serial Montgomery multiplication based on hardware duplication. However, they did not specify the types of errors their design can detect, and they only reported the overhead theoretically. Moreover, recomputation-based fault detection schemes are considered as one of the most important types of fault detection schemes [18], [19], [20], if not carefully designed, can introduce significant overhead and exhibit poor fault detection rates under certain fault models.

In this brief, we present a fault detection scheme based on partial recomputation for Montgomery modular multiplication. Our simulations and FPGA implementations show that the proposed scheme provides high error coverage against various attack models, with a modest area overhead of 25%.

II. Preliminaries

One of the most efficient methods for performing modular multiplication on large numbers is multiple-precision Montgomery multiplication. In this approach, the inputs are divided into smaller words that the processor can handle efficiently depending on the architecture. The algorithm first converts the inputs into Montgomery form, and through precomputation, it replaces the resource-intensive division operation with simple shifting. This enables the efficient computation of $uvb^{-n} \bmod N$ without any division. This modification also results in a scalable design that is highly suitable for cryptographic applications. Algorithm 1 presents the multiple-precision Montgomery multiplication method.

III. Proposed Fault Detection Architecture

In this section, we propose our fault detection method, which includes three countermeasures. The core countermeasure is based on

Algorithm 1 Multiple-Precision Montgomery multiplication

```
Input: $u = (u_{n-1}...u_0)_b$, $v = (v_{n-1}...v_0)_b$, $N$, $w$, and $b = 2^w$
Output: $Result = (u \times v \times b^{-n}) \bmod N$
————————— Precomputation —————————
1: $N' = -N^{-1} \bmod b$
————————— Main Body —————————
2: $t \leftarrow 0$
3: for $i = 0$ to $n-1$ do
4:     $m_i \leftarrow ((t_0 + u_i v_0)N') \bmod b$
5:     $t \leftarrow (t + u_i v + m_i N)/b$
6: end for
7: While $t \geqslant N$
8:     $t \leftarrow t - N$
9: $Result \leftarrow t$
10: return $Result$
```

recomputation, but to further increase the efficiency of our design, we add two extra countermeasures.

A. Countermeasure 1

The first countermeasure is the recomputation module where the output is computed twice and the results are compared. If no error or fault has occurred, the results should match. However, to detect permanent faults, the recomputation must be performed on encoded inputs; otherwise, such faults may go undetected regardless of number of execution. Additionally, both the encoding and decoding methods must be lightweight to avoid increasing the overhead of the design.

1) Total Recomputation: Based on Algorithm 1, the inputs to the Montgomery multiplication are u and v, and the output is $\text{Result}_1 = uvb^{-n} \bmod N$. Encoding is only required for the recomputation phase. A simple yet effective encoding scheme is to use $u_{2e} = u + k_u N$ and $v_{2e} = v + k_v N$ where k_u and k_v are small random numbers. Moreover, since the algorithm fixes one operand and processes the other word by word, altering the order in which the operands are executed introduces a clear distinction between the intermediate results. This makes it significantly harder for an attacker to achieve the same fault effect during both computation stages.

Moreover, because $(v + k_v N)(u + k_u N)b^{-n} \equiv uvb^{-N} \bmod N$, the two results should match in the absence of faults. However, while full recomputation schemes offer high fault coverage, they also double the computational cost. Hence, a more efficient and lightweight approach is desirable.

2) Partial Recomputation: To address the limitations of total recomputation, we employ partial recomputation. In this approach, during the main computation, we store the intermediate result after l iterations. Then, during recomputation, we run the algorithm only up to that point and compare the partial results. More specifically, since the for-loop must iterate n times, we first perform the main computation on u and v to obtain $\text{Result} = uvb^{-n} \bmod N$. During this computation, we store the intermediate result after l iterations as Result_{p1}. For the recomputation phase, after encoding the inputs, the algorithm runs on them but only up to iteration $i = l$ and stores the partial result as Result_{p2}.

