Message-recovery Laser Fault Injection Attack on Code-based Cryptosystems

Pierre-Louis Cayrel[1], Brice Colombier[1], Vlad-Florin Drăgoi[2,3], Alexandre Menu[4], and Lilian Bossuet[1]

[1] Univ Lyon, UJM-Saint-Etienne, CNRS, Laboratoire Hubert Curien UMR 5516, F-42023, SAINT-ETIENNE, France.
{pierre.louis.cayrel, b.colombier, lilian.bossuet}@univ-st-etienne.fr
[2] Department of Mathematics and Computer Sciences, "Aurel Vlaicu" University of Arad, Bd. Revoluţiei, No. 77, 310130-Arad, Romania.
vlad.dragoi@uav.ro
[3] LITIS, University of Rouen Normandie, Avenue de l'université, 76801 Saint-Étienne-du-Rouvray, France.
[4] IMT, Mines Saint-Etienne, Centre CMP, Equipe Commune CEA Tech - Mines Saint-Etienne F-13541 Gardanne FRANCE.
alexandre.menu@emse.fr

Keywords: code-based cryptography · syndrome decoding problem · laser fault injection

Abstract. Code-based public-key cryptosystems are promising candidates for standardisation as quantum-resistant public-key cryptographic algorithms. Their security is based on the hardness of the syndrome decoding problem. Computing the syndrome in a finite field, usually $\mathbb{F}_2$, guarantees the security of the constructions. We show in this article that the problem becomes considerably easier to solve if the syndrome is computed in $\mathbb{N}$ instead. By means of laser fault injection, we illustrate how to force the matrix-vector product in $\mathbb{N}$ by corrupting specific instructions, and validate it experimentally. To solve the syndrome decoding problem in $\mathbb{N}$, we propose a reduction to an integer linear programming problem. We leverage the computational efficiency of linear programming solvers to obtain real time message recovery attacks against all the code-based proposals to the NIST Post-Quantum Cryptography standardisation challenge. We perform our attacks on worst-case scenarios, *i.e.* random binary codes, and retrieve the initial message within minutes on a desktop computer. When considering parameters of the code-based submissions to the NIST PQC standardisation challenge, all of them can be attacked in less than three minutes.

1 Introduction

For the last three decades, public key cryptography has been an indispensable component of digital communications. Communication protocols rely on three core cryptographic functionalities: public key encryption (PKE), digital signatures, and key exchange. These are implemented using Diffie-Hellman key exchange [15], the RSA cryptosystem [43], and elliptic curve cryptosystems [25, 37]. Their security stands on the difficulty of number theoretic problems such as the Integer Factorization or the Discrete Logarithm Problem. Shor proved that quantum computers can efficiently solve each of these problems [45], rendering all public-key cryptosystems (PKC) based on such assumptions impotent.

Since then, cryptographers proposed alternative solutions which are safe in the quantum era. These schemes are called *post-quantum secure* [8]. In 2015, the National Institute of Standards and Technology (NIST) made a call to the community to propose post-quantum secure solutions for standardisation. Several candidates have been submitted based on various hard problems (lattices, error-correcting codes, multivariate systems of equations and hash functions). Here we choose to analyze code-based candidates.

The hardness of general decoding for a linear code is an $\mathcal{NP}$-complete problem in coding theory [7], which makes it an appealing candidate, especially for code-based post-quantum cryptography. From the original scheme proposed by McEliece [34] to the latest variants proposed for the NIST competition [3, 36, 4], the majority of these PKCs assess their security on the syndrome decoding problem (SDP). Informally, for a binary linear code $\mathcal{C}$ of length n and dimension k, having a parity-check matrix $\boldsymbol{H}$, the SDP is defined as follows: find a binary vector $\boldsymbol{x}$ having less than t values equal to one, such that $\boldsymbol{H}\boldsymbol{x} = \boldsymbol{s}$, where $\boldsymbol{s} \in \mathbb{F}_2^{n-k}$ is given.

A possible solution to solve the general decoding problem is to use Integer Linear Programming (ILP). The idea was first proposed by Feldman [18] and later improved by Feldman *et al.* [17]. Since the initial problem is nonlinear, some relaxation was proposed in order to decrease the complexity. For more details on these aspects, we refer the reader to the excellent review of Helmling *et al.* [21]. One of the latest proposals [48], introduces a new method for transforming the initial decoding problem into an ILP, formalism that fits perfectly the ideas that we will put forward in this article. Let us briefly explain the idea of Tanatmis *et al.* [48].

The general decoding problem can be tackled using the well-known Maximum-Likelihood decoder. Let $\mathcal{C}$ be a binary linear code of length n and dimension k, with parity-check matrix $\boldsymbol{H}$. The integer linear programming formulation of maximum-likelihood decoding is given in Equation (1).

$$\min\{\boldsymbol{v}\boldsymbol{x}^t \mid \boldsymbol{H}\boldsymbol{x} = 0 \ , \boldsymbol{x} \in \{0,1\}^n\}, \tag{1}$$

where $\boldsymbol{v}$ is the log-likelihood ratio (see [31, 17]). Tanatmis *et al.* proposed to introduce an auxiliary positive variable $\boldsymbol{z} \in \mathbb{N}^{n-k}$ and define a new problem given in Equation (2).

$$\min\{\boldsymbol{v}\boldsymbol{x}^t \mid \boldsymbol{H}\boldsymbol{x} = 2\boldsymbol{z} \ , \boldsymbol{x} \in \{0,1\}^n, \boldsymbol{z} \in \mathbb{N}^{n-k}\}. \tag{2}$$

The advantage of (2) compared to (1) is that $\boldsymbol{z}$ introduced real/integer constraints, which are much easier to handle for solvers than binary constraints.

Also, there are as many constraints as rows in $\boldsymbol{H}$. Finding an appropriate variable $\boldsymbol{z}$ is not trivial and algorithms such as [48] are constantly modifying the values of $\boldsymbol{z}$ in order to find the correct solution.

Inspired by the ideas of Tanatmis *et al.*, we define the SDP as an ILP. Then we propose to determine a valid constraints integer vector $\boldsymbol{z}$ such as the problem becomes easier to solve. To achieve this, we will put forward a recent result in laser fault injection [13].

Understanding how fault attacks allow to corrupt the instructions executed by a microcontroller has been a vivid topic of research in recent years. While electromagnetic fault injection is probably the most commonly used technique, certainly because of its relatively low cost, it has several drawbacks. Indeed, while the "instruction skip" or "instruction replay" fault models were clearly identified [44], most of the time going down to the instruction set level leaves a lot of question open [38]. As such, only a handful of the observed faults can be tracked down and explained by a modification of the bits in the instruction [29]. Last, but not least, electromagnetic fault injection usually exhibits poor repeatability [12], as low as a few percents in some cases.

Conversely, another actively studied technique is laser fault injection, which exhibits several advantages when it comes to interpreting the observed faults. For example, the instruction skip fault model has been experimentally validated by laser fault injection, with perfect repeatability and the ability to skip one or multiple instructions [16]. On a deeper level of understanding, it has been shown in [13] that it was possible to perform a bit-set on any of the bits of an instruction while it is fetched from the Flash memory of the microcontroller. This modification is temporary since it is performed during the fetch process. As such, the instruction stored in the Flash memory remains untouched. We place ourselves in this framework here, and will show how this powerful fault model gives the possibility to actively corrupt the instructions and allows to mount a fault attack on code-based cryptosystems.

Contributions This article makes the following contributions. First, we propose a new attack on code-based cryptosystems which security relies on the SDP. We show by simulations that, if the syndrome is computed in $\mathbb{N}$ instead of $\mathbb{F}_2$, then SDP can be solved in polynomial time by linear programming. Second, we experimentally demonstrate that such a change of set is feasible by corrupting the instructions executed during the syndrome computation. To this end, we rely on backside laser fault injection in Flash memory in order to transform an addition over $\mathbb{F}_2$ into an addition over $\mathbb{N}$. We perform this by corrupting the instruction when it is fetched from Flash memory, thereby replacing the exclusive-OR operation with an add-with-carry operation. We then show, starting with the faulty syndrome, that the secret error-vector can be recovered very efficiently by linear programming. By means of software simulations we show that, in particular, this attack scales to cryptographically strong parameters for the considered cryptosystems. Finally, we highlight a very practical feature of the attack, which is that only a fraction of the syndrome entries need to be faulty in order for the attack to be successful. On top of that, this fraction decreases when the cryptographic parameters grow. This has important practical consequences, since the attack can be carried out even if the fault injection is not perfectly repeatable. Moreover, this also drastically reduces the number of inequalities to be considered in the linear programming problem, thereby making the problem easier to solve.

We perform a message recovery attack (MRA) against code-based cryptosystems based on Niederreiter's model. Specifically, we recover the message from one faulty syndrome and the public key. The attacker must have physical access to the device, where he performs a laser fault injection during encryption, i.e., in the matrix-vector multiplication. The total number of faults the attacker must inject is upper-bounded by the code dimension.

Our attack was performed on a real microcontroller, embedding an ARM Cortex-M3 core, where we corrupted the XOR operation and obtained the faulty outputs. As in our case, one needs to perform single-bit and double-bit faults, in a repeatable and controlled manner. This method strongly relies on the work of Colombier et al. [13] and thus can be verified and repeated experimentally. These faults are transient and the content of the Flash memory is unchanged. We stress out that constant-time implementations are of great help for this attack setting.

We chose to attack here two multiplication methods, i.e., the schoolbook and the packed version. The former is considered in the NTL library, while the later is the reference implementation of the Classic McEliece proposal.

The article is organised as follows. In Section 2, we detail the main code-based cryptosystems, and in particular the candidates to the NIST post-quantum cryptography competition. Section 3 defines the SDP in $\mathbb{N}$ and shows how it relates to linear programming. In Section 4, we show how the corruption of instructions by laser fault injection allows to switch from $\mathbb{F}_2$ to $\mathbb{N}$ during the syndrome computation. Section 5 presents experimental results following the attack path, from laser fault injection to the exploitation of the faulty syndrome by linear programming. Finally, we conclude this article in Section 6.

2 Code-based cryptosystems

2.1 NIST competition

The main goal of the process started by the NIST is to replace three standards that are considered the most vulnerable to quantum attacks, *i.e.*, FIPS 186-4[5] (for digital signatures), NIST SP 800-56A[6] and NIST SP 800-56B[7](both for keys establishment in public-key cryptography). For the first round of this competition, 69 candidates met the minimum criteria and the requirements imposed by

[5] `https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf`

[6] `https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-56Ar2.pdf`

[7] `https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-56Br1.pdf`

the NIST. 26 out of 69 were announced on January 30, 2019 for moving to the second round. From these, 17 are public-key encryption and/or key-establishment schemes and 9 are digital signature schemes. The majority of these proposals are based on the two important theoretical fields, i.e., coding theory and lattice theory.

