Efficient Error Detection Schemes for ECSM Window Method Benchmarked on FPGAs

Kasra Ahmadi, Saeed Aghapour, Mehran Mozaffari Kermani, and Reza Azarderakhsh

***Abstract*— Elliptic curve scalar multiplication (ECSM) stands as a crucial subblock in elliptic curve cryptography (ECC), which represents the most widely used prequantum public key cryptography. Hardware constructions of cryptographic systems utilizing ECSM have been subject to permanent or transient errors. In cryptographic systems, it is important to validate the correctness of the underlying computation performed on hardware or software to identify such errors. In this article, we present new fault detection schemes in window method scalar multiplication, which, to the best of our knowledge, has not been previously investigated. Our approach involves introducing refined algorithms and implementations that can effectively counter both permanent and transient errors. We assess this by simulating a fault model, ensuring that the evaluations conducted reflect the obtained results. As a result, we achieve a significantly extensive coverage of errors. Finally, we benchmark our proposed error detection scheme on ARMv8 and field-programmable gate array (FPGA) to demonstrate the implementation and resource overhead. On Cortex-A72 processors, we maintain a clock cycle overhead of under 3%. In addition, when implementing our error detection method on different FPGAs, including Zynq Ultrascale+, Artix-7, and Kintex Ultrascale+, we achieve comparable throughput while introducing a mere 2% increase in area compared with the original hardware implementations.**

***Index Terms*— Fault detection, field-programmable gate array (FPGA), reliability, window method.**

I. Introduction

In 1985, Miller [1] and Koblitz [2] each independently introduced the application of elliptic curves in the field of cryptography. Elliptic curve cryptography (ECC) has garnered significant attention among public key cryptographic algorithms due to its shorter key sizes. To ensure efficient modular reduction, the National Institute of Standards and Technology (NIST) has suggested using Solinas prime as the modulus for short Weierstrass curves. It is noted that, researchers in [3] have raised concerns about potential backdoors in the NIST curves and have put forward Curve25519 and Curve448, which were developed by Hamburg [4], [5], as a viable alternative variants. Shor's [6] algorithm indicates that quantum computing has the potential to break the majority of current cryptographic systems. The emergence of quantum computers has raised concerns about the security of ECC, and it is expected to be eventually replaced with postquantum cryptography (PQC). Despite the need for transitioning to PQC due to potential quantum threats, ECC remains widely used for several reasons. Efficient algorithms have been introduced for the elliptic curve scalar multiplication (ECSM) operation, such as the binary method, double-and-add-always, Montgomery ladder, and window method [7]. To resist side channel attacks, it is essential to design the scalar multiplication algorithm to be regular, ensuring that its operation flow remains independent of the input scalar.

Manuscript received 23 August 2023; revised 3 November 2023; accepted 2 December 2023. Date of publication 18 December 2023; date of current version 27 February 2024. This work was supported by the U.S. National Science Foundation (NSF) under Award SaTC-1801488. *(Corresponding author: Mehran Mozaffari Kermani.)*

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

Color versions of one or more figures in this article are available at https://doi.org/10.1109/TVLSI.2023.3341147.

Digital Object Identifier 10.1109/TVLSI.2023.3341147

A. Related Works

A number of prior studies focus on the implementations and fault detection in various arithmetic aspects of both classical and PQC, encompassing ECSM, among others [8], [9], [10], [11], [12], [13], [14]. While fault detection procedures have been explored for cryptosystems, there has been limited research focusing on fault detection at the algorithm level of ECC.

Dominguez-Oviedo and Hasan [15] and Dominguez-Oviedo [16] addressed this gap by presenting fault detection schemes at the algorithm level for scalar multiplication on general curves. Their research introduced error detection schemes for the double-and-add-always and Montgomery ladder ECSM algorithm for nonsupersingular elliptic curves. It was found that the overhead of error detection algorithm on double-and-add-always is about 27% for projective coordinates and less than 1% for affine coordinates. Nevertheless, unlike prior studies, the process of selecting coordinates has not significantly influenced our proposed error detection approach for window method multiplication. In other words, such adoptions to other coordinates can also be performed.