However, the previous encoding scheme does not work in this case. Because adding a multiple of the modulus to the inputs does not affect the final output, but it does affect the intermediate values. This is because the impacts of these additional terms is gradually canceled out over the course of the algorithm in each iteration, and fully eliminated by the final iteration. On the other hand, since we are comparing intermediate results in partial recomputation, these effects are still present and lead to incorrect comparisons.

Algorithm 2 Proposed Fault Detection Method

```
Input: $u = (u_{n-1}...u_0)_b$, $v = (v_{n-1}...v_0)_b$, $N$, $w$, and $b = 2^w$
Output: $Result = (u \times v \times b^{-n}) \bmod N$ and $Flag$
————————— Precomputation —————————
1: $N' = -N^{-1} \bmod b$
——————- Main Computation on $u$ and $v$ ——————
2: $Sum_1 \leftarrow 0$
3: $t \leftarrow 0$
4: $t_{temp} \leftarrow 0$
5: for $i = 0$ to $n-1$ do
6:     $m_i \leftarrow ((t_0 + u_i v_0)N') \bmod b$
7:     $t \leftarrow (t + u_i v + m_i N)/b$
8:     $Sum_1 = Sum_1 + u_i$
9:     If $((t \times b) \neq (t_{temp} + u_i v + m_i N))$
10:            $Flag \leftarrow 0$
11:            Break
12:     $t_{temp} \leftarrow t$
13:     If $(i == l)$
14:            $Result_{p1} \leftarrow t$
15: end for
16: While $t \geqslant N$
17:     $t \leftarrow t - N$
18: $Result \leftarrow t$
———— Recomputation on $u$ and $v \leftarrow v << l_v$ ————
19: $Sum_2 \leftarrow 0$
20: $t \leftarrow 0$
21: for $i = 0$ to $l$ do
22:     $m_i \leftarrow ((t_0 + u_i v_0)N') \bmod b$
23:     $t \leftarrow (t + u_i v + m_i N)/b$
24:     $t \leftarrow t - ((u_i v) - (u_i v >> l_v))/b$
25:     $Sum_2 = Sum_2 + u_i$
26: end for
27: $Result_{p2} \leftarrow t$
28: for $i = l + 1$ to $n - 1$ do
29:     $Sum_2 = Sum_2 + u_i$
30: end for
——————————-- Comparison ——————————
31: If $(Sum_1 == Sum_2$ and $Result_{p1} == Result_{p2})$
32:     $Flag \leftarrow 1$
33:     return $Result$
```

Thus, we use a different encoding method and encode $v_{2e} = v << l_v$, i.e., a left-shifted version of v. But, based on Algorithm 1, aside from v_0 (the first word of v), the entire input v is multiplied by u_i in step 5 of each iteration. Therefore, to make a valid comparison of the results, the effect of the encoded input must be decoded before comparing them. This is achieved efficiently by subtracting $((u_i v) - (u_i v >> l_v))/b$ in line 24 of Algorithm 2.

The effectiveness of this module in detecting errors is directly influenced by the value of l. A larger l improves the detection rate but also increases system delay. Therefore, additional countermeasures are needed to maintain high detection rates even when l is small. In our implementation and simulation, we set $l_v = 2$.

B. Countermeasure 2

As mentioned earlier, the Montgomery multiplication algorithm uses one operand entirely in each iteration, while the other operand is processed word by word (in Algorithm 1, operand v is fixed and u is shifted). Therefore, faults injected into the fixed operand can be effectively detected by the recomputation module. However, detecting faults in the second operand requires additional countermeasures. The

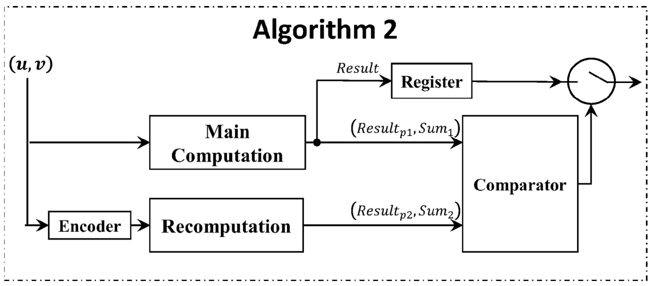

Fig. 1. Overall schematic of the proposed approach.