Here, we will analyse solutions relying on error-correcting codes in the Hamming metric. The main code-based candidates for PKE and KEM (Key Encapsulation Mechanism) are: Classic McEliece and NTS-KEM, BIKE, HQC, LEDAcrypt. All of these have in common the same hard problem on which they sustain their security, i.e., the binary SDP.

2.2 Coding theory – preliminaries

We choose the following conventions and notations. A finite field is denoted by $\mathbb{F}$, and the ring of integers by $\mathbb{N}$. Vectors (column vectors) are written in bold, *e.g.*, a binary vector of length n is $\boldsymbol{x} \in \{0,1\}^n$. Matrices are written in bold capital letters, *e.g.*, an $m \times n$ integer matrix is $\boldsymbol{A} = (a_{i,j})_{\substack{0 \le i \le m-1 \\ 0 \le j \le n-1}} \in \mathcal{M}_{m,n}(\mathbb{N})$. A row sub-matrix of $\boldsymbol{A}$ indexed by a set $I \subseteq \{0,\ldots,m-1\}$ is denoted by $\boldsymbol{A}_{I,} = (a_{i,j})_{i \in I, 0 \le j \le n-1}$. The same applies to column vectors, *i.e.*, $\boldsymbol{x}_I$ is the sub-vector induced by the set I on $\boldsymbol{x}$.

Error correcting codes We say that $\mathcal{C}$ is an $[n,k]$ linear error-correcting code, or simply a linear code, over a finite field $\mathbb{F}$ if $\mathcal{C}$ is a linear subspace of dimension k of the vector space $\mathbb{F}^n$, where k,n are positive integers with $k < n$. The elements of $\mathcal{C}$ are called codewords. The support of a codeword $\mathsf{Supp}(\boldsymbol{c})$ is the set of non-zero positions of $\boldsymbol{c}$. We will represent a code either by its generator matrix, $\boldsymbol{G} \in \mathcal{M}_{k,n}(\mathbb{F})$ ($\mathtt{rank}(\boldsymbol{G}) = k$), or by its parity-check matrix, $\boldsymbol{H} \in \mathcal{M}_{n-k,n}(\mathbb{F})$, ($\mathtt{rank}(\boldsymbol{H}) = n-k$). $\boldsymbol{H}$ satisfies $\boldsymbol{H}\boldsymbol{c}^t = \boldsymbol{0}, \forall \boldsymbol{c} \in \mathcal{C}$.

Decoding The main goal of an error correcting code, as its name strongly suggests, is to correct errors, mainly coming in a communication process, from a noisy channel. To correct errors, R. Hamming [20], proposed to endow the vector space $\mathbb{F}^n$ with a metric, that carries his name. The Hamming distance between two vectors, $\mathsf{d_H}(\boldsymbol{x},\boldsymbol{y})$, is the number of coordinates on which $\boldsymbol{x}$ and $\boldsymbol{y}$ differ. $\mathsf{d_H}$ induces a weight, $\mathsf{wt}(\boldsymbol{x}) = \#\mathsf{Supp}(\boldsymbol{x})$. $\forall\, \mathcal{C}$, we define the minimum distance of $\mathcal{C}$ as $\mathsf{d}_{\min} = \min\{\mathsf{d_H}(\boldsymbol{c},\boldsymbol{c}^*) \mid (\boldsymbol{c},\boldsymbol{c}^*) \in \mathcal{C} \times \mathcal{C}, \boldsymbol{c} \neq \boldsymbol{c}^*\}$. Any vector $\boldsymbol{y} \notin \mathcal{C}$ at distance at most $t = \lfloor (\mathsf{d}_{\min} - 1)/2 \rfloor$ from a codeword $\boldsymbol{c}$ can be uniquely decoded into $\boldsymbol{c}$, *i.e.*, $\boldsymbol{c}$ is the closest codeword to $\boldsymbol{y}$. One general strategy of finding $\boldsymbol{c}$ given $\boldsymbol{y}$ is using its syndrome. Let us suppose, for the sake of simplicity, that $\mathbb{F} = \mathbb{F}_2$ and $\boldsymbol{y} = \boldsymbol{c} \oplus \boldsymbol{x} \in \mathbb{F}_2^n$ where $\boldsymbol{x}$ is the error vector. Since $\boldsymbol{H}\boldsymbol{c}^t = \boldsymbol{0}_{n-k}$ we deduce that $\boldsymbol{H}\boldsymbol{y}^t = \boldsymbol{H}\boldsymbol{x}^t$. Now, we define the SDP.

Definition 1 (SDP).
Input: $\mathbb{F}$ *a field,* $\boldsymbol{H} \in \mathcal{M}_{n-k,n}(\mathbb{F})$ *of rank* $n-k$,
a vector $\boldsymbol{s} \in \mathbb{F}^{n-k}$*, and* $t \in \mathbb{N}^*$*,* $t \neq 0$.
Output: $\boldsymbol{x} \in \mathbb{F}^n$*, with* $\mathsf{wt}(\boldsymbol{x}) \le t$*, such that* $\boldsymbol{H}\boldsymbol{x} = \boldsymbol{s}$

2.3 NIST second-round code-based PKC

There are three major types on KEM/PKE code-based schemes accepted in the second round of the NIST challenge.

The first type is based on binary Goppa codes and it is represented by the recent merge of classic McEliece (classic McEliece) and a variant of the combined McEliece and Niederreiter scheme (NTS-KEM). Here, the private key is a structured code, and the public key its masked variant.

The second type is based on quasi-cyclic Low Density Parity-Check codes (LEDAcrypt) or quasi-cyclic Moderate Density Parity-Check codes (BIKE), in a rather similar construction to the McEliece scheme. The public and private keys follow the same idea as for the first type.

The third type is completely different from the previous ones in the sense that it does not require a masking technique applied to a structured code. It uses, like the second type, quasi-cyclic codes (HQC).

As the first two types of proposals base their construction on the the McEliece [34] and the Niederreiter scheme [40] we choose to illustrate here these two basic schemes. (see Table 1). There are three algorithms defining PKE scheme, i.e., *key generation* (KeyGen), *encryption* (Encrypt) and *decryption* (Decrypt).

Table 1. McEliece and Niederreiter PKE schemes

McEliece PKE	Niederreiter PKE
$\mathsf{KeyGen}(n,k,t) = (\mathsf{pk},\mathsf{sk})$	
$\boldsymbol{H}$-parity-check matrix of $\mathcal{C}$	$\boldsymbol{G}$-generator matrix of $\mathcal{C}$
\\\\ $\mathcal{C}$ an $[n,k]$ that corrects t errors	
An $n \times n$ permutation matrix $\boldsymbol{P}$	
A $k \times k$ invertible matrix $\boldsymbol{S}$	An $(n-k)\times(n-k)$ invertible matrix $\boldsymbol{S}$
Compute $\boldsymbol{G}_{\text{pub}} = \boldsymbol{SGP}$	Compute $\boldsymbol{H}_{\text{pub}} = \boldsymbol{SHP}$
$\mathsf{pk} = (\boldsymbol{G}_{\text{pub}}, t)$	$\mathsf{pk} = (\boldsymbol{H}_{\text{pub}}, t)$
$\mathsf{sk} = (\boldsymbol{S},\boldsymbol{G},\boldsymbol{P})$	$\mathsf{sk} = (\boldsymbol{S},\boldsymbol{H},\boldsymbol{P})$
$\mathsf{Encrypt}(\boldsymbol{m},\mathsf{pk}) = \boldsymbol{z}$	
Encode $\boldsymbol{m} \to \boldsymbol{c} = \boldsymbol{m}\boldsymbol{G}_{\text{pub}}$	Encode $\boldsymbol{m} \to \boldsymbol{e}$
Choose $\boldsymbol{e}$	
\\\\ $\boldsymbol{e}$ a vector of weight t	
$\boldsymbol{z} = \boldsymbol{c} + \boldsymbol{e}$	$\boldsymbol{z} = \boldsymbol{H}_{\text{pub}}\boldsymbol{e}$
$\mathsf{Decrypt}(\boldsymbol{z},\mathsf{sk}) = \boldsymbol{m}$	
Compute $\boldsymbol{z}^* = \boldsymbol{z}\boldsymbol{P}^{-1}$	Compute $\boldsymbol{z}^* = \boldsymbol{S}^{-1}\boldsymbol{z}$
$\boldsymbol{z}^* = \boldsymbol{mSG} + \boldsymbol{e}\boldsymbol{P}^{-1}$	$\boldsymbol{z}^* = \boldsymbol{HPe}$
$\boldsymbol{m}^* = \mathcal{D}ecode(\boldsymbol{z}^*, \boldsymbol{G})$	$\boldsymbol{e}^* = \mathcal{D}ecode(\boldsymbol{z}^*, \boldsymbol{H})$
Retrieve $\boldsymbol{m}$ from $\boldsymbol{m}^*\boldsymbol{S}^{-1}$	Retrieve $\boldsymbol{m}$ from $\boldsymbol{P}^{-1}\boldsymbol{e}^*$

2.4 Security and practical parameters

Basically, all the aforementioned schemes support their security on the hardness of the SDP. Hence, state-of-the-art algorithms for solving the SDP are used to set up the security level. The best strategy in this direction is the so-called class of Information Set Decoding (ISD) algorithms. The original algorithm proposed by Prange [42] was significantly improved [28, 46, 26, 32, 6, 33]. Under the assumption that the public code is indistinguishable from a random code, the ISD techniques are considered the best strategy for tackling the MRA. In all the code-based NIST proposals, the error weight t is of order $o(n)$, when $n \to \infty$. In this case, the time complexity of the ISD variants equals $2^{ct(1-o(1))}$, where c is a constant given by the code rate. Table 2 summarizes the second round code-based proposals, with practical parameters for three security levels, as required by the NIST.

3 Syndrome decoding over $\mathbb{N}$

In this section we will introduce a new problem, extremely similar to the SD, for which we propose an efficient algorithm.

3.1 Description of the problem

Definition 2 ($\mathbb{N}$-SDP).
Input: $\boldsymbol{H} \in \mathcal{M}_{n-k,n}(\mathbb{N})$ *with* $h_{i,j} \in \{0,1\}$ *for all* i, j
$\boldsymbol{s} \in \mathbb{N}^{n-k}$ *and* $t \in \mathbb{N}^*$ *with* $t \neq 0$.
Output: $\boldsymbol{x} \in \mathbb{N}^n$ *with* $x_i \in \{0,1\}$ *and* $\mathsf{wt}(\boldsymbol{x}) \leq t$,
such that $\boldsymbol{Hx} = \boldsymbol{s}$.

