Politecnico di Torino

AES CACHE-COLLISION TIMING ATTACKS

Embedded Systems Security
Prof. Stefano Di Carlo

Gaspard M.
Thomas R.
Marius H.

Co-funded by the European Union
La Région Auvergne-Rhône-Alpes
AGENCE ERASMUS+ FRANCE · EDUCATION FORMATION
Erasmus+

Acknowledgments

- This material was initially developed as part of an assignment for the Operating Systems for embedded systems course delivered at Politecnico di Torino by Prof. Stefano Di Carlo during the academic year 2022/20023.

- Credits for the preparation of this material go to:
  - Gaspard M. : gaspard.michel01@gmail.com
  - Thomas R. : thomasrio0506@gmail.com
  - Marius H. : hln.marius@gmail.com

Summary

- Side channel attacks
- AES
- Weaknesses of AES (Bernstein)
- Bernstein Attack
- Bonneau Attack
- Implementation of Bernstein's attack
- Potential Countermeasures

Side-channel attacks

What is a side-channel attack ?

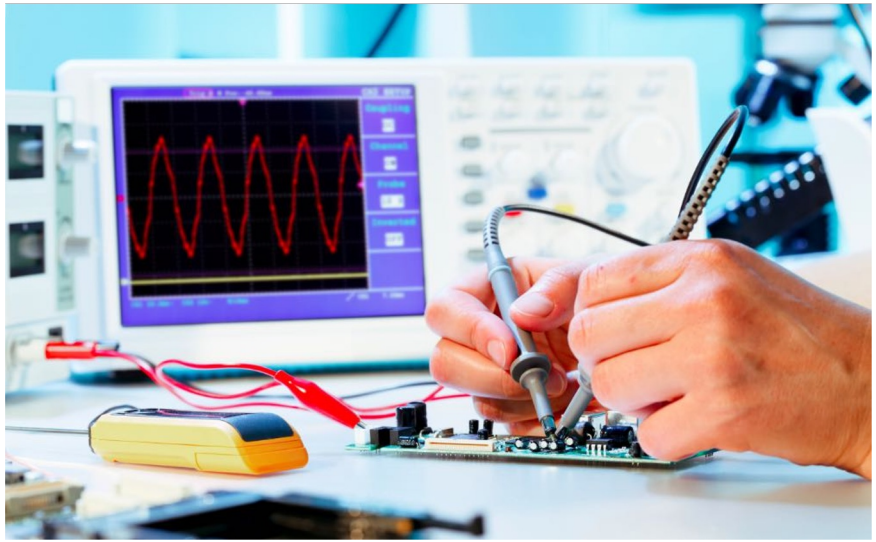
Principle :

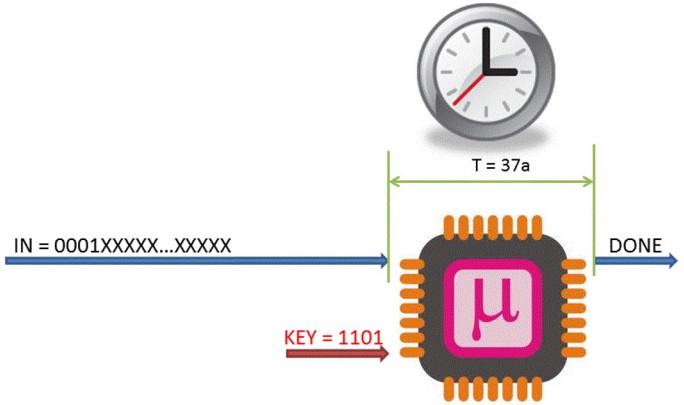
- No attack on the system itself
- Attack the system by an alternative way

Consequences :

- Allow the attacker to find very sensitive data
- Very difficult to detect and to counter
- Are able to bypass well-designed secure systems
- BUT difficult to implement and difficult to realize

