Message-recovery Horizontal Correlation Attack on *Classic McEliece*

CASCADE 2025

Brice Colombier, Vincent Grosso, Pierre-Louis Cayrel, Vlad-Florin Drăgoi

April 2, 2025

Laboratoire Hubert Curien
UMR ▪ CNRS ▪ 5516 ▪ Saint-Étienne

AUREL VLAICU UNIVERSITY of ARAD

Context

Feb. 2016 NIST PQC competition announcement at PQCrypto

Jul. 2022 CRYSTALS-KYBER selected for standardization
4th round candidates:

- BIKE
- *Classic McEliece*
- HQC
- SIKE

Context

Feb. 2016 NIST PQC competition announcement at PQCrypto

Jul. 2022 CRYSTALS-KYBER selected for standardization
4th round candidates:

- BIKE
- *Classic McEliece*
- HQC
- SIKE

Dec. 2024 CASCADE'25 winter submission deadline

Mar. 2025 HQC selected for standardization

Apr. 2025 CASCADE'25

Classic McEliece

Classic McEliece[1] is a **Key Encapsulation Mechanism**

- `KeyGen()` $\rightarrow (\mathbf{H}_{\text{pub}}, \mathsf{k}_{\text{priv}})$
- `Encap(`$\mathbf{H}_{\text{pub}}$`)` $\rightarrow (\mathbf{s}, \mathsf{k}_{\text{session}})$
- `Decap(`$\mathbf{s}, \mathsf{k}_{\text{priv}}$`)` $\rightarrow (\mathsf{k}_{\text{session}})$

The `Encap` algorithm (Niederreiter encryption[2]) encapsulates a secret value to be shared.

- `Encap(`$\mathbf{H}_{\text{pub}}$`)` $\rightarrow (\mathbf{s}, \mathsf{k}_{\text{session}})$

  Generate a random vector $\mathbf{e} \in \mathbb{F}_2^{\mathbf{n}}$ of Hamming weight $\mathbf{t}$ (($\mathbf{n}$; $\mathbf{t}$): security parameters)

  Compute $\mathbf{s} = \mathbf{H}_{\text{pub}}\mathbf{e}$

  Compute the hash: $\mathsf{k}_{\text{session}} = \mathrm{H}(1, \mathbf{e}, \mathbf{s})$

[1] Martin R. Albrecht et al. **Classic McEliece: Conservative Code-Based Cryptography**. 2022.

[2] Harald Niederreiter. "Knapsack-Type Cryptosystems and Algebraic Coding Theory". In: **Problems of Control and Information Theory** (1986).

Classic McEliece[1] is a **Key Encapsulation Mechanism**

- `KeyGen()` $\rightarrow$ ($\mathbf{H}_{\text{pub}}$, k_{priv})
- `Encap(`$\mathbf{H}_{\text{pub}}$`)` $\rightarrow$ ($\mathbf{s}$, k_{session})
- `Decap(`$\mathbf{s}$, k_{priv}`)` $\rightarrow$ (k_{session})

The `Encap` algorithm (Niederreiter encryption[2]) encapsulates a secret value to be shared.

- `Encap(`$\mathbf{H}_{\text{pub}}$`)` $\rightarrow$ ($\mathbf{s}$, k_{session})

  Generate a random vector $\mathbf{e} \in \mathbb{F}_2^{\mathbf{n}}$ of Hamming weight $\mathbf{t}$ (($\mathbf{n}$; $\mathbf{t}$): security parameters)

  Compute $\mathbf{s} = \mathbf{H}_{\text{pub}}\mathbf{e}$

  Compute the hash: $k_{\text{session}} = \mathrm{H}(1, \mathbf{e}, \mathbf{s})$

[1] Martin R. Albrecht et al. **Classic McEliece: Conservative Code-Based Cryptography**. 2022.

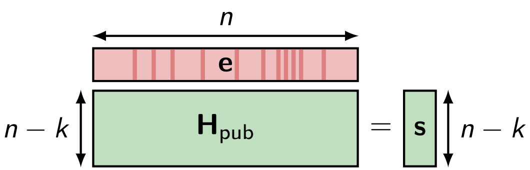
[2] Harald Niederreiter. "Knapsack-Type Cryptosystems and Algebraic Coding Theory". In: **Problems of Control and Information Theory** (1986).

Syndrome decoding problem

Syndrome decoding problem

Input: a binary parity-check matrix $\mathbf{H} \in \mathbb{F}_2^{(n-k)\times n}$
a binary vector $\mathbf{s} \in \mathbb{F}_2^{n-k}$
a scalar $t \in \mathbb{N}^*$

Output: a binary vector $\mathbf{x} \in \mathbb{F}_2^n$ with a Hamming weight $\mathrm{HW}(\mathbf{x}) \leq t$ such that: $\mathbf{Hx} = \mathbf{s}$



n	k	$(n-k)$	t	$\mathbf{H}_{\text{pub}}$ size
3488	2720	768	64	261 kB
4608	3360	1248	96	524 kB
6688	5024	1664	128	1.04 MB
6960	5413	1547	119	1.04 MB
8192	6528	1664	128	1.35 MB

The public key $\mathbf{H}_{\text{pub}}$ is **very large**.

Embedded software / hardware implementations

Embedded software / hardware implementations are now feasible[3][4][5][6][7].

New threats

That makes them vulnerable to **physical attacks** (fault injection & side-channel analysis)

[3] Johannes Roth et al. "Classic McEliece Implementation with Low Memory Footprint". In: **CARDIS**. Nov. 2020.
[4] Ming-Shing Chen et al. "Classic McEliece on the ARM Cortex-M4". In: **TCHES** (2021).
[5] Po-Jen Chen et al. "Complete and Improved FPGA Implementation of Classic McEliece". In: **TCHES** (2022).
[6] Cyrius Nugier et al. "Acceleration of a Classic McEliece Postquantum Cryptosystem With Cache Processing". In: **IEEE Micro** (2024).
[7] Peizhou Gan et al. **Classic McEliece Hardware Implementation with Enhanced Side-Channel and Fault Resistance**. 2024. URL: https://eprint.iacr.org/2024/1828. Pre-published.

Message-recovery attacks on *Classic McEliece*

For a KEM, a **message-recovery** attack recovers the **shared secret**:

Ref.	**Principle**	👎
[8]	recovers chunks of **e** from timing info	multiple decryption queries
[9]	examines different types of leakage	simulation only
[10]	instruction corruption `XOR` $\rightarrow$ `ADD`	laser fault injection **$$$**
[11][12]	templates on Hamming weight	profiled

Horizontal correlation with Hamming distance leakage: **unprofiled** and **single-trace**.

[8] Norman Lahr et al. "Side Channel Information Set Decoding Using Iterative Chunking - Plaintext Recovery from the "Classic McEliece" Hardware Reference Implementation". In: **ASIACRYPT**. Dec. 2020.

[9] Anna-Lena Horlemann et al. "Information-Set Decoding with Hints". In: **CBCrypto**. June 2021.

[10] Pierre-Louis Cayrel et al. "Message-Recovery Laser Fault Injection Attack on the Classic McEliece Cryptosystem". In: **EUROCRYPT**. Oct. 2021.

[11] Brice Colombier et al. "Profiled Side-Channel Attack on Cryptosystems Based on the Binary Syndrome Decoding Problem". In: **IEEE TIFS** (2022).

[12] Vincent Grosso et al. "Punctured Syndrome Decoding Problem - Efficient Side-Channel Attacks Against Classic McEliece". In: **COSADE**. Apr. 2023.

Syndrome computation

Matrix-vector multiplication

The $\mathbf{s} = \mathbf{H}_{\text{pub}}\mathbf{e}$ multiplication is performed over $\mathbb{F}_2$.
Bitwise operations are **bitsliced** and bits are **packed** into words of size w.

