Hardware & Embedded Security

Prof Daniele Rossi


Via G. Caruso 16, room B-1-03


daniele.rossi1@unipi.it


050 221 7611

Example of Applications of PUFs

Lecture 7 - DR

Brief outline

- Design and evaluation principles of PUF
- Quality metrics for PUF evaluation
- **PUFs generating/storing cryptographic keys**
- PUFs for entity authentication protocols
- PUFs for hardware metering

Examples of possible applications

3

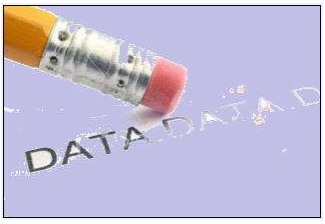
Cryptographic Key Generation

Current approach:

Root keys are typically generated outside the chip after manufacturing stage of silicon devices, and subsequently stored in an on-chip memory

- **Problems:**

1. Manufactures may be malicious
2. Storing the keys in a non-volatile memories makes it vulnerable to readouts attacks by malicious software, wherein a malware can gain an unauthorized access to a device's memory[1]. Another security risk is data **remanence attacks**[2] which makes it feasible for secret information to be deduced from memories even if they have been erased



1. P. Stewin and I. Bystrov, "Understanding DMA Malware," in *Detection of Intrusions and Malware, and Vulnerability Assessment: 9th International Conference, DIMVA 2012, Heraklion, Crete, Greece.*
2. S. Skorobogatov, "Data remanence in flash memory devices," presented at the Proceedings of the 7th international conference on Cryptographic hardware and embedded systems, Edinburgh, UK, 2005.

2

4

We've seen PUFs, we will now see applications examples.

1. PUF generating / storing cryptographic keys.

Why? Storing sensitive info in NVM has its own risk. Plus, for NVM, which are flash memories, they are subject to REMANENCE ATTACKS: even after deleting, there's a trace of the memorized information. This is related to the writing mechanism of the Flash memories. Recall that you inject electrons in a floating gate through tunnel effect to write 1. But tunneling effect introduces some damages that when deeply analyzed can give you critical info.

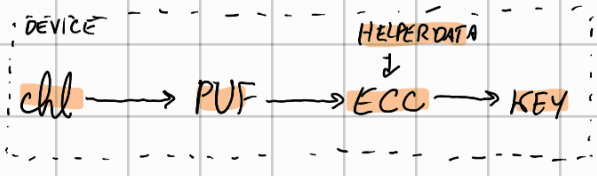
So, I don't store key but I generate it whenever I need it. How to do that in a non-inferrable way? With a PUF!

To generate a key, you supply a challenge, get a response, and use it as a key.

PROBLEM:

- With the same chl I want same response. But I have no control over operative conditions of device, and I have to keep in mind aging problems. You can have RELIABILITY problems.

Solution: we can use ERROR CORRECTION CODES, that must be supported by some "HELPER DATA"



- ECC are used frequently when you have volatile memories. Because they are a bit more "unstable". A high enough glitch in an SRAM cell can flip the state of the bistable. Energetic particles can alter the capacitor's charge in a DRAM memory cell. That's why you use ECC.

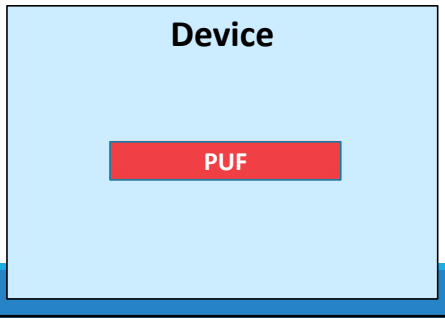
Basic idea is memorize info that gives you hints if whether you have errors or not (helper data).

