POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

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

Tesi di Laurea Magistrale in
Ingegneria Informatica
Computer Science and Engineering

Author: **Andrea Cavezzale**

Student ID: 10712332
Advisor: Prof. Alessandro Barenghi
Co-advisors: Prof. Gerardo Pelosi, Marco Gianvecchio
Academic Year: 2024-25

Abstract

Due to the advancements of quantum computing, cryptographic community's attention has shifted to Post-Quantum Cryptography, a relatively new branch of the discipline that studies quantum-resistant algorithms. This work focuses on one of the current candidates in the most recent NIST call for quantum-resistant algorithms, CROSS, specifically on the implementation designed for embedded devices. Such devices are the primary targets of side-channel attacks, a category of attacks targeting cryptosystems and digital signature schemes. They might be cheap to conduct even though they might achieve significant results, such as a complete key recovery.

This work achieves two main results. The first is the analysis and identification of vulnerabilities in the ARM Cortex-M4 implementation of CROSS regarding active side-channel attacks. The second is the development of a new version of the signature primitive that incorporates a countermeasure based on Keccak, capable of mitigating the most severe attacks identified during the analysis.

Keywords: Post-Quantum Cryptography, Side-Channel Attacks, Digital Signature, ARM Cortex-M4, Embedded Devices

Abstract in lingua italiana

A causa dell'avanzamento del calcolo quantistico, l'attenzione della comunità crittografica si è spostata sulla Crittografia Post-Quantistica, una nuova branca della disciplina che si occupa di studiare algoritmi resistenti ai computer quantistici. Questo lavoro si concentra su CROSS, uno dei candidati nel più recente percorso di selezione, relativo ad algoritmi resistenti ai computer quantistici, bandito dal NIST. Nello specifico, la nostra attenzione sarà rivolta all'implementazione sviluppata per i sistemi embedded. Tali dispositivi infatti sono i bersagli primari degli attacchi side-channel, una categoria di attacchi rivolta ai crittosistemi e ai sistemi di firma digitale. Nonostante i risultati a cui tali attacchi possono condurre, come un recupero completo della chiave privata, essi possono essere implementati con costi contenuti.

Questo lavoro consegue due risultati principali. Il primo è l'analisi e l'identificazione delle vulnerabilità presenti nell'implementazione di CROSS per ARM Cortex-M4 rispetto ad attacchi side-channel attivi. Il secondo è lo sviluppo di una nuova versione della primitiva di firma che integra una contromisura basata su Keccak, capace di mitigare gli attacchi più pericolosi identificati durante l'analisi.

Parole chiave: Crittografia Post-Quantistica, Attacchi Side-Channel, Firma Digitale, ARM Cortex-M4, Sistemi Embedded

Contents

Introduction

Since 1976, when Whitfield Diffie and Martin E. Hellman published their revolutionary article [18], the cryptographic community agreed on the potential of public key, or asymmetric, cryptosystems. Historically, many algorithms were developed using a symmetric paradigm in which the encryption and the decryption key were the same (e.g., Caesar cipher, Enigma machine, DES). However, there was no established secure way to exchange the key and safely start a conversation on a channel. These asymmetric cryptosystems provided a solution to the crucial problem of key exchange. These new approaches were based on *hard* problems such as factorizing a number into its two prime factors or solving the discrete logarithm problem.

The scenario radically changed in 1994, when Peter W. Shor developed a new algorithm [46] capable of efficiently solving the aforementioned problems when executed on a quantum machine. Given the high security margins used in modern protocols, such an algorithm will not endanger the current communications, but data might be collected and decrypted when the technology is sufficiently mature (Harvest Now, Decrypt Later—HNDL). Thus, the development of new algorithms, based on different problems, became the primary objective for the cryptographic community. This new field of research was named Post-Quantum Cryptography (PQC).

Public key cryptosystems are the basis of the communications between digital devices and are also commonly used to execute digital signatures. Differently from Shor's algorithm, such new cryptosystems should be developed and deployed on classical machines. In 2016 [36], in order to standardize them and focus the research efforts, the National Institute of Standards and Technology (NIST) initiated a process of selection of new algorithms to counter the advancements of quantum computing. In 2022 [37], NIST called for additional digital signature schemes in order to diversify its portfolio.

This work will focus on one of these new digital signature schemes, CROSS (Codes and Restricted Objects Signature Scheme). CROSS is currently a candidate in the second round of selection began in 2022, together with thirteen other schemes. Specifically, this thesis will focus on side-channel attacks against the implementation of CROSS meant to

run on embedded devices.

These devices are commonly used to apply a digital signature; hence, they are the primary targets of side-channel attacks. Such attacks do not focus directly on the cryptosystem (unlike cryptanalysis), but they are designed and executed against its implementations. Once implemented, side-channel attacks might be relatively cheap to conduct, and they could be extremely effective (i.e. they can lead to a complete key recovery [30]); thus, they are widely studied in both academia and industry.

This thesis will start by providing a comprehensive background to understand CROSS, its main concepts, and how they are implemented. CROSS is an articulated protocol based on various concepts. The core of CROSS is the Restricted Syndrome Decoding Problem, a relatively new version of an old problem related to coding theory. The framework that contains such a core is built through the implementation of Zero-Knowledge identification schemes and Fiat-Shamir transformation.

Afterwards, we will conduct an in-depth analysis of the primitives of the protocol in order to find possible targets of attacks. Additionally, we will present some attacks that were already published or that were published contemporaneously with our work.

Once we have identified the most relevant ones, we will analyze them in code by focusing on the Cortex-M4 implementation of CROSS. This implementation relies on ARM architecture and on the Cortex-M family of microprocessors.

Lastly, we developed a new version of the signature primitive in order to counter the most dangerous attacks against it. We will take into account the constraints of embedded devices. Finally, we will also provide some benchmarks in order to analyze the trade-off between the increase of security margin and the changes in performance.

1 | Background

In this chapter we will introduce the required background. We will use the following notation:

- Vectors will be indicated as bold letters: $\mathbf{v}$;
- Matrices will be indicated as bold and capital letters: $\mathbf{H}$;
- Finite fields will be indicated as: $\mathbb{F}_q$, where q is the cardinality of the field; their corresponding multiplicative group will be indicated as $\mathbb{F}_q^*$.
- Vectors of length n having coordinates in a finite field $\mathbb{F}_q$ will be defined as: $\mathbf{v} \in \mathbb{F}_q^n$;
- Matrices of size $n \times k$ having coordinates in a finite field $\mathbb{F}_q$ will be defined as: $\mathbf{H} \in \mathbb{F}_q^{n \times k}$.

1.1. Digital Signature Schemes

A digital signature is a cryptographic transformation of data that guarantees three fundamental properties: authenticity, integrity, and non-repudiation. The FIPS 186-5 guideline published by NIST in 2023 - [38] - identifies three techniques to generate and verify a digital signature.

Rivest-Shamir-Adleman (RSA) Algorithm — RSA is an asymmetric (or public-key) cryptosystem. In an asymmetric cryptoscheme, each counterpart owns a keypair, composed by a private key and a public key. In RSA, as shown in Figure 1.1, Alice signs a message using her private key (1), sends it together with the signature to Bob (2), and the signature is verified (3) using the public key of Alice [42].

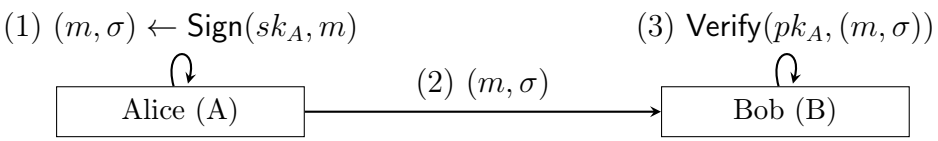

Figure 1.1: Signing a message using RSA.

The private elements, for each keypair, are two prime factors p and q of the same bit size. Their product, n, should have a minimum bit-length; NIST suggests 2048 or 3072 bit-lenghts [39].

A key element is that p and q are secret while $n = p \cdot q$ is public. The recovery of p and q from n cannot be done efficiently (i.e., in polynomial time) by classic machines. However, a polynomial time algorithm developed by Peter Shor [46], meant to run on quantum machines, is able to break RSA. Due to these reasons, RSA is defined as non-quantum safe.

Elliptic Curve Digital Signature Algorithm (ECDSA) — ECDSA [28] is the elliptic-curve variant of Digital Signature Algorithm (DSA). ECDSA is based on the Elliptic Curve Discrete Logarithm Problem [26] where an attacker, given a finite field $\mathbb{F}_q$, an elliptic curve E over $\mathbb{F}_q$ characterized by the equation $y^2 = x^3 + ax + b$, a point $P \in E(\mathbb{F}_q)$ (that generates a subgroup having a large prime order n, $|\langle P \rangle| = n$) and a point $Q \in \langle P \rangle$, has to find the integer $l \mid 0 \leq l \leq n-1$ such that $Q = [l]P$. The ECDLP is a special case of DLP (Discrete Logarithm Problem) in which an attacker, given a cyclic group G, a generator g of G (hence $G = \langle g \rangle$) and an element $y \in G$, has to find x such that $g^x = y$. To compute a valid signature, the signer has to sample, using a CSPRNG (Cryptographically Secure Pseudo-Random Number Generator), an integer $k \mid 1 \leq k \leq n-1$.

The keypair is composed of the public element Q and the private element l, however the underlying problem of ECDSA is fundamentally broken by the Shor's algorithm, similarly to RSA, hence it is categorized as non-quantum safe.

Edwards-Curve Digital Signature Algorithm (EdDSA) — EdDSA can be considered a descendant of ECDSA as they share the same underlying problem (thus they share the same vulnerabilities against Shor's algorithm). However, EdDSA implements a different family of elliptic curves (i.e., Edwards Curves characterized by the equation $x^2 + y^2 = 1 + dx^2y^2$, specifically Twisted Edwards Curve where the equation is generalized as $ax^2 + y^2 = 1 + dx^2y^2$). Furthermore, Edwards curves provide a unified formula for sum and duplication of points. A commonly used curve is $-x^2 + y^2 = 1 - \frac{121665}{121666}x^2y^2$.

Security properties — Security properties are a crucial characteristic of a signature scheme. We provide a description of two of them: sEUF-CMA (Strong Existential UnForgeability under Chosen Message Attacks, often called simply SUF-CMA) and EUF-CMA (Existential UnForgeability under Chosen Message Attacks) security; as they will be particularly relevant for CROSS. We define:

Adversary ($\mathcal{A}$)		**Challenger** ($\mathcal{C}$)
	for $i = 1, \ldots, n$	
$\mathcal{A}$ chooses m_i	$\xrightarrow{m_i}$	Reception of message
		$M \leftarrow M \cup \{m_i\}$
Reception of signature	$\xleftarrow{\sigma_i}$	$\sigma_i \leftarrow \mathsf{Sign}(m_i)$
Creation of the final pair	$\xrightarrow{(m^*, \sigma^*)}$	Reception of the final pair
		$\mathsf{Verify}(pk, (m^*, \sigma^*)) = \texttt{true} \wedge m^* \notin M$

Figure 1.2: The relationship between the adversary and the challenger (EUF-CMA).

- A challenger ($\mathcal{C}$) that generates a keypair (sk, pk) and that provides pk to the adversary;
- An adversary ($\mathcal{A}$) that wants to execute the forgery.

The adversary can ask for signatures on messages that he chooses (i.e., by sending m_i to the challenger he will receive σ_i). Note that the requests may not be done all at once. A scheme is defined as EUF-CMA if no adversary has a non-negligible advantage in assembling a pair (m^*, σ^*) such that:

1. m^* has not been sent to the challenger;
2. (m^*, σ^*) correctly verifies using pk.

sEUF-CMA is more restrictive. Given the aforementioned setup, a scheme is defined as sEUF-CMA if no adversary has a non-negligible advantage in assembling a pair (m^*, σ^*) such that:

1. (m^*, σ^*) was not provided by the challenger;
2. (m^*, σ^*) correctly verifies using pk.

A representation of the two scenarios above is presented in Figure 1.2 and Figure 1.3.

1.2. Post-Quantum Digital Signature Schemes

In December 2016, NIST opened the Post-Quantum Cryptography (PQC) Standardization Process [36]. Three signature schemes were selected as standards: CRYSTALS-

Adversary ($\mathcal{A}$)		**Challenger** ($\mathcal{C}$)
	for $i = 1, \ldots, n$	
$\mathcal{A}$ chooses m_i	$\xrightarrow{m_i}$	Reception of message
		$\sigma_i \leftarrow \mathsf{Sign}(m_i)$
Reception of signature	$\xleftarrow{\sigma_i}$	$S \leftarrow S \cup \{(m_i, \sigma_i)\}$
Creation of the final pair	$\xrightarrow{(m^*, \sigma^*)}$	Reception of the final pair
		$\mathsf{Verify}(pk, (m^*, \sigma^*)) = \mathtt{true} \wedge (m^*, s^*) \notin S$

Figure 1.3: The relationship between the adversary and the challenger (sEUF-CMA).

Dilithium (ML-DSA), Falcon (FN-DSA) and SPHINCS$^+$. Both ML-DSA and FN-DSA are based on structured lattices, in order to diversify the variety of accepted signature schemes a new round of selection began in September 2022 [37]. Fourteen schemes succeeded in the first two rounds of selections based on six different mathematical problems.

Code-based Signatures — These schemes rely on coding theory, a branch of mathematics originally applied in telecommunication as it provides efficient methods to introduce redundancy and error-correction in communication. CROSS [4] is part of this family of schemes, and it is based on Restricted Syndrome Decoding Problem.

Isogeny Signatures — The mathematical foundation is built upon the relationship between Supersingular Elliptic Curves and Quaternion Algebras. The only scheme that relies on this class of problems is SQISign [1].

Lattice-based Signatures — A lattice is the set of all integer linear combinations of vectors from a basis. HAWK [13], the only remaining scheme on this family, relies on Lattice Isomorphism Problem (LIP). This problem consists, given two isomorphic lattices, in the identification of such isomorphism.

MPC-in-the-Head — MPCitH (Multi-Party Computation-in-the-Head) is a family of techniques that transforms a multi-party computation protocol into a zero-knowledge protocol for message signatures. It can be seen as a way to condense a distributed protocol in a unique prover that has to prove the validity of a statement without revealing the secret content. MPCitH proved to be a successful paradigm, and it has been declined in many ways such as SDitH (Syndrome Decoding in the Head) [33], MiRitH (MinRank in the

Head) later merged in Mirath [2], and MQOM (Multivariate Quadratics on my Mind) [10].

Multivariate Signatures — These digital signatures schemes rely on the hardness of solving systems of non-linear equations over finite fields. The hash-and-sign variants of these schemes typically provide small signatures at the cost of big public keys.

Symmetric-based Signatures — These schemes leverage symmetric cryptographic blocks (e.g., like the ones implemented in AES). FAEST [8], the only representative of symmetric-based signature schemes, combines AES and VOLEitH (Vector Oblivious Linear Evaluation in the Head) in its structure.

1.3. Zero-Knowledge Protocols

As defined by Goldreich in [24] a proof is a randomized protocol between a prover $\mathcal{P}$ and a verifier $\mathcal{V}$ that has to preserve two properties. The first one is the completeness property, $\mathcal{P}$ should be able to convince $\mathcal{V}$ of the validity of any true assertion. The second one is soundness, $\mathcal{P}$ should not be able to force $\mathcal{V}$ into accepting any false assertion. Such properties should hold with high probability, hence an error probability, if negligible, is admitted.

Zero-Knowledge (ZK) is a property that holds if the interaction between $\mathcal{P}$ and $\mathcal{V}$ does not provide any knowledge apart from the fact that an assertion is true. In our scenario we will discuss ZK identification schemes where the communication typically includes a secret key (sk) known only by $\mathcal{P}$ and a public key (pk) known by everyone. If $\mathcal{P}$ has to convince $\mathcal{V}$ that he knows sk, the scheme becomes a ZKPOK (Zero-Knowledge Proof Of Knowledge).

A ZK protocol typically consists of multiple passes, in Figure 1.4 we show an example of a five-passes ZK protocol. In Figure 1.4 $\mathcal{V}$ proposes two distinct challenges to $\mathcal{P}$, by providing the correct answer to these challenges $\mathcal{P}$ is able to convince $\mathcal{V}$ of the asserted statement. The commitment (cmt) is usually randomized in order to avoid attacks that leverage repetition of previous communications between the counterparts. If we define ϵ as the soundness error, the probability that a malicious prover is able to successfully convince a verifier is ϵ^t where t is the number of rounds.

1.4. Fiat-Shamir Transformation

In Section 1.3 we introduced the concept of ZK identification schemes. Such schemes were presented as interactive, hence $\mathcal{P}$ and $\mathcal{V}$ must be online at the same time in order to

Prover $(\mathcal{P})$		**Verifier** $(\mathcal{V})$
$cmt = f_0(sk)$	$\xrightarrow{I}$	Reception of commitment
Reception of first challenge	$\xleftarrow{II}$	Sampling of ch_1
$f_1(sk, cmt, ch_1) = resp_1$	$\xrightarrow{III}$	Reception of first response
Reception of second challenge	$\xleftarrow{IV}$	Sampling of ch_2
$f_2(sk, cmt, ch_1, resp_1, ch_2) = resp_2$	$\xrightarrow{V}$	Reception of second response
		$\mathsf{Verify}\ (pk, cmt, ch_1, resp_1, ch_2, resp_2)$

Figure 1.4: An example of a five-passes ZK identification protocol.

make the protocol work. Fiat and Shamir, in [21], presented a solution to this problem by turning an originally interactive protocol into a proper digital signature scheme. This is done by moving the challenge sampling from $\mathcal{V}$ to $\mathcal{P}$. In order to be effective the challenges have to be equally sampled by $\mathcal{P}$, this is achieved using a hash function. $\mathcal{P}$ has no control on the hash function output and this is equivalent to the reception of random challenges from a verifier.

In Figure 1.5 we apply the Fiat-Shamir transformation on the scheme presented in Figure 1.4. $\mathcal{P}$ will compute the first challenge by feeding the commitment into a hash function and the second one similarly, by feeding into a hash function the commitment, the first challenge, and the first response. Finally, it will send the commitment and the two responses to $\mathcal{V}$ that will recompute the challenges and check that everything matches with the public key of $\mathcal{P}$.

However, a digital signature scheme is useful only if we are able to attach a signature to a message. $\mathcal{P}$ might execute this simple transformation:

$$ch_1 = cmt \rightarrow ch_1 = \mathsf{Hash}(cmt \mid msg)$$

By doing so the message is now a part of the scheme. A real protocol should include a salt, omitted in the presented scheme. The salt introduces randomness and counter replay attacks. The final scheme is presented in Figure 1.6

Lastly, we introduce the concept of $q2$-Identification scheme. A ZK protocol is classified as a $q2$-Identification scheme when it is characterized by the following properties:

Prover ($\mathcal{P}$)		**Verifier** ($\mathcal{V}$)
$cmt = f_0(sk)$		
$ch_1 = \mathsf{Hash}(cmt)$		
$resp_1 = f_1(sk, cmt, ch_1)$		
$ch_2 = \mathsf{Hash}(cmt, ch_1, resp_1)$		
$resp_2 = f_2(sk, cmt, ch_1, resp_1, ch_2)$		
	$\xrightarrow{cmt, resp_1, resp_2}$	
		$ch_1 = \mathsf{Hash}(cmt)$
		$ch_2 = \mathsf{Hash}(cmt, ch_1, resp_1)$
		$\mathsf{Verify}\ (cmt, ch_1, resp_1, ch_2, resp_2)$

Figure 1.5: Fiat-Shamir transformation applied to a five-passes ZK protocol.

Prover ($\mathcal{P}$)		**Verifier** ($\mathcal{V}$)
$cmt = f_0(sk)$		
$ch_1 = \mathsf{Hash}(cmt \mid msg)$		
$resp_1 = f_1(sk, cmt, ch_1)$		
$ch_2 = \mathsf{Hash}(cmt, ch_1, resp_1)$		
$resp_2 = f_2(sk, cmt, ch_1, resp_1, ch_2)$		
	$\xrightarrow{cmt, resp_1, resp_2, msg}$	
		$ch_1 = \mathsf{Hash}(cmt \mid msg)$
		$ch_2 = \mathsf{Hash}(cmt, ch_1, resp_1)$
		$\mathsf{Verify}\ (cmt, ch_1, resp_1, ch_2, resp_2)$

Figure 1.6: A digital signature scheme based on Fiat-Shamir transformation.