TABLE I
IMPLEMENTATION RESULT ON AMD/XILINX ARTIX ULTRASCALE+ XCAU15P-FFVB676-1LV-I

		1024 bits input		2048 bits input	
		Baseline	Ours	Baseline	Ours
Area	LUT	11963	14966	24269	30010
	FF	8918	11247	17177	21623
	DSP	16	16	16	16
Timing	Clock Cycles	755	896	2531	2784
	Total Time (μs)	13.731	17.211	57.696	63.701
Power @ 40 MHz (W)		0.036	0.036	0.072	0.072
Energy (μJ)		0.494	0.619	4.154	4.586

second countermeasure we introduce is a comparator for the sum of u. During the algorithm, u is divided into small word sizes u_i, and we incorporate an adder to compute the sum of these chunks. Similarly, we implement this method for the recomputation stage. However, since recomputation stops at $i = l$, we must also compute the sum of u from l to n.

C. Countermeasure 3

The previous countermeasures are mostly effective in detecting faults injected into the inputs. To enhance the detection of faults injected into intermediate values, we introduce an additional register, t_{temp}, and incorporate a comparator to verify whether the relation $t \times b = t_{\text{temp}} + u_i v + m_i N$ holds at each iteration. This check links the current computed value to the results of the previous iteration, enabling the detection of injected faults that affect the intermediate computations. Additionally, since $u_i v$, $m_i N$, and t are already computed and $t \times b$ can be done by a simple shift to the left, this countermeasure can be implemented efficiently. Moreover, if no faults are injected into any intermediate results, this check is always correct, as t_{temp} is simply the t value from the previous iteration.

Fig. 1 and Algorithm 2 depict the overall schematic and the details of our method. Note that since there is no dependency between the main computation and recomputation modules, the design could be implemented parallel or sequential.

IV. Implementation Results

We implemented our design on AMD/Xilinx Artix UltraScale+ FPGA to evaluate its performance. As described earlier, our design can be implemented parallel or sequential. We chose a sequential approach to optimize area efficiency. The additional circuitry in our design includes comparators, additional registers, a *Sum* adder, and a simple shift encoder/decoder. We conducted evaluations with input sizes of 1024 and 2048 bits, both using a word size of 64 bits and $l = 2$. Moreover, Verilog and Vivado tool were used as the programming language, and the analysis tool to assess area, delay, and power metrics. All implementations were carried out by default "Vivado implementation" settings, and to ensure timing constraints were met, the clock was set to 40 MHz. Additionally, although we used the Artix UltraScale+ FPGA as our benchmarking platform, the design is not platform-dependent. Implementation results are detailed in Table I.

V. Error Coverage and Simulation Results

A. Methodology

For simulations we implemented our design in *C* using the GMP library [21] optimized for large numbers arithmetic. We employed 2048-bit number N and 2000-bit u and v, with a word size of 64 and run each simulation 10 000 times. In this study, we assumed that the comparators are fault-free, and faults are injected only on the inputs and intermediate values.

B. Error Coverage

To comprehensively analyze the effectiveness of our design, we have considered two fault types (random and burst), three injection models (bit flip and bit stuck at 1 or 0), two simulation scenarios, various fault targets, and two kinds of faults (transient and permanent). In random type, all of the injected bits (or iterations) are chosen randomly while in the burst type, after choosing a random bit (iteration) the next consecutive bits (iterations) are targeted for injection. For the model of fault injection, bit flipping means "XOR"ing the current value with 1 which will flip the value. Stuck at 1 (0) means "OR"ing ("AND"ing) the target bit with 1 (0).

Moreover, faults are injected based on two independent scenarios: either actively during the execution of the algorithm (Scenario 1) or into the inputs before the algorithm starts (Scenario 2). For instance, in Scenario 1, Algorithm 2 runs for $(|N|/|\text{word} - \text{size}|)$ iterations. In Table II, n_1 specifies how many of these iterations could become faulty. After selecting faulty iterations and a specific target, n_2 indicates how many bits of the target should change based on the chosen fault type and model. Additionally, in Scenario 2, we assumed that faults are injected into the inputs before the algorithm begins. In this case, both the main and recomputation modules operate on faulty inputs, but there is no active fault injection during the execution of the algorithm. In this Scenario, n_3 and n_4 determine the number of faulty bits for each target in the main and recomputation stages, respectively.