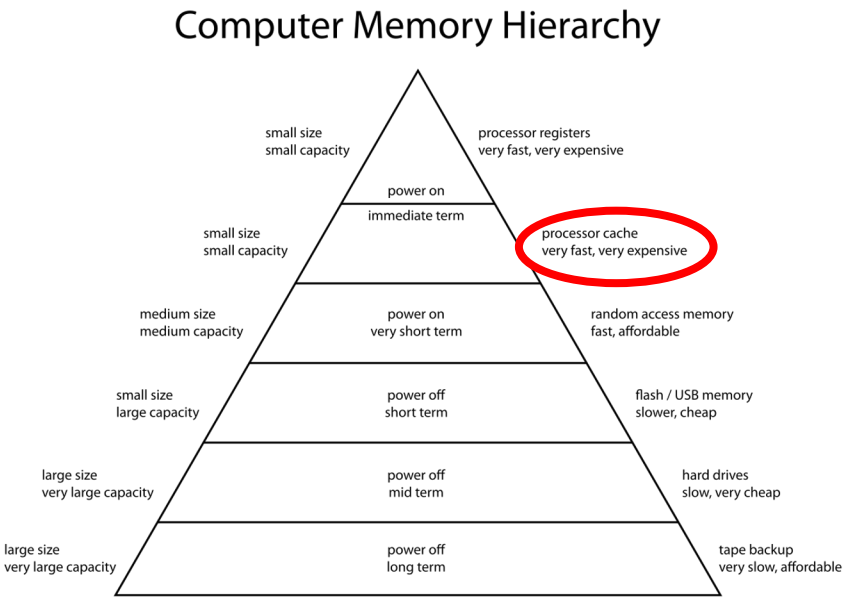
Types of side-channel attack

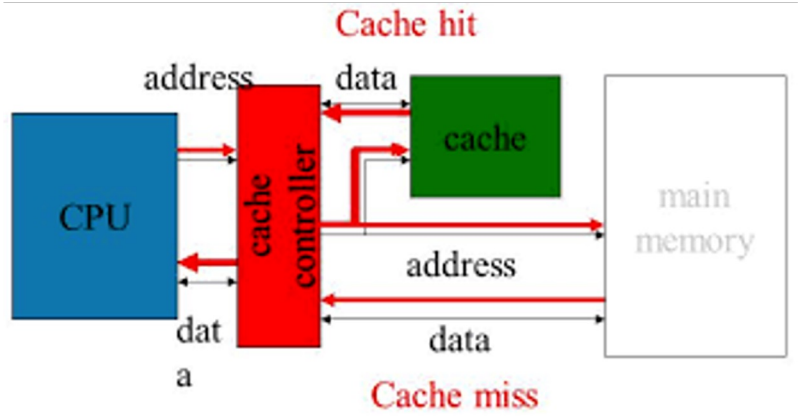




Picture is taken from www.clker.com

Key concept : Cache Hit





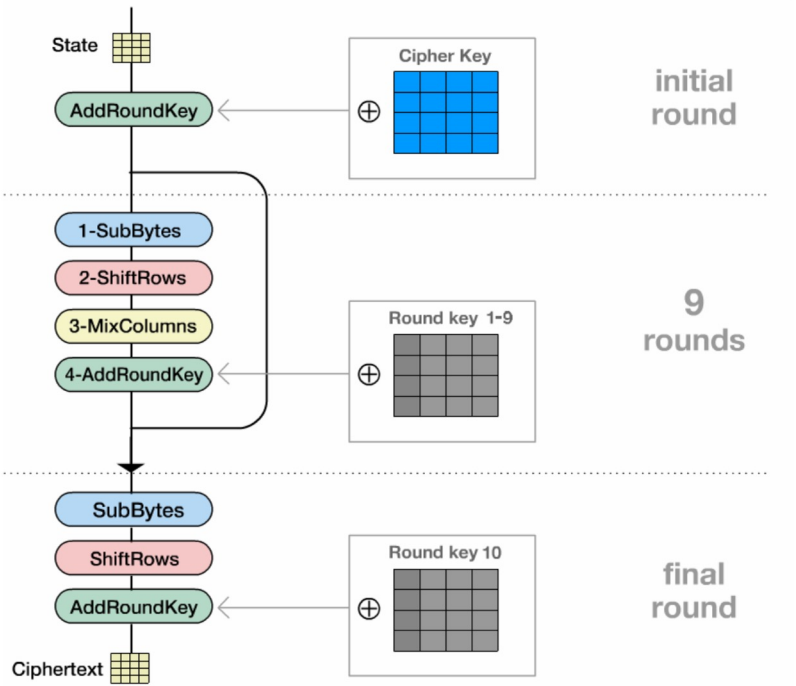
AES

What is AES ?

Advanced Encryption Standard