1. $|C_1|$ (i.e., cardinality of challenge 1) $= q$;
2. $|C_2|$ (i.e., cardinality of challenge 2) $= 2$;

We will see that CROSS respects these properties. It is proven that the application of the Fiat-Shamir transformation over parallel executions of a $q2$-Identification scheme translates into a signature scheme that is EUF-CMA secure [16].

1.5. Restricted Syndrome Decoding Problem

The main mathematical foundation of CROSS is the (Restricted) Syndrome Decoding Problem (R-SDP). However, before introducing it, we provide a brief overview of coding theory and linear codes as this knowledge is required to better understand the R-SDP.

Coding theory — Codes were originally developed as a way to correct errors on noisy channels [32]. A basic scheme is shown in Figure 1.7. We provide some basic definitions [43] that we will use later.

Definition 1.1 (Code). *A (n, k) code over a finite alphabet F is a non-empty subset $\mathcal{C} \subseteq F^n$ having size $|F|^k$. The parameter n is called the code length while k is the code dimension. The elements of a code are called codewords.*

A way to quantify the distance between codewords is required. In our case we will use the concept of *Hamming distance* or *Hamming weight* (wt). The wt associated to a vector is the number of non-zero coordinates of such vector. The Hamming distance can be used to quantify the distance between two vectors as:

$$d(\mathbf{x}, \mathbf{y}) = \mathrm{wt}(\mathbf{x} - \mathbf{y})$$

We will denote a code as (n, k, d) where d is the minimum distance between the codewords of $\mathcal{C}$.

Definition 1.2 (Linear code). *Given a finite field $\mathbb{F}_q$, a (n, k, d) code $\mathcal{C}$ over $\mathbb{F}_q$, is called linear if $\mathcal{C}$ is a linear subspace of $\mathbb{F}_q^n$ over $\mathbb{F}_q$. That is, if for every two codewords $\mathbf{x}_1, \mathbf{x}_2 \in \mathcal{C}$ and two scalars $a_1, a_2 \in \mathbb{F}_q$ we have $a_1\mathbf{x}_1 + a_2\mathbf{x}_2 \in \mathcal{C}$.*

Usually, we will omit the specification of the minimum distance. Hence, we will represent a code as a (n, k) linear code over $\mathbb{F}_q$. The difference $n - k$ is called *redundancy* of $\mathcal{C}$.

For any codeword $\mathbf{x} \in \mathcal{C}$ it should hold:

$$\mathbf{x}\mathbf{H}^\top = 0$$

The matrix $\mathbf{H}$ is called *parity-check matrix*. To obtain a codeword given a message we use a *generator matrix* $\mathbf{G}$ defined as a matrix whose rows form a basis of the code. Then, given the message $\mathbf{u}$ the corresponding codeword is obtained as follows:

$$\mathbf{x} = \mathbf{u}\mathbf{G}$$

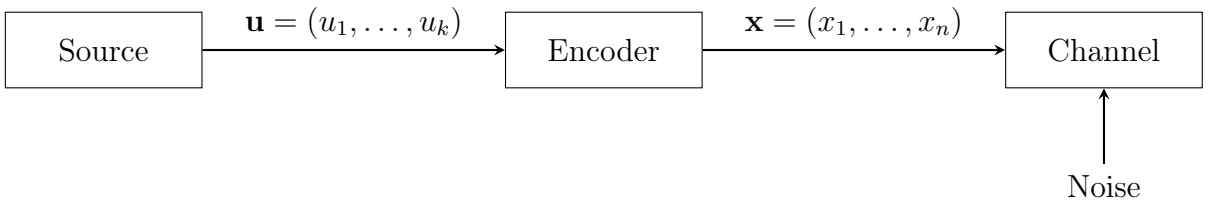

Figure 1.7: Communication over a channel using codes.

$\mathbf{G}$ and $\mathbf{H}$ are related by the following identity:

$$\mathbf{H}\mathbf{G}^\top = 0$$

Now we can define the noise as an *error vector* $\mathbf{e} = (e_1, \ldots, e_n)$. The error vector is additive, hence the receiver will receive a vector $\mathbf{y} = \mathbf{x} + \mathbf{e}$. The concepts we have introduced can be extended to any finite field $\mathbb{F}$ of cardinality q (i.e., $\mathbb{F}_q$); in such case the vectors $(\mathbf{u}, \mathbf{x}, \mathbf{e}, \mathbf{y})$ and the matrices $(\mathbf{H}, \mathbf{G})$ have components in $\mathbb{F}_q$ and the aforementioned addition will be executed modulo q.

Syndrome — The syndrome is the vector computed by a receiver via the following identity:

$$\mathbf{s} = \mathbf{y}\mathbf{H}^\top$$

Thus, the syndrome is a vector of length $n - k$. Straightforwardly, the syndrome is zero if and only if $\mathbf{y}$ is a codeword, hence if no errors of bounded weight occurred:

$$\mathbf{s} = \mathbf{y}\mathbf{H}^\top = (\mathbf{x} + \mathbf{e})\mathbf{H}^\top = \mathbf{x}\mathbf{H}^\top + \mathbf{e}\mathbf{H}^\top = \mathbf{e}\mathbf{H}^\top$$

This identity will be crucial for understanding the properties of CROSS:

$$\mathbf{s} = \mathbf{e}\mathbf{H}^\top$$

Syndrome Decoding Problem — Before introducing Restricted Syndrome Decoding Problem (R-SDP) we provide a definition of the basic form:

Definition 1.3 (Syndrome Decoding Problem). *Given a parity matrix* $\mathbf{H} \in \mathbb{F}_p^{(n-k)\times n}$*, a syndrome* $\mathbf{s} \in \mathbb{F}_p^{n-k}$*, identify the error vector* $\mathbf{e} \mid wt(\mathbf{e}) \leq t$ *such that* $\mathbf{s} = \mathbf{e}\mathbf{H}^\top$*.*

Usually, with SDP, we will define the decisional version of such problem:

Definition 1.4 (Decisional Syndrome Decoding Problem). *Given a parity matrix* $\mathbf{H} \in \mathbb{F}_p^{(n-k)\times n}$*, a syndrome* $\mathbf{s} \in \mathbb{F}_p^{n-k}$*, decide if an error vector* $\mathbf{e} \mid wt(\mathbf{e}) \leq t$ *exists, such that*

$\mathbf{s} = \mathbf{e}\mathbf{H}^\top$.

With the condition $wt(\mathbf{e}) \leq t$ we limit the error vectors to the ones that have a Hamming weight of less than (or equal to) t. The SDP is proven to be NP-hard in [11].

Restricted Syndrome Decoding Problem — In [5, 6], Baldi et al. presented a variation of such problem, called Restricted Syndrome Decoding Problem. In addition to the previous concepts we introduce a subgroup $\mathbb{E}$ generated by an element $g \in \mathbb{F}_p^*$. Thus, $\mathbb{E} = \{g^i \mid i \in \{1, \dots, z\}\} = \langle g \rangle \subset \mathbb{F}_p^*$.

Definition 1.5 (Restricted Syndrome Decoding Problem – R-SDP)**.** *Given a parity matrix* $\mathbf{H} \in \mathbb{F}_p^{(n-k)\times n}$*, a syndrome* $\mathbf{s} \in \mathbb{F}_p^{n-k}$*, a subgroup* $\mathbb{E} = \{g^i \mid i \in \{1, \dots, z\}\} \subset F_p^*$ *having* $g \in \mathbb{F}_p^*$ *as generator, decide if an error vector* $\mathbf{e} \in \mathbb{E}^n \mid wt(\mathbf{e}) \leq t$ *exists, such that* $\mathbf{s} = \mathbf{e}\mathbf{H}^\top$.

The R-SDP can be seen as a special case of the basic SDP, in fact, if $\mathbb{E} = \mathbb{F}_p^*$ then the two problems converge to the same one. The R-SDP is proven to be NP-hard in [5]. Now let's consider the group $(\mathbb{E}^n, \star)$, where $\star$ is the component-wise vector multiplication. Such group is abelian and transitive (as it inherits the properties from the $\star$ operations) and by defining the following relation:

$$l : \mathbb{E}^n \to \mathbb{F}_z^n$$
$$\mathbf{x} = (g^{l_1}, \dots, g^{l_n}) \mapsto l(\mathbf{x}) = (l_1, \dots, l_n)$$

we can prove the existence of an isomorphism between $(\mathbb{E}^n, \star)$ and $(\mathbb{F}_z^n, +)$. This new representation provides two advantages. The first one is the reduction of the representation size of a vector from $n\lceil \log_2(p) \rceil$ to $n\lceil \log_2(z) \rceil$ making it more compact. The second one is that multiplications in $\mathbb{F}_p$ become additions in $\mathbb{F}_z$. The counterpart of a vector $\mathbf{v} \in \mathbb{E}^n$, hence $l(\mathbf{v})$, will be indicated as $\overline{\mathbf{v}} \in \mathbb{F}_p^n$ and will often be indicated as "exponent" vector.

We define a final variant, starting from the definition of the subgroup $(G, \star) \leq (\mathbb{E}^n, \star)$ as:

$$G = \langle \mathbf{a}_1, \dots, \mathbf{a}_m \rangle = \left\{ \prod_{i=1}^{m} \mathbf{a}_i^{u_i} \mid u_i \in \{1, \dots, z\} \right\}$$

with $m < n$. Informally, we are picking m vectors from $\mathbb{E}^n$ and using them as generators of G. From these concepts we define the Restricted Syndrome Decoding Problem with a Subgroup G (R-SDP(G)).

Definition 1.6 (Restricted Syndrome Decoding Problem in a Subgroup – R-SDP(G))**.** *Given a parity matrix* $\mathbf{H} \in \mathbb{F}_p^{(n-k)\times n}$*, a syndrome* $\mathbf{s} \in \mathbb{F}_p^{n-k}$*, a subgroup* $\mathbb{E} = \{g^i \mid i \in$

$\{1,\dots,z\}\} \subset F_p^*$ *having* $g \in \mathbb{F}_p^*$ *as generator, and a group* $G = \langle \mathbf{a}_1,\dots,\mathbf{a}_m\rangle$ *(*$\forall i \in [1,m], \forall \mathbf{a}_i \in \mathbb{E}^n$*) decide if an error vector* $\mathbf{e} \in G$ *exists, such that* $\mathbf{s} = \mathbf{e}\mathbf{H}^\top$.

This problem is still NP-hard. The representation size is even more compact as only $m\lceil \log_2(z)\rceil$ bits are required to represent a vector. In CROSS we will widely use a generator matrix for G, called $\mathbf{M}_G \in \mathbb{F}_z^{m\times n}$, defined as follows:

$$\mathbf{M}_G = \begin{pmatrix} l(\mathbf{a}_1)_1 & \cdots & l(\mathbf{a}_1)_n \\ \vdots & \ddots & \vdots \\ l(\mathbf{a}_m)_1 & \cdots & l(\mathbf{a}_m)_n \end{pmatrix} = \begin{pmatrix} l(\mathbf{a}_1) \\ \vdots \\ l(\mathbf{a}_m) \end{pmatrix}$$

The sampling of $\mathbf{e}$ is equivalent of sampling it from the usual field $\mathbb{F}_z^m$ as we can use the following group isomorphism:

$$l_G : G \to \mathbb{F}_z^m$$
$$\mathbf{e} = \mathbf{a}_1^{u_1} \star \cdots \star \mathbf{a}_m^{u_1} \mapsto l_G(\mathbf{e}) = (u_1, \dots u_m)$$

hence, sampling $\mathbf{e} \in G$ is equivalent to sampling $l_G(\mathbf{e}) \in \mathbb{F}_z^m$. This family of problems is the core of CROSS, and together with the concepts of ZK identification schemes and Fiat-Shamir transformation we are able to build a signature scheme.

1.6. Keccak - SHAKE

Keccak is a family of sponge functions that use one of seven possible permutation as its building block [12]. A sponge function is a function characterized by an internal state that maps input strings of any length to output strings of desired length. Keccak, as a sponge function, is characterized by an absorbing phase and a squeezing phase. During the absorbing phase the input is processed and mixed into the state while in the squeezing phase the output is extracted from the state by taking into account its length. A high-level pseudocode is provided in Algorithm 1.

SHAKE (Secure Hash Algorithm Keccak) is a primitive defined by NIST standard FIPS202 [19] as part of extendable-output family of function (XOFs). Specifically, two variants, SHAKE128 and SHAKE256 were approved. In CROSS, SHAKE is employed as a cryptographic hash function and as a Cryptographically Secure Pseudo-Random Number Generator (CSPRNG). To prevent collisions between the two uses, a domain separator is introduced.

Keccak/SHAKE became even more relevant in CROSS during the development of Cortex-

Algorithm 1: The sponge construction $[f, \text{pad}, r]$ [12]

Input: $M \in \{0,1\}^*$, $\ell > 0$
Output: $C \in \{0,1\}^l$
Data: r: bitrate, b: width of permutation f

// 0. Initialization of the state
$P \leftarrow M \,||\, \text{pad}[r](|M|)$
$s \leftarrow \underbrace{0 \cdots 0}_{b}$
// 1. Absorbing Phase
for $i = 0$ *to* $|P|_r - 1$ **do**
    $s \leftarrow s \oplus (P_i \,||\, \underbrace{0 \cdots 0}_{b-r})$
    $s \leftarrow f(s)$
// 2. Squeezing Phase
$C \leftarrow s[0:r]$
while $|C|_r r < \ell$ **do**
    $s \leftarrow f(s)$
    $C \leftarrow C \,||\, s[0:r]$
return $C[0:\ell]$

M4 implementation. In fact, their capabilities of preserving an internal state made them the most indicated tools for incremental hashing optimization, crucial to reduce the memory impact of the algorithm.

1.7. Codes and Restricted Objects Signature Scheme (CROSS)

Now that we have introduced the required concepts we will provide a concise description of how CROSS is structured. CROSS-ID is a ZK, 5-pass protocol turned into a digital signature scheme via Fiat-Shamir transformation. Its first challenge ($\texttt{chall}_1$ from now on) is sampled from the multiplicative group $\mathbb{F}_p^*$, hence $\texttt{chall}_1 \in \mathbb{F}_p^*$, while the second challenge ($\texttt{chall}_2$ from now on) is sampled from the set $\{0,1\}$. Due to these reasons, it qualifies as a $q2$-identification scheme; as a consequence the signature is EUF-CMA secure.

We present the protocol in its original form, its implementation is actually different as the group isomorphism presented in Section 1.5 is widely used. A pseudocode of the final implementation, provided by the original authors, will be introduced and commented in the following chapter.

Commitments — The protocol is presented using its R-SDP(G) variant as, by setting

$G = \mathbb{E}^n$, we obtain R-SDP. Using a seed ($\mathtt{Seed} \leftarrow \{0,1\}^\lambda$, where λ is the security parameter), $\mathcal{P}$ computes a random error vector $\mathbf{e}' \in G$ and a vector $\mathbf{u}' \in \mathbb{F}_p^n$ via a CSPRNG; in addition $\mathcal{P}$ computes a map $\mathbf{v} \in G \mid \mathbf{v} \star \mathbf{e}' = \mathbf{e}$. Then the syndrome $\mathbf{s}'$ is computed as follows:

$$\mathbf{s}' = (\underbrace{\mathbf{v} \star \mathbf{u}'}_{\mathbf{u}})\mathbf{H}^\top$$

The two commitments are computed by hashing and sent to $\mathcal{V}$:

$$\mathtt{cmt}_0 = \mathsf{Hash}\left(\mathbf{s}' \mid \mathbf{v}\right)$$
$$\mathtt{cmt}_1 = \mathsf{Hash}\left(\mathbf{u}' \mid \mathbf{e}'\right)$$

First challenge and first response — Once it receives the two commitments, $\mathcal{V}$ randomly samples $\mathtt{chall}_1$ from the multiplicative group $\mathbb{F}_p^*$. Once received the challenge, $\mathcal{P}$ will compute the vector $\mathbf{y}$ as follows:

$$\mathbf{y} = \mathbf{u}' + \mathtt{chall}_1 \mathbf{e}'$$

and, after hashing, it will send the response to $\mathcal{V}$:

$$\mathtt{resp}_1 = \mathsf{Hash}\left(\mathbf{y}\right)$$

Second challenge and second response — The second challenge is sampled by $\mathcal{V}$ from the set $\{0,1\}$ and sent to $\mathcal{P}$. If $\mathtt{chall}_2 = 0$ then $\mathcal{P}$ has to send $(\mathbf{y}, \mathbf{v})$ in order to allow the verification of $\mathtt{cmt}_0$. If $\mathtt{chall}_2 = 1$ then $\mathcal{P}$ has to send $\mathtt{Seed}$ in order to allow the verification of $\mathtt{cmt}_1$. Once received the challenge $\mathcal{P}$ assembles the response:

$$\begin{cases} \mathtt{resp}_2 = (\mathbf{y}, \mathbf{v}) & \text{if } \mathtt{chall}_2 = 0 \\ \mathtt{resp}_2 = (\mathtt{Seed}) & \text{if } \mathtt{chall}_2 = 1 \end{cases}$$

and send to $\mathcal{V}$ that, if $\mathtt{chall}_2 = 0$:

$$\mathbf{y}' = \mathbf{v} \star \mathbf{y}$$
$$\mathbf{s}' = \mathbf{y}'\mathbf{H}^\top - \mathtt{chall}_1 \mathbf{s}$$
$$\text{Accept if } \mathsf{Hash}\left(\mathbf{y}\right) = \mathtt{resp}_1 \wedge \mathsf{Hash}\left(\mathbf{s}' \mid \mathbf{v}\right) = \mathtt{cmt}_0 \wedge \mathbf{v} \in G$$

Hence, $\mathcal{V}$ verifies if the vector $\mathbf{v}$ is restricted (i.e., in the subgroup G) and it checks the

first commitment. If $\texttt{chall}_2 = 1$:

$$(\mathbf{e}', \mathbf{u}') = \mathsf{CSPRNG}(\texttt{Seed})$$
$$\mathbf{y}' = \mathbf{u}' + \texttt{chall}_1 \mathbf{e}'$$
$$\text{Accept if } \mathsf{Hash}(\mathbf{y}) = \texttt{resp}_1 \wedge \mathsf{Hash}(\mathbf{u}' \mid \mathbf{e}') = \texttt{cmt}_1$$

In this case, $\mathcal{V}$ verifies the second commitment.

Fiat-Shamir transformation — The previous protocol is turned into a digital signature scheme using the following logic applied to t rounds of CROSS-ID:

$$\texttt{chall}_1[1 \dots t] = \mathsf{Hash}(\texttt{Msg}, \texttt{cmt}_0[1 \dots t], \texttt{cmt}_1[1 \dots t], \texttt{Salt})$$
$$\texttt{resp}_1[1 \dots t] = \mathsf{Hash}(\mathbf{y}[1 \dots t])$$

While the second challenge is generated by hashing together the aforementioned digests, as shown in Section 1.4

Weight of the challenges and CROSS variants — $\texttt{chall}_2$ is the challenge that has the biggest impact on signature size as the response when $\texttt{chall}_2 = 1$ is much smaller than the response if $\texttt{chall}_2 = 0$. As a consequence, the weight of the vector $\texttt{chall}_2$ (from now on w) is extremely relevant. The protocol is optimized using tree structures, with improvements depending on the protocol variant.

If w is close to t then the responses are small and the protocol can be optimized by using a Merkle tree for the commitments and a seed tree for the seeds. This choice of parameters leads to CROSS-small (optimized for signature size) and CROSS-balanced.

If $w \sim \frac{t}{2}$, the same optimizations do not lead to significant advantages and new tree structures are introduced. This leads to CROSS-fast (optimized for speed).

To provide a complete picture of the protocol we include a description of the parameters in Table 1.1.

1.8. Tree Structures

In the following section we present the tree structures used by CROSS. As we will see in the next chapter their knowledge is important as they will be susceptible to active side channel attacks.