Notice that $\boldsymbol{H}$ and $\boldsymbol{x}$ are binary, as in the SDP, whereas $\boldsymbol{s}$ is integer. Basically, $\boldsymbol{H}$ and $\boldsymbol{x}$ are sampled exactly as for the SDP, it is only the operation, *i.e.*, matrix-vector multiplication, that changes, and thus its result.

Possible solutions Based on the similarities with SDP, one might try to solve $\mathbb{N}$-SDP using techniques from coding theory. We briefly enumerate three possible solutions.

1. The simplest solution is to solve it as a linear system. If we consider the system $\boldsymbol{Hx} = \boldsymbol{s}$, it has $n-k$ equations and n unknowns, and hence, can be solved efficiently. However, there are k free variables, and 2^k possible solutions, since $\boldsymbol{x} \in \{0,1\}$. For each instance, compute the Hamming weight, and stop when the value is smaller than or equal to t. This procedure is not feasible in practice for cryptographic parameters, due to the values of k.
2. Another possible solution is combinatorial. One can choose subsets of s_i elements from $\mathsf{Supp}(\boldsymbol{H}_{i,})$ for increasing values of i, until it finds the correct combinations. This solution can be further optimised by choosing subsets from a smaller set at each iteration, where previously selected positions are

Table 2. Code-based cryptosystems in the second round NIST challenge.

Security level	Scheme (variant)	n	k	t
128	Classic McEliece	3488	2720	64
	NTS-KEM	4096	3328	64
	BIKE (2-CPA)	20 326	10 163	134
	BIKE (3-CPA)	22 054	11 027	154
	BIKE (2-CCA)	23 558	11 779	134
	BIKE (3-CCA)	24 538	12 269	154
	LEDAcrypt (CPA)	21 706	10 853	133
	LEDAcrypt (CCA)	39 626	19 813	131
	HQC (1)	24 667	256	77
192	Classic McEliece	4608	3360	96
	NTS-KEM	8192	7152	80
	BIKE (2-CPA)	39 706	19 853	199
	BIKE (3-CPA)	43 366	21 683	226
	BIKE (2-CCA)	49 642	24 821	199
	BIKE (3-CCA)	54 086	27 043	226
	LEDAcrypt (CPA)	41 962	20 981	198
	LEDAcrypt (CCA)	76 138	38 069	196
	HQC (1)	43 669	256	117
	HQC (2)	46 747	256	117
256	Classic McEliece	6688	5024	128
	Classic McEliece	6960	5413	119
	Classic McEliece	8192	6528	128
	NTS-KEM	8192	6424	136
	BIKE (2-CPA)	65 498	32 749	264
	BIKE (3-CPA)	72 262	36 131	300
	BIKE (2-CCA)	81 194	40 597	264
	BIKE (3-CCA)	89 734	44 867	300
	LEDAcrypt (CPA)	70 234	35 117	263
	LEDAcrypt (CCA)	122 422	61 211	261
	HQC (1)	63 587	256	153
	HQC (2)	67 699	256	153
	HQC (3)	70 853	256	153

rejected from the updated set. Even so, the time complexity will be dominated by a product of binomial coefficients that is asymptotically exponential in t.

3. A modified ISD to the integer requirements. Let us choose the original algorithm of Prange [41],that is randomly permuting the matrix $\boldsymbol{H}$ (denote $\boldsymbol{P}$ such a permutation) until the support of the permuted $\boldsymbol{x}$ is included in the set $\{0, \ldots, n-k-1\}$, *i.e.*, the set where the $\boldsymbol{HP}$ is under upper triangular form. To put an integer matrix under upper triangular form, one has to use

the equivalent of the Gaussian elimination for the integers, *i.e.*, the Hermite normal form. So, by computing an integer matrix $\boldsymbol{A}$ and $\boldsymbol{H}^*$, s.t. $\boldsymbol{H}^*$ is upper triangular on its first $n-k$ positions we obtain:

$$\boldsymbol{AHP}\left(\boldsymbol{P}^t\boldsymbol{x}\right) = \boldsymbol{AH}'\boldsymbol{x}' = \boldsymbol{H}^*\boldsymbol{x}' = \boldsymbol{As}. \tag{3}$$

If $\mathsf{Supp}(\boldsymbol{x}') \subseteq \{0, \ldots, n-k-1\}$ then the new syndrome $\boldsymbol{s}^* = \boldsymbol{As}$ has rather small integer entries, that directly allow the computation of $\boldsymbol{x}'$. This algorithm has time complexity similar to the classic ISD, and hence, remains exponential in t.

Since all these methods are not feasible in practice for cryptographic parameters, we propose another solution. For that, let us notice the following fact.

Remark 1. As for the maximum-likelihood decoding problem, we can reformulate $\mathbb{N}$-SDP as an optimization problem, i.e.,

$$\min\{\mathsf{wt}(\boldsymbol{x}) \mid \boldsymbol{Hx} = \boldsymbol{s}, \boldsymbol{x} \in \{0,1\}^n\}, \tag{4}$$

where $\boldsymbol{H}$ and $\boldsymbol{s}$ are given as in Definition 2.

This fact leads us to searching for mathematical optimization techniques, such as integer linear programming.

3.2 Integer Linear programming

ILP was already used in a cryptographic context, mainly for studying stream ciphers [11, 10, 39]. The authors of [39] implemented ILP-based methods that gave practical results for Enocoro-128v2, as well as for calculating the number of active S-boxes for AES. In [10, 11] ILP was used for studying the Trivium stream cipher and the lightweight block cipher Ktantan. In all of these, the technique was to reformulate the original cryptographic problems by means of ILP, and use some well-known solvers in order to obtain practical evidence of their security. Typically, in [11] the authors used the CPLEX solver. There are mainly three big solvers for LP and ILP problems: lpSolve[8], IBM CPLEX[9] and Gurobi[10], recently tested for various types of practical problems [30].

We point here some necessary facts about ILP, as we will use ILP as a tool only. Interested readers might check [9, 22] for a deeper introspection.

Definition 3 (ILP problem). *Let n, m be two positive integers and $\boldsymbol{b} \in \mathbb{N}^n, \boldsymbol{s} \in \mathbb{N}^m$ and $\boldsymbol{A} \in \mathcal{M}_{m,n}(\mathbb{N})$. The ILP problem is defined as the optimization problem*

$$\min\{\boldsymbol{b}^t\boldsymbol{x} | \boldsymbol{Ax} = \boldsymbol{c}, \boldsymbol{x} \in \mathbb{N}^n, \boldsymbol{x} \geq 0\}. \tag{5}$$

Any vector $\boldsymbol{x}$ satisfying $\boldsymbol{Ax} = \boldsymbol{s}$ is called a feasible solution. If a feasible solution $\boldsymbol{x}^*$ satisfies the minimum condition in (5) then $\boldsymbol{x}^*$ is optimal. In order to redefine our initial problem, *i.e.*, (4) into an ILP problem, we need to redefine the Hamming weight of a vector as a linear operation.

[8] `http://lpsolve.sourceforge.net/5.5/`
[9] `https://www.ibm.com/products/ilog-cplex-optimization-studio`
[10] `https://www.gurobi.com`

3.3 Solving $\mathbb{N}$-SDP using ILP

Theorem 1. *Let us suppose that there exists a unique vector* $\boldsymbol{x}^* \in \{0,1\}^n$ *with* $\mathsf{wt}(\boldsymbol{x}^*) = t$*, solution to the* $\mathbb{N}$*-SDP. Then* $\boldsymbol{x}^*$ *is the optimum solution of an ILP problem.*

Proof. Suppose that such an $\boldsymbol{x}^*$ exists and is unique, *i.e.*, $\boldsymbol{H}\boldsymbol{x}^* = \boldsymbol{s}$ with $\boldsymbol{s} \in \mathbb{N}^{n-k}$ and $\mathsf{wt}(\boldsymbol{x}^*) = t$. We will construct an ILP problem for which $\boldsymbol{x}^*$ is the optimum solution. For that, we simply set $\boldsymbol{A}^t = \boldsymbol{H}, \boldsymbol{c} = \boldsymbol{s}$, and $\boldsymbol{b}^t = (1, \ldots, 1)$ in (5). Since $\boldsymbol{x} \in \{0,1\}^n$ $\mathsf{wt}(\boldsymbol{x}) = \sum_{i=1}^{n} x_i = (1, \ldots, 1) \cdot \boldsymbol{x}$, but this is equal to $\boldsymbol{b}^t\boldsymbol{x}^*$. The ILP problem we need to solve can now be defined as:

$$\min\{\boldsymbol{b}^t\boldsymbol{x} | \boldsymbol{H}^t\boldsymbol{x} = \boldsymbol{s}, \boldsymbol{x} \in \{0,1\}^n\}, \tag{6}$$

which is exactly (4). This implies that $\boldsymbol{x}^*$ is a feasible solution to (6), and as $\boldsymbol{x}^*$ is the unique vector satisfying $\boldsymbol{H}^t\boldsymbol{x}^* = \boldsymbol{s}$ with $\mathsf{wt}(\boldsymbol{x}^*) \leq t$, $\boldsymbol{x}^*$ is optimum for the minimum weight condition.

ILP problems are defined as LP problems with integer constraints, hence any algorithm for solving an LP problem could potentially be used as a subroutine for solving the corresponding ILP problem. Usually, these are formalised in a sequential process, where the solution to one LP problem is close to the solution to the next LP problem, and so on, until eventually the ILP problem is solved. One of the most efficient method for solving ILP problems is the *branch and cut* method. In a branch and cut algorithm, an ILP problem is relaxed into an LP problem that is solved using an algorithm for LP problems. If the optimal solution is integral then it gives the solution to the ILP problem. There are mainly two famous methods for solving the linear problem: the simplex and the interior point method.

The simplex algorithm, introduced by Dantzig in [14], is one of the most popular methods for solving LP problems. The idea of this algorithm is to move from one vertex to another on the underlying polytope, as long as the solution is improved. The algorithm stops when no more neighbours of the current vertex improve the objective function. It is known to be really efficient in practice, by solving a large class of problems in polynomial time. However, it was proved in [24] that there are instances where the simplex falls into the exponential time complexity class.

Interior point algorithms are alternative algorithms to the simplex method, and were first proposed by [23]. Several variants improved the initial method, also by providing polynomial time complexity [49, 27]. As the name suggests, this method starts by choosing a point in the interior of the feasible set. Moving inside the polyhedron, this point is improved, until the optimal solution is found.

Efficient solutions using interior point methods were proposed for the maximum-likelihood decoding of binary codes [47, 51, 52]. These have running times dominated by low-degree polynomial functions in the length of the code. Also, they are in particular very efficient for large scale codes [47, 51]. For these particular interesting arguments, we choose the interior point method for solving the $\mathbb{N}$-SDP.