B. Major Contributions

To the best of our knowledge, no prior research has explored fault detection in the context of window method scalar multiplication. The potential also exists for our error detection method to be utilized in the τ-nonadjacent form (NAF) conversion of the scalar within Koblitz curves, in cases where the window method ECSM is applied.

1) We have proposed an algorithm level fault detection scheme on window method ECSM. Our proposed fault detection algorithm has been mathematically proven to determine the extent of error coverage it provides. As such, we have simulated error injection on our proposed approach. According to the simulation results, our proposed scheme demonstrates the ability to detect a wide range of fault variants with a high level of error coverage.
2) We have implemented our proposed fault detection scheme on ARMv8 and field-programmable gate array (FPGA) architectures to assess its viability. For less constrained usage models, we selected Cortex-A72 from ARMv8 family, which is used in Raspberry Pi 4. Our results show that we can achieve very high error coverage while adding only 3% more clock cycles on software. Furthermore, our error detection technique is executed on different FPGAs, including Zynq Ultrascale+, Artix-7, and Kintex Ultrascale+, delivering comparable throughput with a mere 2% rise in area as opposed to the initial method.

II. Preliminaries

A. Window Method

Given the availability of additional memory, the window method significantly improves the efficiency of scalar multiplication by

Algorithm 1 Window Method Scalar Multiplication

Input: P(point), k(scalar), ω(window length)
Output: kP
1: precomp = *empty_array*, current_point = P
2: result = *initial_point*, counter = 0
3: k_{bits} = binary(k)
4: **for** _ **in** range(2^{ω}):
5: precomp.append(current_point)
6: current_point = point_add(P,current_point)
7: **for** bit **in** k_{bits}:
8: counter += 1;
9: window += bit;
10: **if** len(window) = ω **or** counter = len(k_{bits}):
11: value = int(window, 2) // change window value from binary to decimal
12: **for** _ **in** range(len(window)):
13: result = point_double(result)
14: **if** value != 0:
15: result = point_add(result,precomp[value]-1)
16: clear window
17: **return** result

utilizing a table of precomputed points. Window algorithms share similarities with binary algorithms, but they differ in that each iteration of the former concentrates on a window of ω scalar bits, rather than processing individual bits. In other words, each loop iteration handles a digit of the scalar represented in radix 2^{ω}. The window method, which is described in Algorithm 1, involves two stages. The first stage is the precomputation, which is a one-time process, and the second stage is the window method arithmetic.

III. Error Detection in Window Method ECSM

In this section, we introduce the proposed error detection scheme in window-based scalar multiplication. The process involves coherency check (CC) validation, which examines the intermediate or final output to ensure it adheres to a valid pattern in any given algorithm. Essentially, such validation is capable of confirming the specific relationship between intermediate variables and the output of any routine.

Our error detection approach, which is described in Algorithm 2, involves modifying Algorithm 1 to validate the occurrence of various patterns within a window of a specific length. For a window length of ω, there will be 2^{ω} different patterns in the binary form of a scalar. In our proposed error detection scheme, we keep track of the count for each pattern both before and during the ECSM operation. The function known as pattern_counter, outlined in Algorithm 3 and utilized in line 9 of Algorithm 2, is responsible for computing the occurrences of scalar patterns with respect to length of window, ω, before commencing the primary operation of window method scalar multiplication. In line 19 of Algorithm 2, the occurrences of scalar patterns are computed once again and stored in post_pattern_counter. At the conclusion of Algorithm 2, if there is a disparity between the values of pre_pattern_counter and post_pattern_counter, it indicates the occurrence of at least one error during the ECSM operation. This coherency checking process takes place at line 21 of Algorithm 2.

IV. Simulations and Mathematical Analysis

Within this section, we introduce the employed fault model and subsequently conduct a mathematical analysis and derive proofs for each error detection scheme in relation to this fault model. Following that, we present the simulation results obtained through Python implementation, utilizing the proposed fault model and error detection schemes.