Seed trees — In CROSS, seed trees (also known as GGM-PRF (Goldreich-Goldwasser-Micali-Pseudo-Random Functions) trees [25]) are used to derive t round seeds $\texttt{Seed}[i]$

Table 1.1: Parameter choices, keypair and signature sizes recommended for both CROSS-R-SDP and CROSS-R-SDP(G), assuming NIST categories 1, 3, and 5, respectively.

Algorithm and Security Category	Optim. Corner	p	z	n	k	m	t	w	Pri. Key Size (B)	Pub. Key Size (B)	Signature Size (B)
CROSS-R-SDP **1**	**fast**	127	7	127	76	-	157	82	32	77	18432
	balanced	127	7	127	76	-	256	215	32	77	13200
	small	127	7	127	76	-	520	488	32	77	12480
CROSS-R-SDP **3**	**fast**	127	7	187	111	-	239	125	48	115	41406
	balanced	127	7	187	111	-	384	321	48	115	29925
	small	127	7	187	111	-	580	527	48	115	28463
CROSS-R-SDP **5**	**fast**	127	7	251	150	-	321	167	64	153	74590
	balanced	127	7	251	150	-	512	427	64	153	53623
	small	127	7	251	150	-	832	762	64	153	50914
CROSS-R-SDP(G) **1**	**fast**	509	127	55	36	25	147	76	32	54	11980
	balanced	509	127	55	36	25	256	220	32	54	9168
	small	509	127	55	36	25	512	484	32	54	9008
CROSS-R-SDP(G) **3**	**fast**	509	127	79	48	40	224	119	48	83	26772
	balanced	509	127	79	48	40	268	196	48	83	22536
	small	509	127	79	48	40	512	463	48	83	20524
CROSS-R-SDP(G) **5**	**fast**	509	127	106	69	48	300	153	64	106	48102
	balanced	509	127	106	69	48	356	258	64	106	40196
	small	509	127	106	69	48	642	575	64	106	36550

starting from a unique `Seed`. They are computed using a top-down technique in which the root is a `Seed`. The children are computed by splitting the output of a CSPRNG, having the `Seed` as input; the procedure is repeated, using the children as input, until we reach the desired number of leaves. In CROSS, seed trees are used to compress the elements required in the signature by computing a `Path`. Such `Path`, in balanced and small versions, contains only the nodes required to re-compute the `Seed`$[i]$ needed by $\mathcal{V}$, hence only the ones such that $\texttt{chall}_2 = 1$. In the fast version, `Path` contains directly the required `Seed`$[i]$.

Merkle trees — Merkle trees play a crucial role in CROSS as they are an efficient solution to compress the commitments associated to each round. This reduces the size of the final signature, at the cost of a small overhead during the recomputation of commitments from `Proof`. CROSS requires two commitments in each round, only $\texttt{cmt}_0$ are compressed using a Merkle tree. The Merkle tree is built using a bottom-up approach starting from the leaves where a commitment is stored in each leaf. The recursive construction process revolves around a call to the hashing primitive.

Usage in post-quantum signature schemes — Seed/GGM trees are widely used in post-quantum signature schemes and in general in ZK proofs. For instance, Mirath (specifically optimized with BAVC (Batch All-but-one Vector Commitment) abstraction

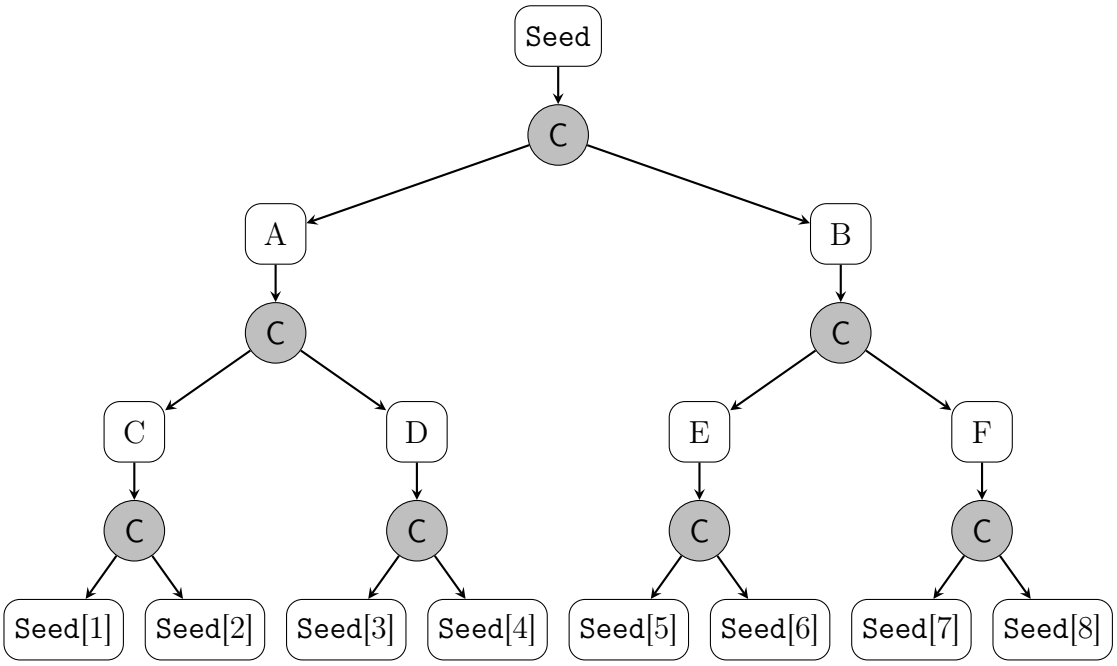

Figure 1.8: An example of a seed tree.

described in [9]) - [2] - and SDitH - [33] - use TCitH-GGM approach, starting from the TCitH framework presented by Feneuil and Rivain in [20]. As a consequence, an in-depth analysis of such structure is relevant not only for CROSS but also for other signature schemes.

Merkle trees are the core of hash-based signature schemes (a comprehensive overview is presented by Pacurar et al. in [40]). Originally and independently developed by Lamport - [31] - and Merkle - [34] - this framework (with various modifications) is still in use by SPHINCS$^+$ family of schemes. However, some weaknesses against active attacks and fault injections were shown by Castelnovi, Martinelli and Prest in [15] and by Genêt et al. in [23] while countermeasures were presented by Genêt in [22]. In CROSS, Merkle trees are not the core concept of the scheme, but they are used as a way to compress signature size and optimize verification.

Trees in CROSS — In CROSS, two distinct types of trees are used accordingly to the chosen variant of CROSS (i.e., balanced/small or fast). In general, a seed tree is a top-down structure having round seeds on its leaves while a Merkle tree is a bottom-up structure having commitments on its leaves. An example of a seed tree is visible in Figure 1.8 while an example of a Merkle tree is visible in Figure 1.9. CROSS differs in the actual structure of the trees, as they are not always complete.

Tree structure in balanced/small variant of CROSS — In balanced/small version,

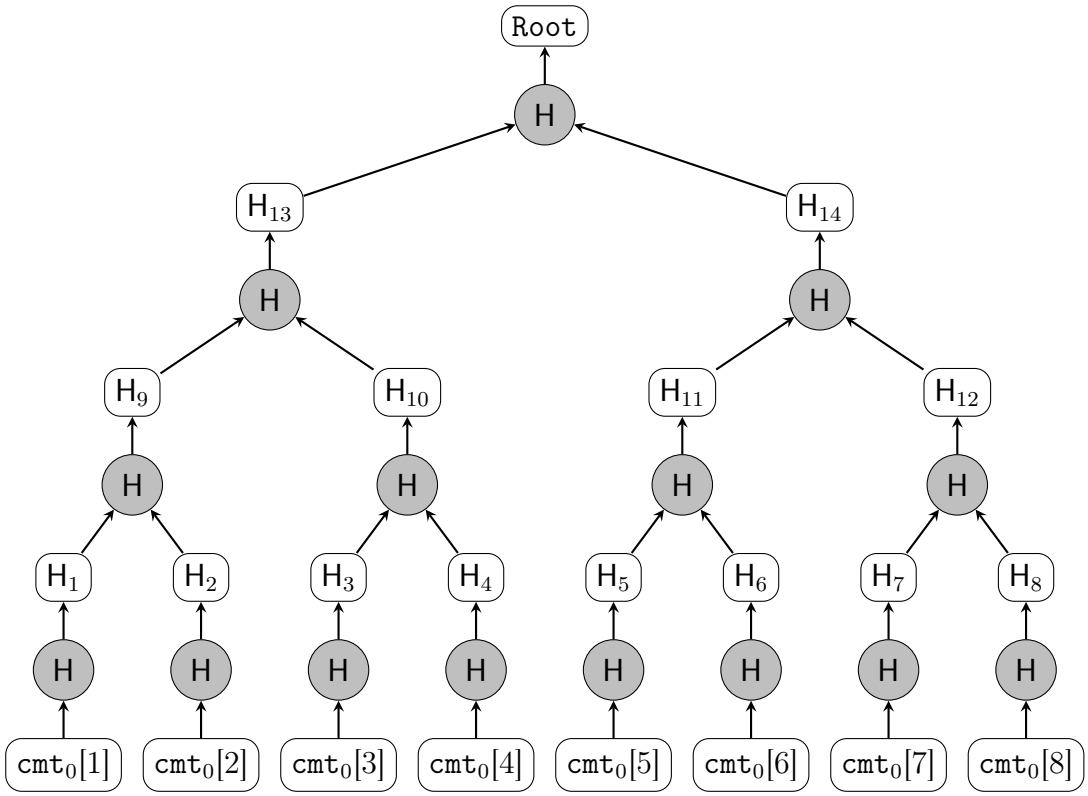

Figure 1.9: An example of a Merkle tree.

CROSS operates with a conventional binary tree having $2t - 1$ nodes. However, t is not always a power of two, hence the tree itself will not always be complete. An example of a tree, for $t = 7$ is in Figure 1.10:

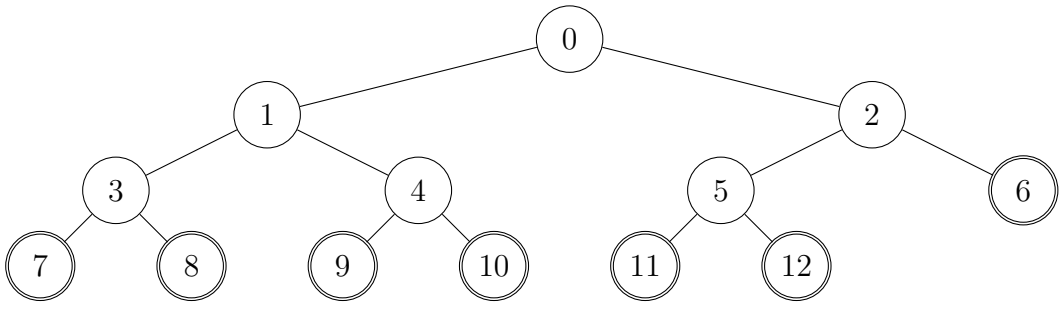

Figure 1.10: Tree for balanced/small version of CROSS having $t = 7$.

The tree is constructed in a way such that it is composed by multiple binary subtrees. To efficiently operate on such tree, CROSS uses five pre-computed arrays (that depend only on t):

- `npl`: nodes per level;
- `lpl`: leaves per level;

- `off`: for offsets, used to compute the parent/child index when moving between levels;
- `lsi`: leaves start indices, it provides the indices of the leftmost leaf for each level;
- `ncl`: number of consecutive leaves.

In case of Figure 1.10:

- `lsi` $\rightarrow [7, 6]$, in fact the index 7 is the index of the leftmost leaf in the last level and 6 is the index of the leftmost leaf of the second level.
  Levels without leaves are not represented. These values can be found by recursively searching for subtrees left-to-right and by keeping track of the depth.
- `ncl` $\rightarrow [6, 1]$.

Tree structure in fast variant of CROSS — In the fast version of CROSS, a more compact tree is used. Such tree is characterized by a root, four intermediate nodes and t leaves equally assigned to the intermediate nodes, hence it is no longer a binary tree. Such intermediate node will have at most $\lfloor t/4 \rfloor$ children. The first three intermediate nodes could have one additional node if:

- Node 1 $\rightarrow t \mod 4 \geq 1$:
- Node 2 $\rightarrow t \mod 4 \geq 2$:
- Node 3 $\rightarrow t \mod 4 \geq 3$:

An example of a tree for fast version of CROSS is visible in Figure 1.11.

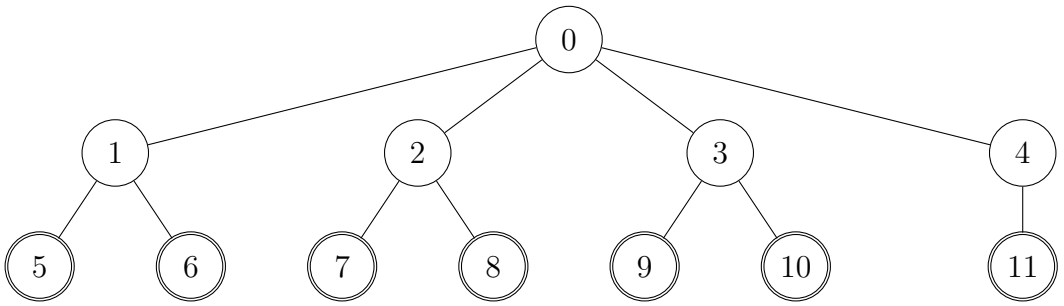

Figure 1.11: Tree for fast version of CROSS having $t = 7$.

In fast version of CROSS, `Path` and `Proof` will not be used as a way to "compress" the information but instead as a way to recover the information in a quicker way.

In the code the usage of either balanced/small or fast version of the algorithms is selected by the use of #`defined`(`flag`) preprocessor directives. This removes runtime selection paths and, marginally, the code size.

1.9. ARM Cortex-M4

The ARM Cortex-M4 [3] is a processor designed for embedded applications. It features low gate count and low interrupt latency.

The Cortex-M is a family of 32-bit RISC ARM processors. Its main difference from the Cortex-A family (widely used on smartphones) is the absence of a MMU (i.e., Memory Management Unit). Hence, it has no support for virtual memory.

Processor features — We provide a brief overview of the ARM Cortex-M4 features ([3]) that are most relevant for the Cortex-M4 implementation of CROSS.

Cortex-M4 allows the usage of SIMD (Single Instruction Multiple Data) directives. These instructions allow the processor to perform an operation over multiple elements packed in a single 32-bit register. This is achieved by dividing a single 32-bit register into either two 16-bit ones or four 8-bit ones. For instance, the instruction `UADD8` executes four 8-bit unsigned additions. More complex instructions are available like `SMLAD` (Signed Multiply Accumulate Dual) that performs two Signed Multiply Accumulate operations on 16-bit operands. All these operations are executed in a single clock cycle.

The main units involved in the arithmetic of CROSS are the ALU (Arithmetic Logic Unit) and the DSP (Digital Signal Processing) extension. The ALU allows the basic arithmetic (32-bit addition, subtraction, and multiplication) and it provides a hardware divide unit via the `SDIV`/`UDIV` instructions. The DSP allows for the SIMD logic presented below and the MACs units (Multiply-Accumulate).

Furthermore, the Cortex-M4 provides the DWT (Data Watchpoint and Trace) unit that is particularly useful for debugging and benchmarking. The DWT provides many counters to track the performance of the processor, like the `CYCCNT` (Clock Cycle Counter) or the `LSUCNT` (Load Store Unit Counter) to monitor the load-store operations.

We provide a summary table (Table 1.2) that presents the key elements of the ARM Cortex-M4.

1.10. Changes introduced by Cortex-M4 implementation

Our analysis will focus mainly on Cortex-M4 implementation of CROSS as embedded devices are typically targets of active side channel attacks.

Table 1.2: ARM Cortex-M4 data sheet.

Component	Specification
Architecture	32-bit ARMv7-M
Instruction Set	Thumb instruction set, hardware divide DSP extensions
Registers	13 general-purpose registers (r0-r12), Stack Pointer (SP) Link Register (LR), Program Counter (PC)
Memory Protection	Optional Memory Protection Unit (MPU)

Introduced optimizations — The main focus of the Cortex-M4 implementation is the reduction of the memory footprint of the algorithm. This is achieved by implementing some optimizations, presented by the authors in [45]:

- *Memory sharing*: many elements in CROSS do not need to be kept in memory for the entire duration of the algorithm. Reusing the same memory areas is a relatively simple optimization that does not alter the flow of the program. In addition, $\texttt{cmt}_0[i]$ can be stored directly on the Merkle tree (as they are basically the pre-hashed leaves), hence they do not need a separate structure.
- *Time-memory trade-off*: if we allow the possibility of recomputing some elements then we allow further optimizations. The first one is the introduction of incremental hashing based on three primitives:
  - HashInit: it initializes a given Keccak state, passed as parameter, to zero;
  - HashUpdate: it takes as parameter a Keccak state, an input, and the input length. It updates the state using the absorbing technique shown in Section 1.6;
  - HashFinAndSqueeze: it takes as parameter a Keccak state, an array to store the digest and a domain separator constant. It condenses the finalization and squeezing phase into a given array.

  where the state will revolve around $\texttt{cmt}_1[i]$ or $\mathbf{y}$ accordingly to where in the code the computation is executed. This optimization is not so straightforward as we need to execute some computation multiple times. This translates into the reconstruction of the trees (both seed and Merkle) multiple times during the execution of the algorithm.
- *Changes in how trees are managed*: these optimizations are discussed below in detail.

Management of trees and `Proof`/`Path` in Sign fast — Tree structure does not change from the reference implementation (a unique root, four intermediate nodes and equally distributed children). However, we do not keep track of every `Seed`$[i]$, but we store only the four intermediate nodes. Then:

- If the round index aligns with remainders we sample the first child from the related intermediate nodes;
- If the round index does not align with remainders we use the first child as input of CSPRNG, then the result will be the next child and next input and so on.

A similar approach is used for Merkle tree where, as soon as we have enough commits, we hash them and save only the intermediate result. `Proof`/`Path` behave as in the reference implementation, their computation is only executed later in the algorithm.

Management of trees and `Proof`/`Path` in Sign balanced/small — Seed trees are managed by computing them in a depth-first approach and by storing both children for each level. This reduces the memory used by the tree from $2t - 1$ nodes to $2\log_2(t)$ nodes. The tree is visited using a pre-order visit and by using a static array of $2\log_2(t)$ cells periodically updated. A visual example is showed in Figure 1.12 and 1.13. Merkle trees follow a similar approach but only a single node per level is stored and when the second node is available the algorithm computes the parent and goes up by a level. This approach requires $\log_2(t)$ nodes. Some auxiliary variables are introduced to make this tree management viable. `Proof`/`Path` are computed using some auxiliary functions that exploits a flag tree to find which nodes should be published and which not.

1.11. Side-Channel attacks

Introduction — Side-channel attacks (SCAs) are a family of methods that typically targets cryptographic algorithms. Differently from conventional cryptanalysis, that focuses on flaws in the design of such systems, side-channel attacks target their implementations. SCAs are typically grouped in two categories: passive and active.

In passive attacks, an attacker wants to obtain sensitive information that are inadvertently leaked by the implementation. This can be achieved with a variety of methods, some examples are:

- *Cache attacks*: attacks that target cache memories. An example is Spectre [29], that targeted the speculative execution implemented in modern microprocessor in order to leak data through cache memory.

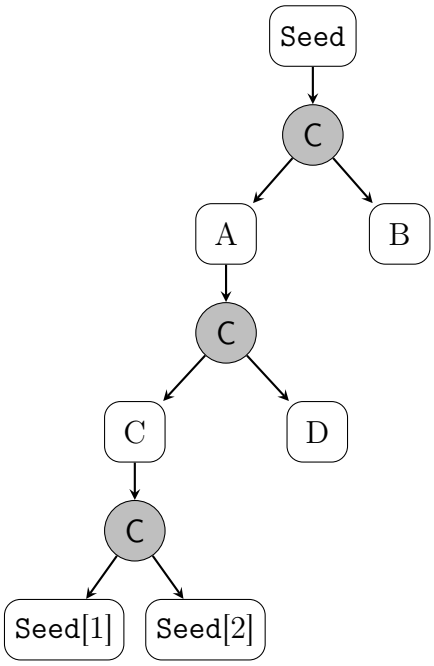

Figure 1.12: An example of which nodes are kept in memory during the first round of Sign for seed trees.