To ensure error correction, I need some redundacy. Imagine I have two events to encode, one with 0, one with 1. If I read 0 or 1, who tells me I didn't have an error? What if I do 00 and 11? 2 bits allow me to encode 4 events but I only use 00 and 11. So if I see 01 or 10, I still can't find original config, but I can tell there's an error. If I increase redundacy and work with 3 bits, I can decode working with majority. So if there's 1 error, good correction. If 2 errors you have a wrong correction.

Cryptographic Key Generation

A Solution:

- Use PUF to generate an encryption key on demand
- Already available commercially (e.g. INTRINSIC ID)

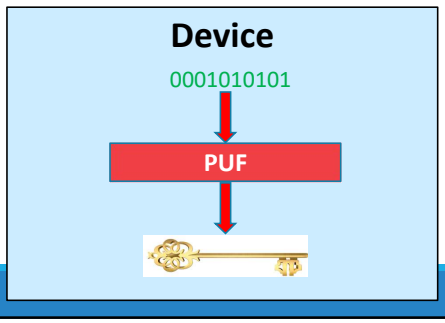


5

Cryptographic Key Generation

A Solution:

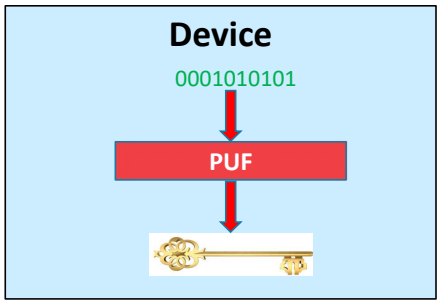
1. Embed a PUF into the design
2. Characterise the PUF challenge/response behaviour (at the post fabrication stage).
3. To generate a key, apply a challenge use the response to generate the key



6

Cryptographic Key Generation

A problem: Environmental variations (power supply, temperature) and ageing may cause the PUF to generate different responses to the same challenge, which means we cannot reproduce the same encryption key.

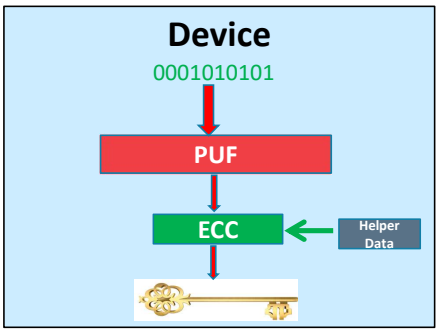


7

Cryptographic Key Generation

Solution:

- Use Error Correction Codes



8

Basic Principles of Error Correction Codes

Repetition Codes Example

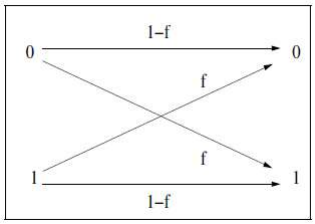


Figure: Binary symmetric channel with noise f

Source	Transmitted
0	000
1	111

9

Basic Principles of Error Correction Codes

Example Source message $\mathbf{s} = 0010110$ Encoder makes $\mathbf{t}$ from $\mathbf{s}$ and sends it.Channel "adds" sparse noise vector $\mathbf{n}$ to $\mathbf{t}$ for receiver to get $\mathbf{r}$

s	0	0	1	0	1	1	0
t	000	000	111	000	111	111	000
n	000	001	000	000	101	000	000
r	000	001	111	000	010	111	000

10

Basic Principles of Error Correction Codes

Decoding by Majority Rule

s	0	0	1	0	1	1	0
t	000	000	111	000	111	111	000
n	000	001	000	000	101	000	000
r	000	001	111	000	010	111	000
$\hat{s}$	0	0	1	0	0	1	0
errors	.	$\oplus$	.	.	$\ominus$	.	.

11

Basic Principles of Error Correction Codes

For a code to encode k data bits, it needs to have at least 2^k codewords (i.e., it can encode k information bits)

Example: the repetition code C = {000, 111} can encode one data bit (k=1)

- With m information bits, the number of check bits I need to add is $\log_2 m + 1$, so acceptable. Redundancy decreases the larger the word gets. But this is for one error detection. With a lot of words, there's a lot of probability of having more than 1 error.