Furthermore, we assumed that faults could be either transient or permanent. In transient faults, the applied injections in the main computation are removed in the recomputation stage, while with permanent faults, the faulty bits remain for the recomputation. This distinction is crucial, since an adversary can inject the same fault into both branches, hoping to produce the same erroneous output. Table II presents our simulation results, and together with Table I, confirms that our design can detect various types of faults with approximately 25% area overhead and 10% time overhead.

C. Comparison and Discussion

As mentioned, due to its critical role and widespread use in both classical and post-quantum cryptography, Montgomery multiplication has been the focus of numerous improvements across different aspects. For example, the work in [22] explores fully systolic structures using DSP48E2 slices on Ultrascale FPGAs, leading to optimized and parallelized computations. In [23], it was demonstrated that replacing the three multiplications within each iteration with three compressions and one addition can significantly reduce latency for radices greater than 2. Additionally, Grale and Swartzlander [24]

TABLE II
OUR SIMULATION RESULTS FOR 2048-BIT INPUTS, INCLUDING DIFFERENT FAULT TYPES, MODELS, AND SCENARIOS

Type	Model	Scenario 1					Scenario 2							
		Transient faults					Transient faults						Permanent fault	
		# faults		Target			# faults		Target					
		n_1	n_2	u_i	m_i	t	n_3	n_4	u_1	v_1	u_2[1]	v_2[1]	(u_1, u_2)	(v_1, v_2)
Random	Flipping	1	1	100%	100%	100%	1	1	100%	100%	100%	99.9%	100%	100%
		1	2	100%	100%	100%	3	3	100%	100%	99.9%	100%	99.9%	100%
		2	1	99.3%	100%	100%	4	4	99.9%	100%	99.9%	100%	99.9%	100%
		2	2	99.9%	100%	100%	11	11	100%	100%	100%	100%	100%	100%
	Stuck at 1	1	1	100%	100%	100%	1	1	100%	100%	99.9%	99.9%	100%	100%
		1	2	100%	100%	100%	3	3	100%	100%	99.9%	99.9%	99.8%	99.9%
		2	1	100%	100%	100%	4	4	100%	100%	99.9%	99.9%	99.9%	99.9%
		2	2	100%	100%	100%	11	11	100%	100%	100%	100%	100%	100%
	Stuck at 0	1	1	100%	100%	100%	1	1	100%	100%	99.9%	99.9%	100%	100%
		1	2	100%	100%	100%	3	3	100%	100%	100%	99.9%	99.8%	99.9%
		2	1	100%	100%	100%	4	4	100%	100%	99.9%	100%	99.9%	100%
		2	2	100%	100%	100%	11	11	100%	100%	100%	100%	100%	100%
Burst	Flipping	1	2	100%	100%	100%	2	2	100%	100%	100%	100%	100%	100%
		1	3	100%	100%	100%	4	4	100%	100%	100%	100%	100%	100%
		2	2	99.5%	100%	100%	5	5	100%	100%	100%	100%	100%	100%
		2	3	99.8%	100%	100%	11	11	100%	100%	100%	100%	100%	100%
	Stuck at 1	1	2	100%	100%	100%	2	2	100%	100%	99.9%	99.9%	100%	100%
		1	3	100%	100%	100%	4	4	100%	100%	99.9%	99.9%	95.4%	95.2%
		2	2	100%	100%	100%	5	5	100%	100%	99.9%	99.9%	94.8%	94.3%
		2	3	100%	100%	100%	11	11	100%	100%	100%	100%	94.2%	95.1%
	Stuck at 0	1	2	100%	100%	100%	2	2	100%	100%	99.9%	99.9%	100%	100%
		1	3	100%	100%	100%	4	4	100%	100%	99.9%	99.9%	95.4%	95%
		2	2	100%	100%	100%	5	5	100%	100%	100%	99.9%	95.1%	95.2%
		2	3	100%	100%	100%	11	11	100%	100%	100%	100%	94%	94%

[1] In these cases, the result will not be erroneous, yet our design detects the injected faults.