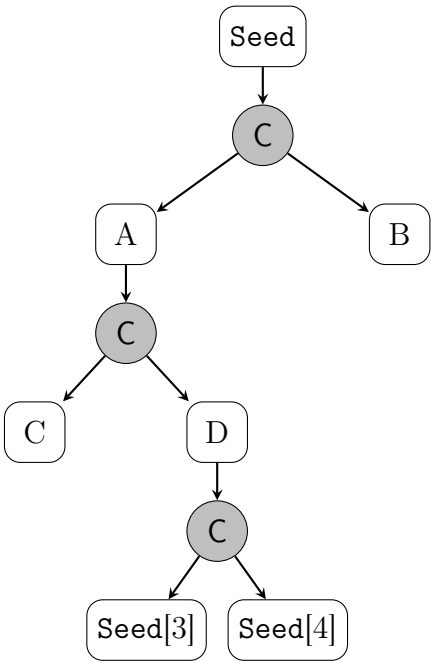

Figure 1.13: An example of which nodes are kept in memory during the third round of Sign for seed trees.

- *Timing attacks*: attacks that observe the variations in execution time of cryptographic primitives. Such attacks might even lead to complete private key recovery [30].

- *Power-analysis attacks*: such attacks might be declined in different ways like Simple Power Analysis (SPA), Differential Power Analysis (DPA), Template Attacks (TA), Correlation Power Analysis (CPA). An overview of these methods is presented by Randolph-Diehl in [41].

Active side-channel attacks typically revolves around fault injections, where the normal computation (e.g., the signature or the verification procedure), is altered. These attacks (as described by Barenghi et al. in [7]) might be relatively cheap to conduct. This category includes underpowering of the device, temporary brown-outs or well-timed power spikes, tampering with the clock signal or blanking memory. More expensive attacks encompass the usage of laser technologies.

The first category is particularly relevant for our scenario. In fact, low-cost attacks (i.e., attacks that require less than 3000$ to set them up), should be kept into serious consideration during the implementation of a cryptosystem as they can be executed by any seriously motivated attacker.

Passive attacks countermeasures — Regarding passive side channel attacks some counterstrategies can be identified. Timing attacks can be countered using constant-time implementations, for instance via constant time Look-Up Tables (LUTs).

SPA can be countered by removing conditional statements. This can be achieved by turning `if` constructs into predicated instructions. These instructions have to be implemented in the processor (ARM and Power ISAs support them natively) and allow the retirement of the result of an instruction only if a particular condition is verified (i.e., add two registers together only if a third one is zero). The control flow will always be the same, hence SPA is no longer useful.

DPA is more complex to counter. The simplest strategy is masking: given the basic operation $c = p \oplus k$, the algorithm might add (or xor) the same random quantity to the input and to the key ($c = p \oplus k \rightarrow c = (p \oplus r) \oplus (k \oplus r)$). This will make DPA ineffective as r will change at every run. Masking translates into delay in computation time and in the usage of more gates. However, attacks against masking already exist; these attacks are called high order DPA. They require the construction of a power model independent of random values. Another defensive strategy involves the usage of a morphic engine. Such engine, built using a compiler, will employ different, yet semantically equivalent, variants of the same algorithm. It is possible to include such engine into the decode unit of a CPU.

Active attacks countermeasures — Countermeasures against fault injections have been widely studied, a comprehensive analysis is presented in [14]. We provide a brief

overview:

- *Encryption-based countermeasures*: such approaches use cryptographic techniques (e.g., block ciphers in CTR mode, sponge-based primitives, encoding of program state, ...) to encrypt the instructions and validate them at runtime;
- *Spatial and timing redundancy*: they implement mechanisms like duplication of signals;
- *Signature based approaches*: this family of methods enhances security by comparing pre-calculated instructions traces with runtime ones or by introducing signature updates independent of the used paths in the Control Flow Graph (CFG).

Side-channel attacks and CROSS — In the context of post-quantum cryptographic schemes, side-channel attacks remain a relevant security threat. In fact, the new schemes and algorithms that were designed to resist attacks from quantum computers are not immune to side-channel vulnerabilities and a significant effort is being expended on the analysis and reinforcements of such schemes (as an example see the analysis on SPHINCS+ by Genêt in [22]).

Some side-channel attacks against CROSS were already presented, both passive (by Schupp-Sigl in [44] and by Czuprynko et al. in [17]) and active (by Mondal et al. in [35] and by Jendral et al. in [27]). Our analysis will focus on active side channel attacks. Our threat model will be presented in the next chapter.

2 | Analysis and strengthening of CROSS

In this chapter we will present our contribution. Firstly, we will analyze the pseudocode and the implementation of CROSS (i.e., its three main primitives), and some possible active attacks that might be executed against them. The pseudocode was provided by the original authors, and it will be used as a first contact point with the protocol. At the same time, we will identify how we can translate such attacks against the Cortex-M4 implementation of CROSS and its actual code (available at the official repository [47]). Then we will implement the most relevant attacks and assess the consequences (i.e., construction of a valid signature by an attacker).

Secondly, we will implement a new version of the signature procedure that will introduce some countermeasures against the most dangerous attacks. The current Cortex-M4 implementation of CROSS relies on the `pqm4` framework. We will make the code compilation process independent of such framework. Finally, we will benchmark the impact on performances when compared with the original version of the signature primitive.

2.1. Threat model

Our threat model assumes that an attacker is able to:

- Blank, partially or entirely, a selected area of memory;
- Conduct an instruction skipping during computation.

These two effects can be achieved by implementing the techniques presented in [7]. The analysis starts by analyzing the pseudocodes of the three primitives of CROSS together with the dependencies graphs of their different components. Then we will focus on the seed tree, a crucial data structure that is used during the signature and the verification process. A particular focus will be given on the most recent Cortex-M4 implementation – [45] – as microcontrollers are the most vulnerable devices against side-channel attacks.

In the pseudocodes, we will highlight with a red box the lines that can be targeted by an active side-channel attack. We will provide short snippets of Assembly, in order to better explain the attacks. Such Assembly was dumped from the compiled object file corresponding to R-SDP-5-small with optimization flag `-O2`.

2.2. Analysis of KeyGen

The analysis of KeyGen focuses on the core elements of the Restricted Syndrome Decoding Problem (described in Section 1.5), such as error vector-parity matrix multiplication and on the parity matrix. In KeyGen, the keypair is generated starting from the secret key, used to derive both the public one and the seed used to sample the error vector. The error vector is later used to compute the public syndrome jointly with the parity-check matrix derivable from the public key.

The critical points of KeyGen are related to the two secret elements managed during the execution: $\mathtt{Seed}_{\mathtt{sk}}$ and $\mathbf{e}$. The first one might be attacked directly while the second one might be attacked during the multiplication with the parity-check matrix $\mathbf{H}$.

Algorithm 2: KeyGen()

Input: None
Output: $\mathtt{sk}$: $\mathtt{Seed}_{\mathtt{sk}}$: secret key seed;
$\mathtt{pk}$: $(\mathtt{Seed}_{\mathtt{pk}}, \mathbf{s})$ public key;
Data: λ: security parameter;
$g \in \mathbb{F}_p^*$: generator of $\mathbb{E}$;

// Sampling seeds

1 $\mathtt{Seed}_{\mathtt{sk}} \xleftarrow{\$} \{0,1\}^{2\lambda}$

2 $(\mathtt{Seed}_{\mathtt{e}}, \mathtt{Seed}_{\mathtt{pk}}) \leftarrow \mathsf{CSPRNG} - \{0,1\}^{2\lambda} \times \{0,1\}^{2\lambda}\,(\mathtt{Seed}_{\mathtt{sk}} \mid 3t+1)$

// Sampling random matrices $\mathbf{H}$ and $\overline{\mathbf{M}}$

3 $(\overline{\mathbf{W}}, \mathbf{V}) \leftarrow \mathsf{CSPRNG} - \mathbb{F}_z^{m\times(n-m)} \times \mathbb{F}_p^{(n-k)\times k}\,(\mathtt{Seed}_{\mathtt{pk}} \mid 3t+2)$ | $\mathbf{V} \leftarrow \mathsf{CSPRNG} - \mathbb{F}_p^{(n-k)\times k}\,(\mathtt{Seed}_{\mathtt{pk}} \mid 3t+2)$

4 $\mathbf{H} \leftarrow [\mathbf{V} \mid \mathrm{Id}_{n-k}]$

// Computing $\mathbf{e}$

$\overline{\mathbf{M}} \leftarrow [\overline{\mathbf{W}} \mid \mathrm{Id}_m]$
5 $\overline{\mathbf{e}}_G \leftarrow \mathsf{CSPRNG} - \mathbb{F}_z^m\,(\mathtt{Seed}_{\mathtt{e}} \mid 3t+3)$
$\overline{\mathbf{e}} \leftarrow \overline{\mathbf{e}}_G\overline{\mathbf{M}}$
| $\overline{\mathbf{e}} \leftarrow \mathsf{CSPRNG} - \mathbb{F}_z^n\,(\mathtt{Seed}_{\mathtt{e}} \mid 3t+3)$

for j **from** 1 **to** n **do**
6 $\quad \mathbf{e}_j \leftarrow g^{\overline{\mathbf{e}}_j}$

```
// Computing the syndrome s
7  s ← eH^⊤
// Return secret key sk and public key pk
8  sk ← Seed_sk
9  pk ← (Seed_pk, s)
10 return (sk, pk);
```

Alteration of secret key — If an attacker is able to alter the secret key - $\texttt{Seed}_{\texttt{pk}}$ - the entire protocol is compromised, as it is the only requirement to sign a message. In the Cortex-M4 implementation, the following primitive is called:

```
randombytes(SK->seed_sk,KEYPAIR_SEED_LENGTH_BYTES)
```

where `randombytes` is a primitive provided by NIST that guarantees a safe source of randomness. The corresponding Assembly is:

```
movs    r1, #16
add     r0, sp, #152
bl      0 <randombytes>
```

The `bl` instruction might be skipped via a single instruction skipping as its implementation does not include an `inline` directive. The register `r0`, corresponding to `SK->seed_sk`, will retain its old value (i.e., it might be zero).

Another relevant vulnerability is shown against blanking attacks, where, in case of a complete blanking of `SK->seed_sk`, an attacker has complete knowledge of the protocol.

Disclosure of secret vector — The alteration of the parity-check matrix $\mathbf{H}$ at **Row 4** (directly or through alteration of $\mathbf{V}$, **Row 3**) might be helpful for an attacker, as it might disclose a fraction of the coordinates of $\mathbf{e}$. This attack requires the zeroization of the entire matrix $\mathbf{V}$. Equivalently an attacker might skip the multiplication loop and achieve a similar effect (**Row 7**). A trivial example is the following, given $n = 4, k = 1$ and a zeroized $\mathbf{V}$:

$$\mathbf{V} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

$$\mathbf{H} = [\mathbf{V} \mid \mathrm{Id}_3] = \left[\begin{array}{c|ccc} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{array}\right]$$

The resulting syndrome will reveal the last $n - k$ coordinates of $\mathbf{e}$:

$$\mathbf{s} = \begin{bmatrix} \mathbf{e}_{n-(k+1)} & \mathbf{e}_{n-k} & \mathbf{e}_n \end{bmatrix}$$

The amount of revealed coordinates is estimated easily with:

$$\frac{n-k}{n}$$

In **Section 5.1** of the specification document we observe that, given n and k specified in the table, such attack provides $\approx 40\%$ of coordinates of $\mathbf{e}$ (the best results are achieved against R-SDP, while R-SDP(G) typically provides worse results). The drawback of such procedure is the production of a faulted syndrome; a verifier would be able to see the incompatibility of the re-generated $\mathbf{H}$ matrix from $\mathtt{Seed}_{\mathtt{pk}}$ (not altered by an attacker) with the public syndrome (produced by the multiplication of $\mathbf{e}$ with an altered $\mathbf{H}$). A working attack that targets the same elements was proposed by Jendral et al. in [27].

Cost estimation — For a complete recovery an attacker has to compute the k remaining coordinates. It holds that:

$$\mathbf{e} \in \mathbb{F}_p^n$$

Apparently, an attacker has to exhaustively research $|\mathbb{F}_p|^k = p^k$ values. However, $\mathbf{e}$ is generated from $\overline{\mathbf{e}} \in \mathbb{F}_z^n$ via the map f:

$$f(\overline{\mathbf{x}}) : \mathbb{F}_z^n \mapsto \mathbb{F}_p^n$$
$$f(\overline{\mathbf{e}}_1, \ldots, \overline{\mathbf{e}}_n) = (g^{\overline{\mathbf{e}}_1}, \ldots, g^{\overline{\mathbf{e}}_n}) \to (\mathbf{e}_1, \ldots, \mathbf{e}_n)$$

where in code it is usually indicated as $\mathbf{e} = g^{\overline{\mathbf{e}}}$. Hence, an attacker will work in the research space z^k.

KeyGen dependencies analysis — In Figure 2.1, orange arrows indicate the paths followed exclusively when in R-SDP(G), same for teal arrows in R-SDP. As shown, all the elements in KeyGen originates from the secret seed, that is the primary target for an attacker. The attacks can be directed towards both variants of the problems, with minimal adjustments.

The right branch, linked to the public seed, provides some attack scenarios (matrices $\mathbf{H}, \mathbf{V}$ and vector-matrix multiplication for syndrome computation). Such attacks target the core of the Syndrome Decoding Problem, the syndrome computation.

The left branch, that is composed mainly by the secret error vectors, cannot be attacked directly by an active attacker. Instead, active attacks will focus on operations that involve the secret vector.

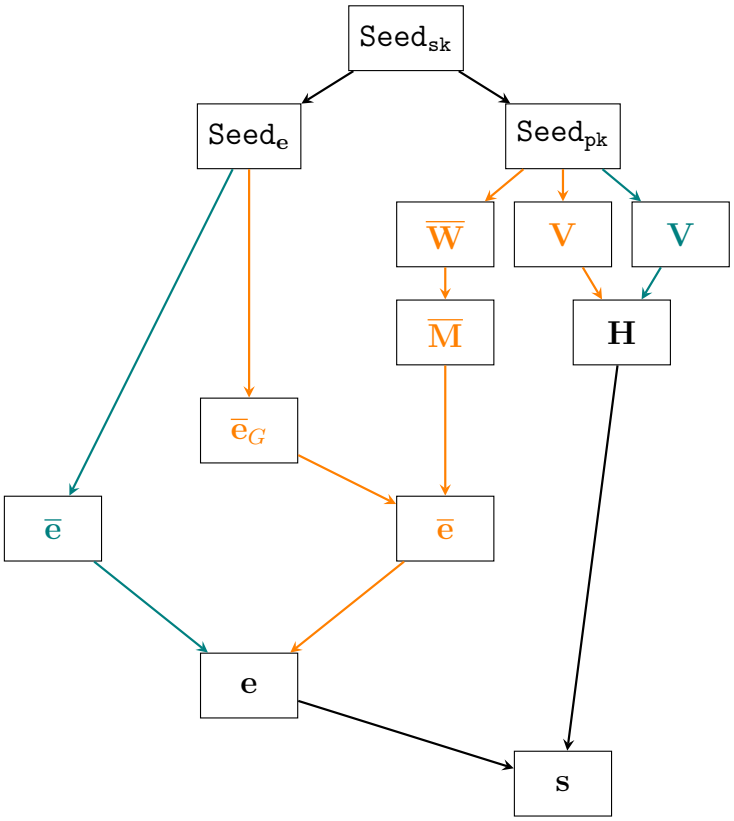

Figure 2.1: KeyGen dependencies.

2.3. Analysis of Sign

Sign is the primitive that computes a signature given a message and the secret key. Sign starts by expanding the secret key, the procedure is similar to the one in KeyGen with the addition of the retrieval of the error vector. Then the master/root seed is sampled, together with the salt. Using the seed tree, starting from the root, a new round seed is sampled for each round; such round seed is the input of the CSPRNG, together with the salt and a domain separation constant. The CSPRNG returns the secret ephemeral error vector ($\overline{\mathbf{e}}'[i]$) and an auxiliary vector ($\mathbf{u}'[i]$). The error vector and the ephemeral one are linked by the vector $\overline{\mathbf{v}}[i]$, and the syndrome is computed by multiplying the vector $\mathbf{u}[i]$, obtained via the component-wise multiplication of $\mathbf{v}[i]$ with $\mathbf{u}'[i]$, with the parity-check matrix. Finally, the two commitments are computed. The described procedure is repeated for t rounds.

The second part of the algorithm computes the digests of the messages and of the commitments together with the data structures `Proof` and `Path`. The two challenges are sampled in this section. The first one is used to multiply the ephemeral error vector ($\texttt{chall}_1[i]\mathbf{e}'[i]$); the product is added to the auxiliary vector $\mathbf{u}'[i]$ and the result is stored in $\mathbf{y}[i]$. The second challenge is used to decide if the round responses (containing $\mathbf{y}[i], \overline{\mathbf{v}}[i]$ and the second commitments) should be published or not. At the end, the signature is assembled.

The analysis of Sign will be organized into two different branches. The first one will focus on attacks that will target single rounds. The second one will target the structures (described in detail in Section 1.8) shared by the different primitives.

Algorithm 3: $\mathsf{Sign}(\mathtt{sk}, \mathtt{Msg})$

Input: $\mathtt{sk}$: secret key $\mathtt{Seed}_{\mathtt{sk}} \in \{0,1\}^{2\lambda}$;
$\mathtt{Msg} \in \{0,1\}^*$: message;
Output: $\mathtt{Sgn}$: signature;
Data: λ: security parameter;
c: constant defined as $c = 2t - 1$;
$g \in \mathbb{F}_p^*$: generator of $\mathbb{E}$;
t: number of rounds;
w: weight of the second challenge;

// Expanding secret key

$\overline{\mathbf{e}}, \overline{\mathbf{e}}_G, \mathbf{H}, \overline{\mathbf{M}} \leftarrow \mathsf{ExpandSK}(\mathtt{Seed}_{\mathtt{sk}})$ | $\overline{\mathbf{e}}, \mathbf{H} \leftarrow \mathsf{ExpandSK}(\mathtt{Seed}_{\mathtt{sk}})$

// Computing the commitments

$\mathtt{Seed} \xleftarrow{\$} \{0,1\}^{\lambda}$, $\mathtt{Salt} \xleftarrow{\$} \{0,1\}^{2\lambda}$

$(\mathtt{Seed}[1], \ldots, \mathtt{Seed}[t]) \leftarrow \mathsf{SeedLeaves}(\mathtt{Seed}, \mathtt{Salt})$

// Compute $\mathbf{v}[i]$ such that $\mathbf{v}[i] \star \mathbf{e}'[i] = \mathbf{e}$

for i **from** 1 **to** t **do**

$\overline{\mathbf{e}}'_G[i], \mathbf{u}'[i] \leftarrow \mathsf{CSPRNG} - \mathbb{F}_z^m \times \mathbb{F}_p^n\,(\mathtt{Seed}[i] \,|\, \mathtt{Salt} \,|\, i + c)$ | $\overline{\mathbf{e}}'[i], \mathbf{u}'[i] \leftarrow \mathsf{CSPRNG} - \mathbb{F}_z^n \times \mathbb{F}_p^n\,(\mathtt{Seed}[i] \,|\, \mathtt{Salt} \,|\, i + c)$

$\overline{\mathbf{v}}_G[i] \leftarrow \overline{\mathbf{e}}_G - \overline{\mathbf{e}}'_G[i]$
$\overline{\mathbf{e}}'[i] \leftarrow \overline{\mathbf{e}}'_G[i]\overline{\mathbf{M}}$

$\overline{\mathbf{v}}[i] \leftarrow \overline{\mathbf{e}} - \overline{\mathbf{e}}'[i]$

  for j **from** 1 **to** n **do**

    $\mathbf{v}[i]_j \leftarrow g^{\overline{\mathbf{v}}[i]_j}$

  $\mathbf{u}[i] \leftarrow \mathbf{v}[i] \star \mathbf{u}'[i]$

  $\mathbf{s}'[i] \leftarrow \mathbf{u}[i]\mathbf{H}^\top$

  $\mathtt{cmt}_0[i] \leftarrow \mathsf{Hash}(\mathbf{s}'[i] \mid \overline{\mathbf{v}}_G[i] \mid \mathtt{Salt} \mid i + c)$ | $\mathtt{cmt}_0[i] \leftarrow \mathsf{Hash}(\mathbf{s}'[i] \mid \overline{\mathbf{v}}[i] \mid \mathtt{Salt} \mid i + c)$