ECC chosen will decrease memory I can use to save effective information!

12

To correct one error I need to have a minimum HD of 3. This is for all single correction codes (codice a correz. singola). $C=\{000,111\} \Rightarrow 3$.

$2t+1$ is the number of bits I need to save that error

t = number of errors I want to correct

Basic Principles of Error Correction Codes

Definition: The minimum distance $d = d(C)$ of a code C is the minimum Hamming distance between any pair of distinct code-words, that is,

$$d = d(C) = \min\{d(a, b) \mid a, b \in C, a \neq b\}$$

Example: Let C be the following length 3 repetition code over the set $Z_2 = [0,1]$
C = {000, 111}

What is the minimum distance of C ?

13

Basic Principles of Error Correction Codes

Theorem. A code C can *detect* up to **s** errors if $d(C) \geq s + 1$. A code C can *correct* up to t errors if $d(C) \geq 2t + 1$.

↳ So a code of distance three allows me to detect 2 errors.

Example: Let C be the following length 3 repetition code over the set $Z_2 = [0,1]$
C = {000, 111}

How many errors can this code detect? At most 2, because if 000 goes to 111 I accept it.

To correct t errors for a data word that has a length of k bits we need to append each data word with extra check bits

14

# of bits I need to add to have a SEC-DED code is $\log_2 K + 2$.

↳ # of data bits that code encodes

SEC-DED code: code that allows you to correct one error and detect up to two. To get a SEC-DED code, your number of bits you need to add a number of bits of $\lceil \log_2 k \rceil + 2$, where K is the # of data bits that your code encodes.

Imagine you have a code of distance 4: you can detect 2 errors! With distance 3 you mistake 2 errors for 1.

Nel SEC DED quello che correggo e 2 errori ne deve essere analogo a quello che posso correggere con 1 errore. Quindi dobbiamo 2 errori vogliamo essere sicuri che siano 2. Se ho $h(d) \geq s+1$, so che posso rivelare fino a 2 errori ma posso confondere con 1 errore da correggere.

> If d(C) is 3, assume C is {000,111}, we can detect up to 2 errors (s + 1 = 3). This means We are not actually sure of WHEN there are two errors, but we know that if we receive 001, we may have got two errors. Correct? Otherwise this formula doesn't make sense

Exactly, Giovanni — you've nailed the subtle but super important point here.

When we say **"we can detect up to 2 errors"**, it doesn't mean we can *tell exactly how many* errors happened — it means that **if 1 or 2 bits get flipped**, the result **won't accidentally match another valid codeword**, so we **know** something's wrong.

Let's walk through your example with $C = \{000, 111\}$, which has a Hamming distance $d(C) = 3$.

What that means:

- **Detect up to 2 errors**:
  Any word with **1 or 2 bit flips** from a valid codeword will **not** land on the *other* valid codeword. So if we receive something like `001` or `101`, it's not a codeword — and we can detect that an error occurred.

Basic Principles of Error Correction Codes

Table: Check Bit Requirements for Single Error Correction Codes

Data Bits (K)	Check Bits	Codeword Length(n)
8	4	12
16	5	21
32	6	38
64	7	71
128	8	136
256	9	265

+1 if you want SEC-DED code

This works for $K > 2$

$\#check\ bits = \lceil \log_2 K \rceil + 1$ In general you have the ceiling here

- It should be noted here that although the overheads of extra bits decreases with longer codes, the complexity of coding/decoding circuitry increases sharply.