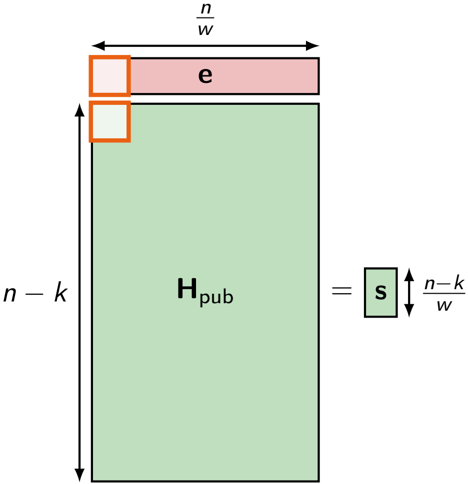
Implementation	w
Reference *Classic McEliece*	8
ARM Cortex-M4[13]	32
Optimized *Classic McEliece*	64

[13] Ming-Shing Chen et al. "Classic McEliece on the ARM Cortex-M4". In: **TCHES** (2021).

Matrix-vector multiplication for $w = 8$

Exemple implementation for $w = 8$:

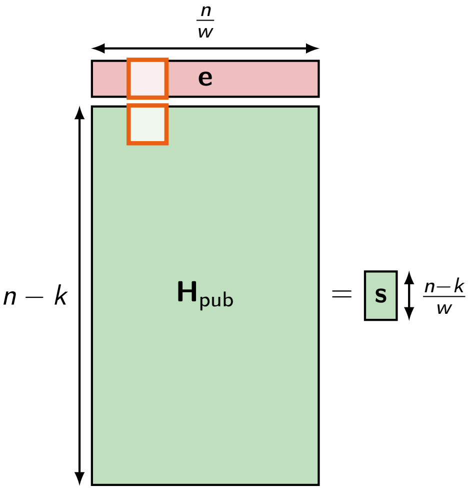
```
for (size_t row = 0; row < n - k; row++)
{
  b = 0;
  for (size_t col = 0; col < n / 8; col++)
    b ^= H[row][col] & e[col];
  b ^= b >> 4;
  b ^= b >> 2;
  b ^= b >> 1;
  b &= 1;
  s[row / 8] |= (b << (row % 8));
}
```



Matrix-vector multiplication for $w = 8$

Exemple implementation for $w = 8$:

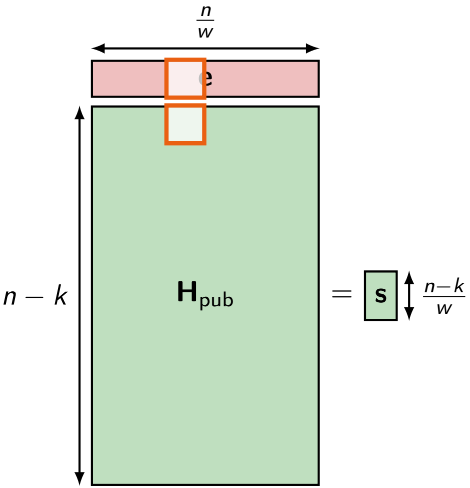
```
for (size_t row = 0; row < n - k; row++)
{
  b = 0;
  for (size_t col = 0; col < n / 8; col++)
    b ^= H[row][col] & e[col];
  b ^= b >> 4;
  b ^= b >> 2;
  b ^= b >> 1;
  b &= 1;
  s[row / 8] |= (b << (row % 8));
}
```



Matrix-vector multiplication for $w = 8$

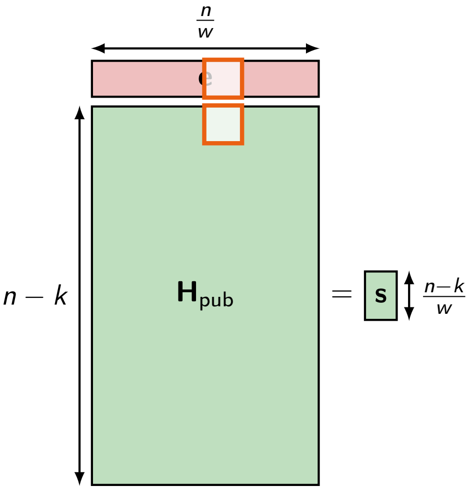
Exemple implementation for $w = 8$:

```
for (size_t row = 0; row < n - k; row++)
{
  b = 0;
  for (size_t col = 0; col < n / 8; col++)
    b ^= H[row][col] & e[col];
  b ^= b >> 4;
  b ^= b >> 2;
  b ^= b >> 1;
  b &= 1;
  s[row / 8] |= (b << (row % 8));
}
```



Exemple implementation for $w = 8$:

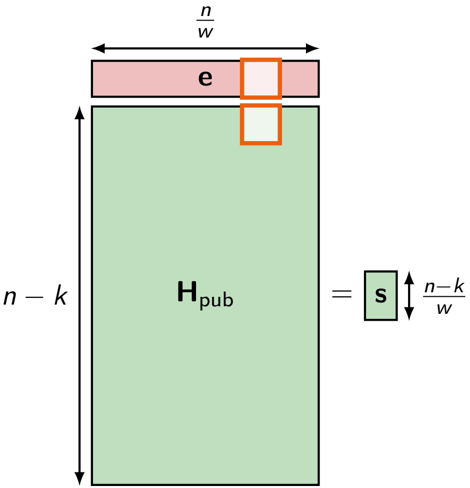
```
for (size_t row = 0; row < n - k; row++)
{
  b = 0;
  for (size_t col = 0; col < n / 8; col++)
    b ^= H[row][col] & e[col];
  b ^= b >> 4;
  b ^= b >> 2;
  b ^= b >> 1;
  b &= 1;
  s[row / 8] |= (b << (row % 8));
}
```



Matrix-vector multiplication for $w = 8$

Exemple implementation for $w = 8$:

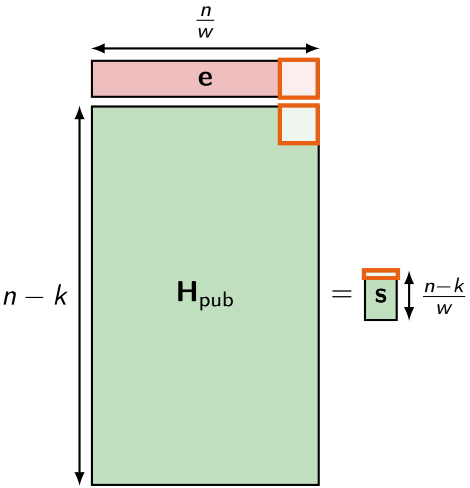
```
for (size_t row = 0; row < n - k; row++)
{
  b = 0;
  for (size_t col = 0; col < n / 8; col++)
    b ^= H[row][col] & e[col];
  b ^= b >> 4;
  b ^= b >> 2;
  b ^= b >> 1;
  b &= 1;
  s[row / 8] |= (b << (row % 8));
}
```



Matrix-vector multiplication for $w = 8$

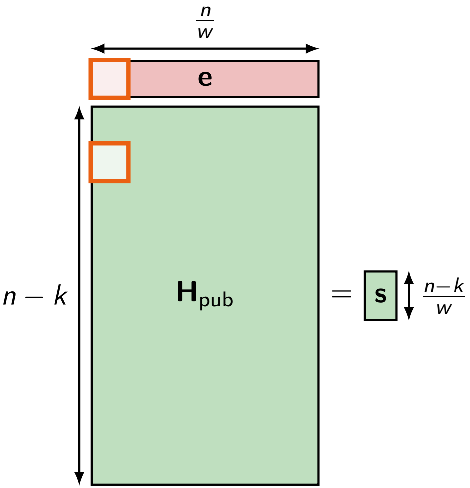
Exemple implementation for $w = 8$:

```
for (size_t row = 0; row < n - k; row++)
{
  b = 0;
  for (size_t col = 0; col < n / 8; col++)
    b ^= H[row][col] & e[col];
  b ^= b >> 4;
  b ^= b >> 2;
  b ^= b >> 1;
  b &= 1;
  s[row / 8] |= (b << (row % 8));
}
```