  $\mathtt{cmt}_1[i] \leftarrow \mathsf{Hash}(\mathtt{Seed}[i] \mid \mathtt{Salt} \mid i + c)$

$\mathtt{digest}_{\mathtt{cmt}_0} \leftarrow \mathsf{TreeRoot}(\mathtt{cmt}_0[1], \cdots, \mathtt{cmt}_0[t])$

$\mathtt{digest}_{\mathtt{cmt}_1} \leftarrow \mathsf{Hash}(\mathtt{cmt}_1[1] \mid \cdots \mid \mathtt{cmt}_1[t])$

$\mathtt{digest}_{\mathtt{cmt}} \leftarrow \mathsf{Hash}(\mathtt{digest}_{\mathtt{cmt}_0} \mid \mathtt{digest}_{\mathtt{cmt}_1})$

// Computing first challenge

$\mathtt{digest}_{\mathtt{Msg}} \leftarrow \mathsf{Hash}(\mathtt{Msg})$

$\mathtt{digest}_{\mathtt{chall}_1} \leftarrow \mathsf{Hash}(\mathtt{digest}_{\mathtt{Msg}} \mid \mathtt{digest}_{\mathtt{cmt}} \mid \mathtt{Salt})$

$\mathtt{chall}_1 \leftarrow \mathsf{CSPRNG} - (\mathbb{F}_p^*)^t\,(\mathtt{digest}_{\mathtt{chall}_1} \mid t + c)$

// Computing first response

for i **from** 1 **to** t **do**

  for j **from** 1 **to** n **do**

    $\mathbf{e}'[i]_j \leftarrow g^{\overline{\mathbf{e}}'[i]_j}$

  $\mathbf{y}[i] \leftarrow \mathbf{u}'[i] + \mathtt{chall}_1[i]\mathbf{e}'[i]$

// Computing second challenge
21 $\texttt{digest}_{\texttt{chall}_2} \leftarrow \mathsf{Hash}(\mathbf{y}[1] \mid \cdots \mid \mathbf{y}[t] \mid \texttt{digest}_{\texttt{chall}_1})$
22 $\texttt{chall}_2 \leftarrow \mathsf{CSPRNG} - \mathcal{B}_{(t,w)}\left(\texttt{digest}_{\texttt{chall}_2} \mid t + c + 1\right)$
// Computing second response
23 $\texttt{Proof} \leftarrow \mathsf{TreeProof}(\texttt{cmt}_0[1], \cdots, \texttt{cmt}_0[t], \texttt{chall}_2)$
24 $\texttt{Path} \leftarrow \mathsf{SeedPath}(\texttt{Seed}, \texttt{Salt}, \texttt{chall}_2)$
25 **for** i **from** 1 **to** t **do**
26 **if** $\texttt{chall}_2[i] = 0$ **then**
27 $\texttt{resp}[i]_0 \leftarrow (\mathbf{y}[i], \overline{\mathbf{v}}_G[i])$ | $\texttt{resp}[i]_0 \leftarrow (\mathbf{y}[i], \overline{\mathbf{v}}[i])$
$\texttt{resp}[i]_1 \leftarrow \texttt{cmt}_1[i]$
// Assembling signature
28 $\texttt{Sgn} \leftarrow (\texttt{Salt}, \texttt{digest}_{\texttt{cmt}}, \texttt{digest}_{\texttt{chall}_2}, \texttt{Path}, \texttt{Proof}, \texttt{resp})$
29 **return** Sgn

2.3.1. Single round analysis

Disclosure of secret vector — The first target for an attacker is the subtraction between the secret (exponent) error vector and the (exponent) round error vector. The following operations might be executed:

1. Annulment of $\overline{\mathbf{e}}'[i]$ (**Row 5**);
2. Skip subtraction of $\overline{\mathbf{e}}'[i]$ from $\overline{\mathbf{v}}[i] \leftarrow \overline{\mathbf{e}} - \overline{\mathbf{e}}'[i]$ (**Row 5**).
3. Annulment of $\mathbf{u}'[i]$ (**Row 20**);
4. Skip addition of $\mathbf{u}'[i]$ from $\mathbf{y}[i] \leftarrow \mathbf{u}'[i] + \texttt{chall}_1[i]\mathbf{e}'[i]$ (**Row 20**).

Case 1 — This first case requires blanking of $\overline{\mathbf{e}}'[i]$ as:

$$\overline{\mathbf{v}} \leftarrow \overline{\mathbf{e}} - \overline{\mathbf{e}}'[i]$$
$$\overline{\mathbf{v}} \leftarrow \overline{\mathbf{e}} - \mathbf{0}$$
$$\overline{\mathbf{v}} \leftarrow \overline{\mathbf{e}}$$

Then $\overline{\mathbf{e}}$ is publicly available through $\overline{\mathbf{v}}$ if, for the associated round i, it holds that $\texttt{chall}_2[i] = 0$. In such case:

$$\texttt{resp}[i]_0 \leftarrow (\mathbf{y}[i], \overline{\mathbf{v}})$$

Then an attacker simply has to apply the transformation:

$$g^{\overline{\mathbf{v}}} = g^{\overline{\mathbf{e}}} = \mathbf{e}$$

Case 2 — To analyze the second case we need to go more in-depth into the code. The operation:

$$\overline{\mathbf{v}}[i] \leftarrow \overline{\mathbf{e}} - \overline{\mathbf{e}}'[i]$$

is represented by the following C code:

```
static inline
void fz_vec_sub_n(FZ_ELEM v_bar[N],
                  const FZ_ELEM e_bar[N],
                  const FZ_ELEM e_bar_prime[N]){
    for(int i = 0; i < N; i++){
        v_bar[i]= FZRED_SINGLE(e_bar[i] + FZRED_OPPOSITE(e_bar_prime[i]));
    }
}
```

where `FZRED_SINGLE` is a macro for modulo reduction – `(x & 0x07) + (x >> 3)` – and `FZRED_OPPOSITE` is a macro – `x ^ 0x07` – for sign inversion in modulo. Note that such macros are highly specific for the Mersenne modulo chosen in CROSS.

For `FZRED_SINGLE` we start by looking at Table 1.1, we observe that z can be either 7 (R-SDP) or 127 (R-SDP(G)) and they are both Mersenne primes (a Mersenne prime follows the structure $2^n - 1$). In our example $z = 7$, we recall that:

$$2^n \equiv 1 \mod 2^n - 1 \rightarrow 2^3 \equiv 1 \mod 7$$

We observe that any number can be divided into:

$$x = x_{HI} \cdot 2^n + x_{LO}$$

Hence, when we compute $x \mod 2^n - 1$ as:

$$\begin{aligned} x \mod 2^n - 1 &= (x_{HI} \cdot 2^n + x_{LO}) && \mod 2^n - 1 \\ &= x_{HI} + x_{LO} && \mod 2^n - 1 \end{aligned}$$

By doing so we avoided an expensive division. The macro exactly replicates the identity above, in fact:

$$\mathtt{x}_{HI} = \mathtt{x} \gg 3 \rightarrow \text{Drop the last three (rightmost) bits}$$

$$\mathtt{x}_{LO} = \mathtt{x} \ \&\ 0\mathtt{x}07 \rightarrow \text{Take the rightmost three bits}$$

For `FZRED_OPPOSITE` we apply similar logic using the XOR operator to turn subtractions into additions.

By looking back at the code the following Assembly instructions are executed:

```
ldrb.w r2, [r3, #1]!        // r2 = e_bar[i]
```

```
ldrb.w r1, [r3, #127]       // r1 = e_bar_prime[i], 127 (N) bytes after e[i]
eor.w r1, r1, #7            // FZRED_OPPOSITE (r1 ^= 0x07)
add r2, r1                  // r2 += r1
and.w r1, r2, #7            // FZRED_SINGLE, left branch
add.w r2, r1, r2, lsr #3    // FZRED_SINGLE, right branch
```

Please note that the compiler does not introduce an explicit iteration variable, but it indexes the coordinates directly by increasing pointers. The register `r3` behaves like i, in fact `[r3, #1]!` executes a write-back as `r3 += 1`, it will be used later in a `bne`. In addition, optimization typically encourages the usage of loop unrolling, hence the jump back might be done after several iterations and not after only one. This makes the managements of the pipeline and the register use easier for the processor. The subtraction can be skipped, coordinate by coordinate, if an attacker is able to skip:

```
add r2, r1
```

Alternatively an attacker might skip the first loop backedge and bypass the rest of the subtraction. If the subtraction $\overline{\mathbf{e}} - \overline{\mathbf{e}}'[i]$ is skipped then $\overline{\mathbf{v}}[i] \leftarrow \overline{\mathbf{e}}$. The second case converges into the first one and $\mathbf{e}$ can be recovered.

Case 3 — The third attack involves blanking again. If an attacker is able to alter the following operation as follows:

$$\begin{aligned}\mathbf{y}[i] \leftarrow &\mathbf{u}'[i] + \mathtt{chall}_1[i]\mathbf{e}'[i] \\ &\mathbf{0} + \mathtt{chall}_1[i]\mathbf{e}'[i] \\ &\mathtt{chall}_1[i]\mathbf{e}'[i]\end{aligned}$$

Then, $\mathtt{chall}_1[i]$ is derivable from public elements, and if $\mathtt{chall}_2[i] = 0$ then $(\mathbf{y}[i], \overline{\mathbf{v}}[i])$ is public too. To obtain $\mathbf{e}'[i]$:

$$\mathbf{e}'[i] \leftarrow \mathtt{chall}_1^{-1}[i]\mathbf{y}[i]$$

It is known that $\mathtt{chall}_1[i] \in \mathbb{F}_p^*$ hence to get the inverse it is sufficient to build a lookup table (with $(p-1)^2$ multiplications). The attacker now has both $\overline{\mathbf{v}}$ and $\mathbf{e}'[i]$, it has to compute $\overline{\mathbf{e}}'[i]$. To do so we require z progressive exponentiation of g and then do a lookup for each coordinate of $\mathbf{e}'[i]$. Once $\overline{\mathbf{e}}'[i]$ is computed then the original error vector is:

$$\overline{\mathbf{e}} \leftarrow \overline{\mathbf{v}}[i] + \overline{\mathbf{e}}'[i]$$

Then:

$$\mathbf{e} = g^{\overline{\mathbf{e}}}$$

Case 4 — The fourth attack strategy requires instruction skipping. If an attacker is able to

alter the following operation as follows:

$$\mathbf{y}[i] \leftarrow \cancel{\mathbf{u}'[i]} + \texttt{chall}_1[i]\mathbf{e}'[i]$$

Then for that specific round it will result:

$$\mathbf{y}[i] \leftarrow \texttt{chall}_1[i]\mathbf{e}'[i]$$

The C code is the following (some casting is done to avoid overflow, not shown for simplicity):

```
static inline
void fp_vec_by_restr_vec_scaled(FP_ELEM y[N],
                                const FZ_ELEM e_bar_prime[N],
                                const FP_ELEM chall_1,
                                const FP_ELEM u_prime[N]){
    for(int i = 0; i < N; i++){
        y[i] = FPRED_DOUBLE(u_prime[i] +
                            RESTR_TO_VAL(e_bar_prime[i]) * chall_1);
    }
}
```

The macro `FPRED_DOUBLE` apply a Mersenne reduction similar to the one shown before while `RESTR_TO_VAL` simply apply the transformation $\mathbf{e}'[i] = g^{\overline{\mathbf{e}}'[i]}$ using a LUT. For simplicity, we do not show the complete Assembly translation but only the most relevant instructions:

```
ldrb.w lr, [r3, #1]!            // Load chall_1
[...]
ldrb.w r3, [r2, #255]           // Load u_prime[i]
ldrb.w sl, [r2, #1]!            // Load e_bar_prime
ldrb.w sl, [r4, sl]             // LUT look-up for e_bar_prime[i]
smlabb r3, sl, lr, r3           // Multiply and Accumulate
```

The ARM instruction `smlabb` (Signed Multiply Long Accumulate Bottom Bottom) executes a multiplication between the lower 16 bits of `sl` and the lower 16 bits of `lr` then it adds the result to `r3`. An attacker cannot simply skip it as $\mathbf{y}[i]$ will retain its old value that is $\mathbf{u}[i]$. Hence, the most viable path for this attack is blanking as shown in third case.

Cost estimation — For blanking attacks suppose that the probability of executing a successful attack is p. Then to be certain to obtain an adjacent $\texttt{chall}_2[i] = 0$ it has to repeat the attack t times (one for each round). The final probability P is:

$$P = \prod_{i=1}^{t} p = \underbrace{p \cdots p}_{t \text{ times}} = p^t$$

Instead, if it focuses only on a specific round (for instance, the first) then the probability is:

$$P = p \cdot \frac{t-w}{t}$$

In fact the probability that a specific position in the challenge vector is 0 is:

$$P(\texttt{chall}_2[i] = 0) = \frac{t-w}{t}$$

This value can be derived by dividing the number of compatible permutation $\binom{t-1}{w}$ by the number of possible permutations $\binom{t}{w}$:

$$\begin{aligned}
\frac{\binom{t-1}{w}}{\binom{t}{w}} = \frac{\frac{(t-1)!}{w!(t-1-w)!}}{\frac{t!}{w!(t-w)!}} &= \frac{(t-1)!}{w!(t-1-w)!} \cdot \frac{w!(t-w)!}{t!} \\
&= \frac{(t-1)!}{w!(t-1-w)!} \cdot \frac{w!(t-w)(t-w-1)!}{t(t-1)!} \\
&= \frac{\cancel{(t-1)!}}{\cancel{w!}\cancel{(t-1-w)!}} \cdot \frac{\cancel{w!}(t-w)\cancel{(t-w-1)!}}{t\cancel{(t-1)!}} \\
&= \frac{t-w}{t} = 1 - \frac{w}{t}
\end{aligned}$$

This holds for all blanking attacks.

For instruction skipping an attacker has to inject a precise fault N times to obtain a full recovery. We suppose that such probability is always the same, and it is equal to p. In addition, it has to target a round having $\texttt{chall}_2[i] = 0$. The final probability is:

$$P = P(\texttt{chall}_2[i] = 0) \cdot \prod_{i=1}^{N} p = \frac{t-w}{t} \cdot p^N$$

However, if we suppose that the attacker skips the loop back-edge then it has to recover the remaining coordinates. This requires a single fault injection, combined with a round having $\texttt{chall}_2[i] = 0$.

2.3.2. Analysis of data structures

The most critical data structure is clearly the seed tree. If an attacker is able to know even one $\texttt{Seed}[i]$ associated with a $\texttt{chall}_2[i] = 0$ then it can recover the original error vector $\mathbf{e}$. In fact, suppose that an attacker knows $\texttt{Seed}[0]$ and that $\texttt{chall}_2[0] = 0$:

$$\begin{aligned}
\overline{\mathbf{e}}'[0], \mathbf{u}'[0] &\leftarrow \mathsf{CSPRNG} - \mathbb{F}_z^n \times \mathbb{F}_p^n \left(\texttt{Seed}[0] \,|\, \texttt{Salt} \,|\, i + c\right) \\
\overline{\mathbf{v}}[0] &\leftarrow \texttt{resp}_0[0] \\
\overline{\mathbf{e}} &\leftarrow \overline{\mathbf{v}}[0] + \overline{\mathbf{e}}'[0]
\end{aligned}$$

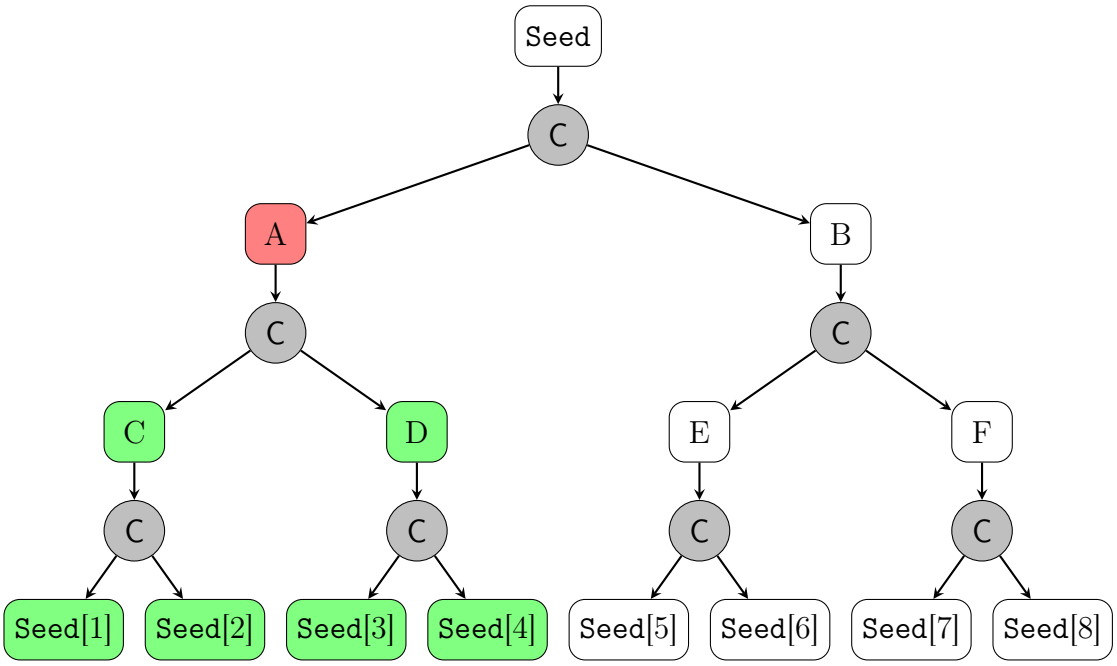

Figure 2.2: An example of a faulted seed tree.

The same holds if an attacker is able to know even an inner node as it can generate all the seeds that depends on such node as shown in Figure 2.2 where the red is the known node and the green ones are the ones that can be derived from the red one.

As we are interested in fault injection we transpose the previous observations into active attacks. In such cases, the attacker injects a known value (ideally 0) either in **Row 2**, **Row 5** or **Row 24**.

In case of **Row 2** the C code is straightforward:

```
randombytes(root_seed,SEED_LENGTH_BYTES);
```

and so is its Assembly translation:

```
movs r1, #16
add.w r0, r7, #112
bl 0 <randombytes>
```

If the attacker is able to skip the call to the subroutine, the master seed will potentially be 0. Blanking the memory area associated to the root seed is viable too.

The attack against **Row 5** is similar. In the Cortex-M4 implementation of management of trees greatly differs from the reference implementation, as shown in Section 1.10. However, the logic is similar as before, injecting a fault (or equivalently blanking) in a position known by an attacker translates in complete knowledge of the related seeds.

An indirect attack might target **Row 24** in order to force the publication in the `Path` of a node that is not supposed to be published. Some caution should be used as `Path` size is established at compile-time, hence the publication of an unexpected seed might break the array used to keep track of the structure.

Cost estimation — The attacks against the seed tree might be extremely interesting for an attacker. In fact, the all the attacks require the injection of a single precise fault. In case of **Row 5** the injection should be combined with a compatible $\texttt{chall}_2[i] = 0$, as shown before, the final probability is:

$$P = p \cdot \frac{t - w}{t}$$

Sign dependencies analysis — Due to the complexity of the primitive, two distinct graphs for R-SDP(G) (Figure 2.3) and R-SDP (Figure 2.4) are provided. Dashed arrows are followed only if $\texttt{chall}_2[i] = 0$ pre-condition is verified. The targeted elements are, predictably, related to secret elements (e.g., secret key, ephemeral secret seed, round seed) by some dependencies' path. In fact from $\overline{\mathbf{v}}$ we have one path that leads $\texttt{Seed}_{\texttt{sk}}$ and one that leads to $\texttt{Seed}[i]$. Similar relationships hold for $\mathbf{y}$. The number of nodes between the secret elements and the vulnerable ones in small, this means that the attacker focuses on shorter paths in order to minimize the required steps to recover the desired elements.