15

Basic Principles of Error Correction Codes

- To be able to correct multiple errors, more restrictions on the code-words must be applied, so there are fewer of them, and they are further apart
- **Golay codes** are 3-bit error correcting cyclic codes. They have been used in space missions by NASA since the early eighties. They have the following properties $(n, k, d) = (23, 12, 7)$
- The **B**ose, **C**haudhuri, and **H**ocquenghem (**BCH**) codes and **Reed-Solomon** codes are important cyclic codes which can correct burst of errors. For example, BCH(127,36,31) code will correct up to 15 errors out of 127 bits

Very high redundancy and complex circuitry to be added

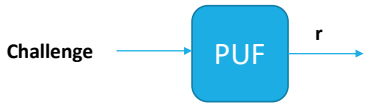
16

How to use ECC to Generate Reliable Response ?

Using Code-Offset Construction:

Stage 1: Helper Data Generation

1. Generate a response *r* for a given challenge



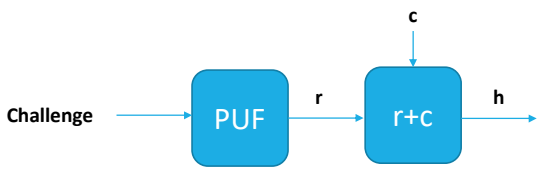
17

How to use ECC to Generate Reliable Response ?

Using Code-Offset Construction:

Stage 1: Helper Data Generation

1. Generate a response *r* for a given challenge
2. Add a random codeword c of a given code C(n, k, d) to r



18

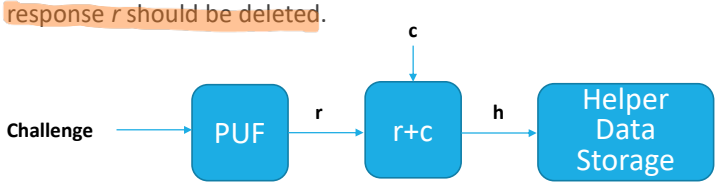
We have said that we need helper data to apply ECC. But we don't want to save anything that might be related to keys. So, I apply chl, get response and select an ECC, for example Hamming Code. Response 2 is summed bit to bit to our response. Suppose code word is on 128 bits and can have all possible configurations on bits. So 2^{128} possibilities. So after adding them together I get h which is our helper data. Reading h gives me no info on the original r or c, because c is one of the possible combinations.

How to use ECC to Generate Reliable Response ?

Using Code-Offset Construction:

Stage 1: Helper Data Generation

1. Generate a response *r* for a given challenge
2. Add a random codeword c of a given code C(n, k, d) to r
3. Store the result *h = r + c* in the *Helper Data Storage* (+: mod 2 addition - XOR)
4. The response *r* should be deleted.



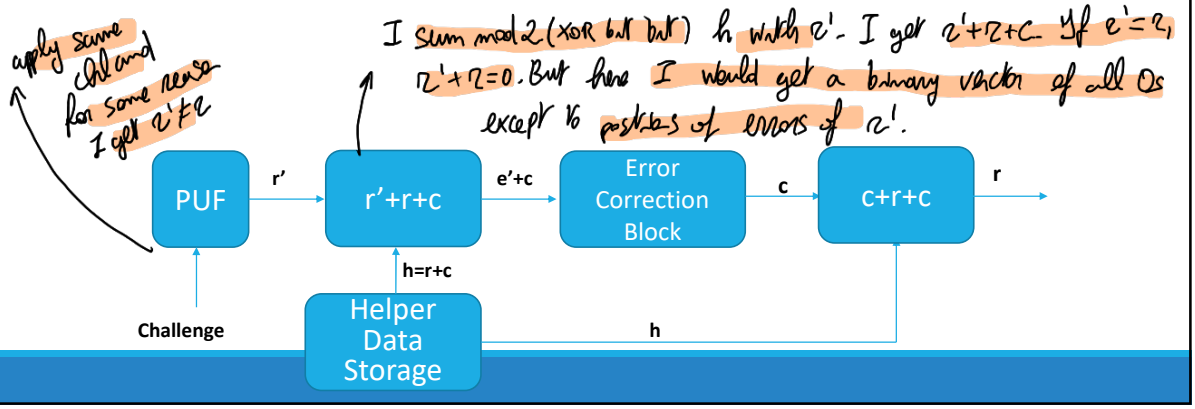
19

How to use ECC to Generate Reliable Response ?