TABLE III
COMPARISON TABLE

	Approach	Implementation	Fault Types Considered	Additional Comments
[12]	Parity check	Sequential/Parallel	Single and multiple fault	Only stuck at faults High delay in some cases
[15]	Redundancy	Parallel	Only single faults	Multi-faults detection increases area
[16]	Time redundancy	Parallel	Only single faults	Works only on pipeline multipliers Detects only single fault
[17]	Hardware duplication	Sequential	N/A	High area overhead
[20]	Total Recomputation	Parallel	N/A	High area overhead
Ours	Partial Recomputation	Sequential	Single and multiple faults Burst, random, stuck, and bit flip	Considered various fault targets Has parallelization potential

proposed a modified version of the Montgomery multiplication algorithm by introducing a predefined level of parallelism. This modification enables a new tradeoff between area and latency. Our fault detection architecture is compatible with all these implementations; however, for generality, we applied it to the baseline design and reported the overhead.

Table III compares our work with related work qualitative. Due to having different nature and applications, they cannot be compared quantitatively. For example, some related works reported the overhead of their designs only theoretically, while others reported the total overhead when their method was applied within a larger design (such as ECC or RSA) or a specialized implementation (like RNS or pipelined). Moreover, to the best of our knowledge, our work is the only one that considers various parameters as targets for fault injection. Furthermore, related works either assessed fault detection rates theoretically or focused exclusively on stuck-at faults, while our work not only considered bit-flipping and stuck at faults, but also conducted various simulations to show its practicality.

Additionally, to assess area and time overheads, our design was evaluated through RTL simulation and post-implementation analysis using the AMD/Xilinx Vivado toolchain. For evaluating the detection rate against fault injections, we used *C* to simulate the injections. While these results provide valuable insights into performance and resource utilization, deploying the design on actual FPGA hardware presents its own challenges. Although the overhead is expected to be close to the reported values, and the detection rates similar to those observed in simulations, validating the system's robustness against real-world factors—such as timing variations, environmental noise, and physical fault injections—remains a key direction for future work.

VI. CONCLUSION

Reliability of an implementation is one of the most crucial aspects of sensitive algorithms, especially in cryptographic applications where even single or few bit changes could cause devastating outcomes. In this brief, by focusing on Montgomery modular multiplication algorithm which is widely used in many cryptosystems, we proposed a lightweight and effective fault detection approach benchamrked on AMD/Xilinx FPGAs. Our simulations and implementations indicate that our proposed scheme could detect various errors with high coverage while imposing a modest area and time overhead which shows its practicality on resource-constrained devices.

REFERENCES

[1] D. Boneh, R. DeMillo, and R. Lipton, "On the importance of eliminating errors in cryptographic computations," *J. Cryptol.*, vol. 14, pp. 101–119, 2001.

[2] A. Barenghi, L. Breveglieri, I. Koren, and D. Naccache, “Fault injection attacks on cryptographic devices: Theory, practice, and countermeasures,” *Proc. IEEE*, vol. 100, no. 11, pp. 3056–3076, Nov. 2012.

[3] M. M. 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.

[4] M. Ciet and M. Joye, “Practical fault countermeasures for Chinese remaindering based RSA (extended abstract),” in *Proc. Workshop Fault Detect. Tolerance Cryptogr.*, 2005, pp. 124–131.

[5] A. Dominguez-Oviedo and M. Hasan, “Error detection and fault tolerance in ECSM using input randomization,” *IEEE Trans. Depend. Secure Comput.*, vol. 6, no. 3, pp. 175–187, Jul.–Sep. 2009.

[6] 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.

[7] S. Saha, M. Alam, A. Bag, D. Mukhopadhyay, and P. Dasgupta, “Learn from your faults: Leakage assessment in fault attacks using deep learning,” *J. Cryptol.*, vol. 36, no. 3, p. 19, 2023.

[8] P. Montgomery, “Modular multiplication without trial division,” *Math. Comput.*, vol. 44, no. 170, pp. 519–521, 1985.

