POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Executive Summary of the Thesis

Hardening the Cortex-M4 Implementation of CROSS Against Active Side-Channel Attacks

Laurea Magistrale in Computer Science and Engineering - Ingegneria Informatica

Author: Andrea Cavezzale

Advisor: Prof. Alessandro Barenghi

Co-advisors: Prof. Gerardo Pelosi, Marco Gianvecchio

Academic year: 2024-2025

1. Introduction

Nowadays, the biggest challenge for the cryptographic community is the advancements of quantum computers. Such devices are capable of efficiently solving problems that classical machines are not able to solve equally efficiently. These problems, like the discrete logarithm problem or prime factorization, are the core of public key cryptography, the standard way to exchange keys to communicate between digital devices. Even though quantum algorithms existed for a long time (see Shor's algorithm [6]), quantum computers are not already capable of running them against current protocols; albeit they might be in the relatively near future. Hence, data might be collected now and deciphered later; this practice is called Harvest Now, Decrypt Later (HNDL). This was the reason why, in 2016 and again in 2022, the National Institute of Standards and Technology (NIST) called for the creation of new schemes, based on different problems, to be used as new standards. While the 2016 call ended with the definition of three new schemes as standards (for key encapsulation and for digital signature), the 2022 one is still running with fourteen remaining candidates. This new branch of cryptography took the name of Post-Quantum Cryptography (PQC). These new schemes immediately became targets of thorough analyses. Such analyses focus on two aspects: the protocol itself or its implementation. Our work will revolve around the second type of analysis. The attacks that target the implementation of a cryptographic protocol take the name of Side-Channel Attacks (SCAs). SCAs exploit the protocol in two distinct ways. The first is through information leakage; hence, they are called passive attacks. An attacker tries to extract information (like the encryption key) by exploiting vulnerabilities in the implementation. Such vulnerabilities might be related to usage of cache memory, timing or power consumption. The second one is through fault injection, where an attacker modifies the computation by blanking memory or skipping instructions; these attacks are called active attacks. Our work will focus on the active SCAs and on the digital signature scheme CROSS.

2. CROSS

2.1. Key elements of the protocol

Codes and Restricted Objects Signature Scheme (CROSS) is a digital signature scheme based on

the concept of Restricted Syndrome Decoding Problem (R-SDP), Zero-Knowledge Proofs and Fiat-Shamir transformation [1]. The R-SDP is a relatively new variant of the widely studied and established Syndrome Decoding Problem: This variant is proved to be NP-complete in [2], and it is believed to be a hard problem not only for classical machines but also for quantum computers.

2.2. Cortex-M4 implementation

SCAs typically target embedded devices as they are the most common way to run digital signatures algorithms. CROSS has a newly introduced implementation meant to be deployed over such devices [5]. This version introduced many optimizations and changes in the implementation of the primitives. These optimizations were obtained via a memory-time trade-off (allowing recomputation in order to save memory), via memory sharing between compatible components of the algorithms, and via a redesign of the data structures.

3. Attacking CROSS through Side-Channel analysis

Our work started with the definition of a threat model. As shown in [3], active side-channel attacks might be relatively cheap to conduct if we compromise on the types of fault injection that an attacker can do. We adopted a relatively strict fault injection model where an attacker might: blank (entirely or partially) selected portion of memory or skip an instruction. These two faults can be obtained via the techniques presented in [3], typically by tampering with the clock frequency (instruction skipping) or by using UV lamps (blanking).

Our first contribution consisted in the identification of the vulnerable parts of the three main primitives of CROSS: KeyGen, Sign, and Verify. The analysis started with the construction of dependencies graphs, we provide an example of the simplest one, the one related to KeyGen in Figure 1.
Such construction required a bottom-up analysis of the information flow, in order to correctly identify the relationship between the different elements.

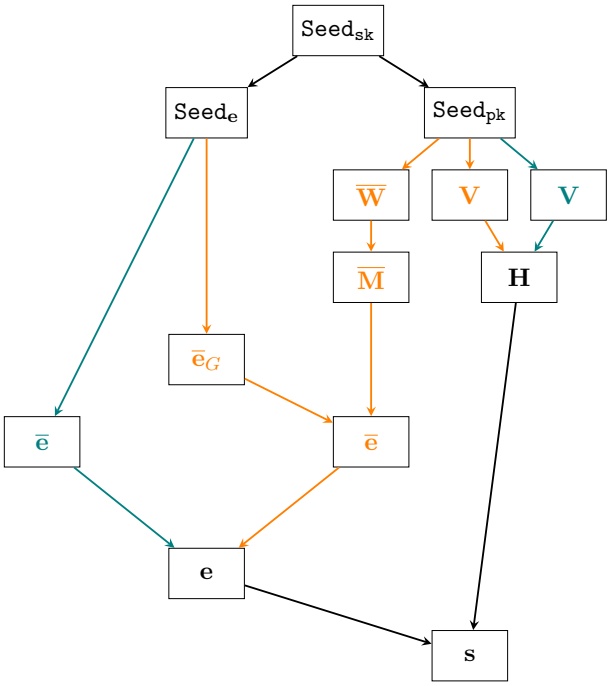

Figure 1: KeyGen dependencies.