Exemple implementation for $w = 8$:

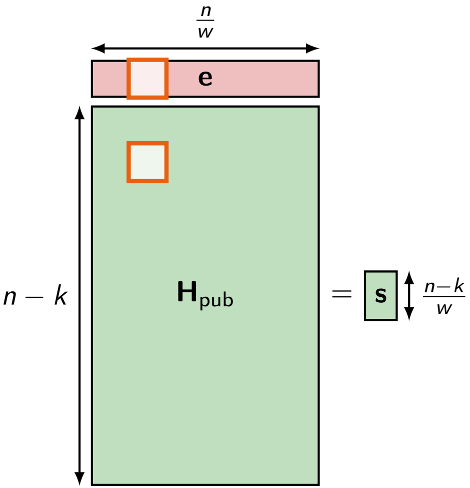
```
for (size_t row = 0; row < n - k; row++)
{
  b = 0;
  for (size_t col = 0; col < n / 8; col++)
    b ^= H[row][col] & e[col];
  b ^= b >> 4;
  b ^= b >> 2;
  b ^= b >> 1;
  b &= 1;
  s[row / 8] |= (b << (row % 8));
}
```



Matrix-vector multiplication for $w = 8$

Exemple implementation for $w = 8$:

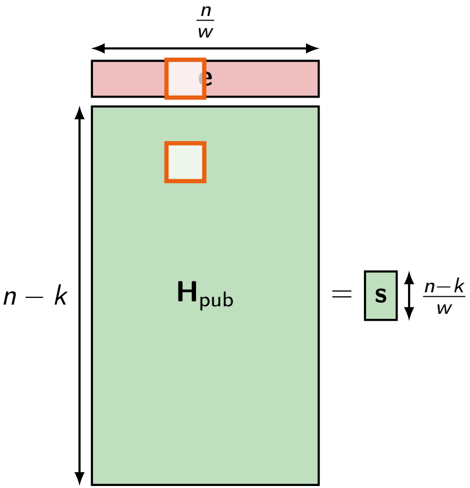
```
for (size_t row = 0; row < n - k; row++)
{
  b = 0;
  for (size_t col = 0; col < n / 8; col++)
    b ^= H[row][col] & e[col];
  b ^= b >> 4;
  b ^= b >> 2;
  b ^= b >> 1;
  b &= 1;
  s[row / 8] |= (b << (row % 8));
}
```



Matrix-vector multiplication for $w = 8$

Exemple implementation for $w = 8$:

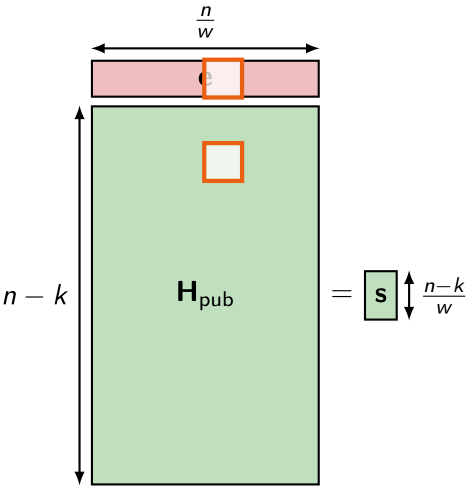
```
for (size_t row = 0; row < n - k; row++)
{
  b = 0;
  for (size_t col = 0; col < n / 8; col++)
    b ^= H[row][col] & e[col];
  b ^= b >> 4;
  b ^= b >> 2;
  b ^= b >> 1;
  b &= 1;
  s[row / 8] |= (b << (row % 8));
}
```



Matrix-vector multiplication for $w = 8$

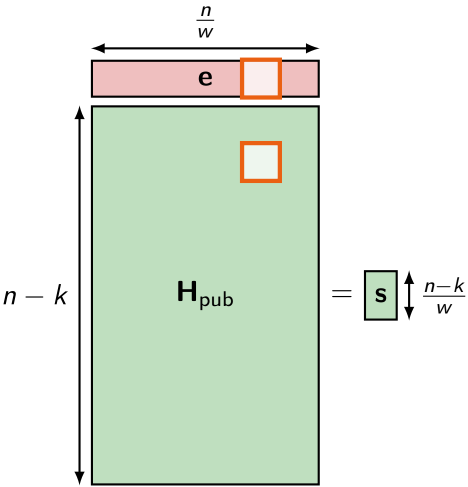
Exemple implementation for $w = 8$:

```c
for (size_t row = 0; row < n - k; row++)
{
  b = 0;
  for (size_t col = 0; col < n / 8; col++)
    b ^= H[row][col] & e[col];
  b ^= b >> 4;
  b ^= b >> 2;
  b ^= b >> 1;
  b &= 1;
  s[row / 8] |= (b << (row % 8));
}
```



Exemple implementation for $w = 8$:

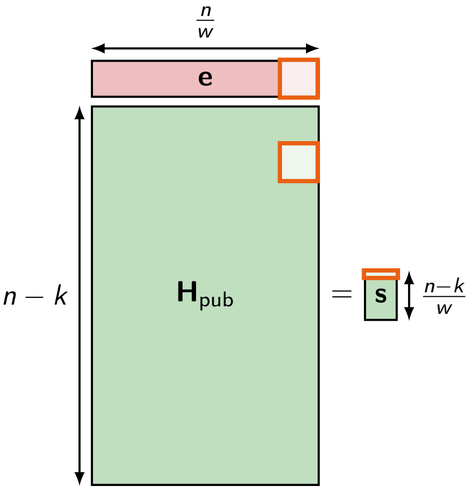
```
for (size_t row = 0; row < n - k; row++)
{
  b = 0;
  for (size_t col = 0; col < n / 8; col++)
    b ^= H[row][col] & e[col];
  b ^= b >> 4;
  b ^= b >> 2;
  b ^= b >> 1;
  b &= 1;
  s[row / 8] |= (b << (row % 8));
}
```



Matrix-vector multiplication for $w = 8$

Exemple implementation for $w = 8$:

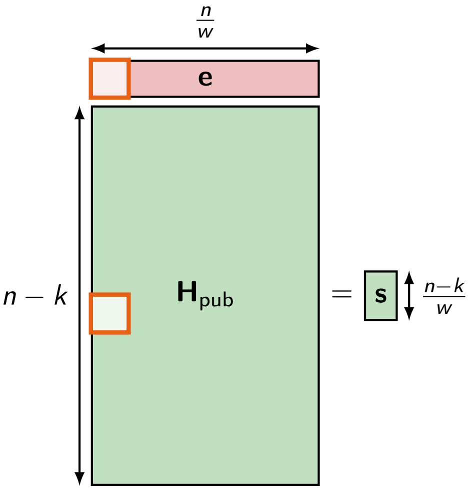
```
for (size_t row = 0; row < n - k; row++)
{
  b = 0;
  for (size_t col = 0; col < n / 8; col++)
    b ^= H[row][col] & e[col];
  b ^= b >> 4;
  b ^= b >> 2;
  b ^= b >> 1;
  b &= 1;
  s[row / 8] |= (b << (row % 8));
}
```



Matrix-vector multiplication for $w = 8$

Exemple implementation for $w = 8$:

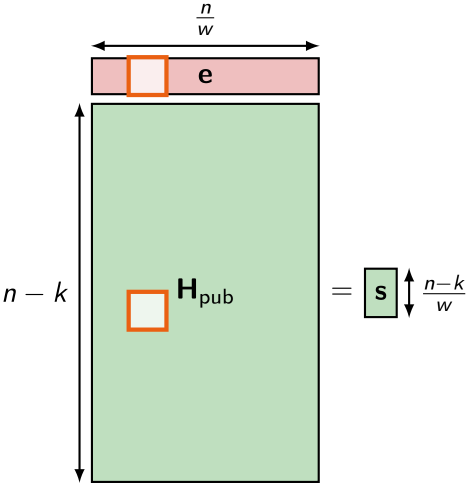
```
for (size_t row = 0; row < n - k; row++)
{
  b = 0;
  for (size_t col = 0; col < n / 8; col++)
    b ^= H[row][col] & e[col];
  b ^= b >> 4;
  b ^= b >> 2;
  b ^= b >> 1;
  b &= 1;
  s[row / 8] |= (b << (row % 8));
}
```



Matrix-vector multiplication for $w = 8$

Exemple implementation for $w = 8$:

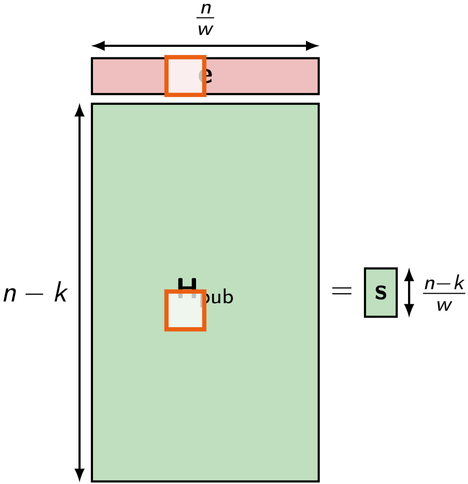
```
for (size_t row = 0; row < n - k; row++)
{
  b = 0;
  for (size_t col = 0; col < n / 8; col++)
    b ^= H[row][col] & e[col];
  b ^= b >> 4;
  b ^= b >> 2;
  b ^= b >> 1;
  b &= 1;
  s[row / 8] |= (b << (row % 8));
}
```



Matrix-vector multiplication for $w = 8$

Exemple implementation for $w = 8$:

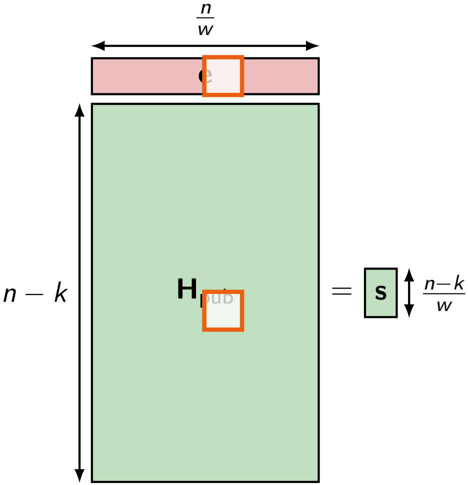
```
for (size_t row = 0; row < n - k; row++)
{
  b = 0;
  for (size_t col = 0; col < n / 8; col++)
    b ^= H[row][col] & e[col];
  b ^= b >> 4;
  b ^= b >> 2;
  b ^= b >> 1;
  b &= 1;
  s[row / 8] |= (b << (row % 8));
}
```



Matrix-vector multiplication for $w = 8$

Exemple implementation for $w = 8$:

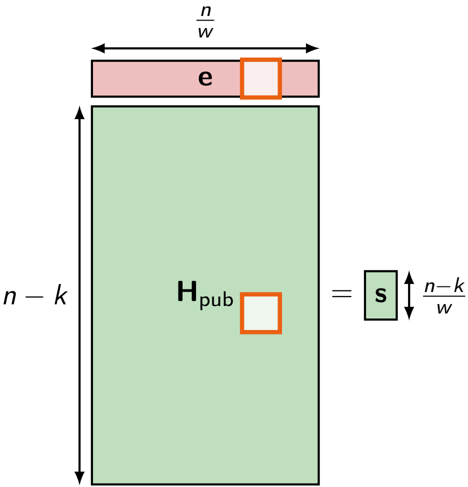
```
for (size_t row = 0; row < n - k; row++)
{
  b = 0;
  for (size_t col = 0; col < n / 8; col++)
    b ^= H[row][col] & e[col];
  b ^= b >> 4;
  b ^= b >> 2;
  b ^= b >> 1;
  b &= 1;
  s[row / 8] |= (b << (row % 8));
}
```



Matrix-vector multiplication for $w = 8$

Exemple implementation for $w = 8$:

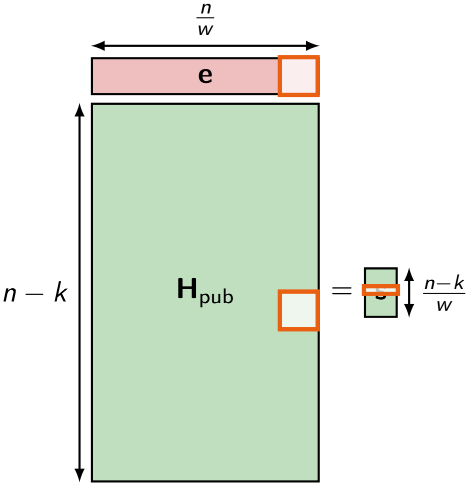
```
for (size_t row = 0; row < n - k; row++)
{
  b = 0;
  for (size_t col = 0; col < n / 8; col++)
    b ^= H[row][col] & e[col];
  b ^= b >> 4;
  b ^= b >> 2;
  b ^= b >> 1;
  b &= 1;
  s[row / 8] |= (b << (row % 8));
}
```



Matrix-vector multiplication for $w = 8$

Exemple implementation for $w = 8$:

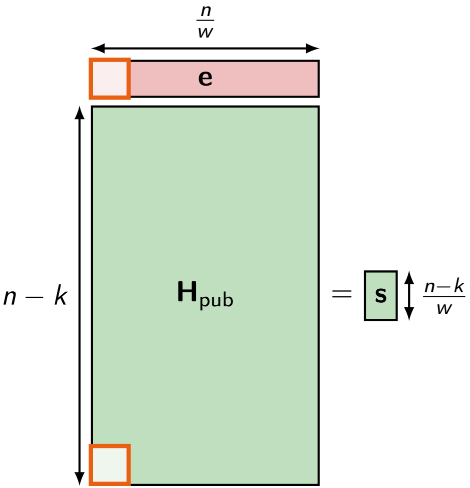
```
for (size_t row = 0; row < n - k; row++)
{
  b = 0;
  for (size_t col = 0; col < n / 8; col++)
    b ^= H[row][col] & e[col];
  b ^= b >> 4;
  b ^= b >> 2;
  b ^= b >> 1;
  b &= 1;
  s[row / 8] |= (b << (row % 8));
}
```



Matrix-vector multiplication for $w = 8$

Exemple implementation for $w = 8$:

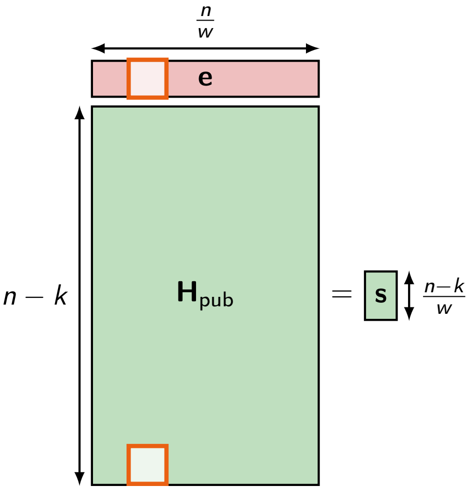
```
for (size_t row = 0; row < n - k; row++)
{
  b = 0;
  for (size_t col = 0; col < n / 8; col++)
    b ^= H[row][col] & e[col];
  b ^= b >> 4;
  b ^= b >> 2;
  b ^= b >> 1;
  b &= 1;
  s[row / 8] |= (b << (row % 8));
}
```



Matrix-vector multiplication for $w = 8$

Exemple implementation for $w = 8$:

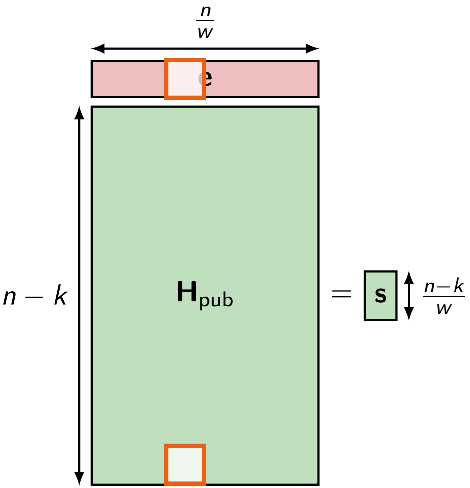
```
for (size_t row = 0; row < n - k; row++)
{
  b = 0;
  for (size_t col = 0; col < n / 8; col++)
    b ^= H[row][col] & e[col];
  b ^= b >> 4;
  b ^= b >> 2;
  b ^= b >> 1;
  b &= 1;
  s[row / 8] |= (b << (row % 8));
}
```



Matrix-vector multiplication for $w = 8$

Exemple implementation for $w = 8$:

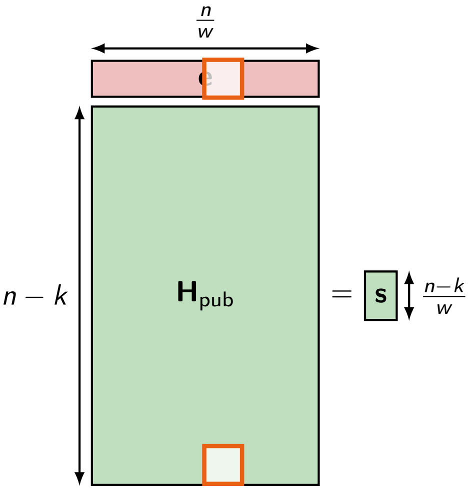
```
for (size_t row = 0; row < n - k; row++)
{
  b = 0;
  for (size_t col = 0; col < n / 8; col++)
    b ^= H[row][col] & e[col];
  b ^= b >> 4;
  b ^= b >> 2;
  b ^= b >> 1;
  b &= 1;
  s[row / 8] |= (b << (row % 8));
}
```



Matrix-vector multiplication for $w = 8$

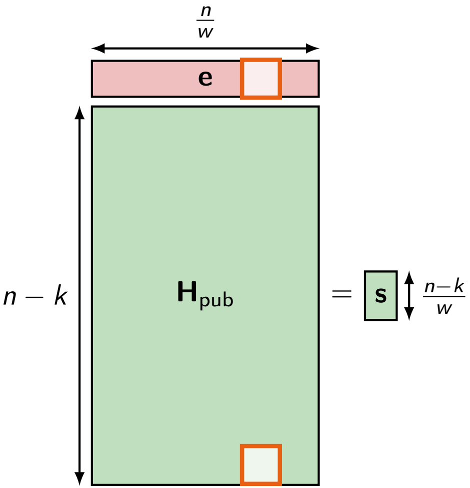
Exemple implementation for $w = 8$:

```
for (size_t row = 0; row < n - k; row++)
{
  b = 0;
  for (size_t col = 0; col < n / 8; col++)
    b ^= H[row][col] & e[col];
  b ^= b >> 4;
  b ^= b >> 2;
  b ^= b >> 1;
  b &= 1;
  s[row / 8] |= (b << (row % 8));
}
```



Exemple implementation for $w = 8$:

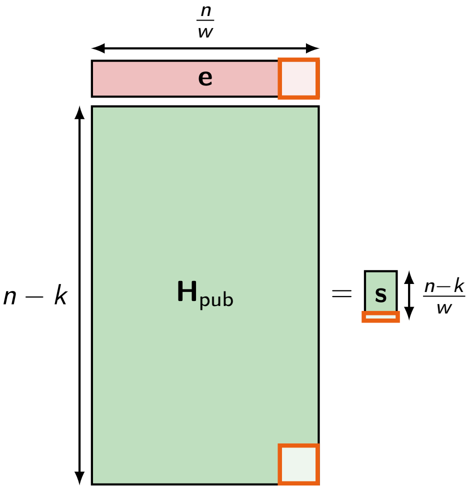
```
for (size_t row = 0; row < n - k; row++)
{
  b = 0;
  for (size_t col = 0; col < n / 8; col++)
    b ^= H[row][col] & e[col];
  b ^= b >> 4;
  b ^= b >> 2;
  b ^= b >> 1;
  b &= 1;
  s[row / 8] |= (b << (row % 8));
}
```



Matrix-vector multiplication for $w = 8$

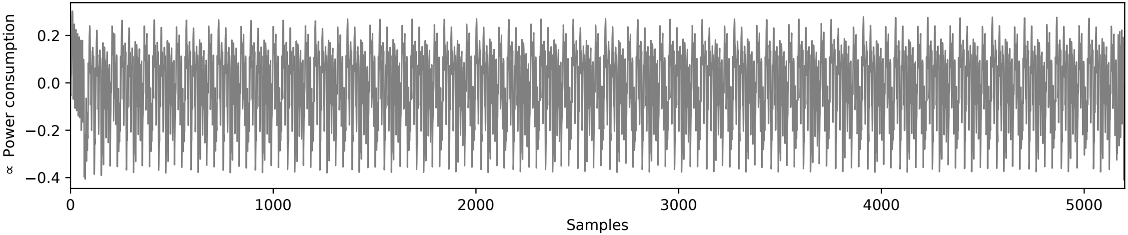
Exemple implementation for $w = 8$:

```
for (size_t row = 0; row < n - k; row++)
{
  b = 0;
  for (size_t col = 0; col < n / 8; col++)
    b ^= H[row][col] & e[col];
  b ^= b >> 4;
  b ^= b >> 2;
  b ^= b >> 1;
  b &= 1;
  s[row / 8] |= (b << (row % 8));
}
```

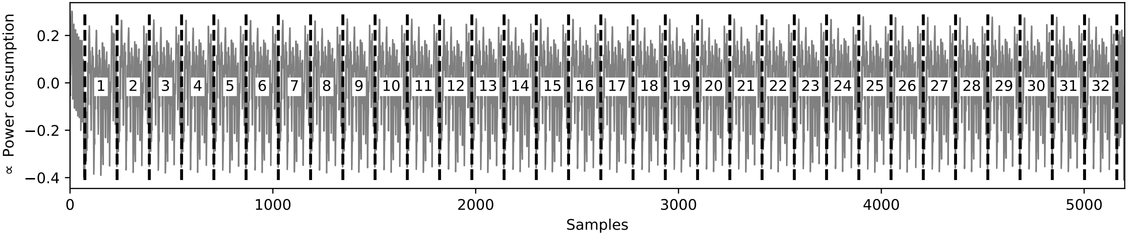


Horizontal side-channel attack

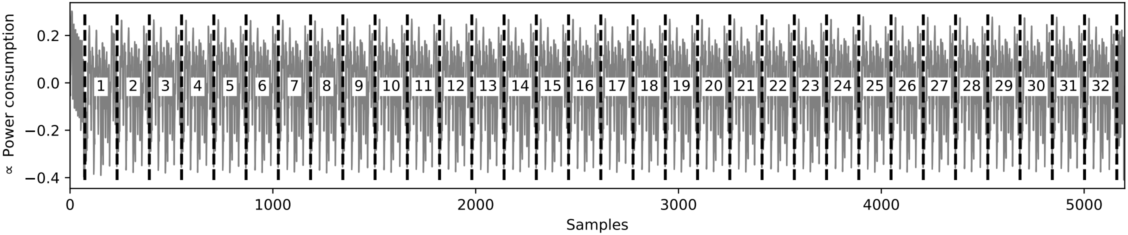
Example side-channel trace for $n - k = 32$