Algorithm 2 Proposed Error Detection Scheme in Window Method Scalar Multiplication

Input: P, k, ω
Output: kP
1 : precomp = *empty_array*, current_point = P
2 : result = *initial_point*, counter = 0
3 : pre_pattern_counter = *empty_dictionary*
4 : post_pattern_counter = *empty_dictionary*
5 : k_{bits} = binary(k)
6 : **for** _ **in** range(2^{ω}):
7 : pre_comp.append(current_point)
8 : current_point = point_add(P,current_point)
9 : pre_pattern_counter = pattern_counter(k_{bits}, ω)
10: **for** bit **in** k_{bits}:
11: counter += 1;
12: window += bit;
13: **if** len(window) = ω **or** counter = len(k_{bits}):
14: value = int(window, 2) // change window value from binary to decimal
15: **for** _ **in** range(len(window)):
16: result = point_double(result)
17: **if** value != 0:
18: result = point_add(result, precomp[value]-1)
19: post_pattern_counter[value]+=1
20: clear window
21: **if** pre_pattern_counter != post_pattern_counter:
22: **return** "Error detected"
23: **else**:
24: **return** result

Algorithm 3 Pattern_Counter Function

Input: k, ω
Output: pattern_counter
1 : pattern_counter = *empty*_array, counter = 0
2 : width_precomp = *empty_array*
3 : k_{bits} = binary(k)
4 : **for** (i=0; i<ω; i++)
5 : width_precomp[i] = 2^i
6 : **for** (i = 0; i<$\frac{log\, k}{\omega}$; i++):
7 : win_value = 0
8 : **for**(j=1; j$\leqslant\omega$;j++):
9 : **if**(k_{bits}[$log\, k$-((i.width)+j)] = 1):
10: win_value += width_precomp[ω-j]
11: pattern_counter[win_value] += 1
12: **return** pattern_counter

A. Fault Model

The window method calculation in Algorithm 2 may experience the occurrence of random or burst errors during its execution. To simulate faults, we utilized the pseudo-program counter (PC) model. According to this model, after executing each instruction,

Algorithm 4 Fault Calculation Function

Input: k, ω
Output: faulty_instructions
1: faulty_instructions = *empty_array*
2: max_fault = $5log\,k + \frac{2log\,k}{\omega}$
3: **for** _ **in** range(error_number):
4: faulty_instructions.append(random(1,max_fault))
5: **return** faulty_instructions

a counter (PC) is incremented. In this particular model, we made assumptions that each line of code is an instruction, and all instructions carry an equal workload. Consequently, if a specific instruction within an algorithm executes more, the likelihood of encountering an error during that particular instruction increases. Algorithm 4 outlines our fault calculation function, which is responsible for randomly choosing faulty instructions prior to executing Algorithm 2.

Once the fault calculation function has been executed, a check is performed before each instruction execution. This check determines whether the counter corresponds to a line of faulty code or not. If the current counter is associated with a faulty instruction, it indicates that a fault should occur, leading to a random alteration of the output produced by that particular instruction. We assumed any fault introduced would result in a modification of the instruction's output.

B. Mathematical Analysis

Lines 10–13 and 16 in Algorithm 2 execute $\log k$ times, while lines 15 and 18 execute a maximum of $\log k/\omega$ times, where ω represents the length of window and k is the scalar. In Algorithm 4, line 2 computes the possible_fault_domain, which corresponds to the maximum value of the PC counter. Our error detection scheme is capable of identifying errors that occur in all the instructions except for the point_double and point_add operations (corresponding to lines 16 and 18). The cause for this is that any alterations made to the instructions, with the exception of point_double and point_add, have a direct impact on the post_pattern_counter array. This array is later compared with the pre_pattern_counter array at the conclusion of Algorithm 2.

The probability of undetected fault is $\alpha = (log\,k + log\,k/\omega)/(5\log k + 2\log k/\omega)$. Error detection ratio in our scheme will be $1 - \alpha^m$, where m is the number of faults, which will happen in the window method scalar multiplication. It is evident that the error detection ratio increases in scenarios with a higher number of faults.

C. Simulation Results

Using Python 3, we conducted simulations utilizing the presented fault models and error detection methods. The simulation used million samples, each utilizing varying ω. As described in Table I, employing a smaller window length leads to a higher error detection ratio in cases of single fault occurrence. Moreover, as the number of faults increases, we can achieve an approximately 100% error detection ratio across all window lengths. Burst errors can be identified with higher error coverage as well.