[9] R. Wu et al., “Efficient high-radix GF(p) montgomery modular multiplication via deep use of multipliers,” *IEEE Trans. Circuits Syst. II, Exp. Briefs*, vol. 69, no. 12, pp. 5099–5103, Dec. 2022.

[10] B. Zhang, Z. Cheng, and M. Pedram, “An iterative montgomery modular multiplication algorithm with low area-time product,” *IEEE Trans. Comput.*, vol. 72, no. 1, pp. 236–249, Jan. 2023.

[11] Z. Ahmadpour and G. Jaberipur, “Up to 8k-bit modular montgomery multiplication in residue number systems with fast 16-bit residue channels,” *IEEE Trans. Comput.*, vol. 71, no. 6, pp. 1399–1410, Jun. 2022.

[12] A. Hariri and A. Reyhani-Masoleh, “Concurrent error detection in montgomery multiplication over binary extension fields,” *IEEE Trans. Comput.*, vol. 60, no. 9, pp. 1341–1353, Sep. 2010.

[13] K. Ahmadi, S. Aghapour, M. M. Kermani, and R. Azarderakhsh, “Efficient error detection cryptographic architectures benchmarked on FPGAs for montgomery ladder,” *IEEE Trans. Very Large Scale Integr. (VLSI) Syst*, vol. 32, no. 11, pp. 2154–2158, Nov. 2024, doi: 10.1109/TVLSI.2024.3419700.

[14] M. Bedoui, B. Bouallegue, H. Mestiri, B. Hamdi, and M. Machhout “An improvement of both security and reliability for elliptic curve scalar multiplication montgomery algorithm,” *Multimedia Tools App/.* vol. 82, no. 8, pp. 11973–11992, Aug. 2023.

[15] J. C. Bajard, J. Eynard, and F. Gandino, “Fault detection in RNS montgomery modular multiplication,” in *Proc. IEEE Int. Symp. Comp. Arithmetic*, 2013, pp. 119–126.

[16] C. Chiou, C. Lee, A. Deng, and J. Lin, “Concurrent error detection in montgomery multiplication over $GF(2^m)$,” *IEICE Trans. Funda. Electron. Comm. Comput. Sci.*, vol. 89, no. 2, pp. 566–574, 2006.

[17] M. Prabu and R. Shanmugalakshmi, “A modified bit-serial montgomery multiplier algorithm in fault detection method,” in *Electrical Power Systems and Computers*, vol. 99. Berlin, Germany: Springer, 2011.

[18] X. Guo and R. Karri, “Recomputing with permuted operands: A concurrent error detection approach,” *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.,* vol. 32, no. 10, pp. 1595–1608, Oct. 2013.

[19] S. Liu, P. Reviriego, X. Tang, W. Tang, and F. Lombardi, “Result-based re-computation for error-tolerant classification by a support vector machine,” *IEEE Trans. Artif. Intell.*, vol. 1, no. 1, pp. 62–73, Aug. 2020.

[20] B. Sargunam and R. Dhanasekaran, “Duplication based error detection for montgomery multiplier over galois field,” in *Proc. IEEE Int. Adv. Commun. Cont. Comput. Tech.*, 2014, pp. 502–506.

[21] “The GNU multiple precision arithmetic library (GMP).” Accessed: May 2025. [Online]. Available: https://gmplib.org/

[22] L. Noyez, N. E. Mrabet, O. Potin, and P. Veron, “Montgomery multiplication scalable systolic designs optimized for DSP48E2,” *ACM Trans. Reconfig. Technol. Syst.*, vol. 17, no. 1, pp. 1–31, 2024.

[23] B. Zhang, Z. Cheng, and M. Pedram, “High-radix design of a scalable montgomery modular multiplier with low latency,” *IEEE Trans. Comput.*, vol. 71, no. 2, pp. 436–449, Feb. 2022.

[24] T. J. Grale and E. E. Swartzlander, “Improved montgomery multiplication,” in *Proc. IEEE Int. Symp. Comp. Arithmetic,* 2023, pp. 60–67.