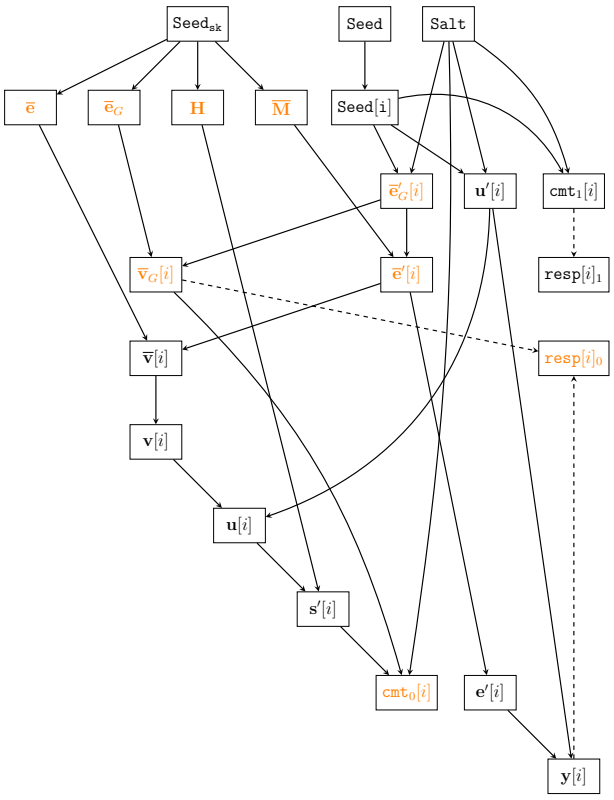

Figure 2.3: Intra-round dependencies for R-SDP(G) Sign.

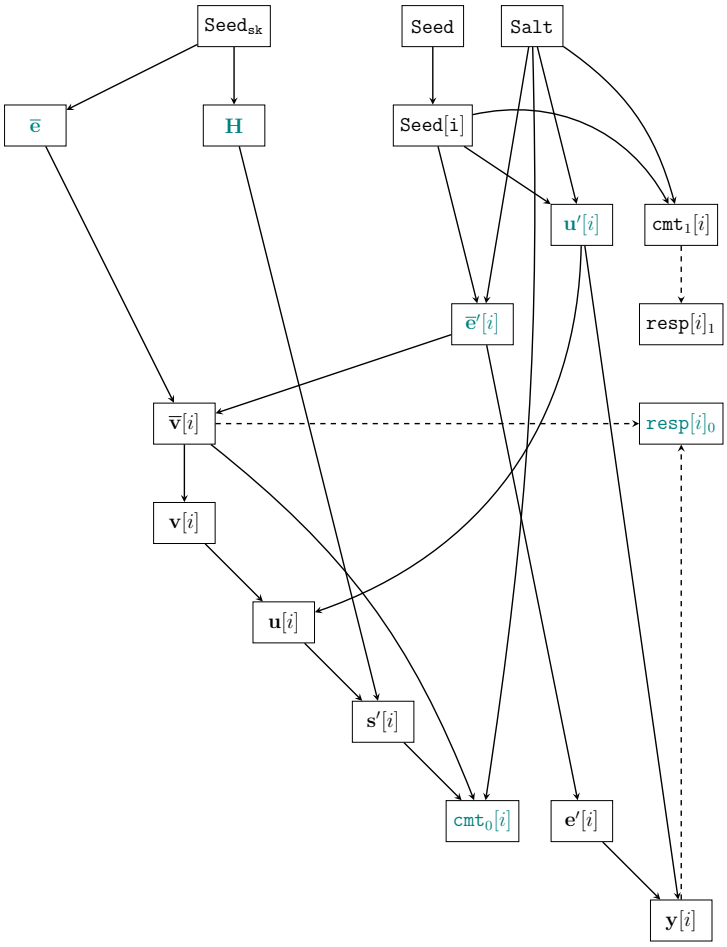

Figure 2.4: Intra-round dependencies for R-SDP Sign.

Summary table for attacks against Sign — To sum up, we present a table (Table 2.1) that recaps the attacks and the attacks strategies presented in the section as our work is based mainly on these attacks. In the last column we provide the cheapest cost if multiple attacks are available.

Table 2.1: Summary of active attacks against Sign.

Attack	Blanking	Instruction Skipping	Computational Cost
$\overline{\mathbf{v}} \leftarrow \overline{\mathbf{e}} - \overline{\mathbf{e}}'[i]$	✓	✓	$p \cdot \frac{t-w}{t}$
$\mathbf{y}[i] \leftarrow \mathbf{u}'[i] + \mathtt{chall}_1[i]\mathbf{e}'[i]$	✓	×	$p \cdot \frac{t-w}{t}$
$\mathtt{Seed} \xleftarrow{\$} \{0,1\}^\lambda$	✓	✓	p
$\mathsf{CSPRNG} - \mathbb{F}_z^m \times \mathbb{F}_p^n \left(\mathtt{Seed}[i] \mid \mathtt{Salt} \mid i + c\right)$	✓	✓	$p \cdot \frac{t-w}{t}$
Path	×	✓	p

2.4. Analysis of Verify

Verify is the primitive that, given a message, a signature and a public key, will accept or reject the signature for the given message. This will be done starting from the expansion of the public key, similarly to what is done in KeyGen. Then the digest of the message is computed and subsequently the two challenge vectors. Before the main loop the seeds linked to a round i such that $\texttt{chall}_2[i] = 1$ are recovered.

The main loop, articulated in t rounds, is composed by two branches. The selection of a branch depends exclusively on the value of $\texttt{chall}_2[i]$. In case $\texttt{chall}_2[i] = 1$ then the second commitment will be checked. This is achieved via the usage of a CSPRNG seeded with the reconstructed round seed, the salt and the domain separation constant; then $\mathbf{y}[i]$ is recomputed. In case $\texttt{chall}_2[i] = 0$ then the first commitment will be checked. Verify will check that the vector $\overline{\mathbf{v}}[i]$ (obtained from the public response, similarly to $\mathbf{y}[i]$) is restricted and, it will recompute the round syndrome. The first commitment is then recomputed using a CSPRNG seeded with the recomputed elements, the salt and the domain separation constant.

After the main loop the digests associated with the commitments and the digest related to $\mathbf{y}$ vectors will be recomputed. The final check will compare the aforementioned digests with the one provided in the signature.

The analysis of the verification primitive focuses on forcing the acceptance of forged signatures. The identified attacks are targeted against two core concepts of the primitive. The first one attack the first challenge, hence it is related to the core concepts of the Fiat-Shamir transformation. The second one is strictly related to the implementation, and it targets the final check used to accept or reject the signature.

Algorithm 4: Verify$(\texttt{pk}, \texttt{Msg}, \texttt{Sgn})$

Input: $\texttt{pk}$: $(\texttt{Seed}_{\texttt{pk}}, \mathbf{s})$ public key;
$\texttt{Msg} \in \{0,1\}^*$: message;
$\texttt{Sgn}$: $(\texttt{Salt}, \texttt{digest}_{\texttt{cmt}}, \texttt{digest}_{\texttt{chall}_2}, \texttt{Path}, \texttt{Proof}, \texttt{resp})$ signature;
Output: $\{\texttt{True}, \texttt{False}\}$;
Data: λ: security parameter;
g: generator of $\mathbb{E}$;
t: number of rounds; w: weight of second challenge;
c: constant defined as $2t - 1$;

// Recovering public key

$(\overline{\mathbf{W}}, \mathbf{V}) \leftarrow \text{CSPRNG} - \mathbb{F}_z^{m\times(n-m)} \times \mathbb{F}_p^{(n-k)\times k}\,(\texttt{Seed}_{\texttt{pk}} \mid 3t+2)$ | $\mathbf{V} \leftarrow \text{CSPRNG} - \mathbb{F}_p^{(n-k)\times k}\,(\texttt{Seed}_{\texttt{pk}} \mid 3t+2)$

$\mathbf{H} \leftarrow [\mathbf{V} \mid \mathrm{Id}_{n-k}]$

$\overline{\mathbf{M}} \leftarrow [\overline{\mathbf{W}} \mid \mathrm{Id}_m]$

// Computing challenges

$\texttt{digest}_{\texttt{Msg}} \leftarrow \mathsf{Hash}(\texttt{Msg})$

$\texttt{digest}_{\texttt{chall}_1} \leftarrow \mathsf{Hash}(\texttt{digest}_{\texttt{Msg}} \mid \texttt{digest}_{\texttt{cmt}} \mid \texttt{Salt})$

$\texttt{chall}_1 \leftarrow \text{CSPRNG} - (\mathbb{F}_p^*)^t\,(\texttt{digest}_{\texttt{chall}_1} \mid t + c)$

$\texttt{chall}_2 \leftarrow \text{CSPRNG} - \mathcal{B}_{(t,w)}\,(\texttt{digest}_{\texttt{chall}_2} \mid t + c + 1)$

// Computing commitments
$(\mathsf{Seed}[i])_{i:\mathtt{chall}_2[i]=1} \leftarrow \mathsf{RebuildLeaves}(\mathtt{Path}, \mathtt{chall}_2, \mathtt{Salt})$
for i **from** 1 **to** t **do**
  if $\mathtt{chall}_2[i] = 1$ **then**
    $\mathtt{cmt}_1[i] \leftarrow \mathsf{Hash}(\mathsf{Seed}[i] \mid \mathtt{Salt} \mid i + c)$
    $\overline{\mathbf{e}}'_G[i], \mathbf{u}'[i] \leftarrow \mathsf{CSPRNG} - \mathbb{F}_z^m \times \mathbb{F}_p^n\ (\mathsf{Seed}[i] \mid \mathtt{Salt} \mid i + c)$ | $\overline{\mathbf{e}}'[i], \mathbf{u}'[i] \leftarrow \mathsf{CSPRNG} - \mathbb{F}_z^n \times \mathbb{F}_p^n\ (\mathsf{Seed}[i] \mid \mathtt{Salt} \mid i + c)$
    $\overline{\mathbf{e}}'[i] \leftarrow \overline{\mathbf{e}}'_G[i]\overline{\mathbf{M}}$
    for j **from** 1 **to** n **do**
      $\mathbf{e}'[i]_j \leftarrow g^{\overline{\mathbf{e}}[i]_j}$
    $\mathbf{y}[i] \leftarrow \mathbf{u}'[i] + \mathtt{chall}_1[i]\mathbf{e}'[i]$
  if $\mathtt{chall}_2[i] = 0$ **then**
    $\mathtt{cmt}_1[i] \leftarrow \mathtt{resp}[i]_1$
    $(\mathbf{y}[i], \overline{\mathbf{v}}_G[i]) \leftarrow \mathtt{resp}[i]_0$ | $(\mathbf{y}[i], \overline{\mathbf{v}}[i]) \leftarrow \mathtt{resp}[i]_0$
    Check if $\overline{\mathbf{v}}_G[i] \in \mathbb{F}_z^m$ | Check if $\overline{\mathbf{v}}[i] \in \mathbb{F}_z^n$
    $\overline{\mathbf{v}}[i] \leftarrow \overline{\mathbf{v}}_G[i]\overline{\mathbf{M}}$
    for j **from** 1 **to** n **do**
      $\mathbf{v}[i]_j \leftarrow g^{\overline{\mathbf{v}}[i]_j}$
    $\mathbf{y}'[i] \leftarrow \mathbf{v}[\mathbf{i}] \star \mathbf{y}[i]$
    $\mathbf{s}'[i] \leftarrow \mathbf{y}'[i]\mathbf{H}^\top - \mathtt{chall}_1[i]\mathbf{s}$
    $\mathtt{cmt}_0[i] \leftarrow \mathsf{Hash}(\mathbf{s}'[i] \mid \overline{\mathbf{v}}_G[i] \mid \mathtt{Salt} \mid i + c)$ | $\mathtt{cmt}_0[i] \leftarrow \mathsf{Hash}(\mathbf{s}'[i] \mid \overline{\mathbf{v}}[i] \mid \mathtt{Salt} \mid i + c)$
// Checking digests
$\mathtt{digest}_{\mathtt{cmt}_0} \leftarrow \mathsf{RecomputeRoot}(\mathtt{cmt}_0, \mathsf{Proof}, \mathtt{chall}_2)$
$\mathtt{digest}_{\mathtt{cmt}_1} \leftarrow \mathsf{Hash}(\mathtt{cmt}_1[1] \mid \cdots \mid \mathtt{cmt}_1[t])$
$\mathtt{digest}'_{\mathtt{cmt}} \leftarrow \mathsf{Hash}(\mathtt{digest}_{\mathtt{cmt}_0} \mid \mathtt{digest}_{\mathtt{cmt}_1})$
$\mathtt{digest}'_{\mathtt{chall}_2} \leftarrow \mathsf{Hash}(\mathbf{y}[1] \mid \cdots \mid \mathbf{y}[t] \mid \mathtt{digest}_{\mathtt{chall}_1})$
if $\mathtt{digest}_{\mathtt{cmt}} = \mathtt{digest}'_{\mathtt{cmt}}$ **and** $\mathtt{digest}_{\mathtt{chall}_2} = \mathtt{digest}'_{\mathtt{chall}_2}$ **then**
  return True
return False

Acceptance of a forged signature - attacking the first challenge — Suppose that an attacker runs a custom Sign, by knowing a pk in which:

- He will recover $\mathbf{H}^\top$ from pk;
- He will alter the computation of $\mathbf{y}[i] \leftarrow \mathbf{u}'[i] + \mathtt{chall}_1[i]\mathbf{e}'[i]$ by setting $\mathtt{chall}_1[i] = 0$. By doing so $\mathbf{y}[i] \leftarrow \mathbf{u}'[i]$.

He will end up with a correctly formatted signature but with a $\mathtt{resp}_0[i] = (\mathbf{y}[i], \overline{\mathbf{v}}[i]) = (\mathbf{u}'[i], \overline{\mathbf{v}}[i])$. Now, during the verification process he needs to inject $\mathtt{chall}_1 = 0$. Please note that this requires a fault as for any round $\mathtt{chall}_1[i] \in \mathbb{F}_p^*$, hence it cannot be 0. Then, if $\mathtt{chall}_2[i] = 1$ the verifier will correctly recompute $\mathbf{y}[i] \leftarrow \mathbf{u}'[i]$. If $\mathtt{chall}_2[i] = 0$, from the elements provided

by the attacker, the verifier will recompute:

$$\begin{aligned}(\mathbf{y}[i], \overline{\mathbf{v}}[i]) &\leftarrow \mathtt{resp}_0[i] \\ \mathbf{y}'[i] &\leftarrow \mathbf{v}[i] \star \mathbf{y}[i] \\ &\leftarrow \mathbf{v}[i] \star \mathbf{u}'[i] \\ &\leftarrow \mathbf{u}[i] \\ \mathbf{s}'[i] &\leftarrow \mathbf{u}[i]\mathbf{H}^\top - \cancel{\mathtt{chall}_1[i]\mathbf{s}}\end{aligned}$$

All elements will be recomputed correctly, and any signature will be accepted.

To inject this fault an attacker might blank the memory area that stores $\mathtt{chall}_1$. Alternatively it might inject a fault to skip the following call:

```
csprng_fp_vec_chall_1(chall_1, csprng_state, (uint8_t *)&V_tr_csprng_e_bar_inst);
```

In the actual code, there is no function call, as it is inlined. However, the code revolves around a while loop, the condition is checked with the following Assembly:

```
cmp.w r2, #520                    // Size of chall_1 is T = 520 in our case
                                  // r2 contains the index of the
                                  // currently processed position of chall_1
beq.n 23e <CROSS_verify+0x23e>    // Jump to end
```

The `beq.n` will look at the zero flag set by `cmp.w`. If an attacker is able to skip `cmp.w`, the zero flag will hold the result of the last executed comparison. This flag will likely be 1 (from another while or from other positive comparisons) and the while loop will be completely skipped.

The countermeasure for this attack is simple, and it has negligible impact on performance as it is sufficient to reject the signature if even one of $\mathtt{chall}_1[i]$ is equal to 0. In fact, $\mathtt{chall}_1[i]$ is part of a multiplicative group, and it cannot be zero if not for a fault injection.

Acceptance of a forged signature — Another target for an attacker might be **Row 26**, where the final check is done before the acceptance or the rejection of the provided signature. In code the final control is actually more complex:

```
is_signature_ok = is_signature_ok &&
                      does_digest_cmt_match &&
                      does_digest_chall_2_match &&
                      is_mtree_padding_ok &&
                      is_stree_padding_ok &&
                      is_padd_key_ok &&
                      is_packed_padd_ok;
return is_signature_ok;
```

The most interesting targets for an attacker are the following two variables:

```
does_digest_cmt_match
does_digest_chall_2_match
```

as a forged signature, if well formatted, will not turn the other variables into false state, but it will be detected by the two variables above. If we analyze more in-depth how they are built:

```
int does_digest_cmt_match = (memcmp(digest_cmt_prime,
                                    sig->digest_cmt,
                                    HASH_DIGEST_LENGTH) == 0);
int does_digest_chall_2_match = (memcmp(digest_chall_2_prime,
                                    sig->digest_chall_2,
                                    HASH_DIGEST_LENGTH) == 0);
```

we might identify an attack strategy. By default, `memcmp` will return a 0 if the comparison length (the third argument) is 0. In our case `HASH_DIGEST_LENGTH` is defined at compile time, it is always bigger than 0, and it is typically stored in a register as shown by the Assembly:

```
ldr     r1, [r7, #16]   // Loading the second argument
movs    r2, #32         // Loading the third argument
add.w   r0, r7, #160    // Loading the first argument
bl      0 <memcmp>      // Call to memcmp
```

The call to `memcmp` adopts the usual parameter passing approach via registers. Even though blanking registers is outside our threat model, as it requires expensive technologies, a seriously motivated attacker might blank the `r2` register in order to set it to 0. Then `memcmp` will return a 0; the same should be repeated for the second `memcmp` to force the acceptance of a forged signature.

Cost estimation — The first presented attack requires a single, precise fault, to skip the comparison; hence its probability of success is simply $P = p$. The second attack requires two precise faults, if we keep p as the probability of injecting a fault, the final probability is $P = p^2$.

Signature dependencies — In Figure 2.5 we represented the path followed by the elements that compose the signature. For brevity, $\texttt{cmt}_0$ will represent $\texttt{cmt}_0[1] \mid \cdots \mid \texttt{cmt}_0[t]$, $\mathbf{y}$ will represent $\mathbf{y}[1] \mid \cdots \mid \mathbf{y}[t]$ and so on. Nodes with gray background are stored in the signature. The first presented attack targets $\texttt{chall}_1$. This component is a core element of the dependencies graph as it is the destination of both digests related to commitments. Such digests are also used later to accept or reject the signature. The importance of these elements is highlighted by the second attack that targets them directly.