Using Code-Offset Construction

Stage 2: Response Reproduction

1. Apply the same challenge to obtain an erroneous response r' = r+e', where e' is an additive error



10

20

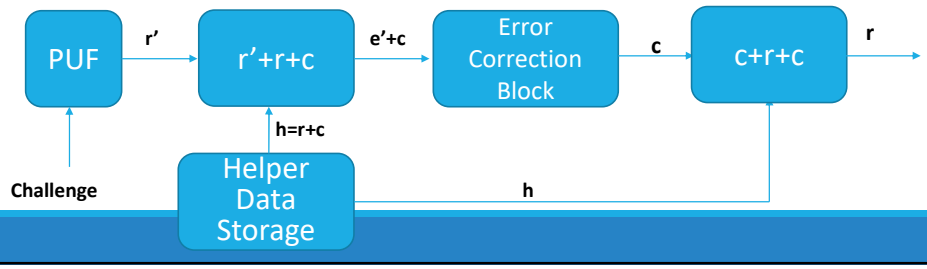
The number of errors I get should be within the correction capacity of ECC. So $c' = e' + c$ is a code word with an error. If I can correct c', I do, and get back c. I apply correction algorithm and I get the random code word I used c. How do I get to r? $h = r \oplus c$ (XOR bitwise), so $c \oplus h = r$. So even w/ wrong puf data, I get correct key.

How to use ECC to Generate Reliable Response ?

Using Code-Offset Construction

Stage 2: Response Reproduction

1. Apply the same challenge to obtain an erroneous response r' = r+e', where e' is an additive error.
2. Use the helper data to generate e'+c



21

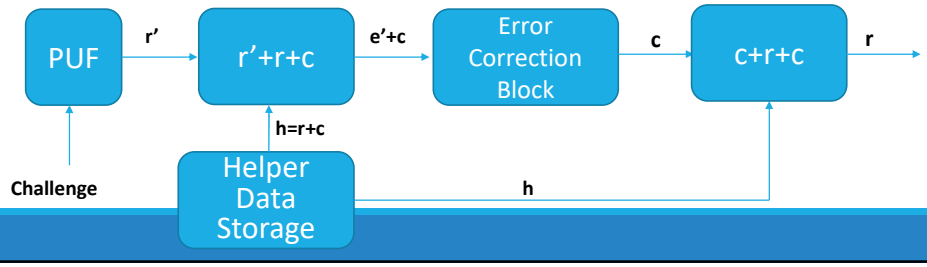
How to use ECC to Generate Reliable Response ?

Using Code-Offset Construction

Stage 2: Response Reproduction

1. Apply the same challenge to obtain an erroneous response r' = r+e', where e' is an additive error.
2. Use the helper data to generate e'+c
3. If the Hamming distance between r and r' (wtH (e') = dH (r, r')) is within the error correction capabilities of the code C(n, k, d), it will be possible to reproduce the original PUF response (r), by first decoding (e'+c) then XORing the result with the helper data (e)

Weight function = # of 1s



22

Any questions?

23

Hamming SEC (7, 4)

↳ # of info bits you started with

↳ # of actual bits: to get SEC you need $2^r \geq$ bits

$$H = \begin{pmatrix} d_0 & d_1 & d_2 & d_3 & c_0 & c_1 & c_2 \\ 1 & 1 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 1 \end{pmatrix}$$

Code word: concatenation of original information bits || code evaluated for correction

CHECK NEXT PAGE

- You construct a matrix by columns by selecting columns 2-to-2 linearly independent, so this means with binary field, 2 different words (excluding 000).
- How many I need to take? Equal to the number of bits associated to the code.

Note that c_0-c_1-c_2 is an Identity matrix.

Starting from this H matrix I can retrieve check bits starting from data bits.