Solving the $\mathbb{N}$*-SDP* The algorithm we propose here to solve the $\mathbb{N}$-SDP can be described as follows. Initiate the parameters from (6), solve a relaxation of the $\mathbb{N}$-SDP (using the interior point methods), round the solution to binary entries (using the method from [35]) and finally verify if the binary solution satisfies the parity-check equations and the weight condition. The relaxation of the ILP problem to LP problem is a common method, more exactly, the LP problem that we have to solve is:

$$\min\{\boldsymbol{b}^t\boldsymbol{x} \mid \boldsymbol{H}\boldsymbol{x} = \boldsymbol{s}, \boldsymbol{0} \preceq \boldsymbol{x} \preceq \boldsymbol{1}, \boldsymbol{x} \in \mathbb{R}^n\}, \tag{7}$$

where the ordering $\preceq$ is defined by $\boldsymbol{x} \preceq \boldsymbol{y}$ if and only if $x_i \leq y_i$ for all $0 \leq i \leq n-1$.

Algorithm 1 ILP solver for $\mathbb{N}$-SDP

Input: $\boldsymbol{H}, \boldsymbol{s}, t$
Output: $\boldsymbol{x}$ solution to $\mathbb{N}$-SDP or ERROR

```
1: Set b = (1, ..., 1)^t
2: Solve equation (7)                          ▷ Using interior point method
3: round the solution x* to x* ∈ {0,1}^n        ▷ using [35]
4: if Hx* = s and wt(x) ≤ t then
5:     return x*
6: else
7:     return ERROR
8: end if
```

3.4 Optimization

In this paragraph we propose an optimisation to Algorithm 1. Let us first define the following sets :

Definition 4. *Let* $0 < \ell < n-k$ *and* $\emptyset \subset I_0 \subset \cdots \subset I_\ell \subseteq \{0, \ldots, n-k-1\}$. *For* $0 \leq i \leq \ell$ *we define* $\mathcal{H}_{I_j} = \{\boldsymbol{x} \in \{0,1\}^n \mid \boldsymbol{H}_{I_j,}\boldsymbol{x} = \boldsymbol{s}_{I_j}\}$, *and* $\mathcal{H} = \{\boldsymbol{x} \in \{0,1\}^n \mid \boldsymbol{H}\boldsymbol{x} = \boldsymbol{s}\}$.

Now, let us prove how to reduce the number of constraints to our initial problem. Firstly, notice that $\mathbb{N}$-SDP can be written as $\min\{\boldsymbol{x}\boldsymbol{b}^t \mid \boldsymbol{x} \in \mathcal{H}\}$. Secondly, we prove that:

Proposition 1. *Let* $0 < \ell < n-k$ *and* $\emptyset \subset I_0 \subset \cdots \subset I_\ell \subseteq \{0, \ldots, n-k-1\}$ *and* $\boldsymbol{x}^*_{I_j} = \min\{\boldsymbol{x}\boldsymbol{b}^t \mid \boldsymbol{x} \in \mathcal{H}_{I_j}\}$, *for any* $0 \leq j \leq \ell$. *Then* $\mathsf{wt}(\boldsymbol{x}^*) \geq \mathsf{wt}(\boldsymbol{x}^*_{I_\ell}) \geq \cdots \geq \mathsf{wt}(\boldsymbol{x}^*_{I_0})$.

Proof. From definition 4 we deduce

$$\mathcal{H} \subseteq \mathcal{H}_{I_l} \cdots \subseteq \mathcal{H}_{I_0}. \tag{8}$$

Now, as the sets $\mathcal{H}_{I_j}$ are finite we can take the minimum and use the inclusion from (8) to deduce the result.

We will thus use Proposition 1 as a reduction of our initial problem to a shorter one, in terms of constrains, or equivalently in the dimension of the system. The resulting algorithm randomly chooses a set of initial row indexed for which it calls the ILP solver for $\mathbb{N}$-SDP (Algorithm 1). If the output is an optimum solution for the full problem then it stops, if not it adds a random row and continues until it finds the solution. This procedure allows us to solve an easier instance and reduce the overall time complexity of our algorithm. As we shall see in Section 5.2, the reduction decreases the experimentally observed time complexity of the $\mathbb{N}$-SDP from $\mathcal{O}(n^3)$ to $\mathcal{O}(n^2)$. In addition, the proposed optimisation allows us to perform practical attacks on all the parameters from Table 2 on a regular desktop computer.

4 Fault injection

As shown in the previous section, computing the syndrome in $\mathbb{N}$ instead of $\mathbb{F}_2$ makes the SDP considerably easier to solve. In order to perform this change, we must have the processor perform the additions in $\mathbb{N}$ instead of $\mathbb{F}_2$ during the syndrome computation. This is done by replacing the exclusive-OR instruction with an addition instruction. Since both these arithmetic instructions are performed by the arithmetic–logic unit of the processor, their associated opcodes are close, in the sense that the Hamming distance between them is small. Therefore, only few bits must be modified to switch from one to the other.

We focus on the Thumb instruction set here since it is widely used in embedded systems. The fact that, in the Thumb instruction set, the exclusive-OR instruction can be transformed into an add-with-carry instruction by a single bit-set can be considered pure luck. This is at least partially true but this is not so as surprising as it seems when given a second thought. Indeed, both these instructions are "data processing" instructions. As such, they are handled by the arithmetic and logic unit. Therefore, the opcode bits are used to generate similar control signals, and it is not surprising that they differ by only a few bits. A few examples of corruptions in other instruction sets are given in Appendix A, showing that this attack could be easily ported to other targets.

4.1 Previous work

The single-bit fault model is a very powerful one and allows an attacker to mount efficient attacks [19]. However, performing a single-bit fault in practice is far from trivial. While these can be performed by global fault injection techniques, such as under-powering [5], further analysis is necessary to filter the exploitable faults.

Indeed, while performing a single-bit fault at an unpicked position is feasible, targeting one bit specifically is much more complicated.

To this end, a more precise fault injection technique is required. In this regard, laser fault injection is a well-suited method. Indeed, as shown in [13], it is possible to perform a single-bit bit-set fault on data which is fetched from the Flash memory. This makes it possible to alter the instruction when it is fetched, before it is executed by the processor. We insist here on the fact that, as detailed in [13], the corruption is temporary, and only performed on the fetched instruction. The content of the Flash memory is left untouched. Therefore, if the instruction is fetched again from the Flash memory while no laser fault injection is performed then the instruction is executed normally.

Colombier *et al.* showed that targeting a single bit in a precise manner is relatively easy, since it only requires to position the laser spot at the right location on the y-axis in the Flash memory [13], aiming at different word lines. Indeed, moving along the x-axis does not change the affected bit, since the same word line is covered by the laser spot. Therefore, targeting a single bit of the fetched instruction is possible. Moreover, they also showed that two adjacent bits can also be set by shooting with sufficient power between two word lines. This single-bit or dual-bit bit-set fault model is the one we use as a framework for the rest of the article.

4.2 Bit-set fault on an exclusive-OR instruction

Using the fault injection technique described above, we now show how to apply it to replace an exclusive-OR instruction with an add-with-carry instruction. Figure 1 shows the Thumb encoding of both instructions, as available from the ARMv7-M Architecture Reference Manual[11]. When comparing both instructions, we observe that only one single-bit bit-set fault, on the bit of index 8, is required to replace the exclusive-OR instruction with an add-with-carry instruction. This is highlighted in red in Figure 1.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Generic EORS: $R_d = R_m$ ^ R_n															
0	1	0	0	0	0	0	0	0	1	R_m			R_{dn}		
Generic ADCS: $R_d = R_m + R_n$															
0	1	0	0	0	0	0	1	0	1	R_m			R_{dn}		

Fig. 1. Thumb encoding of the exclusive-OR (`EORS`) and add-with-carry (`ADCS`) instructions. The bit which must be set by laser fault injection is highlighted in red.

[11] `https://static.docs.arm.com/ddi0403/e/DDI0403E_B_armv7m_arm.pdf`

4.3 Bit-set fault on schoolbook matrix-vector multiplication

Now that we have shown that a single-bit fault can replace an exclusive-OR instruction with an add-with-carry instruction, we will extend it to a matrix-vector multiplication, used to compute the syndrome in code-based cryptosystems. The syndrome computation is typically implemented as shown in Algorithm 2. This is how it is done in the NTL[12] library for instance, which is widely used by NIST PQC competition candidates.

Algorithm 2 Matrix-vector multiplication.

```
function MAT_VEC_MULT(matrix, error_vector)
  for r ← 0 to k − 1 do
    syndrome[r] = 0                                   ▷ Initialisation
  for r ← 0 to k − 1 do
    for c ← 0 to n − 1 do
      syndrome[r] ^= matrix[r][c] & error_vector[c]
                                      ▷ Multiplication and addition
  return syndrome
```

When performing laser fault injection in this setting, an attacker has essentially three delays to tune. According to this implementation, an exclusive-OR instruction will be executed at each run of the inner *for* loop. The delay between the execution of these instructions is constant. We refer to it as t_{inner}. The second delay of interest is between the last and the first exclusive-OR instruction of the inner *for* loop, when one iteration of the outer *for* loop is performed. This delay is constant too. We refer to it as t_{outer}. Finally, the last delay to tune is the initial delay, before the matrix-vector multiplication starts. We refer to it as $t_{initial}$. Figure 2 shows these three delays on an example execution.

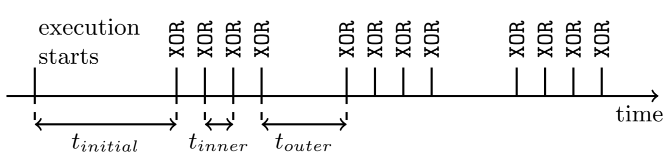


Fig. 2. Laser fault injection delays to tune

These delays are easy to tune, since they are not inter-dependent. The first delay to tune is $t_{initial}$, then t_{inner} and finally t_{outer}. Therefore, performing laser fault injection on the schoolbook matrix-vector multiplication does not induce much additional practical complexity compared with the exclusive-OR instruction alone because of the regularity of the computation.

[12] `https://www.shoup.net/ntl/`

Overall, $(n-k) \times n$ faults are necessary to obtain the full faulty syndrome in $\mathbb{N}$.

4.4 Bit-set fault on a packed matrix-vector multiplication

The matrix-vector multiplication method described in Algorithm 2 makes poor use of the capacity of the machine words when matrix entries are in $\mathbb{F}_2$. Indeed, even if both the matrix and the error-vector are binary, their elements are stored in a full machine word. Although the smallest type available, `unsigned char`, can be used, it still take eight bits to store only one bit of information.

To overcome this, consecutive bits in the rows of the parity-check matrix can be packed together in a single machine word. Typically, eight bits are packed in a byte. In this setting, the dimensions of the matrix, error-vector and syndrome are changed. The parity-check matrix now has k rows and $n/8$ columns. The error-vector now has $n/8$ entries. The syndrome now has $k/8$ entries.