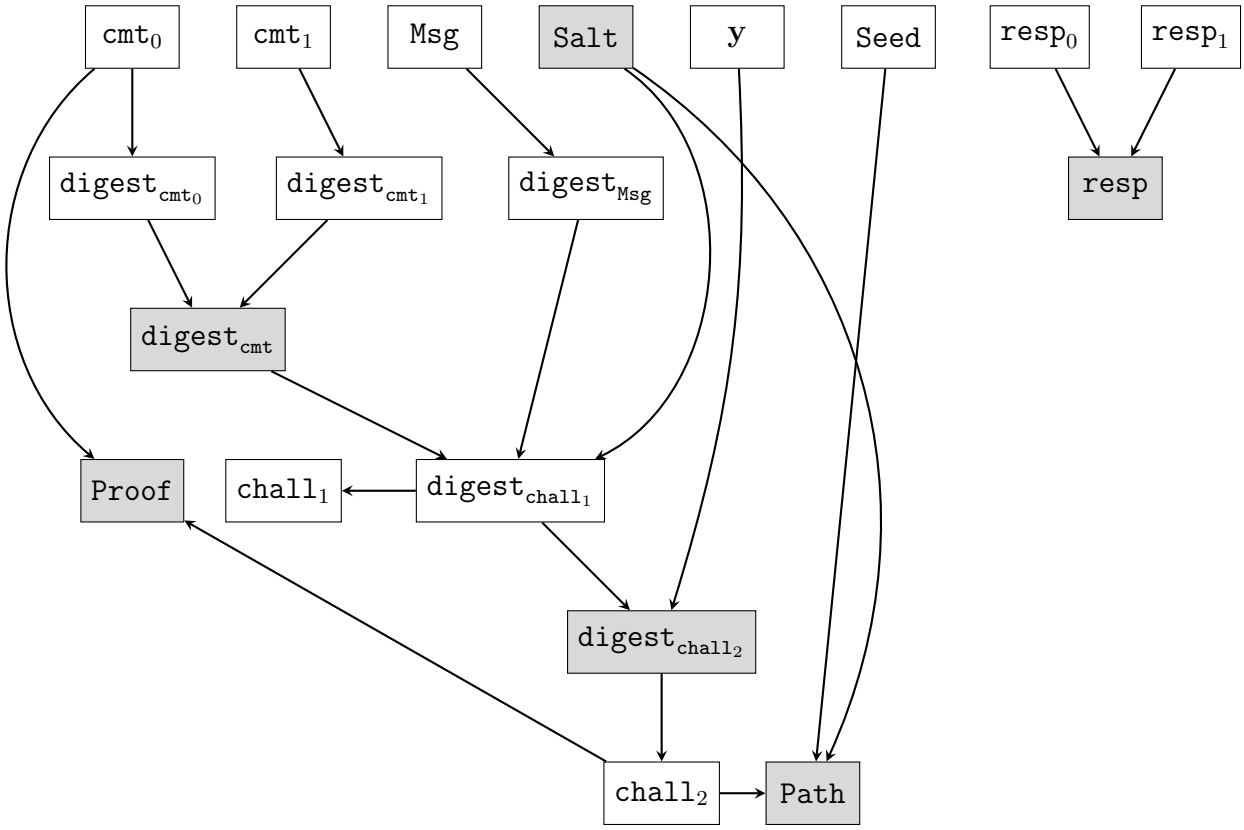

Figure 2.5: Signature dependencies.

Summary table for attacks — We include a summary table (Table 2.2) that provides references to every attack presented in the section.

Table 2.2: Summary of active attacks against primitives of CROSS.

Attack	KeyGen	Sign	Verify
Disclosure of secret key ($\mathtt{Seed}_{\mathtt{pk}}$)	Row 1	—	—
Disclosure of secret vector ($\mathbf{e}$)	Row 3/4/7	Row 2/3/5/20/24	—
Acceptance of forged signatures	—	—	Row 6/26

2.5. Development of a countermeasure

During the development of the countermeasure, we tried to achieve a favorable trade off between the impact on the performance of the Cortex-M4 implementation and the increase of security margin. This was achieved starting from an in-depth analysis of the Cortex-M4 implementation and its core concepts.

Core concepts — The Cortex-M4 implementation heavily relies on recomputation. This was

the most natural choice as microcontrollers have strict memory limits, hence keeping in memory all the elements of the original implementation was not feasible. This translated in the introduction of three `for` cycles of length t during the signature. Additionally, Keccak became an even more relevant primitive as it allowed the compression of the data structures via incremental hashing.

Our contribution tries to combine the two aforementioned concepts by introducing an additional data structure. This structure should follow the same footprints of the Cortex-M4 implementation, hence it must have a low-impact in memory. Such structure is organized as follows:

```
struct countermeasures_struct {
    CSPRNG_STATE_T cm_state;
    uint8_t cm_digest[HASH_DIGEST_LENGTH];
};
```

The idea is to use Keccak to maintain an additional state during Sign. It will be used in two different places. The first one is the first `for` cycle iteration while the second one is the third and last iteration. In fact, an attacker has to introduce a fault during the last iteration in order to preserve the information and successfully recover the private key.

Implementation — In order to show the main logic of the countermeasure we provide the pseudocode related to the Cortex-M4 implementation of the signature procedure. The pseudocode refers to the balanced/small variant, the fast variant differs, as shown in Section 1.10, in the management of the data structures, but the countermeasure implementation is equivalent. The shown pseudocode focuses mainly on the countermeasure itself, hence the seed tree and Merkle tree are represented in a simplified way that only shows their incremental nature. The code lines related to the countermeasure will be represented in purple.

Algorithm 5: HardenedSign($\mathtt{sk}, \mathtt{Msg}$) - balanced/small

Input: $\mathtt{sk}$: secret key $\mathtt{Seed}_{\mathtt{sk}} \in \{0,1\}^{2\lambda}$;
$\mathtt{Msg} \in \{0,1\}^*$: message;
Output: $\mathtt{Sgn}$: signature;
$\mathtt{error\_code}$: either 0 if no errors were detected, or 1 if errors were detected;
Data: λ: security parameter;
c: constant defined as $c = 2t - 1$;
$g \in \mathbb{F}_p^*$: generator of $\mathbb{E}$;
t: number of rounds;
w: weight of the second challenge;

```
// Expanding secret key, V^T is the only requirement to know H
```

1 $\bar{\mathbf{e}}, \bar{\mathbf{e}}_G, \mathbf{V}^\top, \overline{\mathbf{M}} \leftarrow \mathsf{ExpandSK}(\mathtt{Seed}_{\mathtt{sk}})$ | $\bar{\mathbf{e}}, \mathbf{V}^\top \leftarrow \mathsf{ExpandSK}(\mathtt{Seed}_{\mathtt{sk}})$

2 $\mathtt{RSeed} \xleftarrow{\$} \{0,1\}^\lambda, \quad \mathtt{Salt} \xleftarrow{\$} \{0,1\}^{2\lambda}$
3 $\mathsf{HashInit}(\mathtt{cm\_state})$
4 $\mathsf{HashInit}(\mathtt{cmt}_1^{\mathtt{state}})$
5 $\mathtt{sTree} \leftarrow \mathsf{InitializeSeedTree}(\mathtt{RSeed})$

6 **for** i **from** 1 **to** t **do**
7 $\quad \mathsf{Seed} \leftarrow \mathsf{IncrementalSeedTree}(\mathtt{sTree})$

$\overline{\mathbf{e}}'_G, \mathbf{u}' \leftarrow \mathsf{CSPRNG} - \mathbb{F}_z^m \times \mathbb{F}_p^n\,(\mathtt{Seed} \mid \mathtt{Salt} \mid i+c)$	$\overline{\mathbf{e}}', \mathbf{u}' \leftarrow \mathsf{CSPRNG} - \mathbb{F}_z^n \times \mathbb{F}_p^n\,(\mathtt{Seed} \mid \mathtt{Salt} \mid i+c)$
8 $\quad \overline{\mathbf{v}}_G \leftarrow \overline{\mathbf{e}}_G - \overline{\mathbf{e}}'_G$	
$\quad \overline{\mathbf{e}}' \leftarrow \overline{\mathbf{e}}'_G \overline{\mathbf{M}}$	

$\quad$ // Compute $\mathbf{v}$ such that $\mathbf{v} \star \mathbf{e}' = \mathbf{e}$
9 $\quad \overline{\mathbf{v}} \leftarrow \overline{\mathbf{e}} - \overline{\mathbf{e}}'$
$\quad$ // In R-SDP $\mathbf{v}, \mathbf{u}$ share the same memory area, this does not hold in R-SDP(G)
10 $\quad \mathbf{v} \leftarrow g^{\overline{\mathbf{v}}}$
11 $\quad \mathbf{u} \leftarrow \mathbf{v} \star \mathbf{u}'$
12 $\quad \mathbf{s}' \leftarrow \mathbf{u}\mathbf{H}^\top$
$\quad$ // Update the cm_state with Seed, $\overline{\mathbf{e}}'$, and $\mathbf{u}'$
13 $\quad \mathtt{cm\_state} \leftarrow \mathsf{HashUpdate}(\mathtt{Seed})$
14 $\quad \mathtt{cm\_state} \leftarrow \mathsf{HashUpdate}(\overline{\mathbf{e}}')$
15 $\quad \mathtt{cm\_state} \leftarrow \mathsf{HashUpdate}(\mathbf{u}')$
$\quad$ // $\mathtt{cmt}_0^{\mathtt{in}}$ = $\mathbf{s}' \mid \overline{\mathbf{v}}/\overline{\mathbf{v}}_G \mid \mathtt{Salt}$

16 $\quad \mathtt{cmt}_0^{\mathtt{in}} \leftarrow (\mathbf{s}' \mid \overline{\mathbf{v}} \mid \mathtt{Salt})$	$\mathtt{cmt}_0^{\mathtt{in}} \leftarrow (\mathbf{s}' \mid \overline{\mathbf{v}}_G \mid \mathtt{Salt})$

$\quad$ // Commitments are stored directly on the Merkle tree, go up by a level if possible
17 $\quad \mathtt{mTree}[\mathtt{Leaf}_i] \leftarrow \mathsf{Hash}\left(\mathtt{cmt}_0^{\mathtt{in}} \mid i+c\right)$
18 $\quad \mathtt{mTree} \leftarrow \mathsf{IncrementalMerkleTree}(\mathtt{mTree})$
$\quad$ // $\mathtt{cmt}_1^{\mathtt{in}}$ = Seed | Salt | (roundIndex $+ 2t - 1$)
19 $\quad \mathtt{cmt}_1^{\mathtt{in}} \leftarrow (\mathtt{Seed} \mid \mathtt{Salt} \mid i+c)$
20 $\quad \mathtt{cmt}_1^{\mathtt{state}} \leftarrow \mathsf{HashUpdate}(\mathtt{cmt}_1^{\mathtt{in}})$
// Squeezing the first state
21 $\mathtt{cm\_digest} \leftarrow \mathsf{HashFinAndSqueeze}(\mathtt{cm\_state})$
// Concatenation of the two digests
22 $\mathtt{digest}_{\mathtt{cmt}} \leftarrow \mathsf{Hash}(\mathtt{mTree}) \mid \mathsf{HashFinAndSqueeze}(\mathtt{cmt}_1^{\mathtt{state}})$
// Computing first challenge
23 $\mathtt{digest}_{\mathtt{Msg}} \leftarrow \mathsf{Hash}(\mathtt{Msg})$
24 $\mathtt{digest}_{\mathtt{chall}_1} \leftarrow \mathsf{Hash}(\mathtt{digest}_{\mathtt{Msg}} \mid \mathtt{digest}_{\mathtt{cmt}} \mid \mathtt{Salt})$
25 $\mathtt{chall}_1 \leftarrow \mathsf{CSPRNG} - (\mathbb{F}_p^*)^t\,(\mathtt{digest}_{\mathtt{chall}_1} \mid t+c)$
26 $\mathsf{HashInit}(\mathtt{cmt}_1^{\mathtt{state}})$
// Computing first response
27 $\mathtt{sTree} \leftarrow \mathsf{InitializeSeedTree}(\mathtt{RSeed})$
28 **for** i **from** 1 **to** t **do**
$\quad$ // Here a signature round is executed again but $\mathbf{y}$ is stored in the same place of $\mathbf{v}$
29 $\quad \mathsf{Seed} \leftarrow \mathsf{IncrementalSeedTree}(\mathtt{sTree})$

$\overline{\mathbf{e}}'_G, \mathbf{u}' \leftarrow \mathsf{CSPRNG} - \mathbb{F}_z^m \times \mathbb{F}_p^n\,(\mathtt{Seed} \mid \mathtt{Salt} \mid i+c)$	$\overline{\mathbf{e}}', \mathbf{u}' \leftarrow \mathsf{CSPRNG} - \mathbb{F}_z^n \times \mathbb{F}_p^n\,(\mathtt{Seed} \mid \mathtt{Salt} \mid i+c)$
30 $\quad \overline{\mathbf{v}}_G \leftarrow \overline{\mathbf{e}}_G - \overline{\mathbf{e}}'_G$	
$\quad \overline{\mathbf{e}}' \leftarrow \overline{\mathbf{e}}'_G \overline{\mathbf{M}}$	

$\quad \mathbf{e}' \leftarrow g^{\overline{\mathbf{e}}'}$
31 $\quad \mathbf{y} \leftarrow \mathbf{u}' + \mathtt{chall}_1[i]\mathbf{e}'$
32 $\quad \mathtt{cmt}_1^{\mathtt{state}} \leftarrow \mathsf{HashUpdate}(\mathbf{y})$

```
   // Computing second challenge
33 cmt_1^state ← HashUpdate(digest_chall_1)
34 digest_chall_2 ← HashFinAndSqueeze(cmt_1^state)
35 chall_2 ← CSPRNG − B_(t,w) (digest_chall_2 | t + c + 1)
   // Computing second response
36 HashInit(cm_state)
37 sTree ← InitializeSeedTree(RSeed)
38 for i from 1 to t do
       // Recompute v̄/v̄_G
39     Seed ← IncrementalSeedTree(sTree)
       ē'_G, u' ←CSPRNG − F_z^m × F_p^n (Seed | Salt | i + c)  |  ē', u' ←CSPRNG − F_z^n × F_p^n (Seed | Salt | i + c)
40     v̄_G ← ē_G − ē'_G
       ē' ← ē'_G M̄
       v̄ ← ē − ē'
41     v ← g^v̄
42     u ← v ⋆ u'
43     s' ← uH^⊤
44     cmt_0^in ← (s' | v̄ | Salt)  |  cmt_0^in ← (s' | v̄_G | Salt)
       if chall_2[i] = 0 then
45         resp[i]_0 ← (··· , v̄_G)  |  resp[i]_0 ← (··· , v̄)
46     e' ← g^ē'
       // Recompute y
47     y ← u' + chall_1[i]e'
48     cmt_1^in ← (Seed | Salt | i + c)
49     if chall_2[i] = 0 then
50         resp[i]_0 ← (y, ··· )
51         resp[i]_1 ← Hash (cmt_1^in)
52     mTree[Leaf_i] ← Hash (cmt_0^in | i + c)
53     mTree ← IncrementalMerkleTree(mTree)
54     Proof ← ComputeProof(mTree)
55     Path ← SeedPath(sTree)
56     cm_state ← HashUpdate(Seed)
57     cm_state ← HashUpdate(ē')
58     cm_state ← HashUpdate(u')
   // Squeezing the second state
59 digest_ver ← HashFinAndSqueeze(cm_state)
   // Check if the two states are equal
60 if digest_ver = cm_digest then
61     Sgn ← (Salt, digest_cmt, digest_chall_2, Path, Proof, resp)
62     return 0
63 else
64     return 1
```

Countered attacks — The implemented countermeasure counters almost all attacks against Sign or at least makes them significantly harder to achieve. In fact, if an attacker wants to alter

$\overline{\mathbf{e}}'[i], \mathbf{u}'[i]$ or any seeds or inner nodes it has to inject the same exact fault, twice, in distinct for cycles. If an attacker is able to achieve so, the state will remain the same between the two for cycles, and it will not be detected. In any other case, our new version of Sign will not produce a signature, and it will report an error.

3 | Experimental Evaluation

In this chapter we will present the experimental setup used to develop the countermeasure. In addition, we will introduce the framework utilized during testing and benchmarking. Lastly, we will briefly comment the achieved results.

3.1. Experimental setup

Board — The analysis of the Cortex-M4 code was conducted on a `STM32F4DISCOVERY` board. Such board required a connection via a microUSB-USB cable (required for power supply and binary flashing) and a three-pins cable for serial connection (required for I/O with the board). The board has the following characteristics:

Table 3.1: 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

Software setup — The original Cortex-M4 implementation relied on the `pqm4` framework. The framework was developed to allow an easy way to test and execute benchmarks of many post-quantum KEMs and signature algorithms, over multiple boards. In our scenario, to make development easier, we created an independent Makefile that linked CROSS implementation with its dependencies independently of pqm4.

The first step was the identification of the dependencies. CROSS is almost self-contained, however, some external components were required by the protocol and some were required specifically by the Cortex-M4 implementation.

The header and source files `fips202/keccak1600` are required in order to properly use SHAKE and Keccak. In addition, the Makefile includes `keccak1600.S`, written in Assembly, in order to

allow low-level optimizations.

`randombytes` source and header file is required, as it is the primary source of randomness, and must be included. It provides a call to the TRNG (True Random Number Generator) or, if not available, a fallback implementation.

Then we had to include `libopencm3`, an open-source firmware library for many ARM microcontrollers (including Cortex-M4). `libopencm3` implements the `hal.h` library. `hal.h` (Hardware Abstraction Layer) provides a set of headers (interfaces) needed to manage input and output on the board.

Finally, we included the board-specific `.ld` files. Linker script files are required to ensure the correct compilation and organization of the binary on the board. They describe how and where the different sections of the code should be arranged in the memory.

Lastly, by using the `arm-gcc-none-eabi` toolchain and `st-link` package we compiled and deployed the code on the board.

The countermeasure has been tested in two ways. The first one consisted in the implementation of the attacks (presented in Section 2.3.1) in software using our custom build. Such attacks were correctly detected, and the signature was not produced. The second one consisted in the integration of the countermeasure in `pqm4` that provides a rich test suite to check that no edge cases were ignored.

`pqm4` were used also to produce the data collected and presented in the following section. The framework provides an ad-hoc suite for benchmarking, we collected data regarding speed and stack consumption for the eighteen variants of CROSS. For speed testing we used the available `m4opt` implementation of CROSS, for stack size estimation we used the `m4stack` implementation. The countermeasure will be the same on both variants.

Hardware setup — The board had to be connected via:

- microUSB-USB cable: required to supply power to the board and to flash binaries;
- serial-USB interface: required for I/O.

The board communicates via serial interface, specifically:

- Data are received through RXD-PA2 connection;
- Data are transmitted through TXD-PA3 connection.

In addition, a ground (GND) pin has to be connected. The board interacts through the standard `/dev/ttyUSB0` interface. The development was done on a Fedora Linux 43 (x86_64) machine with gcc version 15.2.0.

3.2. Experimental results

We will present some benchmarks in order to estimate the impact of countermeasures on performances. The countermeasure introduces the following amount of new calls:

- Two `hash_init` and two `hash_fin_and_squeeze`;
- Three `hash_update` for each iteration of the for cycle, thus $3 \cdot 2T = 6T$ calls;
- A final `memcmp`.

By comparing the graphs shown in Figure 3.1 and in Figure 3.2 we observe that the impact 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).

The impact on the stack is almost negligible (Figure 3.3 and Figure 3.4), the overhead introduced by the additional function calls, and the newly introduced struct, does not have a significant impact on stack size.

The first version of the countermeasure implemented the update of the state using a container for the input:

```
const uint16_t container_length = SEED_LENGTH_BYTES + sizeof(FZ_ELEM) * N +
                                  sizeof(FP_ELEM) * N + SALT_LENGTH_BYTES;
uint8_t input_state_container[container_length];
```

and it used three `memcpy` to align and format the input (`input` $\leftarrow$ `Seed`$[i] \mid \overline{\mathbf{e}}'[i] \mid \mathbf{u}'[i]$):

```
memcpy(&countermeasures_struct_i.cm_state, csprng_input, SEED_LENGTH_BYTES);
memcpy(&countermeasures_struct_i.cm_state + SEED_LENGTH_BYTES,
        e_bar_prime, sizeof(FZ_ELEM) * N);
memcpy(&countermeasures_struct_i.cm_state + SEED_LENGTH_BYTES + sizeof(FZ_ELEM) * N,
        u_prime, sizeof(FP_ELEM) * N);
```

However, we noticed that the approach based on three consecutive updates provided almost equivalent results in terms of performance. Additionally, we reduced the amounts of memory operations executed on the device. The tables that sum up the results are Table 3.2 for speed and Table 3.3 for stack consumption.

All things considered, the implemented countermeasure proved to be capable of detecting most of the attacks against Sign. Its performance impact highly depends on the variant of the algorithm and on the security category. In particular CROSS-R-SDP(G) is the most impacted variant. Its usage should be kept into consideration in devices that might be targeted by active

side-channel attacks.