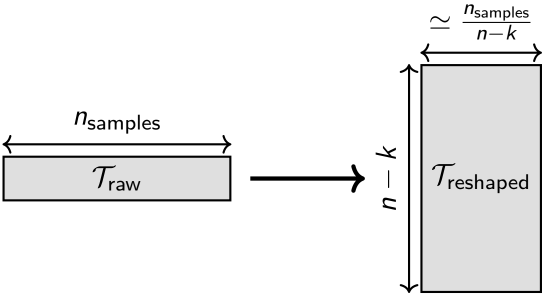


Example side-channel trace for $n - k = 32$



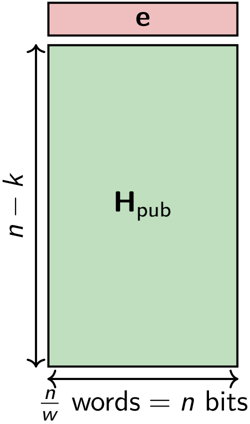
Example side-channel trace for $n - k = 32$

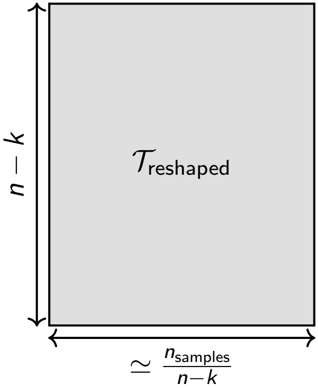




$\mathbf{e}[\mathtt{i}]$	$\mathbf{H}_{\text{pub}}[\mathtt{i},\mathtt{j}]$	$\mathbf{e} \wedge \mathbf{H}_{\text{pub}}$
0	0	0
0	1	0
1	0	0
1	1	1

$\mathtt{b[i]}$	$\mathbf{e} \wedge \mathbf{H}_{\text{pub}}$	$\oplus$	$\mathcal{L}_{HD}$
0	0	0	0
0	1	1	1
1	0	1	0
1	1	0	1

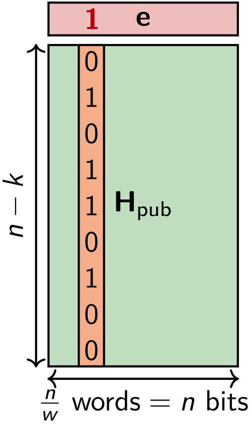


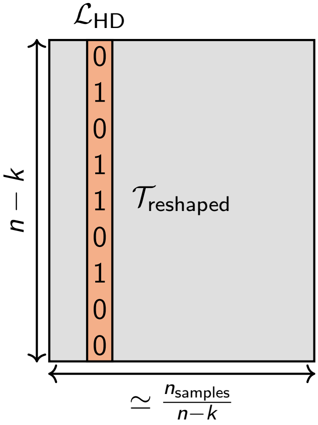


From error positions to Hamming distance leakage

$\mathbf{e}[\mathtt{i}]$	$\mathbf{H}_{\text{pub}}[\mathtt{i},\mathtt{j}]$	$\mathbf{e} \wedge \mathbf{H}_{\text{pub}}$
0	0	0
0	1	0
1	0	0
1	1	1

$\mathtt{b[i]}$	$\mathbf{e} \wedge \mathbf{H}_{\text{pub}}$	$\oplus$	$\mathcal{L}_{HD}$
0	0	0	0
0	1	1	1
1	0	1	0
1	1	0	1

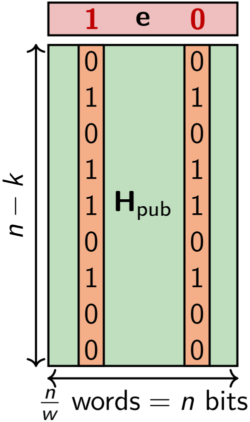


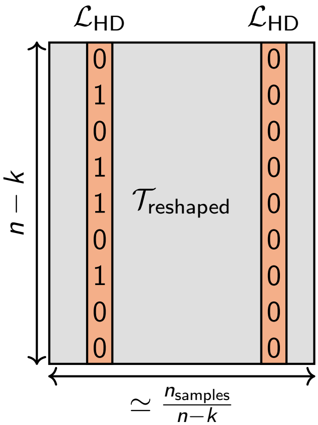


From error positions to Hamming distance leakage

$\mathbf{e}[\mathtt{i}]$	$\mathbf{H}_{\text{pub}}[\mathtt{i},\mathtt{j}]$	$\mathbf{e} \wedge \mathbf{H}_{\text{pub}}$
0	0	0
0	1	0
1	0	0
1	1	1

$\mathtt{b[i]}$	$\mathbf{e} \wedge \mathbf{H}_{\text{pub}}$	$\oplus$	$\mathcal{L}_{HD}$
0	0	0	0
0	1	1	1
1	0	1	0
1	1	0	1

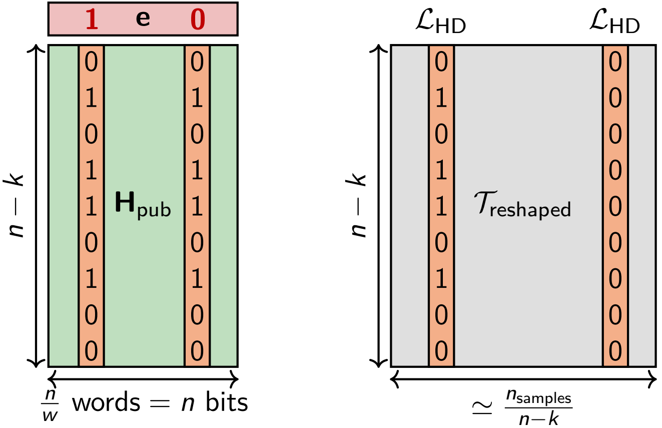




From error positions to Hamming distance leakage

$\mathbf{e}[\mathtt{i}]$	$\mathbf{H}_{\text{pub}}[\mathtt{i},\mathtt{j}]$	$\mathbf{e} \wedge \mathbf{H}_{\text{pub}}$
0	0	0
0	1	0
1	0	0
1	1	1