Algorithm 3 Matrix-vector multiplication with packed bits.

```
 1: function MAT_VEC_MULT_PACKED(matrix, error_vector)
 2:   for r ← 0 to k − 1 do
 3:     syndrome[r/8] = 0                                ▷ Initialisation
 4:   for r ← 0 to k − 1 do
 5:     b = 0
 6:     for c ← 0 to (n − 1)/8 do
 7:       b ^= matrix[r][c] & error_vector[c]    ▷ Multiplication and addition
 8:     b ^= b >> 4;                             ▷
 9:     b ^= b >> 2;                             ▷ Exclusive-OR folding
10:     b ^= b >> 1;                             ▷
11:     b &= 1;                                          ▷ LSB extraction
12:     syndrome[r/8] |= b << (r%8)                      ▷ Bits packing
13:   return syndrome
```

Compared to the schoolbook method shown in Algorithm 2, a variable `b` is used to store the intermediate result of the multiplication and addition (see line 7 of Algorithm 3). Next, a few extra steps are performed on this variable. First, it is necessary to compute the exclusive-OR of all the bits of this variable. This is done by computing the exclusive-OR of the lower half and the upper half, by shifting by four positions (see line 8 of Algorithm 3). This is repeated again by shifting by two and finally one position (see lines 9 and 10 of Algorithm 3). We refer to this technique as exclusive-OR folding. The least-significant bit is then extracted (see line 11 of Algorithm 3). Finally, it is packed into the syndrome byte at the correct position (see line 12 of Algorithm 3).

Compared to the schoolbook matrix-vector multiplication shown in Algorithm 2, several different faults are required here. They are detailed below.

Fault on the multiplication and addition *for* loop The first specific fault to perform on the packed matrix-vector multiplication is on the inner *for* loop found on line 6 of Algorithm 3. Indeed, since the bits of the parity-check matrix are now packed, we cannot perform the sum over $\mathbb{N}$ and expect the final value to be the sum of all individual bits. This is because, when bits are stored in a word, performing the addition in $\mathbb{N}$ will incur carries which will propagate and make the final byte useless, since individual contributions of the rows of the parity check matrix are mixed.

To overcome this issue, we propose to prematurely exit this *for* loop. Before explaining how this can be achieved in practice by laser fault injection, we detail the consequences it has on the packed matrix-vector multiplication.

Consequence of a premature exit of the inner for loop of the packed matrix-vector multiplication If we are able to prematurely exit the inner for loop, then the value of the intermediate variable `b`, which holds the temporary result of the multiplication and addition, is changed. We shall identify the possible values of `b` by induction. Let us refer to the value of `b` after the i-th execution of the *for* loop as $\mathtt{b_i}$.

Let us first identify the base case, that is, exiting after only one execution. We have:

$$\mathtt{b}_0 = \mathtt{matrix}[\mathtt{r}][0] \;\&\; \mathtt{error\_vector}[0] \tag{9}$$

We can now identify the induction step, which corresponds to the subsequent executions of the for loop. We then have:

$$\mathtt{b_i} = \mathtt{b_{i-1}} \;\hat{}\; (\mathtt{matrix}[\mathtt{r}][\mathtt{i}] \;\&\; \mathtt{error\_vector}[\mathtt{i}]) \tag{10}$$

Therefore, we now have the values of `b` from $\mathtt{b}_0$ to $\mathtt{b}_{(\mathtt{n}-1)/8}$. The value $\mathtt{b}_\mathrm{i}$ is obtained by executing the *for* loop i times and prematurely exiting it only then. As mentioned in subsection 4.1, this is feasible since instructions are corrupted "on the fly", only when they are fetched from the Flash memory.

In order to obtain the faulty syndrome entry, that is, the sum over $\mathbb{N}$, we must compute the sum given in Equation (11). We use the Hamming weight (wt) to obtain the sum of the individual bits.

$$\mathtt{wt}(\mathtt{b}_0) + \sum_{i=1}^{(n-1)/8} \mathtt{wt}(\mathtt{b_i} \;\hat{}\; \mathtt{b_{i-1}}) \tag{11}$$

We then obtain a faulty syndrome entry just like the one we got after performing fault injection on the schoolbook matrix-vector multiplication. The next paragraph describes how to perform it practically by laser fault injection.

Premature exit of a for loop by laser fault injection As discussed in [13], prematurely exiting a *for* loop is feasible by corrupting the loop variable increment. Instead of incrementing the loop variable by only 1, we can try to make this increment as large as possible. As shown in Figure 3, the increment of the loop variable at the end of the *for* loop is performed by a 16-bit `ADD` instruction. It

```
mov r1, #0
inner:
...
...
...
add r1, #1
cmp r1, #N/8
ble @inner
```

(a) Typical assembly code of a *for* loop.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Generic ADD: $\mathtt{R_{dn}}$ += imm8															
0	0	1	1	0	$\mathtt{R_{dn}}$			imm8							
ADD r1 #1															
0	0	1	1	0	0	0	1	0	0	0	0	0	0	0	1
ADD r1 #193															
0	0	1	1	0	0	0	1	1	1	0	0	0	0	0	1

(b) Thumb encoding of the `ADD` instruction and two examples with different immediate values. The bits which must be set by laser fault injection are highlighted in red.

Fig. 3. Assembly code of a *for* loop and a way to exit it prematurely by corrupting the loop variable increment.

has been demonstrated in [13] that it is possible to perform a bit-set fault on two adjacent bits of the instruction. Here, we can thus make the increment step as large as 193 by setting the bits of index 6 and 7 of the `ADD` instruction.

As shown in Algorithm 3, the body of the inner *for* loop normally executes $n/8$ times. By performing the previously described fault, we can make the loop variable increment step as large as 193. Therefore, the loop is executed $\left\lceil \frac{n}{8\times 193} \right\rceil = \left\lceil \frac{n}{1544} \right\rceil$ times. Our objective is to exit the *for* loop prematurely. In this regard, for large values of n, executing the loop $\left\lceil \frac{n}{1544} \right\rceil$ times can lead to execute the *for* loop for a few more iterations.

For instance, if $n = 3488$, then the loop should be executed $\frac{n}{8} = 436$ times. If we want to exit after 5 iterations to obtain $\mathtt{b}_5$, then we will in fact obtain:

$$\begin{aligned} \mathtt{b}_5 = \mathtt{b}_4 \ \hat{}\ & \mathtt{matrix[r][5]\ \&\ error\_vector[5]}\ \hat{} \\ & \mathtt{matrix[r][198]\ \&\ error\_vector[198]}\ \hat{} \\ & \mathtt{matrix[r][391]\ \&\ error\_vector[391]} \end{aligned} \tag{12}$$

instead of:

$$\mathtt{b}_5 = \mathtt{b}_4 \ \hat{}\ \mathtt{matrix[r][5]\ \&\ error\_vector[5]} \tag{13}$$

since $391 \equiv 198 \equiv 5 \mod 193$.

Therefore, we have a few parasitic extra elements in the $\mathtt{b_i}$ value. However, since the error-vector has low weight, we can expect the associated bytes, `error_vector[198]` and `error_vector[391]` in Equation 12, to be all zeros and therefore not change the $\mathtt{b_i}$ value.

Another approach would be to obtain multiple values for every $\mathtt{b_i}$, by exploring several increment steps. The correct one could then potentially be extracted as the common pattern of all these values. This will not be investigated further in this article but could be the subject of future research.

Fault on the exclusive-OR folding Now that we obtained a temporary faulty syndrome entry stored in the intermediate variable `b`, we must deal with the exclusive-OR folding (see lines 8 to 10 of Algorithm 3) in order to keep this value intact.

There are two ways to address the exclusive-OR folding. The first possibility is to corrupt the destination register in the instruction. Depending on the level of optimisation used for the compilation, the exclusive-OR folding can be either decomposed into three consecutive shift-exclusive-OR pairs or be performed directly by three consecutive wide exclusive-OR operations. Indeed, as specified in the ARM reference manual, the exclusive-OR instruction can be made wide to include an optional shift of one of the operands (see ARMv7-M Reference Manual). In both cases, corrupting the destination register is easy and consists only in performing a bit-set on the $\mathtt{R_d}$ part of the instruction.

The second possibility, which is the one we consider more practical, is to notice that the sequence of three operations that make up the exclusive-OR folding constitute a permutation over $\mathbb{F}_2^8$. We verified it exhaustively for the 256 possible values. Therefore, rather than performing the destination register corruption described previously, one can simply inverse the permutation.

Fault on the least-significant bit extraction The next operation to address is the least-significant bit (LSB) extraction (see line 11 of Algorithm 3).

Again here, there are two possible faults. Similarly to what was presented before for the exclusive-OR folding, it is also possible to corrupt the destination register. This would leave the source register untouched and preserve the full value of $\mathtt{b_i}$, not only its LSB. The second option is to corrupt the "immediate" operand of the `AND` instruction that performs the masking to extract the LSB. To extract the LSB, this immediate value is `0x01`. The objective here is to set as many bits as possible to 1 in the immediate value, in order for the `AND` masking to reset as few bits as possible. Depending on the level of optimisation used for the compilation, the LSB extraction can be performed in one or two instructions. For the sake of readability, we consider only the case where two 16-bit instructions are used instead of a condensed 32-bit one. However, the idea to apply is exactly the same.

Figure 4a shows the two assembly instructions that perform the LSB extraction. First, the mask value is loaded. It is then used as a mask in the subsequent `AND` instruction. Ideally, we would like to load 255 as a mask instead, so that no bits are reset by the AND masking. However, this requires to perform a bit-set on seven adjacent bits, which is out of reach with a single-spot laser that can

```
mov r1, #1
and r1, r2
```

(a) Assembly code of the LSB extraction.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Generic MOV: R_d = imm8															
0	1	0	0	0	R_d			imm8							
MOV r1 #1															
0	1	0	0	0	0	0	1	0	0	0	0	0	0	0	1
MOV r1 #3															
0	1	0	0	0	0	0	1	0	0	0	0	0	0	**1**	1
MOV r1 #13															
0	1	0	0	0	0	0	1	0	0	0	0	**1**	**1**	0	1
MOV r1 #49															
0	1	0	0	0	0	0	1	0	0	**1**	**1**	0	0	0	1
MOV r1 #193															
0	1	0	0	0	0	0	1	**1**	**1**	0	0	0	0	0	1

(b) Thumb encoding of the `MOV` instruction and the set of four corruptions required to get the full byte value. The bits which must be set by laser fault injection are highlighted in red.

