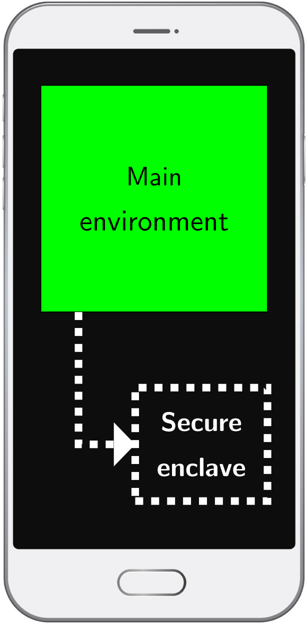
Faulting original McEliece's implementations is possible

How to mitigate this risk?

Vincent Giraud [1,2] Guillaume Bouffard [1,3]

{first name}.{last name}@ens.fr

[1]DIENS, École Normale Supérieure, Université PSL, CNRS
[2]Ingenico
[3]National Cybersecurity Agency of France (ANSSI)



The industry has developed an important need to operate sensitive and secure processes on uncontrolled peripherals.

This necessity mainly concerns COTS[1], in other words, devices we all own today.

Many of them are not equipped with security hardware mechanisms such as secure enclaves for example.

[1] Commercial Off-The-Shelf

A key challenge in this context is to exploit cryptography using a private key.

In these conditions, an implementation :

A key challenge in this context is to exploit cryptography using a private key.

In these conditions, an implementation :

- can be spied on

A key challenge in this context is to exploit cryptography using a private key.

In these conditions, an implementation :

- can be spied on
- can be statically read and modified

A key challenge in this context is to exploit cryptography using a private key.

In these conditions, an implementation :

- can be spied on
- can be statically read and modified
- can be dynamically instrumentalized

In this use case, industrial actors often use symmetrical algorithms.

But asymmetrical ones can also be interesting :

In this use case, industrial actors often use symmetrical algorithms.

But asymmetrical ones can also be interesting :

- for DRM purposes on the client's side

In this use case, industrial actors often use symmetrical algorithms.

But asymmetrical ones can also be interesting :

- for DRM purposes on the client's side
- to strengthen the security server-side

Additional requirements include consideration of :

- the new constraints brought by quantum computing

Additional requirements include consideration of :

- the new constraints brought by quantum computing
- mobile networks' constraints

Additional requirements include consideration of :

- the new constraints brought by quantum computing
- mobile networks' constraints
- COTS hardware and software limitations

The McEliece cryptosystem

The McEliece cryptosystem is a good candidate to deal with these concerns.

It relies on coding operations, permutations and scrambling.

The McEliece cryptosystem is a good candidate to deal with these concerns.

It relies on coding operations, permutations and scrambling.

It is an asymmetrical
cryptographic
algorithm

The McEliece cryptosystem is a good candidate to deal with these concerns.

It relies on coding operations, permutations and scrambling.

It is an asymmetrical cryptographic algorithm

It is considered resistant in a post-quantum context

The McEliece cryptosystem is a good candidate to deal with these concerns.

It relies on coding operations, permutations and scrambling.

It is an asymmetrical cryptographic algorithm

It is considered resistant in a post-quantum context

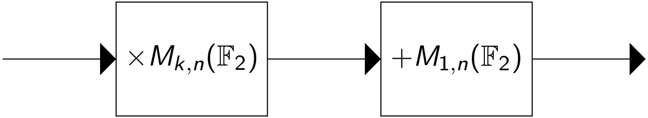
It is mainly based on linear algebra, making it efficient and easy to optimize

The McEliece cryptosystem is a good candidate to deal with these concerns.

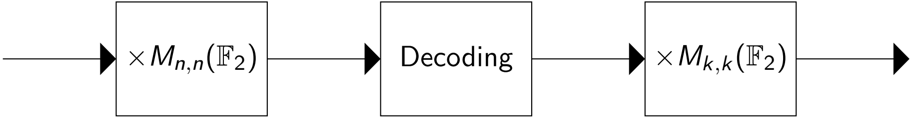
It relies on coding operations, permutations and scrambling.