Then we began the actual analysis on the primitives. Our primary objective when designing an attack against KeyGen and Sign was to leak the private key, or when not available, the error vector linked to it. When attacking Verify our priority was forcing the acceptance of forged signatures. The pseudocode used during the analyses was provided by the original authors [1].

3.1. Attacking KeyGen

Against KeyGen a complete key recovery was not identified. However, two interesting attacks were found: the first one consisted in the publication of $\approx 40\%$ of the coordinates of the error vector by blanking the non-systematic portion of the parity-check matrix. The second one consisted in skipping the extraction of random bytes to assign to the private key, in order to keep it uninitialized.

3.2. Attacking Sign

Most of our effort was directed against Sign as it provided the biggest attack surface. The analysis is divided in two: firstly we analyzed the main (round) `for` loop, secondly we targeted the data structures used to keep track of the seed used to extract the ephemeral error vectors used during rounds (i.e., the seed or GGM trees).

The first analysis translated in the identification of four possible attacks. They consisted in targeting the lines $\overline{\mathbf{v}} \leftarrow \overline{\mathbf{e}} - \overline{\mathbf{e}}'[i]$ and $\mathbf{y}[i] \leftarrow \mathbf{u}'[i] + \texttt{chall}_1[i]\mathbf{e}'[i]$ via blanking and instruction skipping. By analyzing the actual code we observed that the second line cannot be attacked via instruction skipping, reducing the actual attacks to three.

The second analysis identified three targets: the first two were related to the CSPRNG used to sample the round seeds and the master seed. The last one targeted `Path`, used to store the nodes of the seed tree required to reconstruct the needed round seeds during Verify.

3.3. Attacking Verify

Against Verify we identified two attacks with an equivalent outcome: the acceptance of a forged signature. However, only one of them was coherent with our threat model. The first one consisted in the blanking of the $\texttt{chall}_1$ vector. By doing so, and with a custom Sign, an attacker is able to produce a signature using exclusively the public key of the defender to reconstruct the correct parity-check matrix. The second one required the blanking of a register (hence not feasible in our scenario) used to determine a parameter required in two of the variables used to check if a signature is valid. By setting this parameter to zero, Verify accepts any signature, if well formatted.

4. Creation of a countermeasure

Our second contribution consisted in the creation of a countermeasure against the most dangerous attacks. These attacks mostly target Sign. The design of the countermeasure started by analyzing the actual Cortex-M4 implementation and in the development of a stand-alone version of it (the original one relies on `pqm4` framework [4]). This implementation requires three `for` loops, where many components are recomputed. We reused this concept, and we combined it with the introduction of a new Keccak state. Such state will be computed a first time during the first `for` loop and it will be recomputed a second time during the last `for` cycle. By comparing the two results we will be able to find out if an attacker tried to inject a fault and consequently, reject the signature. Naturally, the state should involve all vulnerable components, hence $\texttt{Seed}[i]$ (the round seed), $\overline{\mathbf{e}}'[i], \mathbf{u}'[i]$. Due to the constraints of embedded devices we had to assess the performance of our new version of Sign, in order to find out the impact both in terms of speed and stack consumption. These benchmarks were computed using `pqm4` framework and a STM32F4DISCOVERY board, a small data sheet is provided in Table 1.

Table 1: STM32F4DISCOVERY data sheet.

Component	Specification
Microcontroller	STM32F407VGT6
Processor Core	32-bit Arm Cortex-M4 with FPU
Flash Memory	1 Mb
RAM	192 Kb
Debugging	On-board ST-LINK/V2-A

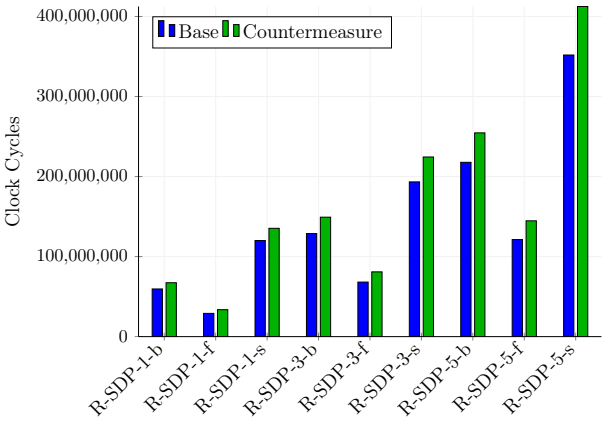

Figure 2: Increase in number of clock cycles when signing with countermeasures enabled (m4opt – R-SDP).