Fig. 4. Assembly code of the LSB extraction and the four necessary corruptions required to prevent it and obtain the full byte value.

at most fault two adjacent bits [13]. Therefore, four intermediate faults are necessary. For each of them, two bits of the mask are set, as shown in Figure 4b, giving the following mask values: `0x03`, `0x0D`, `0x31` and `0xC1`. We refer to the four consecutive faulty byte values as $\texttt{b}_{\#3}$, $\texttt{b}_{\#13}$, $\texttt{b}_{\#49}$ and $\texttt{b}_{\#193}$. Then the correct `b` value, without LSB extraction, is given in Equation (14).

$$\texttt{b} = \texttt{b}_{\#3} \mid \texttt{b}_{\#13} \mid \texttt{b}_{\#49} \mid \texttt{b}_{\#193} \tag{14}$$

Fault on the bits packing operation The previous sections showed how it is possible to keep the `b` value intact. Finally, the last operation to address is the bits packing operation (see line 12 of Algorithm 3). There are two issues to address here. First, we must deal with the left shift that will cause the most significant bits of `b` to be dropped. Second, we must address the eight successive OR operations performed for each syndrome entry.

We will actually start without dealing with the shift. The objective here is to have the `b` stored in the syndrome vector directly, to make them available to the attacker. To this end, we will apply again the idea of modifying the loop increment (as shown in Figure 3 but this time for the outer *for* loop. The pattern to observe is the following. If we increase the loop increment after the first execution of the outer *for* loop, then we have: $\texttt{s}[0] = \texttt{b}$, with `b` not being shifted. All other syndrome entries are altered and unusable. If we increase the loop increment after the ninth execution of the outer *for* loop, then we have: $\texttt{s}[1] = \texttt{b}$, with `b` not being shifted. Again, all other syndrome entries are altered and unusable. We then repeat this process and exit the outer *for* loop after the i-th execution, $i \in \{8m+1 \mid m \in \mathbb{N},\ m < k/8\}$.

This fault leaves us with a syndrome vector which entries contain every eighth faulty syndrome value, those for which the row index r verifies $r \equiv 0 \bmod 8$. Therefore, we only have 12.5 % of the faulty syndrome entries to feed to the linear programming solver. While we show in Section 5.2 that this is actually more than necessary to successfully recover the error-vector by integer linear programming for large values of n, we briefly examine some possibilities to obtain a higher percentage.

The issue here is with the left shift operation, which discards the most significant bits of the byte `b`. This shift is implemented with the `LSL` instruction. As it turns out, performing a one-bit bit-set at different positions of this instruction leads quite a few corrupted instructions. They are listed in Figure 5. The most interesting corruption is probably to turn the `LSL` instruction into a `CMP` instruction, which compares the values stored in the registers and updates the processor flags but does not modify the content of the registers. Therefore, this is the corruption that we pick. Alternatively, other corruptions such as `LSR` (logical shift right) or `SBC` (subtract with carry) could also be exploited, but would require more analysis.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Generic LSL: R_{dn} <<= R_m															
0	1	0	0	0	0	0	0	1	0	R_m			R_{dn}		
Generic LSR: R_{dn} >>= R_m															
0	1	0	0	0	0	0	0	1	1	R_m			R_{dn}		
Generic SBC: R_{dn} -= R_m															
0	1	0	0	0	0	0	1	1	0	R_m			R_{dn}		
Generic CMP: Compare(R_m, R_n)															
0	1	0	0	0	0	1	0	1	0	R_m			R_n		

Fig. 5. Possible corruptions of the `LSL` instruction with a one-bit bit-set fault

At last, the final operation to deal with is the OR operation which packs the bits together without affecting the ones which have already been packed. This must be addressed by premature exit of the outer *for* loop again.

After the row of index $r \equiv 0 \bmod 8$ has been processed, the syndrome entry holds the correct value, as mentioned before, making 12.5 % of the faulty entries readily available. However, if we run the outer *for* loop for one more iteration, the row of index $r \equiv 1 \bmod 8$ is processed. The syndrome entry value is then: $\texttt{b}_{\texttt{r}\equiv 0} \mid (\texttt{b}_{\texttt{r}\equiv 1} << 1)$. If the value $\texttt{b}_{\texttt{r}\equiv 0}$ had many zeroes and the most significant bit of $\texttt{b}_{\texttt{r}\equiv 1}$ is not 1, then the value of $\texttt{b}_{\texttt{r}\equiv 1}$ can be deduced. However, this might not be correct. A trial-and-error process could then be followed, trying to include those new faulty syndrome entries into the problem fed to the solver. Again, in many realistic cases this is not necessary, since 12.5 % of faulty syndrome entries are enough to mount the attack.

Summary and feasibility of faulting the packed matrix-vector multiplication Figure 6 summarises the steps performed in the packed matrix-vector multiplication and the associated faults required to compute the multiplication in $\mathbb{N}$ instead of $\mathbb{F}_2$. Essentially, a lot of required faults involve prematurely exiting the inner and outer *for* loops.

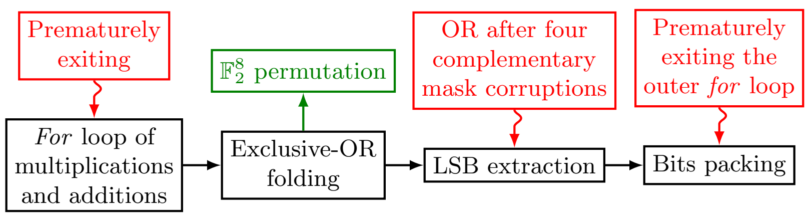


Fig. 6. Summary of the operations found in the packed matrix-vector multiplication and the required associated faults.

For practical reasons, it is worth noting which bits of the instructions must be set. Indeed, this determines the position of the laser spot in the Flash memory. The timing of the laser fault injection can be tuned very precisely, allowing to selectively target one instruction only. However, given the linear speed at which a typical XYZ stage holding the objective lens travels, it is foolish to try to fault consecutive instructions at different bit positions. Premature exit of a *for* loop requires to set the bits of index 7 and 6. Corrupting the `MOV` instruction to avoid LSB extraction, as depicted in Figure 4b, requires to set the bits 7 and 6, then 5 and 4, then 3 and 2, and finally 1. This is thus not feasible with a single-spot laser injection station, but would be possible with a multi-spot station.

5 Experimental results

5.1 Fault injection

We did perform the fault described above by laser fault injection. This allowed us to replace the exclusive-OR instruction by an add-with-carry instruction. We use a laser that emits light in the infrared region, at a wavelength of 1064 nm and perform backside injection on the target. We reused the laser fault injection parameters provided in [13]. The injection power is 1 W. The laser spot has a diameter of 5 µm. The duration of the laser pulse is 135 ns. This is roughly equal to the clock period of the microcontroller, which runs at 7.4 MHz. Laser synchronisation is becoming more precise and circuit with faster clocks are definitely within reach. Then, the fault is achieved by placing the laser spot in the Flash memory of the 32-bit microcontroller. This device embeds an ARM Cortex-M3 core, which is a very common base found in many embedded systems. We validated that the fault injection was indeed correctly performed by comparing input/output pairs with and without fault injection. This confirmed that the exclusive-OR instruction can indeed be replaced with an add-with-carry instruction.

Figure 7 shows a detailed example of instruction corruption by laser fault injection. The example code simply loads an identical constant value into two registers and performs the exclusive-OR of them. The value is then read out from the destination register.

Fault	Assembly code	Binary machine code	Readout

No	`mov r3, #90`
	`mov r4, #90`
	`eors r3, r4`
		`0010 0011 0101 1010`
		`0010 0100 0101 1010`
		`0010 0000 0110 0011`
			`r3 = 0x00`

Yes	`mov r3, #90`
	`mov r4, #90`
	`adcs r3, r4`
		`0010 0011 0101 1010`
		`0010 0100 0101 1010`
		`0010 0001 0110 0011`
			`r3 = 0xB4`

Fig. 7. Detailed example of instruction corruption by laser fault injection. The effects of the fault are highlighted in red

On the first line, no fault injection is performed. Since the value loaded into the registers is the same, the exclusive-OR leads to a byte where all bits are zero, as shown in the readout value of the destination register.

On the second line, a fault is injected. This allows to perform a bit-set on the bit of index 8, as shown in red in the "Binary machine code" column. This in turns changes the exclusive-OR operation into an add-with-carry operation. This is visible in the "Assembly code" column, where the `eors` instruction is replaced with an `adcs` instruction. As a consequence, the value stored in the destination register is different from zero and equal to the sum of both registers instead. Since we observe precisely this value in our experimentation, this validates that the instruction has been successfully corrupted.

Following the fault injection strategies detailed in Section 4, we are able to obtain a syndrome with values in $\mathbb{N}$. The following section describes the actual exploitation of this syndrome to recover the binary error-vector.

5.2 Syndrome decoding over $\mathbb{N}$ with integer linear programming

After obtaining a faulty syndrome with entries in $\mathbb{N}$, we feed it and the parity-check matrix to the programming solver. We used the `linprog` function of the `scipy.optimize` [50] Python module. It implements the interior point method as described in [2]. As mentioned in Section 3 we chose the interior point method over the simplex, for several already known arguments. We still performed a comparison between these two methods for our specific problem, and indeed, the interior point method turned out to be much faster.

In order to remain as general as possible, we consider parity-check matrices of random binary codes. Since no efficient decoding algorithm exists for these, they can be considered the worst-case scenario. Also, all the code-based proposals to the NIST challenge state that the public codes are indistinguishable from random codes. Parity-check matrices which are associated with structured codes thus cannot be harder to handle than the ones of random binary codes. We explore values of n from 10^3 to 10^5 as they are representative of modern parameters for code-based cryptosystems, as shown in Table 2. The $\mathbb{N}$-SDP instance that we have to solve for the code-based candidates have the following properties.

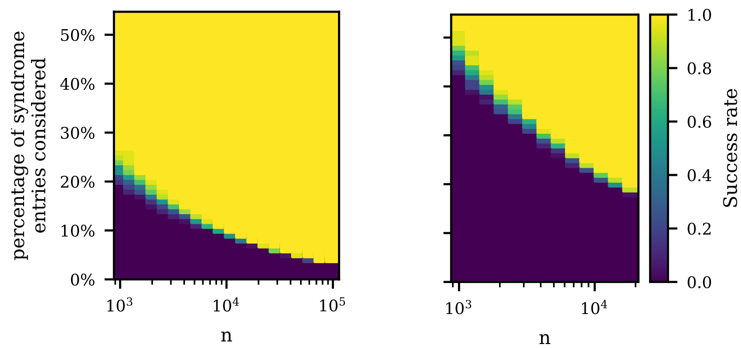
- The parameter t ranges in $t = \mathcal{O}(\sqrt{n})$ or $t = \mathcal{O}(\sqrt{n \log(n)})$ for all the schemes.
- For McEliece, NTS-KEM, BIKE, and LEDAcrypt, the SDP instance that we have to solve comes directly from the parameters specifications, as they follow from the McEliece/Niederreiter scheme.
- For HQC, the SDP instance that we solve is constructed as in [1]. Hence, the parity-check matrix has length $2n$ and dimension n, where n is given in Table 2.