My Identity submatrix corresponds to check bits I need to evaluate. The remaining columns to the data bits I need to encode.

- How do I get check bits? Start from c_0, that starts with a 1 in the first row.

So $c_0 = d_0 \oplus d_1 \oplus d_3$, XOR of all the columns that have a 1 in the first row. Same for others.

$c_1 = d_0 \oplus d_2 \oplus d_3$.

$c_2 = d_1 \oplus d_2 \oplus d_3$

- Since we used identity submatrix, c_i depends only on data bits.

So starting from H matrix built with all different columns excluding 000, and by associating c_i to identity submatrix, I can get the check bits associated from my word:

Code is constructed w/ concatenation of information bits + control bits.
You construct a binary matrix to obtain your code by constructing it by columns. You have to select columns of length equal to the number of control bits you have, 2-to-2 linearly independent. In a binary field this just means different from each other (excluding 0).
So I take configurations and choose the ones I need.
I need a # equal to the # of bits that make up the code word.
So in this case I need to take all of them.
The last part of H is constructed to have a identity submatrix (characteristic of codice lineare e sistematico).
Starting from H I can determine the value of control bits for my informative ones.
My identity matrix corresponds to the check bits I need to evaluate, while the others to the info bits you have.
H allows me to make expression of C_0 as function of d_3.
C_0 has 1 in first row, so I XOR all data bits with 1 in first row and so on.
Since we used identity submatrix, C_0 depends only on data bits.

d_0	d_1	d_2	d_3	c_0	c_1	c_2
0	0	0	0	0	0	0
0	0	0	1	1	1	1
0	0	1	0	0	1	1
0	0	1	1	1	0	0
0	1	0	0	1	0	1
0	1	0	1	0	1	0
0	1	1	0	1	1	0
0	1	1	1	0	0	1
1	0	0	0	1	1	0
1	0	0	1			
1	0	1	0			
1	0	1	1			
1	1	0	0			
1	1	0	1			
1	1	1	0			
1	1	1	1			

$c_0 = d_0 \oplus d_1 \oplus d_3$ EASY

$c_1 = d_0 \oplus d_2 \oplus d_3$ CIRCUIT

$c_2 = d_1 \oplus d_2 \oplus d_3$

- Now suppose our PUF response on 7 bits is $r: 0111001$
And let's assume from our code C we use $c: 0101010$.
So $h = r \oplus c = 0010011$. And I save this helper data.
Reading h (realistically) gives no information on r, our key, brute force is not practical.
Suppose now, after the same challenge we get $r': 0011001$
(we use SEC code, only one error in the 2nd bit).
We now need to combine r' with helper data. NOTE: if I have an error on helper data I'm doomed, but NVM are usually more resistant.
So, $r' \oplus h = 0001010$. This is $e' + c = c'$. Note that $e' = 0100000$.
So I know c' is a code with just a single error. How do I detect error using Hamming SEC?

We use H, and c'. We evaluate an auxiliary quantity called "syndrome d'erreur" that will give us info on "where" we have the error. It is a binary bit equal to the # of control bits (3):

$$\begin{bmatrix} S_0 \\ S_1 \\ S_2 \end{bmatrix} = H \times c'$$

↑ column vector $[0\ 0\ 0\ 1\ 0\ 1\ 0]^T$

$$= \begin{pmatrix} 1 & 1 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 1 \end{pmatrix} \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 1 \\ 0 \end{bmatrix} = \begin{cases} S_0 = 0+0+0+1+0+0+0 \\ S_1 = 0+0+0+1+0+1+0 \\ S_2 = 0+0+0+1+0+0+0 \end{cases} = \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix}$$

S is the second column of matrix H that corresponds to d_1, which tells us that the position of the error is the 2nd bit. If I had 0 errors, S would have been 0.