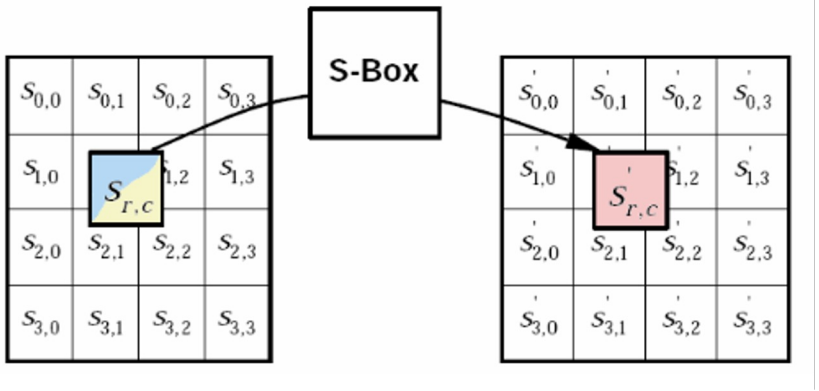
- Encryption algorithm defined as a standard
- Widely used and considered as very secure
- Encode 128 bits blocks using a key with a length of 128, 192 or 256 bits

AES - General view



Figures from the course of J.M. Dutertre written in 2011.