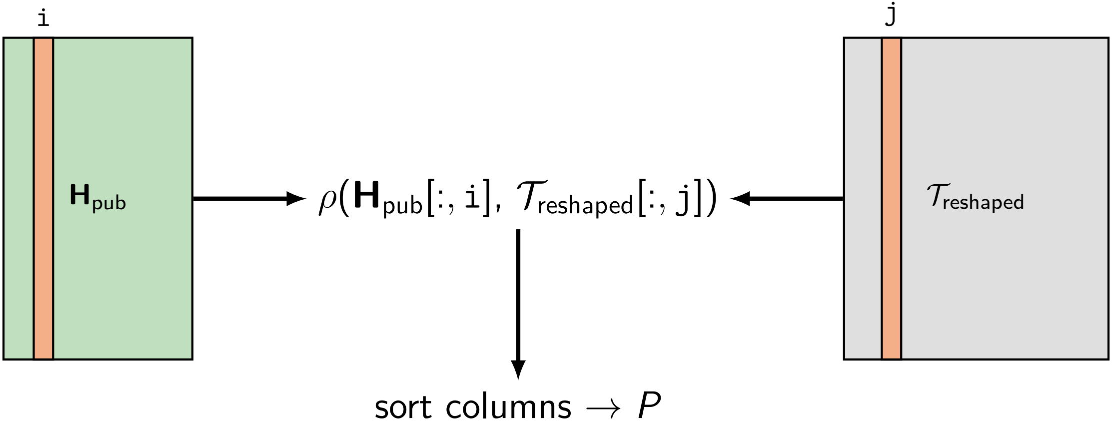
$\mathtt{b}[\mathtt{i}]$	$\mathbf{e} \wedge \mathbf{H}_{\text{pub}}$	$\oplus$	$\mathcal{L}_{HD}$
0	0	0	0
0	1	1	1
1	0	1	0
1	1	0	1



Key observation

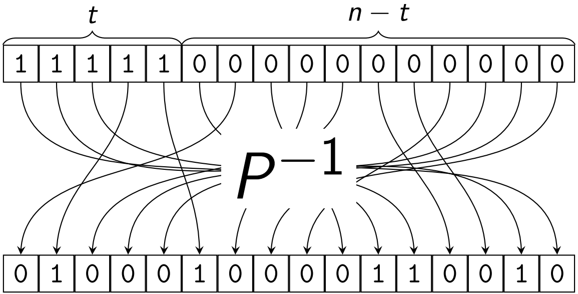
In $\mathcal{T}_{\text{reshaped}}$, t columns of HD leakage must match columns of $\mathbf{H}_{\text{pub}}$ that face a 1 in $\mathbf{e}$.

Correlation



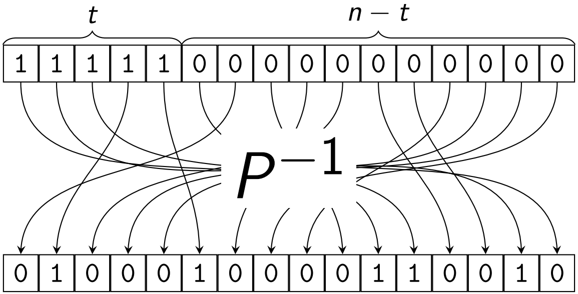
Exploiting the permutation P

(Extremely) lucky case

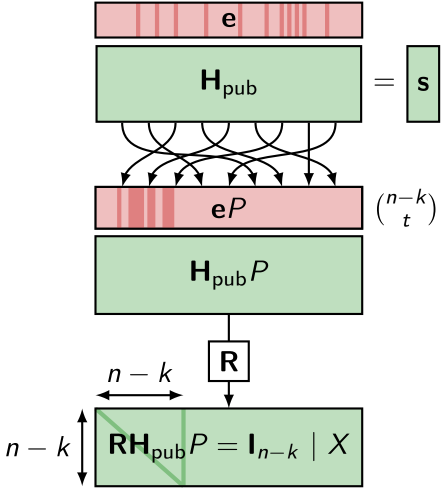


Exploiting the permutation P

(Extremely) lucky case



More realistic: ISD-based strategies



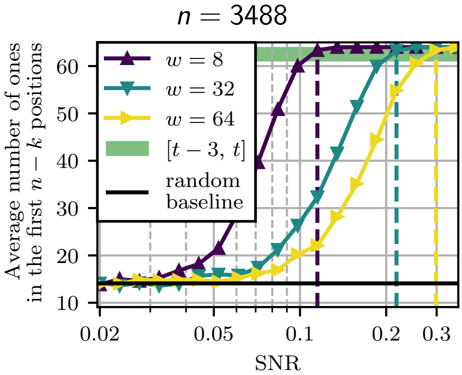
$\rightarrow$ $\mathrm{HW}(\mathbf{R}s) = t$ ✓

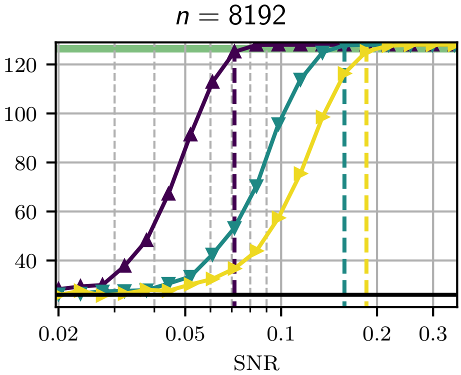
Experimental results

Simulated traces

Simulated leakage after every bitwise operation by concatenating:

- the Hamming weight of the current `b` value and
- the Hamming distance to the previous `b` value.





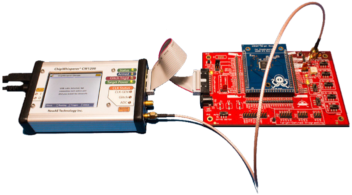
Conclusion

Best success rate with **smaller words** and **larger cryptographic parameters**

Real-life traces

Reference implementation running on the ChipWhisperer[14] platform.

Target device:



- ARM Cortex-M4 core with **32-bit registers**: $w = 32$
- 256 kB of Flash memory (only...)

  Cryptographic parameters (n, k, t) are scaled accordingly[15]
  - $(640, 512, 13)$
  - $(1600, 1280, 30)$
- Compilation optimization level: `-O0`, `-O1`, `-O2`, `-O3` and `-Os`

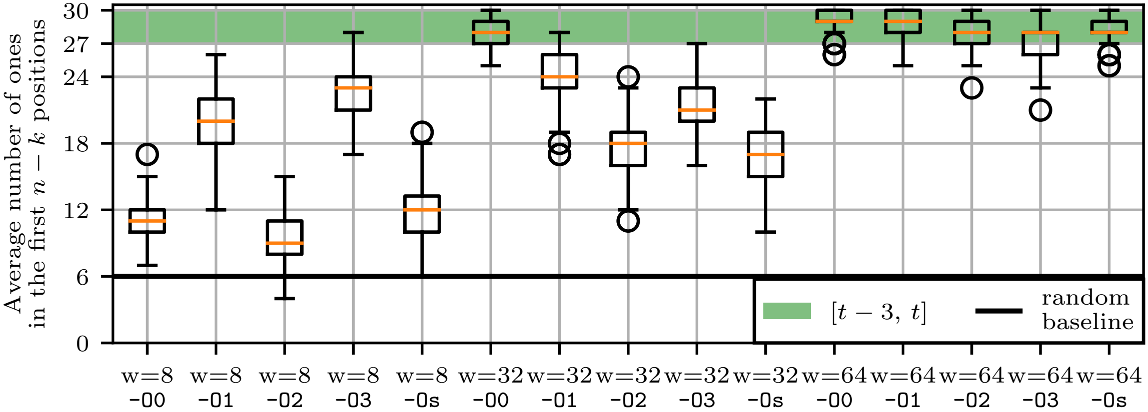
$$n = 640 \quad \#\texttt{cycles}_{\texttt{clk}} \text{ ranging from } 3120 \text{ to } 153$$

$$n = 1600 \quad \#\texttt{cycles}_{\texttt{clk}} \text{ ranging from } 7080 \text{ to } 419$$