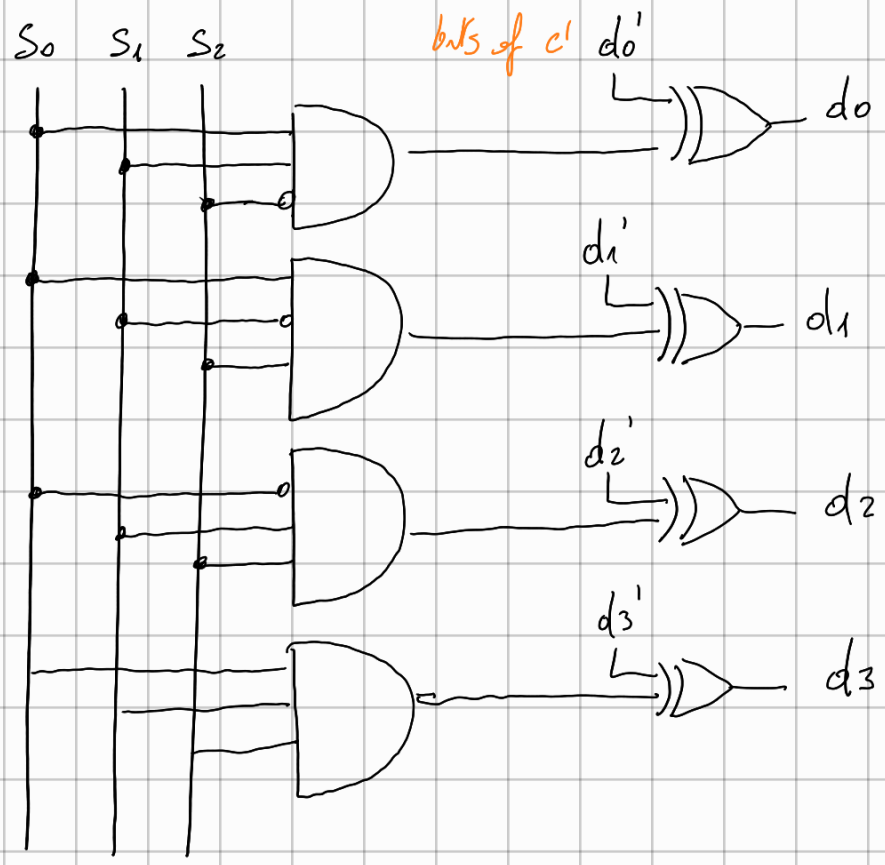
In HW, this job would be done by a decoder: 3 input ANDs that give 1 when we detect the correct position. AND associated to d_1 is $S_0 \cdot \bar{S}_1 \cdot S_2$. And those ANDs for all columns. The result give by AND will be used to correct error we were talking about.

So, we correct $c' = 0001010$ to $c = 0101010$.

We can XOR c with the original additional data h to get r back again.

$c \oplus h = 0111001$.

This circuit requires 3 XOR tree for c_0, c_1, c_2. Other XOR trees for S_0, S_1, S_2, and then a decoding block to identify bit position that has the mistake:



Recall: I needed 7 columns, 1 for each # bits (control + data).
If I have a double error (ex on d_1 and d_2), the syndrome I obtain is the linear combination of the syndromes I would get from d_1 and d_2.
And I obtain that is just 1 error.
To detect 2 errors I need to increase redundancy.

- We work with COLONNE DI PESO DISPARI (with an odd number of 1s).
So columns are of 4 bits. Identity matrix is 4×4.

$$H = \begin{array}{c} \begin{matrix} d_0 & d_1 & d_2 & d_3 & c_0 & c_1 & c_2 & c_3 \end{matrix} \\ \begin{pmatrix} 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \end{array}$$

SEC-DED (8, 4)

If we have a single error, I would have a syndrome of peso dispari. If I have two errors, I would have a syndrome of peso pari. Of course syndrome should be $\neq \underline{0}$.

QUESTION: # of columns I need are (due a due linearmente indipendenti, so different) equal to # of bits I use for my words (control + data).