V. Implementation Benchmarks and Comparison

In order to demonstrate the effectiveness and error detection capabilities of our proposed scheme, we opted to evaluate it using short Weierstrass curve and employed the binary form of the scalar for the analysis. We performed an implementation benchmark with $\omega = 3$, on the Cortex-A72 processor and various FPGAs, including Zynq Ultrascale+, Artix-7, and Kintex Ultrascale+ to assess the overhead of our approach. This analysis clearly illustrates that our methods maintain a reasonable overhead while successfully achieving a high ratio of fault detection.

TABLE I
Error Detection Ratios of Our Proposed Scheme With 1, 2, 4, 8, and 16 Faults for Window Length (ω) of 2, 3, 5, 7, and 10 Incurred in the Implementation of Algorithm 2 Based on the Simulation Results

		Window length (ω)				
		2	3	5	7	10
Number of faults	1	57%	40%	41%	39%	39%
	2	83%	67%	68%	63%	62%
	4	96%	92%	91%	87%	85%
	8	99%	99%	99%	98%	98%
	16	99.9%	99.9%	99.9%	99.9%	99.9%

A. Implementation Results

Through the implementation of our design on ARMv8 and FPGA architectures, we gained valuable insights into the efficacy of our schemes across diverse platforms. For hardware implementation, we have selected AMD/Xilinx FPGA, including Zynq Ultrascale+, Artix-7, and Kintex Ultrascale+. We used high-level synthesis (HLS) Vitis development environment to synthesize our proposed schemes to register transfer-level (RTL) hardware description. We used Vitis resource allocation pragmas to achieve the respective area and performance strategies and to control the trade-off between the time and the required resources. The same pragma rules and clock were used in the baseline and the proposed error detection scheme. In the context of the Vivado synthesis tool, area is defined as a combination of Slices and Digital Signal Processing Units (DSPs). In addition, the equivalence ratio used is that one DSP is considered equal to 100 Slices. The latency was evaluated using identical test vectors for each hardware implementation. Tables II–IV present the area, timing, power, and energy derivations of our proposed error detection scheme based on the aforementioned area and timing effort strategies on three different AMD/Xilinx FPGAs. Our proposed error detection designs successfully reach an approximate maximum operational frequency of 60 MHz across all the FPGAs. Our proposed error detection scheme incurs a maximum overhead of 2% in terms of additional area and adds a latency of up to six clock cycles at most. Our implementation code in HLS and simulation code are available in our GitHub account.[1]

For the ARMv8 architecture, we chose the Raspberry Pi 4 as our platform, equipped with quad 1.5-GHz Cortex-A72 cores. To ensure a suitable environment, we installed Raspberry Pi OS lite 32 bit with kernel version 6.1 as the operating system on the Raspberry Pi 4. For evaluating performance on the ARMv8 architecture, we utilized the Performance Application Programming Interface (PAPI), a well-established framework for measuring system performance [18]. Table V illustrates the clock cycle counts for both the proposed error detection scheme and the baseline approach.

B. Implementation Optimization

We note that the performed implementations are utilized for deriving the overhead of the error detection schemes used, to assess their suitability for constrained usage models; nevertheless, implementations on other hardware platforms (other FPGA families and devices and also ASIC) would result in similar overheads, because the proposed schemes are platform oblivious.

[1]https://github.com/KasraAhmadi/window_method_error_detection.git

TABLE II
AMD/XILINX ZYNQ ULTRASCALE+, XCZU4EV-SFVC784-2-I IMPLEMENTATION RESULTS

Strategy		Area Effort		Timing Effort	
Scheme		Our scheme[1]	Baseline work [2]	Our scheme	Baseline work
Area	LUTs	7,767	7,681	9,245	9,162
	FFs	5,396	5,306	5,947	5,857
	CLBs	1,629	1,624	1,868	1,850
	DSPs	60	60	66	66
Power (W) @ 60 MHz		0.42	0.42	0.44	0.44
Timing	Latency [CCs]	39,711	39,705	38,825	38,819
	Total time [ms]	0.660	0.659	0.644	0.644
Energy (mJ)		0.27	0.27	0.28	0.28

[1]Our Scheme: The design which includes error detection scheme.
[2]Baseline work: The design which does not include any error detection scheme.