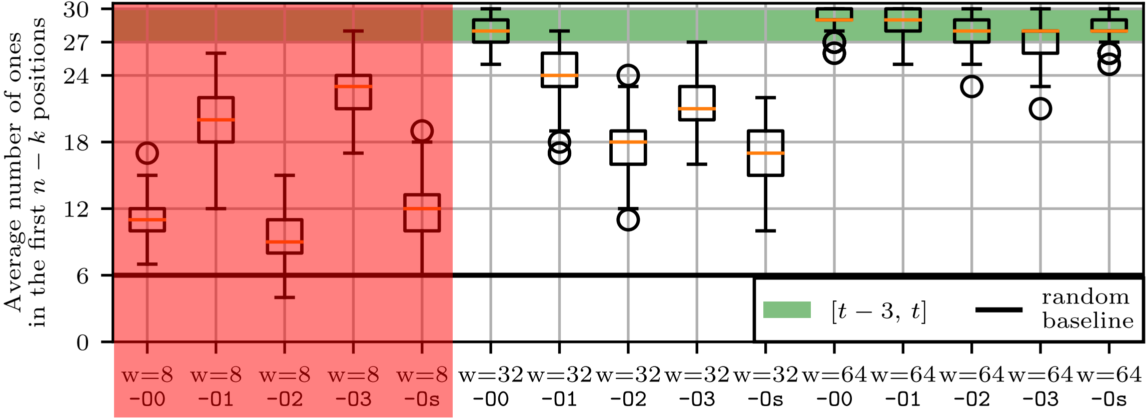
[14] Colin O'Flynn et al. "ChipWhisperer: An Open-Source Platform for Hardware Embedded Security Research". In: **COSADE**. Apr. 2014.

[15] `https://decodingchallenge.org/goppa`

Experimental results $n = 1600$ and $HW(e) = t = 30$



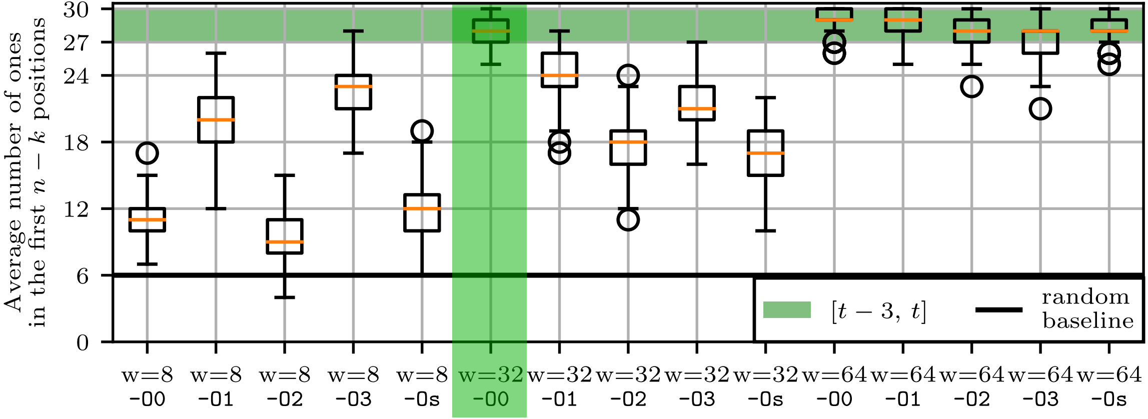
Experimental results $n = 1600$ and $HW(e) = t = 30$



Attack does not work for $w = 8$

- Lots of sub-word-size memory accesses,
- Strong Hamming weight leakage, not much Hamming distance.

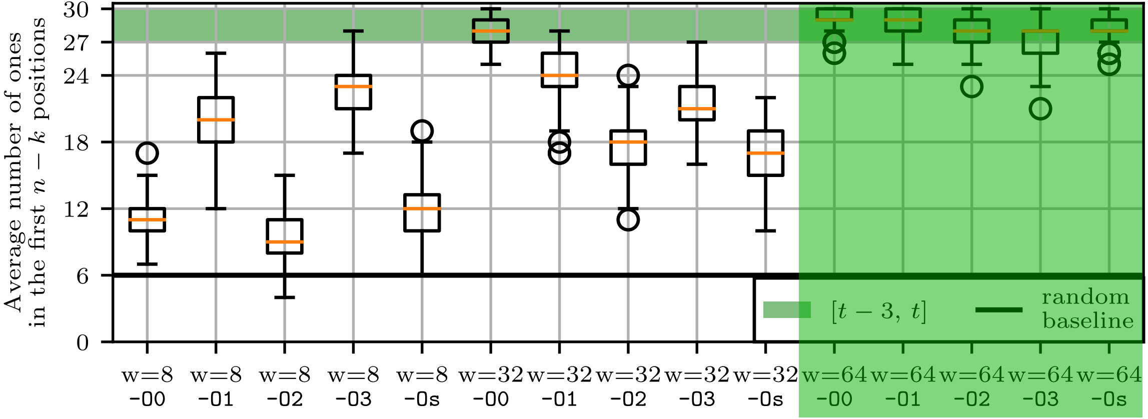
Experimental results $n = 1600$ and $HW(e) = t = 30$



Attack works for $w = 32$ and -00

- Strong Hamming distance leakage,
- `load` $\rightarrow$ `eors` $\rightarrow$ `store` sequence.

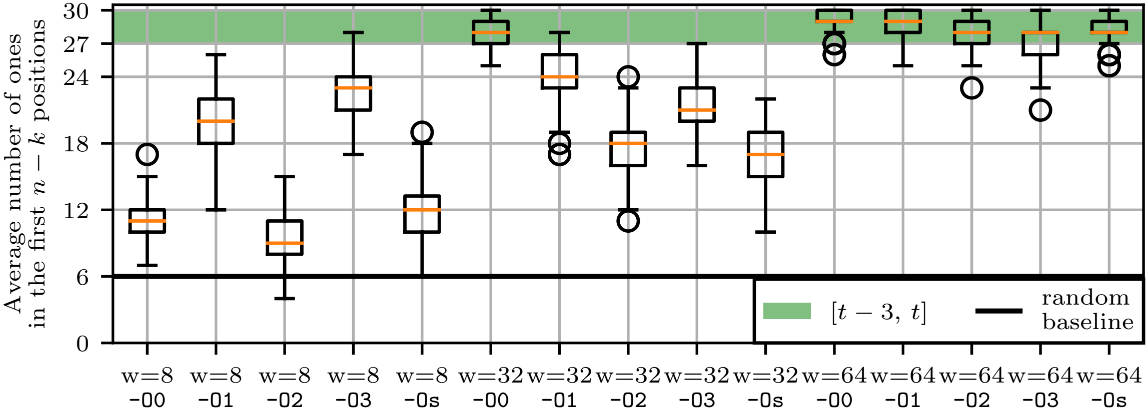
Experimental results $n = 1600$ and $HW(e) = t = 30$



Attack works for $w = 64$

- ???

Experimental results $n = 1600$ and $HW(e) = t = 30$



Conclusion

Experiments contradict simulations: **larger** words are **easier** to attack.

Conclusion

Conclusion & perspectives

Conclusion:

- First **unprofiled single-trace** message-recovery attack on *Classic McEliece*,
- Valitated in **practice**,
- **No clear understanding** of the attack success in practice.

Perspectives:

- Microcontrollers for which $w = 8$ and $w = 64$ are the **native word width**,
- **Hardware** implementations,
- **Assembly**-level countermeasures to prevent Hamming distance leakage.

Conclusion & perspectives

Conclusion:

- First **unprofiled single-trace** message-recovery attack on *Classic McEliece*,
- Valitated in **practice**,
- **No clear understanding** of the attack success in practice.

Perspectives:

- Microcontrollers for which $w = 8$ and $w = 64$ are the **native word width**,
- **Hardware** implementations,
- **Assembly**-level countermeasures to prevent Hamming distance leakage.

— Questions ? —