All experiments are conducted on a standard desktop computer, embedding a 6-core CPU clocked at 2.8 GHz and 32 GB of RAM.

Required percentage of faulty syndrome entries As highlighted in Section 3.4, only a fraction of the parity-check matrix rows and syndrome entries are required to solve the linear programming problem. Figure 8 shows how the percentage of required syndrome entries changes for different values of n. This depends not only on n but also on the weight of the error-vector. Figure 8a shows the required percentage of syndrome entries for $t = \sqrt{n}$. Figure 8b shows the required percentage of syndrome entries for $t = \sqrt{n \log(n)}$. For each value of n and every percentage from 1 to 54, we estimate the success rate by solving the linear programming problem 20 times.

We can clearly see that the number of required syndrome entries decreases when n increases. It drops below the 12.5 % threshold derived in Section 4.4 after $n = 9000$ approximately. This makes most of the cryptosystems listed in Table 2 vulnerable to the attack without dealing with the bits packing if the packed matrix-vector multiplication is used. For large values of n and $t = \sqrt{n}$, the required number of syndrome entries even drops below 5 %.

For $t = \sqrt{n \log(n)}$, as shown in Figure 8b, the required percentage of syndrome entries does not drop as fast. Moreover, this leads to an issue related to large values of t. For example, $n = 10000$ leads to $t = \sqrt{n \log(n)} = 303$. This is already higher than any t found in Table 2. At this number of errors, since n is not so large, the linear programming problem to solve is better satisfied by non-binary vectors. Therefore, it is necessary to add bounds on the variables of the linear programming problem to make sure that they remain in the $[0, 1]$ interval. This dramatically increases the memory requirements of the solver, thereby



(a) $t = \sqrt{n}$ (b) $t = \sqrt{n \log(n)}$

Fig. 8. Success rate of solving the liner programming problem for different values of n and percentage of syndrome entries considered.

limiting the largest value of n to 2×10^4 approximately. Note that this is only dictated by the RAM available on the desktop computer we used and is not an algorithmic limit.

To conclude this part, we stress that only a small percentage of rows suffices to solve the full problem. Also, for $t < \sqrt{n \log(n)}$, the ILP solver finds the binary solution directly, which makes it really efficient. However, for larger values of t, we need to bound the solution to the $[0, 1]$ interval in order to be able to practically solve the ILP.

Execution time Figure 9 shows how the execution time of the linear programming solver changes for different values of n. Two cases are displayed. In the "Full" case, the whole faulty syndrome is fed to the solver. In the "Optimal" case, only the required percentage of syndrome entries are used.

We can observe that taking only the required percentage of syndrome entries drastically reduces the computation time. For $n = 18000$, and $t = \sqrt{n}$, two orders of magnitude of computation time are gained. Above $n = 18000$, the memory available on our desktop computer is not sufficient to handle the $t = \sqrt{n \log(n)}$ case. However, in the case where $t = \sqrt{n}$ and only the required percentage of syndrome entries are considered, then the memory is not limiting, even for values of n as large as 10^5. The computation time remains low too, since it takes only two minutes so solve the problem for $n = 10^5$.

We can empirically observe on Figure 9 that the slope is different for the "Full" and the "Optimal" cases, having respectively $\mathcal{O}(n^3)$ and $\mathcal{O}(n^2)$ time complexities. This difference of exponent could be investigated further in future works.

When considering the parameters of the NIST PQC competition candidates, given in Table 2, we see that they are close to the $t = \sqrt{n}$ case. For the largest parameters, $n = 122\,422$, $k = 61\,211$, $t = 261$, only $4\,\%$ of the syndrome entries

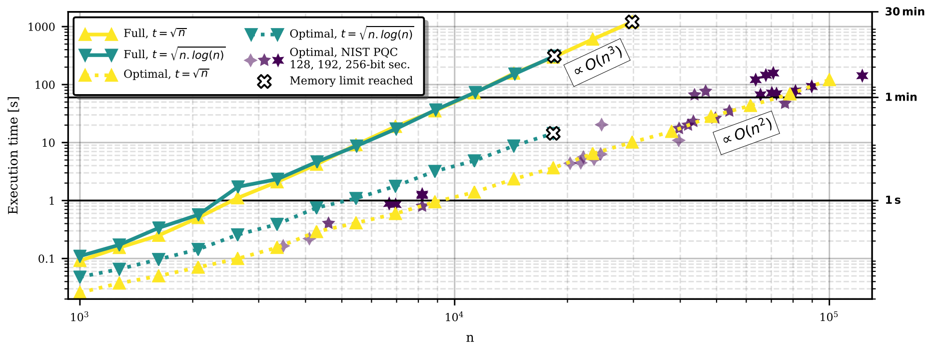


Fig. 9. Execution time of the linear programming solver for different values of n. In the "Full" case, all syndrome entries are considered. In the "Optimal" case, only the required percentage of syndrome entries are considered.

are required to solve the problem, and this is done in 142 s approximately. We did not extend our experiments beyond this limit but this is definitely feasible, even on our desktop computer, without hitting the memory limit. Overall, all NIST PQC competition candidates can be attacked within a few minutes with a desktop computer.

6 Conclusion

We have shown in this paper that, using laser fault injection, we are able to modify one of the building blocks of code-based cryptosystems, *i.e.*, the well-known syndrome decoding problem. We modelled the modified instance by means of an integer linear programming problem, and further solve it experimentally in polynomial time. We have provided real time attacks against all the parameters of the code-based solution in the NIST post-quantum challenge.

Furthermore, we have shown that the number of fault injected can be drastically reduced if we focus on only a few percent of the number of rows of the matrices involved. Combining laser fault injection to obtain an easier problem such as the syndrome decoding problem over $\mathbb{N}$ instead of $\mathbb{F}_2$ and then using linear programming to solve this problem is an interesting combination that potentially could be applied to other interesting problems, such as Shortest Integer Problem or Shortest Vector Problem.

Our results open the road to other research directions. Take for example the maximum-likelihood decoding problem. If one is able to modify the multiplication from $\mathbb{F}_2$ to integer multiplication, the modified problem is almost identical to what we propose here, as we point out in the beginning of our article. Another direction which our article opens is a more theoretical one. Our simulations provide evidence of the polynomial time complexity of this problem. However, known results on this issue, *e.g.*, totally unimodular binary matrices, fixed variable problems, bounded determinant binary matrices (solvable in polynomial time), do not apply to our case. The question is whether the worst case ℕ-SPD is still solvable in polynomial time.

References

1. Aguilar, C., Blazy, O., Deneuville, J.C., Gaborit, P., Zémor, G.: Efficient encryption from random quasi-cyclic codes. arXiv preprint arXiv:1612.05572 (2016)
2. Andersen, E.D., Andersen, K.D.: The mosek interior point optimizer for linear programming: an implementation of the homogeneous algorithm. In: High performance optimization, pp. 197–232. Springer (2000)
3. Aragon, N., Barreto, P., Bettaieb, S., Bidoux, L., Blazy, O., Deneuville, J.C., Gaborit, P., Gueron, S., Guneysu, T., Melchor, C.A., et al.: Bike: bit flipping key encapsulation (2017)
4. Baldi, M., Barenghi, A., Chiaraluce, F., Pelosi, G., Santini, P.: Ledakem: A post-quantum key encapsulation mechanism based on qc-ldpc codes. In: Lange, T., Steinwandt, R. (eds.) Post-Quantum Cryptography. pp. 3–24. Springer International Publishing, Cham (2018)
5. Barenghi, A., Bertoni, G.M., Breveglieri, L., Pellicioli, M., Pelosi, G.: Fault attack on AES with single-bit induced faults. In: Sixth International Conference on Information Assurance and Security. pp. 167–172. IEEE, Atlanta, GA, USA (8 2010). https://doi.org/10.1109/ISIAS.2010.5604061
6. Becker, A., Joux, A., May, A., Meurer, A.: Decoding random binary linear codes in $2^{n/20}$: How $1+1=0$ improves information set decoding. In: Advances in Cryptology - EUROCRYPT 2012. Lecture Notes in Comput. Sci., Springer (2012)
7. Berlekamp, E., McEliece, R., van Tilborg, H.: On the inherent intractability of certain coding problems. IEEE Trans. Inform. Theory **24**(3), 384–386 (May 1978)
8. Bernstein, D.J., Buchmann, J., Dahmen, E. (eds.): Post-Quantum Cryptography. Springer-Verlag (2009)
9. Bertsimas, D., Tsitsiklis, J.N.: Introduction to linear optimization, vol. 6. Athena Scientific Belmont, MA (1997)
10. Borghoff, J.: Mixed-integer linear programming in the analysis of trivium and ktantan. IACR Cryptology ePrint Archive **2012**, 676 (2012)
11. Borghoff, J., Knudsen, L.R., Stolpe, M.: Bivium as a mixed-integer linear programming problem. In: Cryptography and Coding. pp. 133–152. Springer Berlin Heidelberg (2009)
12. Bukasa, S.K., Lashermes, R., Lanet, J., Legay, A.: Let's shock our iot's heart: Armv7-m under (fault) attacks. In: Doerr, S., Fischer, M., Schrittwieser, S., Herrmann, D. (eds.) International Conference on Availability, Reliability and Security. pp. 33:1–33:6. ACM, Hamburg, Germany (8 2018). https://doi.org/10.1145/3230833.3230842
13. Colombier, B., Menu, A., Dutertre, J.M., Moëllic, P.A., Rigaud, J.B., Danger, J.L.: Laser-induced single-bit faults in flash memory: Instructions corruption on a 32-bit microcontroller. In: IEEE International Symposium on Hardware Oriented Security and Trust. pp. 1–10. McLean, VA, USA (5 2019)
14. Dantzig, G.B.: Maximization of a linear function of variables subject to linear inequalities. Activity analysis of production and allocation **13**, 339–347 (1951)
15. Diffie, W., Hellman, M.: New directions in cryptography. IEEE Trans. Inform. Theory **22**(6), 644–654 (Nov 1976)