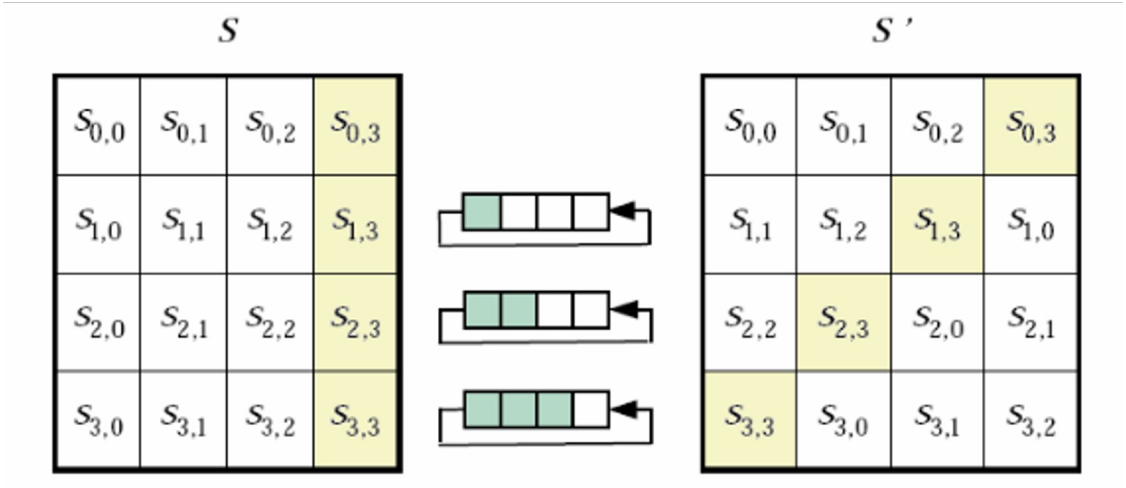
AES - SubBytes



x \ y	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
0	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
1	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
2	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
3	04	c7	23	c3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
4	09	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84
5	53	d1	00	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
6	d0	ef	aa	fb	43	4d	33	85	45	f9	02	7f	50	3c	9f	a8
7	51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
8	cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
9	60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0b	db
a	e0	32	3a	0a	49	06	24	5c	c2	d3	ac	62	91	95	e4	79
b	e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	08
c	ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
d	70	3e	b5	66	48	03	f6	0e	61	35	57	b9	86	c1	1d	9e
e	e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
f	8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16

Figures from the course of J.M. Dutertre written in 2011.

AES - ShiftRows



Figures from the course of J.M. Dutertre written in 2011.

AES - MixColumns

$$\begin{bmatrix} s'_{0,c} \\ s'_{1,c} \\ s'_{2,c} \\ s'_{3,c} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} s_{0,c} \\ s_{1,c} \\ s_{2,c} \\ s_{3,c} \end{bmatrix}$$

$$s'_{0,c} = (\{02\} \bullet s_{0,c}) \oplus (\{03\} \bullet s_{1,c}) \oplus s_{2,c} \oplus s_{3,c}$$

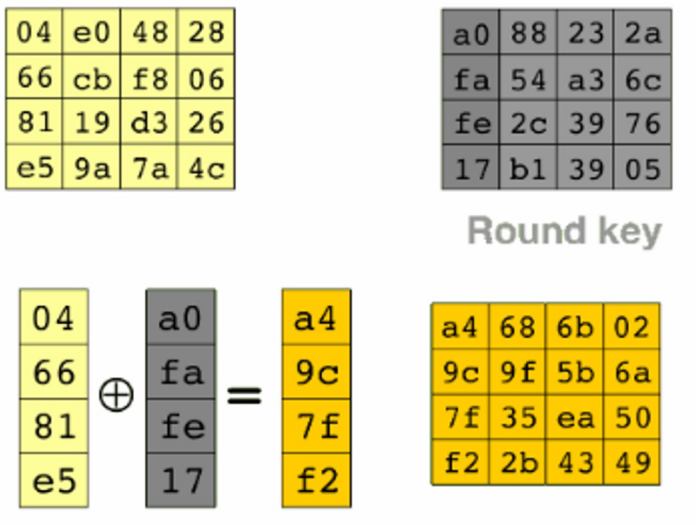
$$s'_{1,c} = s_{0,c} \oplus (\{02\} \bullet s_{1,c}) \oplus (\{03\} \bullet s_{2,c}) \oplus s_{3,c}$$

$$s'_{2,c} = s_{0,c} \oplus s_{1,c} \oplus (\{02\} \bullet s_{2,c}) \oplus (\{03\} \bullet s_{3,c})$$

$$s'_{3,c} = (\{03\} \bullet s_{0,c}) \oplus s_{1,c} \oplus s_{2,c} \oplus (\{02\} \bullet s_{3,c})$$

Figures from the course of J.M. Dutertre written in 2011.