5. Experimental results

By comparing the graphs shown in Figure 2 and in Figure 3 we observe that the impact on speed (measured in clock cycles) is significantly bigger on R-SDP(G). This can be explained by the fact that the amount of total clock cycles is generally much lower in R-SDP(G) than in R-SDP, but the overhead introduced by the countermeasure is almost the same. Such considerations translate into an increase between 10% and 20% in number of clock cycles for R-SDP and between 25% and 40% for R-SDP(G).

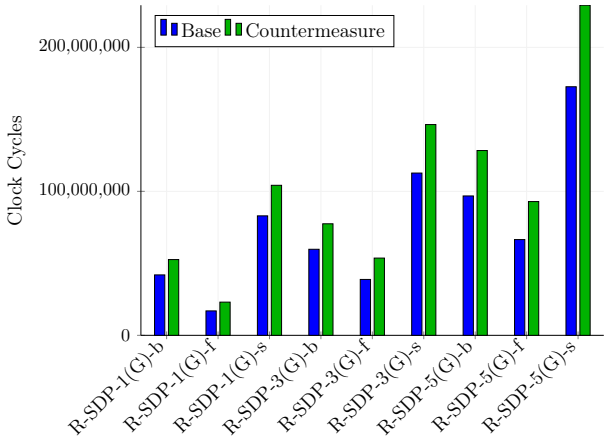

Figure 3: Increase in number of clock cycles when signing with countermeasures enabled (m4opt – R-SDP(G)).

The increase on stack size, shown in Figure 4 and in Figure 5 is not particularly significant as it is smaller than 9% for every schemes. Similar considerations to the ones related to speed hold for stack consumption where R-SDP(G) has slightly worse performance. A complete overview of the benchmarks is provided in Table 2 and in Table 3.

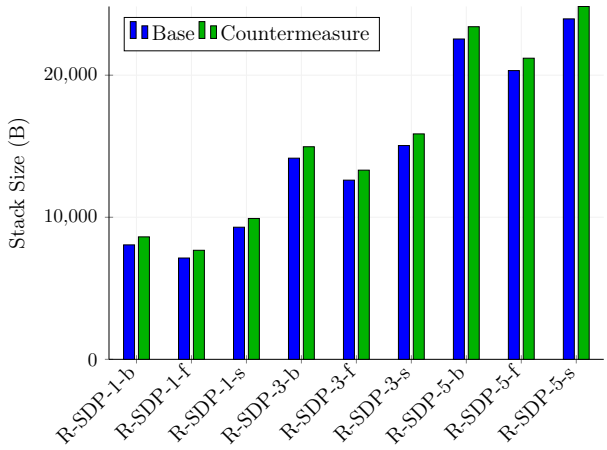

Figure 4: Increase in stack size (B) when signing with countermeasures enabled (m4stack – R-SDP).

6. Conclusions

In our work we achieved two objectives. The first one was the assessment of the vulnerable components of CROSS under the point of view of active side-channel attackers. Our observations will not be useful only to CROSS or other code-based signature schemes but also to

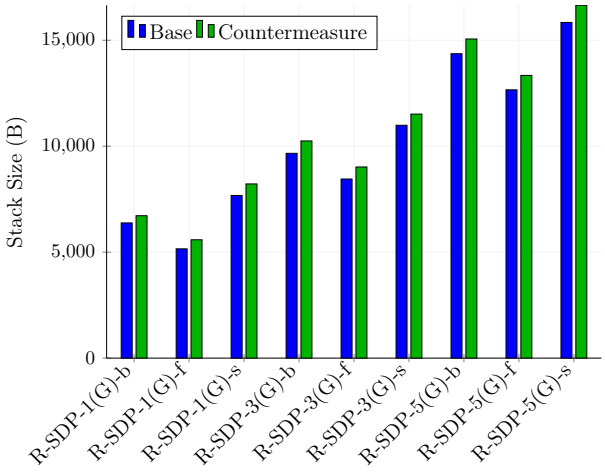

Figure 5: Increase in stack size (B) when signing with countermeasures enabled (m4stack – R-SDP(G)).