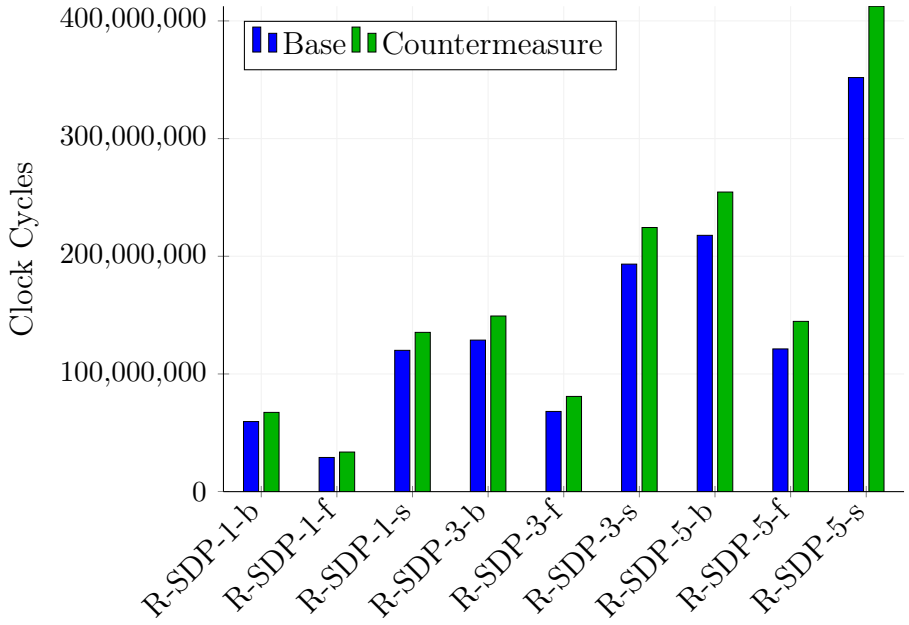

Figure 3.1: Increase in number of clock cycles when signing with countermeasures enabled (m4opt – R-SDP).

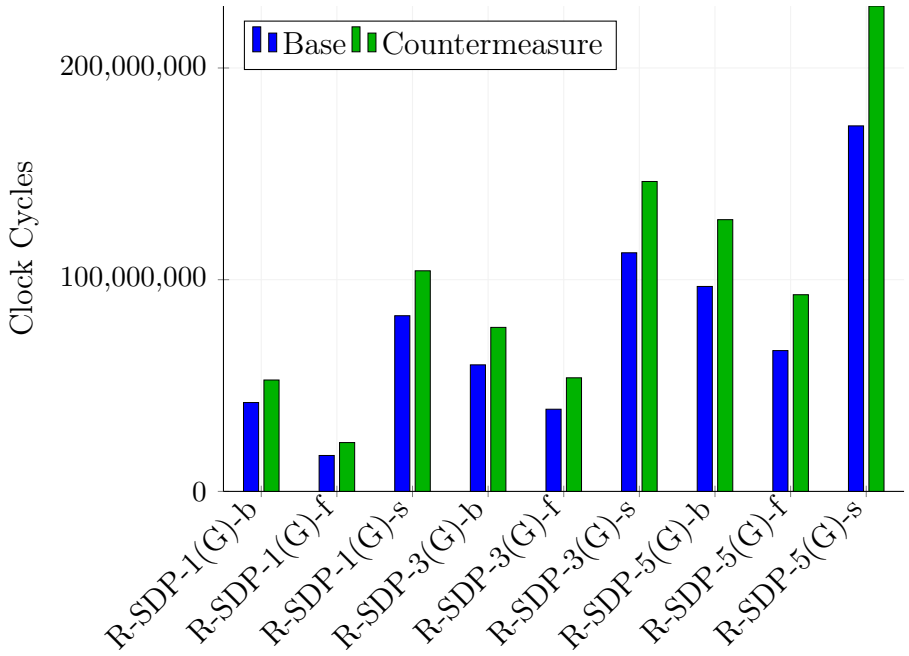

Figure 3.2: Increase in number of clock cycles when signing with countermeasures enabled (m4opt – R-SDP(G)).

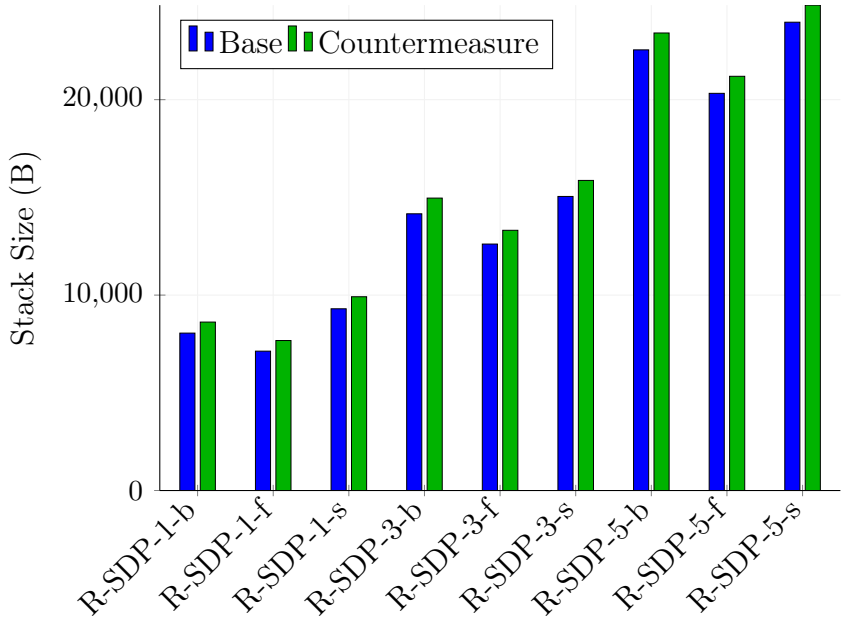

Figure 3.3: Increase in stack size (B) when signing with countermeasures enabled (m4stack – R-SDP).

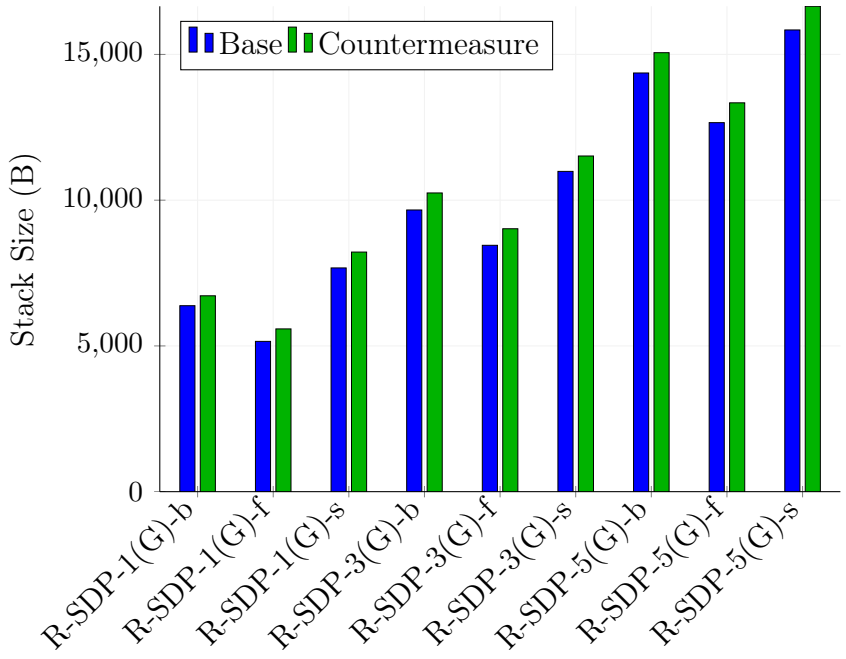

Figure 3.4: Increase in stack size (B) when signing with countermeasures enabled (m4stack – R-SDP(G)).

Table 3.2: Increase in number of clock cycles when signing with countermeasures enabled (m4opt).

Algorithm and Security Category	Optim. Corner	Number of CC (in millions) original	Number of CC (in millions) 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%

Table 3.3: 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%

4 | Conclusion

Our work achieved two distinct results. The first is the identification of the components of CROSS that are vulnerable to active side-channel attacks. The analysis focused not only on the primitives but also on the data structures, specifically the seed trees. Many of our observations may be used as a source of inspiration when conducting similar analyses against other post-quantum digital signature schemes, as many of them share the same basic concepts (e.g., the usage of seed trees).

The second result is the implementation of a new version of the signature primitive. Such version significantly increases the security margin of the Cortex-M4 implementation, by countering the most dangerous active side-channel attacks. Its performance has been benchmarked; the results (shown in Figures 3.1, 3.2 and in Figures 3.3, 3.4), show an increase in the number of clock cycles between 10% and 20% for R-SDP version and 25% and 40% for R-SDP(G). However, the increase in stack consumption is significantly lower, at most 9% for R-SDP(G). This makes our variant viable for embedded devices that already run the Cortex-M4 implementation of CROSS.

Bibliography

[1] M. A. Aardal, G. Adj, D. F. Aranha, A. Basso, I. A. C. Martínez, J. Chávez-Saab, M. C.-R. Santos, P. Dartois, L. D. Feo, M. Duparc, J. K. Eriksen, T. B. Fouotsa, D. L. G. Filho, B. Hess, D. Kohel, A. Leroux, P. Longa, L. Maino, M. Meyer, K. Nakagawa, H. Onuki, L. Panny, S. Patranabis, C. Petit, G. Pope, K. Reijnders, D. Robert, F. R. Henríquez, S. Schaeffler, and B. Wesolowski. *SQISign*. National Institute of Standards and Technology (NIST). URL `https://sqisign.org/`. Submission to the NIST Post-Quantum Cryptography Standardization Process (Round 2).

[2] G. Adj, N. Aragon, S. Barbero, M. Bardet, E. Bellini, L. Bidoux, J.-J. Chi-Domínguez, V. Dyseryn, A. Esser, T. Feneuil, P. Gaborit, R. Neveu, M. Rivain, L. Rivera-Zamarripa, C. Sanna, J.-P. Tillich, J. Verbel, and F. Zweydinger. *Mirath*. National Institute of Standards and Technology (NIST). URL `https://pqc-mirath.org/`. Submission to the NIST Post-Quantum Cryptography Standardization Process (Round 2).

[3] Arm Limited. *Arm® Cortex®-M4 Processor Technical Reference Manual*, 2020. URL `https://developer.arm.com/documentation/100166/0001/`.

[4] 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). URL `https://www.cross-crypto.com/`. Submission to the NIST Post-Quantum Cryptography Standardization Process (Round 2).

[5] M. Baldi, M. Battaglioni, F. Chiaraluce, A.-L. Horlemann-Trautmann, E. Persichetti, P. Santini, and V. Weger. A new path to code-based signatures via identification schemes with restricted errors, 2021. URL `https://arxiv.org/abs/2008.06403`.

[6] M. Baldi, S. Bitzer, A. Pavoni, P. Santini, A. Wachter-Zeh, and V. Weger. Zero knowledge protocols and signatures from the restricted syndrome decoding problem. Cryptology ePrint Archive, Paper 2023/385, 2023. URL `https://eprint.iacr.org/2023/385`.

[7] A. Barenghi, L. Breveglieri, I. Koren, and D. Naccache. Fault injection attacks on cryptographic devices: Theory, practice, and countermeasures. *Proceedings of the IEEE*, 100(11): 3056–3076, 2012. doi: 10.1109/JPROC.2012.2188769.

[8] C. Baum, L. Braun, W. Beullens, C. D. de Saint Guilhem, M. Klooß, C. Majenz, S. Mukherjee, E. Orsini, S. Ramacher, C. Rechberger, L. Roy, and P. Scholl. *FAEST*. National Institute of Standards and Technology (NIST). URL `https://faest.info/`. Submission to the NIST Post-Quantum Cryptography Standardization Process (Round 2).

[9] C. Baum, W. Beullens, S. Mukherjee, E. Orsini, S. Ramacher, C. Rechberger, L. Roy, and P. Scholl. One tree to rule them all: Optimizing GGM trees and OWFs for post-quantum signatures. Cryptology ePrint Archive, Paper 2024/490, 2024. URL `https://eprint.iacr.org/2024/490`.

[10] R. Benadjila, C. Bouillaguet, T. Feneuil, and M. Rivain. *MQOM*. National Institute of Standards and Technology (NIST). URL `https://mqom.org/`. Submission to the NIST Post-Quantum Cryptography Standardization Process (Round 2).

[11] E. R. Berlekamp, R. J. McEliece, and H. C. A. van Tilborg. On the inherent intractability of certain coding problems (corresp.). *IEEE Trans. Inf. Theory*, 24:384–386, 1978. URL `https://api.semanticscholar.org/CorpusID:34892814`.

[12] G. Bertoni, J. Daemen, M. Peeters, and G. V. Assche. The Keccak reference. URL `https://keccak.team/files/Keccak-reference-3.0.pdf`.

[13] J. W. Bos, O. Bronchain, L. Ducas, S. Fehr, Y.-H. Huang, T. Pornin, E. W. Postlethwaite, T. Prest, L. N. Pulles, and W. van Woerden. *HAWK*. National Institute of Standards and Technology (NIST). URL `https://hawk-sign.info/`. Submission to the NIST Post-Quantum Cryptography Standardization Process (Round 2).

[14] R. Boulifa, G. Di Natale, and P. Maistri. Countermeasures against fault injection attacks in processors: A review. *Information*, 16(4), 2025. ISSN 2078-2489. doi: 10.3390/info16040293. URL `https://www.mdpi.com/2078-2489/16/4/293`.

[15] L. Castelnovi, A. Martinelli, and T. Prest. Grafting trees: a fault attack against the SPHINCS framework. Cryptology ePrint Archive, Paper 2018/102, 2018. URL `https://eprint.iacr.org/2018/102`.

[16] M.-S. Chen, A. Hülsing, J. Rijneveld, S. Samardjiska, and P. Schwabe. From 5-pass MQ-based identification to MQ-based signatures. Cryptology ePrint Archive, Paper 2016/708, 2016. URL `https://eprint.iacr.org/2016/708`.

[17] M. Czuprynko, A. Mukherjee, and S. S. Roy. Correlation power analysis of LESS and CROSS. Cryptology ePrint Archive, Paper 2025/839, 2025. URL `https://eprint.iacr.org/2025/839`.

[18] W. Diffie and M. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654, 1976. doi: 10.1109/TIT.1976.1055638.

[19] M. J. Dworkin. Sha-3 standard: Permutation-based hash and extendable-output functions:, 2015-07-01 04:07:00 2015.

[20] T. Feneuil and M. Rivain. Threshold computation in the head: Improved framework for post-quantum signatures and zero-knowledge arguments. Cryptology ePrint Archive, Paper 2023/1573, 2023. URL https://eprint.iacr.org/2023/1573.

[21] A. Fiat and A. Shamir. How to prove yourself: Practical solutions to identification and signature problems. In A. M. Odlyzko, editor, *Advances in Cryptology — CRYPTO' 86*, pages 186–194, Berlin, Heidelberg, 1987. Springer Berlin Heidelberg. ISBN 978-3-540-47721-1.

[22] A. Genêt. On protecting SPHINCS+ against fault attacks. Cryptology ePrint Archive, Paper 2023/042, 2023. URL https://eprint.iacr.org/2023/042.

[23] A. Genêt, M. J. Kannwischer, H. Pelletier, and A. McLauchlan. Practical fault injection attacks on SPHINCS. Cryptology ePrint Archive, Paper 2018/674, 2018. URL https://eprint.iacr.org/2018/674.

[24] O. Goldreich. *Foundations of Cryptography, Volume 1: Basic Tools*. Cambridge University Press, Cambridge, UK, 2001.

[25] O. Goldreich, S. Goldwasser, and S. Micali. How to construct random functions. *J. ACM*, 33(4):792–807, Aug. 1986. ISSN 0004-5411. doi: 10.1145/6490.6503. URL https://doi.org/10.1145/6490.6503.

[26] D. Hankerson and A. Menezes. *Elliptic Curve Discrete Logarithm Problem*, pages 397–400. Springer US, Boston, MA, 2011. ISBN 978-1-4419-5906-5. doi: 10.1007/978-1-4419-5906-5_246. URL https://doi.org/10.1007/978-1-4419-5906-5_246.

[27] S. Jendral, E. Dubrova, Q. Guo, and T. Johansson. Correction fault attack on CROSS under unknown bit flips. Cryptology ePrint Archive, Paper 2025/1885, 2025. URL https://eprint.iacr.org/2025/1885.

[28] D. Johnson, A. Menezes, and S. Vanstone. The elliptic curve digital signature algorithm (ecdsa). *International Journal of Information Security*, 1(1):36–63, Aug. 2001. ISSN 1615-5262. doi: 10.1007/s102070100002. URL https://doi.org/10.1007/s102070100002.

[29] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. Spectre attacks: Exploiting speculative execution, 2018. URL https://arxiv.org/abs/1801.01203.

[30] P. C. Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In N. Koblitz, editor, *Advances in Cryptology — CRYPTO '96*, pages 104–113, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg. ISBN 978-3-540-68697-2.

[31] L. Lamport. Constructing digital signatures from a one-way function, 1979. URL https://www.microsoft.com/en-us/research/wp-content/uploads/2016/12/Constructing-Digital-Signatures-from-a-One-Way-Function.pdf.

[32] F. J. MacWilliams and N. J. A. Sloane. The theory of error-correcting codes. 1977. URL https://api.semanticscholar.org/CorpusID:118260868.

[33] C. A. Melchor, S. Bettaieb, L. Bidoux, T. Feneuil, P. Gaborit, N. Gama, S. Gueron, J. Howe, A. Hülsing, D. Joseph, A. Joux, M. Kulkarni, E. Persichetti, T. H. Randrianarisoa, M. Rivain, and D. Yue. *SDitH*. National Institute of Standards and Technology (NIST). URL https://sdith.org/. Submission to the NIST Post-Quantum Cryptography Standardization Process (Round 2).

[34] R. C. Merkle. Secrecy, authentication, and public key systems, 1979. URL https://www.ralphmerkle.com/papers/Thesis1979.pdf.

[35] P. Mondal, S. Adhikary, S. Kundu, and A. Karmakar. ZKFault: Fault attack analysis on zero-knowledge based post-quantum digital signature schemes. Cryptology ePrint Archive, Paper 2024/1422, 2024. URL https://eprint.iacr.org/2024/1422.

[36] National Institute of Standards and Technology (NIST). Post-quantum cryptography: Additional digital signature schemes. https://csrc.nist.gov/projects/pqc-dig-sig, Dec. 2016.

[37] National Institute of Standards and Technology (NIST). Post-quantum cryptography: Additional digital signature schemes. https://csrc.nist.gov/news/2022/request-additional-pqc-digital-signature-schemes, Sept. 2022.

[38] National Institute of Standards and Technology (NIST). Fips 186-5 digital signature standard (dss). https://csrc.nist.gov/pubs/fips/186-5/final, Feb. 2023.

[39] National Institute of Standards and Technology (NIST). Cryptographic algorithms and key sizes for personal identity verification. https://csrc.nist.gov/pubs/sp/800/78/5/final, July 2024.

[40] C. M. Pacurar, R. Bocu, and M. Iavich. An analysis of existing hash-based post-quantum signature schemes. *Symmetry*, 17(6), 2025. ISSN 2073-8994. doi: 10.3390/sym17060919. URL https://www.mdpi.com/2073-8994/17/6/919.

[41] M. Randolph and W. Diehl. Power side-channel attack analysis: A review of 20 years of study for the layman. *Cryptography*, 4(2), 2020. ISSN 2410-387X. doi: 10.3390/cryptography4020015. URL https://www.mdpi.com/2410-387X/4/2/15.

[42] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 26(1):96–99, Jan. 1983. ISSN 0001-0782. doi: 10.1145/357980.358017. URL https://doi.org/10.1145/357980.358017.

[43] R. Roth. *Linear Codes*, pages 26–49. Cambridge University Press, 2006.

[44] J. Schupp and G. Sigl. A horizontal attack on the codes and restricted objects signature scheme (CROSS). Cryptology ePrint Archive, Paper 2025/116, 2025. URL `https://eprint.iacr.org/2025/116`.

[45] J. Schupp, M. Gianvecchio, A. Barenghi, P. Karl, G. Pelosi, and G. Sigl. Optimizing the post quantum signature scheme CROSS for resource constrained devices. Cryptology ePrint Archive, Paper 2025/1928, 2025. URL `https://eprint.iacr.org/2025/1928`.

[46] P. W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM Journal on Computing*, 26(5):1484–1509, Oct. 1997. ISSN 1095-7111. doi: 10.1137/s0097539795293172. URL `http://dx.doi.org/10.1137/S0097539795293172`.

[47] C. team. Cross - m4 implementation repository. `https://github.com/joschupp/pqm4/blob/cross2.2`.

List of Figures

List of Tables