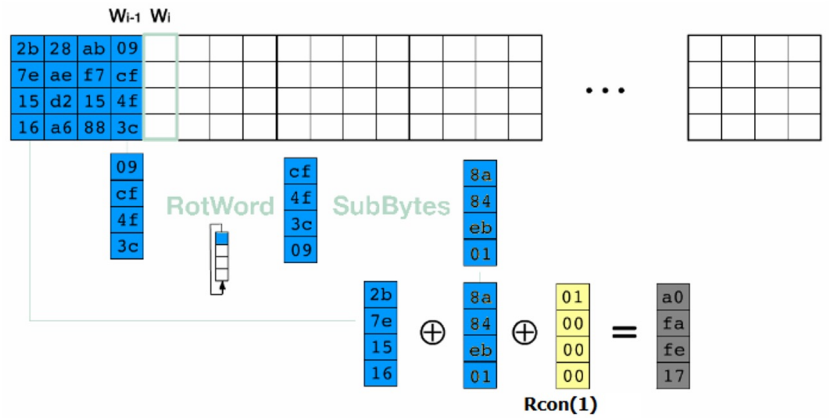
AES - AddRoundKey



Figures from the course of J.M. Dutertre written in 2011.

AES - KeyExpansion - First column

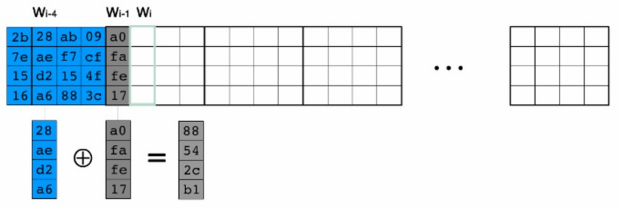
01	02	04	08	10	20	40	80	1b	36
00	00	00	00	00	00	00	00	00	00
00	00	00	00	00	00	00	00	00	00
00	00	00	00	00	00	00	00	00	00

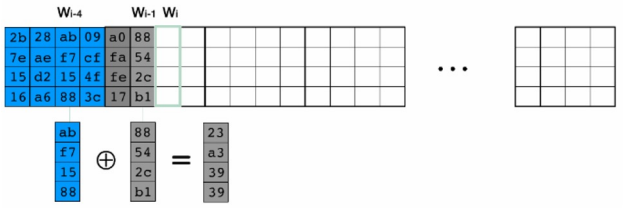


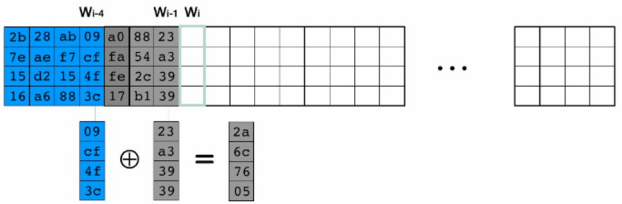
Figures from the course of J.M. Dutertre written in 2011.

AES - KeyExpansion - Other columns

01	02	04	08	10	20	40	80	1b	36
00	00	00	00	00	00	00	00	00	00
00	00	00	00	00	00	00	00	00	00
00	00	00	00	00	00	00	00	00	00

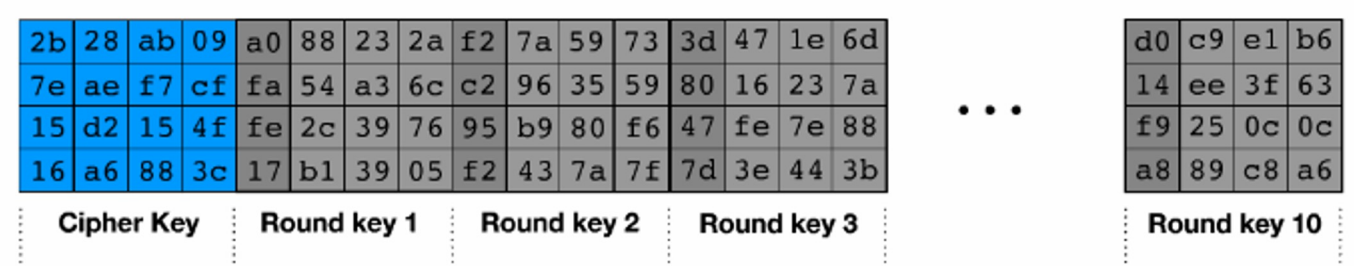






Figures from the course of J.M. Dutertre written in 2011.