It is an asymmetrical cryptographic algorithm	It is considered resistant in a post-quantum context	It is mainly based on linear algebra, making it efficient and easy to optimize	After almost half a century of existence, it is considered reliable and robust

Encryption simply consists in a matrix multiplication and a matrix addition.



Decryption is done with two matrix multiplications and a fast decoding operation.



Everything is done in $\mathbb{F}_2$.

In fact, if :

- S is a scrambling matrix (that is, a random invertible matrix)
- G is a generator matrix for a (n,k)-linear error-correcting code $\mathcal{C}$
- P is a permutation matrix

Then, with $G' = SGP$, for a clear-text block m, we have a cipher-text block c such as :

$$c = mG' + e$$

where e is a random vector of Hamming weight below $\mathcal{C}$'s error-correcting capability.

In fact, if :

- S is a scrambling matrix (that is, a random invertible matrix)
- G is a generator matrix for a (n,k)-linear error-correcting code $\mathcal{C}$
- P is a permutation matrix

Then, with $G' = SGP$, for a clear-text block m, we have a cipher-text block c such as :

$$c = mG' + e$$

where e is a random vector of Hamming weight below $\mathcal{C}$'s error-correcting capability.

To get m back, one must :

- multiply c by P^{-1}
- use a fast algorithm to decode the result
- multiply the outcome by S^{-1}

In fact, if :

- S is a scrambling matrix (that is, a random invertible matrix)
- G is a generator matrix for a (n,k)-linear error-correcting code $\mathcal{C}$
- P is a permutation matrix

Then, with $G' = SGP$, for a clear-text block m, we have a cipher-text block c such as :

$$c = mG' + e$$

where e is a random vector of Hamming weight below $\mathcal{C}$'s error-correcting capability.

To get m back, one must :

- multiply c by P^{-1}
- use a fast algorithm to decode the result
- multiply the outcome by S^{-1}

The McEliece cryptosystem's security rely on the fact that an attacker cannot use a fast algorithm to decode c because of the permutation.

The McEliece cryptosystem is quite robust against fault-injection based attacks.

Such attacks were driven to :

The McEliece cryptosystem is quite robust against fault-injection based attacks.

Such attacks were driven to :

recover the public key

The McEliece cryptosystem is quite robust against fault-injection based attacks.

Such attacks were driven to :

recover the public key

exploit weaknesses when using other error-correcting codes than the Goppa ones, which are specified in the initial paper

The McEliece cryptosystem is quite robust against fault-injection based attacks.

Such attacks were driven to :

- recover the public key
- exploit weaknesses when using other error-correcting codes than the Goppa ones, which are specified in the initial paper
- break the NIST candidate based on, but different than the original one

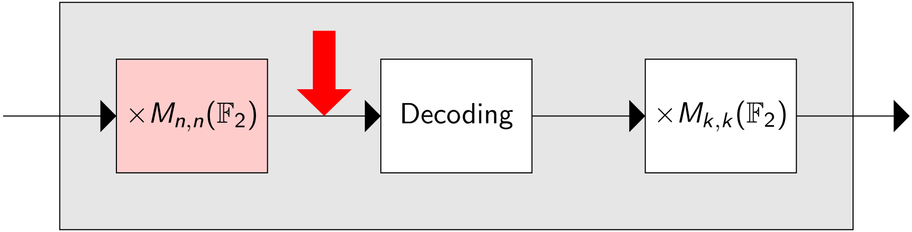
The McEliece cryptosystem is quite robust against fault-injection based attacks.
Such attacks were driven to :

recover the public key	exploit weaknesses when using other error-correcting codes than the Goppa ones, which are specified in the initial paper	break the NIST candidate based on, but different than the original one

Here, we now present an attack based on fault-injection, focusing on the original McEliece specification, aiming at the secret key.