16. Dutertre, J., Riom, T., Potin, O., Rigaud, J.: Experimental analysis of the laser-induced instruction skip fault model. In: Askarov, A., Hansen, R.R., Rafnsson, W. (eds.) Nordic Conference on Secure IT Systems. Lecture Notes in Computer Science, vol. 11875, pp. 221–237. Springer, Aalborg, Denmark (11 2019). https://doi.org/10.1007/978-3-030-35055-0_14
17. Feldman, J., Wainwright, M.J., Karger, D.R.: Using linear programming to decode binary linear codes. IEEE Transactions on Information Theory **51**(3), 954–972 (2005)
18. Feldman, J.: Decoding error-correcting codes via linear programming. Ph.D. thesis, Massachusetts Institute of Technology (2003)
19. Giraud, C.: DFA on AES. In: Dobbertin, H., Rijmen, V., Sowa, A. (eds.) 4th International Advanced Encryption Standard Conference. Lecture Notes in Computer Science, vol. 3373, pp. 27–41. Springer, Bonn, Germany (5 2004). https://doi.org/10.1007/11506447_4
20. Hamming, R.W.: Error detecting and error correcting codes. Bell System Tech. J. **29**, 147–160 (1950)
21. Helmling, M., Ruzika, S., Tanatmis, A.: Mathematical programming decoding of binary linear codes: Theory and algorithms. IEEE transactions on information theory **58**(7), 4753–4769 (2012)
22. Johnson, E.L., Nemhauser, G.L., Savelsbergh, M.W.: Progress in linear programming-based algorithms for integer programming: An exposition. Informs journal on computing **12**(1), 2–23 (2000)
23. Karmarkar, N.: A new polynomial-time algorithm for linear programming. In: Proceedings of the sixteenth annual ACM symposium on Theory of computing. pp. 302–311 (1984)
24. Klee, V., Minty, G.J.: How good is the simplex algorithm. Inequalities **3**(3), 159–175 (1972)
25. Koblitz, N.: Elliptic curve cryptosystems. Mathematics of computation **48**(177), 203–209 (1987)
26. Lee, P.J., Brickell, E.F.: An observation on the security of McEliece's public-key cryptosystem. In: Advances in Cryptology - EUROCRYPT'88. Lecture Notes in Comput. Sci., vol. 330, pp. 275–280. Springer (1988)
27. Lee, Y.T., Sidford, A.: Efficient inverse maintenance and faster algorithms for linear programming. In: 2015 IEEE 56th Annual Symposium on Foundations of Computer Science. pp. 230–249. IEEE (2015)
28. Leon, J.: A probabilistic algorithm for computing minimum weights of large error-correcting codes. IEEE Trans. Inform. Theory **34**(5), 1354–1359 (1988)
29. Liao, H., Gebotys, C.H.: Methodology for EM fault injection: Charge-based fault model. In: Teich, J., Fummi, F. (eds.) Design, Automation & Test in Europe Conference. pp. 256–259. IEEE, Florence, Italy (3 2019). https://doi.org/10.23919/DATE.2019.8715150
30. Luppold, A., Oehlert, D., Falk, H.: Evaluating the performance of solvers for integer-linear programming. Tech. rep. (2018)
31. MacWilliams, F.J., Sloane, N.J.A.: The Theory of Error-Correcting Codes. North–Holland, Amsterdam, fifth edn. (1986)
32. May, A., Meurer, A., Thomae, E.: Decoding random linear codes in $O(2^{0.054n})$. In: Lee, D.H., Wang, X. (eds.) Advances in Cryptology - ASIACRYPT 2011. Lecture Notes in Comput. Sci., vol. 7073, pp. 107–124. Springer (2011)
33. May, A., Ozerov, I.: On computing nearest neighbors with applications to decoding of binary linear codes. In: Oswald, E., Fischlin, M. (eds.) Advances in Cryptology

- EUROCRYPT 2015. Lecture Notes in Comput. Sci., vol. 9056, pp. 203–228. Springer (2015)
34. McEliece, R.J.: A Public-Key System Based on Algebraic Coding Theory, pp. 114–116. Jet Propulsion Lab (1978), dSN Progress Report 44
35. Megiddo, N.: On finding primal-and dual-optimal bases. ORSA Journal on Computing **3**(1), 63–65 (1991)
36. Melchor, C.A., Aragon, N., Bettaieb, S., Bidoux, L., Blazy, O., Deneuville, J.C., Gaborit, P., Hauteville, A., Ruatta, O., Tillich, J.P., et al.: ROLLO-Rank-Ouroboros, LAKE & LOCKER (2018)
37. Miller, V.S.: Use of elliptic curves in cryptography. In: Conference on the Theory and Application of Cryptographic Techniques. pp. 417–426. Springer (1985)
38. Moro, N., Dehbaoui, A., Heydemann, K., Robisson, B., Encrenaz, E.: Electromagnetic fault injection: Towards a fault model on a 32-bit microcontroller. In: Fischer, W., Schmidt, J. (eds.) Workshop on Fault Diagnosis and Tolerance in Cryptography. pp. 77–88. IEEE Computer Society, Los Alamitos, CA, USA (8 2013). https://doi.org/10.1109/FDTC.2013.9
39. Mouha, N., Wang, Q., Gu, D., Preneel, B.: Differential and linear cryptanalysis using mixed-integer linear programming. In: Wu, C.K., Yung, M., Lin, D. (eds.) Information Security and Cryptology. pp. 57–76. Springer Berlin Heidelberg (2012)
40. Niederreiter, H.: Knapsack-type cryptosystems and algebraic coding theory. Problems of Control and Information Theory **15**(2), 159–166 (1986)
41. Prange, E.: Cyclic error-correcting codes in two symbols. Electronics Research Directorate, Air Force Cambridge Research Center (September 1957), no. AFCRC-TN-57-103. ASTIA Document No. AD133749
42. Prange, E.: The use of information sets in decoding cyclic codes. IRE Transactions on Information Theory **8**(5), 5–9 (1962). https://doi.org/10.1109/TIT.1962.1057777, `http://dx.doi.org/10.1109/TIT.1962.1057777`
43. Rivest, R.L., Shamir, A., Adleman, L.M.: A method for obtaining digital signatures and public-key cryptosystems. Commun. ACM **21**(2), 120–126 (1978)
44. Rivière, L., Najm, Z., Rauzy, P., Danger, J., Bringer, J., Sauvage, L.: High precision fault injections on the instruction cache of armv7-m architectures. In: International Symposium on Hardware Oriented Security and Trust. pp. 62–67. IEEE Computer Society, Washington, DC, USA (5 2015). https://doi.org/10.1109/HST.2015.7140238
45. Shor, P.W.: Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. SIAM J. Comput. **26**(5), 1484–1509 (1997)
46. Stern, J.: A method for finding codewords of small weight. In: Cohen, G.D., Wolfmann, J. (eds.) Coding Theory and Applications. Lecture Notes in Comput. Sci., vol. 388, pp. 106–113. Springer (1988)
47. Taghavi, M.H., Shokrollahi, A., Siegel, P.H.: Efficient implementation of linear programming decoding. IEEE transactions on information theory **57**(9), 5960–5982 (2011)
48. Tanatmis, A., Ruzika, S., Hamacher, H.W., Punekar, M., Kienle, F., Wehn, N.: A separation algorithm for improved lp-decoding of linear block codes. IEEE Transactions on Information Theory **56**(7), 3277–3289 (2010)
49. Vaidya, P.M.: Speeding-up linear programming using fast matrix multiplication. In: 30th annual symposium on foundations of computer science. pp. 332–337. IEEE (1989)

50. Virtanen, P., Gommers, R., Oliphant, T.E., Haberland, M., Reddy, T., Cournapeau, D., Burovski, E., Peterson, P., Weckesser, W., Bright, J., van der Walt, S.J., Brett, M., Wilson, J., Jarrod Millman, K., Mayorov, N., Nelson, A.R.J., Jones, E., Kern, R., Larson, E., Carey, C., Polat, İ., Feng, Y., Moore, E.W., Vand erPlas, J., Laxalde, D., Perktold, J., Cimrman, R., Henriksen, I., Quintero, E.A., Harris, C.R., Archibald, A.M., Ribeiro, A.H., Pedregosa, F., van Mulbregt, P., Contributors, S...: SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. Nature Methods **17**, 261–272 (2020). https://doi.org/https://doi.org/10.1038/s41592-019-0686-2
51. Vontobel, P.O.: Interior-point algorithms for linear-programming decoding. In: 2008 Information Theory and Applications Workshop. pp. 433–437. IEEE (2008)
52. Wadayama, T.: An lp decoding algorithm based on primal path-following interior point method. In: 2009 IEEE International Symposium on Information Theory. pp. 389–393. IEEE (2009)

A Possible corruptions of the exclusive-OR instruction in a few other instruction sets

We provide here a few examples of possible corruptions of the exclusive-OR instruction in other instruction sets than the Thumb that we considered in the article. These instruction sets are ARMv7[13], PIC[14] and RISC-V compressed[15].

ARMv7 In the ARMv7 instruction set, the exclusive-OR instruction (`EORS.W`) can be corrupted into a saturated addition instruction (`QADD`) as shown in Figure 10. For readability reasons, we split the 32-bit instructions into two 16-bit blocks.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Generic EORS.W: R_d = R_m ^ R_n (upper half)															
1	1	1	0	1	0	1	0	1	0	0	0	R_n			
Generic QADD: R_d = R_m + R_n (upper half)															
1	1	1	1	1	0	1	0	1	0	0	0	R_n			

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Generic EORS.W: R_d = R_m ^ R_n (lower half)															
0	imm3			R_d				imm2		type		R_m			
Generic QADD: R_d = R_m + R_n (lower half)															
1	1	1	1	R_d				1	0	0	0	R_m			

Fig. 10. ARMv7 encoding of the exclusive-OR instruction and a possible fault feasible by bit-sets

[13] `https://static.docs.arm.com/ddi0403/e/DDI0403E_B_armv7m_arm.pdf`
[14] `http://ww1.microchip.com/downloads/en/devicedoc/31029a.pdf`
[15] `https://riscv.org/specifications/isa-spec-pdf/`

PIC In the PIC instruction set, the exclusive-OR instruction (`XORWF`) can be corrupted into an addition instruction (`ADDWF`) as shown in Figure 11.

13	12	11	10	9	8	7	6	5	4	3	2	1	0
Generic `XORWF: W = W ^` R_f													
0	0	0	1	1	0	d	R_f						
Generic `ADDWF: W = W +` R_f													
0	0	0	1	1	1	d	R_f						

Fig. 11. PIC encoding of the exclusive-OR instruction and a possible fault feasible by bit-set

RISC-V compressed In the RISC-V compressed instruction set, the exclusive-OR instruction (`C.XOR`) can be corrupted into an addition instruction (`C.ADDW`) as shown in Figure 12.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Generic `C.XOR:` R_d `=` R_{s1} `^` R_{s2}															
1	0	0	0	1	1	$R_{s1/d}$			0	1	R_{s2}			0	1
Generic `C.ADDW:` R_d `=` R_{s1} `+` R_{s2}															
1	0	0	1	1	1	$R_{s1/d}$			0	1	R_{s2}			0	1

Fig. 12. RISC-V encoding of the exclusive-OR instruction and a possible fault feasible by bit-set