TABLE III
AMD/XILINX ARTIX-7, XC7A100T-CSG324-3 IMPLEMENTATION RESULTS

Strategy		Area Effort		Timing Effort	
Scheme		Our scheme	Baseline work	Our scheme	Baseline work
Area	LUTs	8,011	7,864	9,550	9,406
	FFs	6,279	6,189	6,830	6,740
	SLICEs	2,909	2,950	3,349	3,318
	DSPs	64	64	70	70
Power (W) @ 60 MHz		0.16	0.16	0.17	0.18
Timing	Latency [CCs]	39,719	39,713	38,833	38,827
	Total time [ms]	0.659	0.659	0.644	0.644
Energy (mJ)		0.10	0.10	0.11	0.11

TABLE IV
AMD/XILINX KINTEX ULTRASCALE+, XCKU5P-SFVB784-3-E IMPLEMENTATION RESULTS

Strategy		Area Effort		Timing Effort	
Scheme		Our scheme	Baseline work	Our scheme	Baseline work
Area	LUTs	7,764	7,761	9,243	9,236
	FFs	5,468	5,378	6,019	5,929
	CLBs	1,579	1,586	1,867	1,915
	DSPs	60	60	66	66
Power (W) @ 60 MHZ		0.55	0.55	0.57	0.55
Timing	Latency [CCs]	39,709	39,703	38,823	38,818
	Total time [ms]	0.659	0.659	0.644	0.644
Energy (mJ)		0.36	0.36	0.36	0.35

TABLE V
ARMV8 IMPLEMENTATION RESULTS

	Window method ECSM	
Scheme	Our Scheme	Baseline work
Clock cycles	27,718	27,118
Clock cycles overhead[1]	2.2%	-

[1]Clock cycles overhead $= \frac{\text{Approach's clock cycles} - \text{Baseline work's clock cycles}}{\text{Baseline work's clock cycles}} \times 100$

While HLS has transitioned the design abstraction from RTL to C/C++, it is often essential, in practice, to engage in substantial source code rewriting, including the insertion of pragmas, to achieve satisfactory performance. We achieved our intended area and timing efforts implementation by strategically placing the following pragmas within our program.

1) *Pragma HLS Inline:* We placed this pragma to limit the resource allocation to control the area usage. Inlining promotes cross optimization among various C functions arranged hierarchically. Differing from a basic optimization method that handles each function independently, this pragma encourages resource sharing to improve performance and reduce the usage of digital resources.
2) *Pragma HLS Pipeline:* We placed this pragma to enable instruction-level pipelining to increase the throughput and clock frequency within the doubling and addition modules, which are the most time-consuming components in the window method ECSM. However, it is worth noting that this optimization comes with the trade-off of consuming additional digital resources.
3) *Pragma HLS Interface:* We used this pragma to integrate our scheme into Vivado as an external IP. This pragma not only simplified the incorporation process but also resulted in an overall performance boost and more efficient utilization of resources, effectively making our IP an integral part of the FPGA design.

VI. DISCUSSION

The presented error detection scheme can be utilized on various curves, such as Montgomery (Curve448 and Curve25519) and Weierstrass, all of which employ the fixed window method ECSM. In addition, it applies to both the binary and NAF of the scalar. Moreover, it has the potential to be employed in the τ-NAF conversion of the scalar within Koblitz curves, where the window method ECSM is utilized. It is important to note that our error detection approach is not reliant on particular coordinates. This signifies that whenever the window method algorithm is chosen for scalar multiplication due to application requirements and hardware limitations, our error detection approach remains applicable.

A. Challenges and Shortcomings

A potential add-on to this work could be an extension, so that our proposed error detection scheme can identify errors occurring within the doubling or addition components. Our research focuses specifically on the algorithmic level of the window method ECSM.