Faulting McEliece implementations

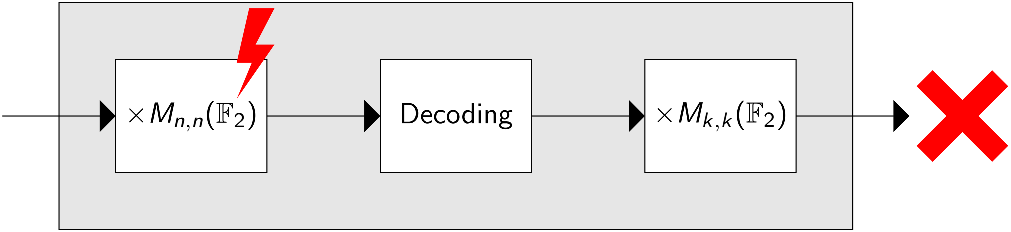
As the permutation matrix is the foundation of this cryptosystem's security, our goal is to obtain information about it or the intermediate variable right after, which is hard in a black-box context, or in case of obfuscation.



Trying to cancel the scrambling and the decoding is futile, as one brings a lot of diffusion, and the other is a surjective function.

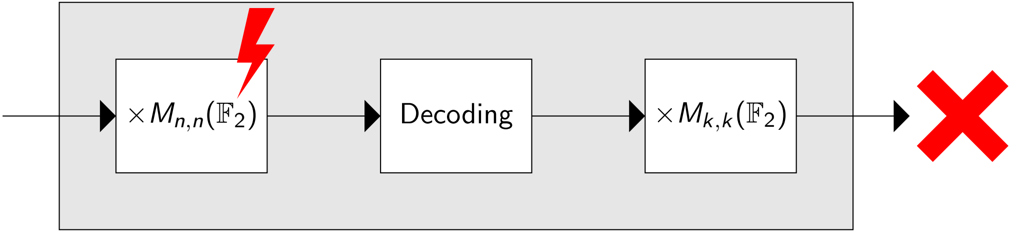
Making a fault-injection based attack work on McEliece is hard because attempts :

- either will be cancelled by the error-correcting code, in which case it will be as if nothing has happened



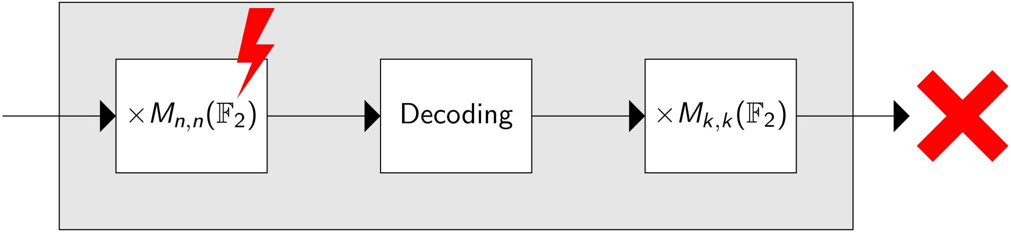
Making a fault-injection based attack work on McEliece is hard because attempts :

- either will be cancelled by the error-correcting code, in which case it will be as if nothing has happened
- or will result in a corrupted output because the data was too much impacted



Making a fault-injection based attack work on McEliece is hard because attempts :

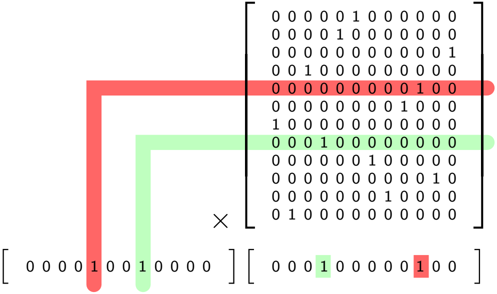
- either will be cancelled by the error-correcting code, in which case it will be as if nothing has happened
- or will result in a corrupted output because the data was too much impacted