AES - KeyExpansion - Result



Figures from the course of J.M. Dutertre written in 2011.

Bernstein Attack

Description (1)

- k[i] : i-th byte of the key
- n[i] : i-th byte of the plaintext
- K_i : integer between 0 and 255

Attack on the first round of the algorithm.

In the AES algorithm, we have to look up in a table.
Correlation between the execution time and the lookup time.
The Cache Hit and the Cache Miss appear during this lookup.

By having a knowing on the execution time and on n[i], we can discover k[i].

We are searching the value n[i] for which the execution time is the highest.

$$k[i] \oplus n[i] = K_i,\ K_i \epsilon [0, 255]$$

Description (2)

First step (study)

Execute the attack with a known key
Purpose : discover K_i

$$k[i] = n[i] \oplus K_i$$

Second step (attack)

With the same processor and the same software, execute the attack with an unknown key
Purpose : discover the key

Simple attack.

To avoid noise in the measurements, the victim sends us the execution time for each processed packet.

Bonneau Attacks

General description

- T_i[x] : value of the table i for x
- x_i^0 : result of (p_i XOR k_i)
- p_i : i-th byte of the plaintext

2 types of attack :

- On the first round
- On the last round

Attacks using also the mechanism of Cache Hit

We did not implement these attacks

First round attack (1)

Introduction :

Attack using the first round of AES
4 bytes using a table form a “family”
Example for T_0 :

$$x_0^0, x_4^0, x_8^0, x_{12}^0$$

Cache Hit arrives when :

$$p_i \oplus k_i = p_j \oplus k_j \Leftrightarrow p_i \oplus p_j = k_i \oplus k_j$$

The plaintexts satisfying the Cache Hit condition may have lower execution time

We will record the execution time in a table for all (i, j) composing a same family

Execution time table :

$$t[i, j, p_i \oplus p_j]$$

First round attack (2)

If an execution table is lower than the mean execution time, we will store :

$$t[i, j, \Delta]$$

We assume in this case that $k_i \oplus k_j = \Delta$

For each family, we obtain 6 equations.
Example for T_0 :

$$k_0 \oplus k_4 = \Delta_1$$
$$k_0 \oplus k_8 = \Delta_2$$
$$k_0 \oplus k_{12} = \Delta_3$$
$$k_4 \oplus k_8 = \Delta_4$$
$$k_4 \oplus k_{12} = \Delta_5$$
$$k_8 \oplus k_{12} = \Delta_6$$

These equations are the only information obtained by the attack.
We cannot discover some bits of the key.

The attack can only discover 60 bits

Not usable in a real case

Last round attack (1)

For the last round, there is no *MixColumns*. So we obtain :

$$C = \{T_4[x_0^{10}] \oplus k_0^{10}, T_4[x_5^{10}] \oplus k_1^{10}, T_4[x_{10}^{10}] \oplus k_2^{10}, T_4[x_{15}^{10}] \oplus k_3^{10},$$
$$T_4[x_4^{10}] \oplus k_4^{10}, T_4[x_9^{10}] \oplus k_5^{10}, T_4[x_{14}^{10}] \oplus k_6^{10}, T_4[x_3^{10}] \oplus k_7^{10},$$
$$T_4[x_8^{10}] \oplus k_8^{10}, T_4[x_{13}^{10}] \oplus k_9^{10}, T_4[x_2^{10}] \oplus k_{10}^{10}, T_4[x_7^{10}] \oplus k_{11}^{10},$$
$$T_4[x_{12}^{10}] \oplus k_{12}^{10}, T_4[x_1^{10}] \oplus k_{13}^{10}, T_4[x_6^{10}] \oplus k_{14}^{10}, T_4[x_{11}^{10}] \oplus k_{15}^{10}\}.$$

$$c_i = k_i^{10} \oplus T_4[x_u^{10}]$$
$$c_j = k_j^{10} \oplus T_4[x_w^{10}]$$

A Cache Hit arrives when :

$$x_u^{10} = x_w^{10} \Leftrightarrow T_4[x_u^{10}] = T_4[x_w^{10}] = \alpha$$

$$c_i = k_i^{10} \oplus \alpha$$
$$c_j = k_j^{10} \oplus \alpha$$

We will record the execution time in a table for all i, j and for all Δ

$$t[i, j, \Delta]$$

$$\Delta = c_i \oplus c_j$$

Last round attack (2)