To address this issue, multiple error detection schemes [8], [9] have been performed for multiplication, addition, inversion, and squaring, which can be integrated into our work. Furthermore, another method to overcome the aforementioned issue is to easily confirm that the doubling and the addition's output lie on a valid elliptic curve. This procedure is referred to as point verification (PV) [15].

VII. Conclusion

In this article, we proposed an error detection scheme for window method ECSM operation, which is a crucial operation in ECC-based cryptosystems. By computing the occurrences of possible scalar patterns in a window length, we provide a coherency function between the scalar as an input and intermediate outputs on the window method scalar multiplication. Through conducting simulations and mathematical analysis, we showed that our proposed error detection scheme achieved high error coverage. Furthermore, we implemented our proposed design on FPGAs and ARMv8. In terms of overhead, the implementation resulted in negligible additional cost in hardware and software.

References

[1] V. S. Miller, "Use of elliptic curves in cryptography," in *Proc. Conf. Theory Appl. Cryptograph. Techn.* (Lecture Notes in Computer Science), vol. 218. Cham, Switzerland: Springer, 1986, pp. 417–426.

[2] N. Koblitz, "Elliptic curve cryptosystems," *Math. Comput.*, vol. 48, no. 177, pp. 203–209, 1987.

[3] *Safecurves: Introduction*. Accessed: Dec. 2023. [Online]. Available: https://safecurves.cr.yp.to/

[4] M. Hamburg, "Ed448-goldilocks, a new elliptic curve," IACR Cryptol. ePrint Arch., Tech. Rep., 625, 2015. [Online]. Available: https://eprint.iacr.org/2015/625

[5] M. Hamburg. *Ed448-Goldilocks, a New High-Strength Curve and Implementation*. Accessed: Dec. 2023. [Online]. Available: https://csrc.nist.gov/csrc/media/events/workshop-on-elliptic-curve-cryptography-standards/documents/presentations/session7-hamburg-michael

[6] P. W. Shor, "Algorithms for quantum computation: Discrete logarithms and factoring," in *Proc. 35th Annu. Symp. Found. Comput. Sci.*, 1994, pp. 124–134.

[7] M. Rivain. (2011). *Fast and Regular Algorithms for Scalar Multiplication Over Elliptic Curves*. [Online]. Available: http://eprint.iacr.org/2011/338.pdf

[8] S. Bayat-Sarmadi and M. A. Hasan, "Concurrent error detection in finite-field arithmetic operations using pipelined and systolic architectures," *IEEE Trans. Comput.*, vol. 58, no. 11, pp. 1553–1567, Nov. 2009.

[9] 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.

[10] G. Bertoni, L. Breveglieri, I. Koren, P. Maistri, and V. Piuri, "Error analysis and detection procedures for a hardware implementation of the advanced encryption standard," *IEEE Trans. Comput.*, vol. 52, no. 4, pp. 492–505, Apr. 2003.

[11] 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.

[12] M. Mozaffari-Kermani and R. Azarderakhsh, "Reliable hash trees for post-quantum stateless cryptographic hash-based signatures," in *Proc. IEEE Int. Symp. Defect Fault Tolerance VLSI Nanotechnol. Syst. (DFTS)*, Oct. 2015, pp. 103–108.

[13] 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.

[14] M. M. Kermani and R. Azarderakhsh, "Reliable architecture-oblivious error detection schemes for secure cryptographic GCM structures," *IEEE Trans. Rel.*, vol. 68, no. 4, pp. 1347–1355, Dec. 2019.

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

[16] A. Dominguez-Oviedo, "On fault-based attacks and countermeasures for elliptic curves cryptosystems," Ph.D. dissertation, Dept. Electr. Comput. Eng., Univ. Waterloo, Waterloo, ON, Canada, 2008.

[17] P.-A. Fouque, R. Lercier, D. Réal, and F. Valette, "Fault attack on elliptic curve Montgomery ladder implementation," in *Proc. 5th Workshop Fault Diagnosis Tolerance Cryptography*, Aug. 2008, pp. 92–98.

[18] D. Terpstra, H. Jagode, H. You, and J. Dongarra, "Collecting performance data with PAPI-C," in *Tools for High Performance Computing*. Heidelberg, Germany: Springer, 2010, pp. 157–173.