In face of this, we propose another approach : **instead of targeting the data and the specification, we target implementations**.

```c
uint32_t accu[1024/32] = {0};
for(int i = 0; i < 1024; i++) {
  if(((vector[i/32] >> (31-(i%32))) & 0x01) != 0) {
    for(int j = 0; j < (1024/32); j++) {
      accu[j] = accu[j] ^ matrix[i*(1024/32)+j];
}}}
```

```
10660   e51b300c    ldr r3, [fp, #-12]
10664   e1a03103    lsl r3, r3, #2
10668   e24b2004    sub r2, fp, #4
1066c   e0823003    add r3, r2, r3
10670   e5131024    ldr r1, [r3, #-36]
10674   e51b2008    ldr r2, [fp, #-8]
10678   e1a03002    mov r3, r2
1067c   e1a03083    lsl r3, r3, #1
10680   e0832002    add r2, r3, r2
10684   e51b300c    ldr r3, [fp, #-12]
10688   e0822003    add r2, r2, r3
1068c   e59f30d8    ldr r3, [pc, #216]
10690   e08f3003    add r3, pc, r3
10694   e7933102    ldr r3, [r3, r2, lsl #2]
10698   e0212003    eor r2, r1, r3
1069c   e51b300c    ldr r3, [fp, #-12]
106a0   e1a03103    lsl r3, r3, #2
106a4   e24b1004    sub r1, fp, #4
106a8   e0813003    add r3, r1, r3
106ac   e5032024    str r2, [r3, #-36]
```



A typical $\mathbb{F}_2$ vector-matrix multiplication implementation loops over the vector and accumulates matrix lines with XOR if the element is set.

31 28	27 26	25	24 21	20	19 16	15 12	11 0
Cond	00	I	OpCode	S	Rn	Rd	Operand 2

Operation Code

0000 = AND - Rd:= Op1 AND Op2
0001 = EOR - Rd:= Op1 EOR Op2
0010 = SUB - Rd:= Op1 - Op2
0011 = RSB - Rd:= Op2 - Op1
0100 = ADD - Rd:= Op1 + Op2
0101 = ADC - Rd:= Op1 + Op2 + C
0110 = SBC - Rd:= Op1 - Op2 + C - 1
0111 = RSC - Rd:= Op2 - Op1 + C - 1
1000 = TST - set condition codes on Op1 AND Op2
1001 = TEQ - set condition codes on Op1 EOR Op2
1010 = CMP - set condition codes on Op1 - Op2
1011 = CMN - set condition codes on Op1 + Op2
1100 = ORR - Rd:= Op1 OR Op2
1101 = MOV - Rd:= Op2
1110 = BIC - Rd:= Op1 AND NOT Op2
1111 = MVN - Rd:= NOT Op2

If we consider a fault injection changing only one bit in the program, making an `EOR` instruction become a `RSB` gives convincing results.

For illustrative purposes,
we use `n = 12`, `t = 2`,
and a fictional processor
with 4-bit registers.

$$
\begin{bmatrix}
0 & 0 & 0 & 0 & 0 & \mathbf{1} & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & \mathbf{1} & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{1} \\
0 & 0 & \mathbf{1} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\mathbf{1} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{1} & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{1} & 0 & 0 \\
0 & 0 & 0 & \mathbf{1} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & \mathbf{1} & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{1} & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{1} & 0 & 0 & 0 & 0 \\
0 & \mathbf{1} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{bmatrix}
$$

For illustrative purposes, we use `n = 12`, `t = 2`, and a fictional processor with 4-bit registers.

$$
\begin{bmatrix}
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{bmatrix}
$$

For illustrative purposes,
we use `n = 12`, `t = 2`,
and a fictional processor
with 4-bit registers.

$$
\begin{bmatrix} 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \end{bmatrix}
\ \textbf{EOR}\
\begin{bmatrix}
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{bmatrix}
$$

For illustrative purposes, we use `n = 12`, `t = 2`, and a fictional processor with 4-bit registers.