We want to find for each (i, j) a value of Δ for which the execution time is significatively lower than the mean execution time.
In this case, we obtain :

$$\Delta_{i,j} = k_i^{10} \oplus k_j^{10}$$

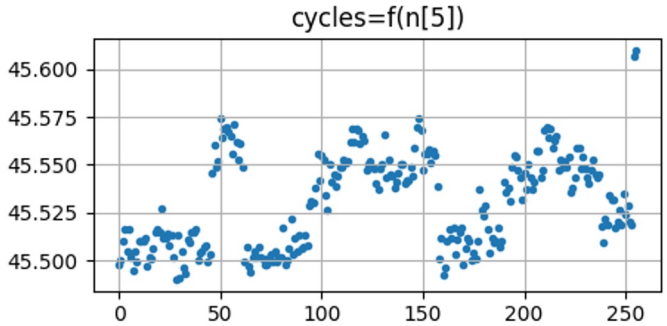
By solving these equations, we can obtain the last 16 bytes of the extended key. With this information, it is easy to recover the original key

These results were obtained:

CPU	L1 cache eviction	L2 cache eviction
Pentium III 1.0 GHz	2^{16}	2^{15}
Pentium IV Xeon 3.2 GHz	$2^{19.9}$	2^{16}
UltraSPARC-III+ 0.9 GHz	$2^{18.7}$	2^{15}

Implementation of Daniel J. Bernstein's Attack for AES-128

Main ideas

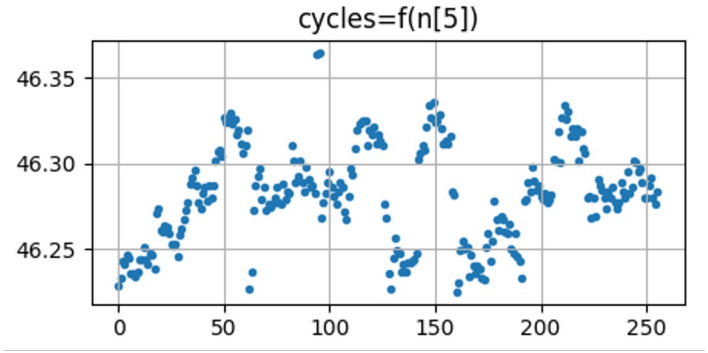


n: 16-byte plaintext
k: 16-byte key

Study (left graph): Max when n[5]=**255**

Attack (right graph):
We look which n[5] gives us the biggest computation time (for n[5]=**95** on right graph).

This n[5] should be equal to k[5] ⊕ 255, due to the nature of the AES implementation.
255 ⊕ 95 = **160 ← This is exactly the byte k[5] of the key used in the experiment !**