Table 2: Increase in number of clock cycles when signing with countermeasures enabled (m4opt).

Algorithm and Security Category	Optim. Corner	Number of CC (in millions) original	with countermeasure	Percentage Increase
CROSS-R-SDP **1**	**balanced**	59,56	67,34	13.07%
	fast	29,04	33,65	15.85%
	small	120,08	135,35	12.72%
CROSS-R-SDP **3**	**balanced**	128,77	149,25	15.91%
	fast	68,15	80,92	18.73%
	small	193,34	224,46	16.09%
CROSS-R-SDP **5**	**balanced**	217,80	254,59	16.89%
	fast	121,28	144,69	19.3%
	small	351,87	412,46	17.22%
CROSS-R-SDP(G) **1**	**balanced**	41,97	52,62	25.37%
	fast	16,97	23,06	35.91%
	small	82,98	104,20	25.57%
CROSS-R-SDP(G) **3**	**balanced**	59,77	77,47	29.61%
	fast	38,82	53,65	38.21%
	small	112,71	146,39	29.88%
CROSS-R-SDP(G) **5**	**balanced**	96,81	128,35	32.59%
	fast	66,51	92,90	39.68%
	small	172,65	229,27	32.79%

other post-quantum algorithms. In fact, many other schemes use the same data structures implemented in CROSS, like the seed trees, and many others are based on concepts like the Fiat-Shamir transformation. Our adopted approach might be useful not only in real-world scenarios, where CROSS might be used, but only as a possible source of inspiration to analyze other cryptosystems against side-channel attacks. The second objective was the development of a countermeasure based on hash-based primitives like Keccak. Our implementation countered the most dangerous attacks against the signature primitive, the ones that lead to private key/error vector exfiltration. Thus, we increased the security margin of the M4 version of CROSS against

Table 3: Increase in stack consumption when signing with countermeasures enabled (m4stack).

Algorithm and Security Category	Optim. Corner	Stack consumption (B) original	Stack consumption (B) with countermeasure	Percentage Increase
CROSS-R-SDP **1**	**balanced**	8056	8620	7%
	fast	7132	7676	7.63%
	small	9300	9916	6.62%
CROSS-R-SDP **3**	**balanced**	14160	14964	5.68%
	fast	12612	13316	5.58%
	small	15048	15868	5.45%
CROSS-R-SDP **5**	**balanced**	22548	23412	3.83%
	fast	20324	21196	4.29%
	small	23964	24836	3.64%
CROSS-R-SDP(G) **1**	**balanced**	6380	6720	5.33%
	fast	5156	5584	8.3%
	small	7676	8220	7.09%
CROSS-R-SDP(G) **3**	**balanced**	9664	10248	6.04%
	fast	8452	9020	6.72%
	small	10988	11516	4.81%
CROSS-R-SDP(G) **5**	**balanced**	14364	15060	4.85%
	fast	12660	13340	5.37%
	small	15840	16652	5.13%

active side-channel attacks.

References

[1] M. Baldi, A. Barenghi, M. Battagliola, S. Bitzer, M. Gianvecchio, P. Karl, F. Manganiello, A. Pavoni, G. Pelosi, F. Pintore, P. Santini, J. Schupp, E. Signorini, F. Slaughter, A. Wachter-Zeh, and V. *CROSS: Codes and Restricted Objects Signature Scheme*. National Institute of Standards and Technology (NIST). Submission to the NIST Post-Quantum Cryptography Standardization Process (Round 2).

[2] Marco Baldi, Massimo Battaglioni, Franco Chiaraluce, Anna-Lena Horlemann-Trautmann, Edoardo Persichetti, Paolo Santini, and Violetta Weger. A new path to code-based signatures via identification schemes with restricted errors, 2021.

[3] Alessandro Barenghi, Luca Breveglieri, Israel Koren, and David Naccache. Fault injection attacks on cryptographic devices: Theory, practice, and countermeasures. *Proceedings of the IEEE*, 100(11):3056–3076, 2012.

[4] Matthias J. Kannwischer, Richard Petri, Joost Rijneveld, Peter Schwabe, and Ko Stoffelen. PQM4: Post-quantum crypto library for the ARM Cortex-M4. `https://github.com/mupq/pqm4`.

[5] Jonas Schupp, Marco Gianvecchio, Alessandro Barenghi, Patrick Karl, Gerardo Pelosi, and Georg Sigl. Optimizing the post quantum signature scheme CROSS for resource constrained devices. Cryptology ePrint Archive, Paper 2025/1928, 2025.

[6] Peter W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM Journal on Computing*, 26(5):1484–1509, October 1997.