EOR

$$
\left[\begin{array}{cccc|cccc|cccc}
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{array}\right]
$$

$$
\begin{bmatrix} 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \end{bmatrix}
\begin{bmatrix} 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}
$$

For illustrative purposes, we use `n = 12`, `t = 2`, and a fictional processor with 4-bit registers.

$$\begin{bmatrix}
0&0&0&0&0&1&0&0&0&0&0&0\\
0&0&0&0&1&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&1\\
0&0&1&0&0&0&0&0&0&0&0&0\\
1&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&1&0&0&0\\
0&0&0&0&0&0&0&0&0&1&0&0\\
0&0&0&1&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&1&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&1&0\\
0&0&0&0&0&0&0&1&0&0&0&0\\
0&1&0&0&0&0&0&0&0&0&0&0
\end{bmatrix}$$

~~EOR~~ RSB

$$\begin{bmatrix} 0&0&0&0&1&0&0&1&0&0&0&0 \end{bmatrix}$$

For illustrative purposes, we use `n = 12`, `t = 2`, and a fictional processor with 4-bit registers.

~~EOR~~ **RSB**

$$
\begin{bmatrix} 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \end{bmatrix}
\begin{bmatrix}
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{bmatrix}
$$

$$
\begin{bmatrix} 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}
$$

For illustrative purposes,

we use `n = 12`, `t = 2`,

and a fictio

with 4-bi

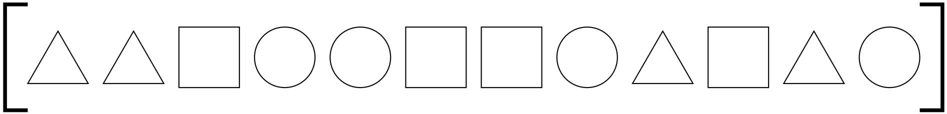
With linear error-correcting codes, a null vector is necessarily a valid value.

~~EOR~~ RSB

$$
\begin{bmatrix}
0&0&0&0&0&1&0&0&0&0&0&0\\
0&0&0&0&1&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&1\\
0&0&1&0&0&0&0&0&0&0&0&0\\
1&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&1&0&0&0\\
 & & & & & &0&0&0&1&0&0\\
 & & & & & &0&0&0&0&0&0\\
0&0&0&0&0&0&1&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&1&0\\
0&0&0&0&0&0&0&1&0&0&0&0\\
0&1&0&0&0&0&0&0&0&0&0&0
\end{bmatrix}
$$

$$
\begin{bmatrix}0&0&0&0&1&0&0&1&0&0&0&0\end{bmatrix}\begin{bmatrix}0&1&1&1&0&0&0&0&0&0&0&0\end{bmatrix}
$$

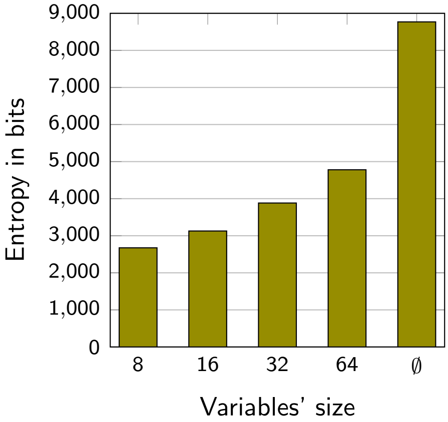
By iterating this process many times with real-sized registers, one can potentially place all the elements in subgroups.



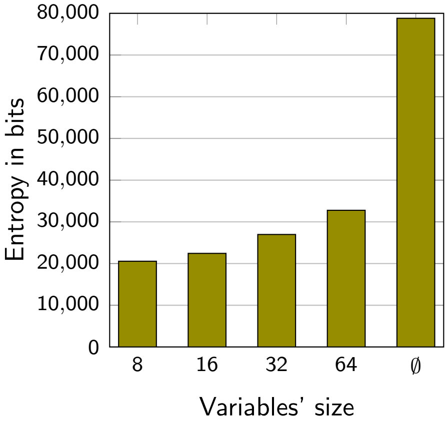
It is important to note that however, we cannot know which subgroup corresponds to which group of columns.

Remaining entropy of the permutation matrix, depending on the variables' size :

with $n = 1024$



with $n = 6960$



Protection on COTS

The fault model considered here focuses on **execution rather than on data**.

In the case of an implementation targeting an open environment, the impacts will differ vastly depending on the technical choices that were made.

The fault model considered here focuses on **execution rather than on data**.

In the case of an implementation targeting an open environment, the impacts will differ vastly depending on the technical choices that were made.

When the solution is simply protected by a layer of obfuscation applied to it, the risk is not mitigated, as long as there is an `EOR` instruction used for the vector-matrix multiplication somewhere

The fault model considered here focuses on **execution rather than on data**.

In the case of an implementation targeting an open environment, the impacts will differ vastly depending on the technical choices that were made.

When the solution is simply protected by a layer of obfuscation applied to it, the risk is not mitigated, as long as there is an `EOR` instruction used for the vector-matrix multiplication somewhere

However by using **white-box techniques**, in particular precalculation of intermediate results, this risk can be completely wiped out

The fault model considered here focuses on **execution rather than on data**.

In the case of an implementation targeting an open environment, the impacts will differ vastly depending on the technical choices that were made.

When the solution is simply protected by a layer of obfuscation applied to it, the risk is not mitigated, as long as there is an `EOR` instruction used for the vector-matrix multiplication somewhere

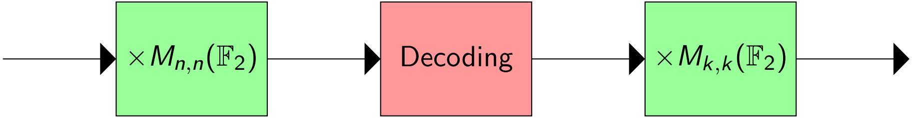
However by using **white-box techniques**, in particular precalculation of intermediate results, this risk can be completely wiped out

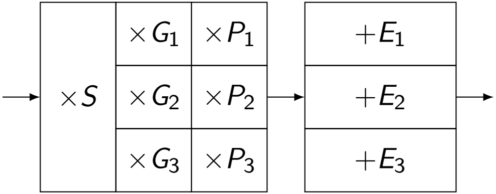
Precalculating results in the McEliece cryptosystem requires caution since this algorithm works on very large values.

$$M = \begin{bmatrix} \begin{bmatrix} M_{0,0} \end{bmatrix} & \begin{bmatrix} M_{0,1} \end{bmatrix} & \cdots & \begin{bmatrix} M_{0,b} \end{bmatrix} \\ \begin{bmatrix} M_{1,0} \end{bmatrix} & \ddots & & \\ \vdots & & & \\ \begin{bmatrix} M_{a,0} \end{bmatrix} & & & \begin{bmatrix} M_{a,b} \end{bmatrix} \end{bmatrix}$$

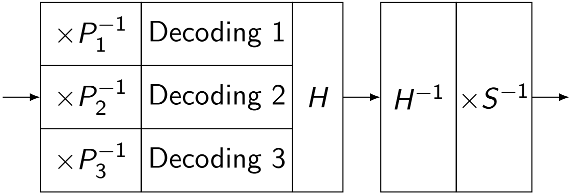
$$u \times M = \left[\sum_{i=0}^{a} u_i \times M_{i,0} \parallel \sum_{i=0}^{a} u_i \times M_{i,1} \parallel \ldots \parallel \sum_{i=0}^{a} u_i \times M_{i,b} \right]$$

If managing the vector-matrix multiplications is possible, precalculating the decoding is impossible as is.





A studied possibility is to divide the code into multiple different subcodes.



While it does modify the specification, it allows to make a version of the cryptosystem protected against the attack.

Adding an external encoding at the output of the decryption, besides the internal one, is completely possible.

Conclusion

While the McEliece cryptosystem is an interesting, robust candidate for asymmetrical post-quantum cryptography, its implementation may be vulnerable to fault-injection based interferences.

These attacks can be deployed in hardware or software. In the latter case, protection via simple obfuscation is not sufficient.

White-box techniques can be applied, under the condition to modify the original specification.

We wish to thank David Naccache for his support during this work.