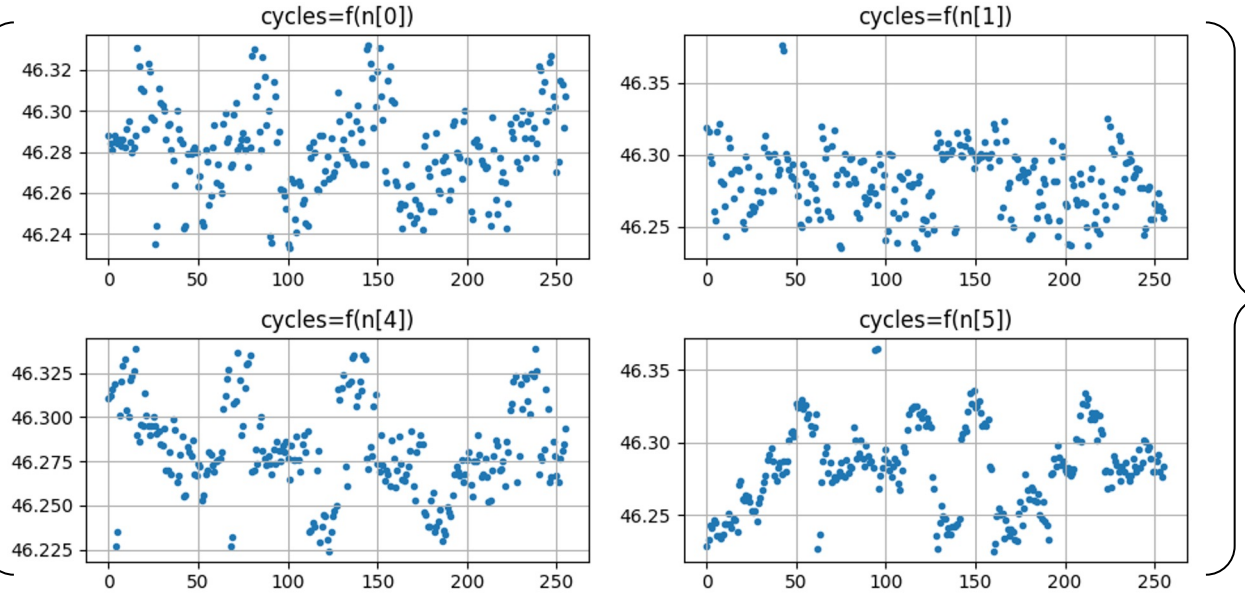


Source: Daniel J. Bernstein. Cache-timing attacks on AES. Department of Mathematics, Statistics, and Computer Science. The University of Illinois at Chicago. 2005.

Main ideas

What about the points more difficult to discern ?

Difficult



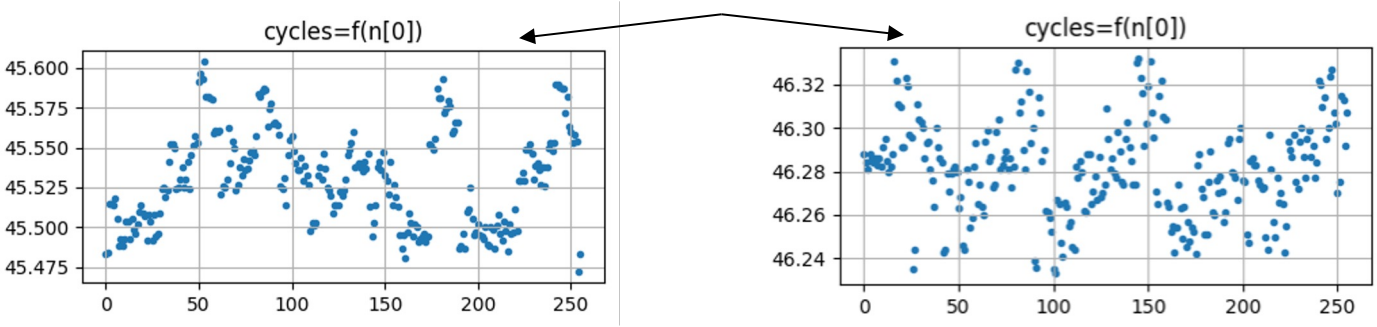
Easy

Source: Daniel J. Bernstein. Cache-timing attacks on AES. Department of Mathematics, Statistics, and Computer Science. The University of Illinois at Chicago. 2005.

Main ideas

How to take advantage of all the points obtained during the study and the attack phase ?

Study: Really different Attack:



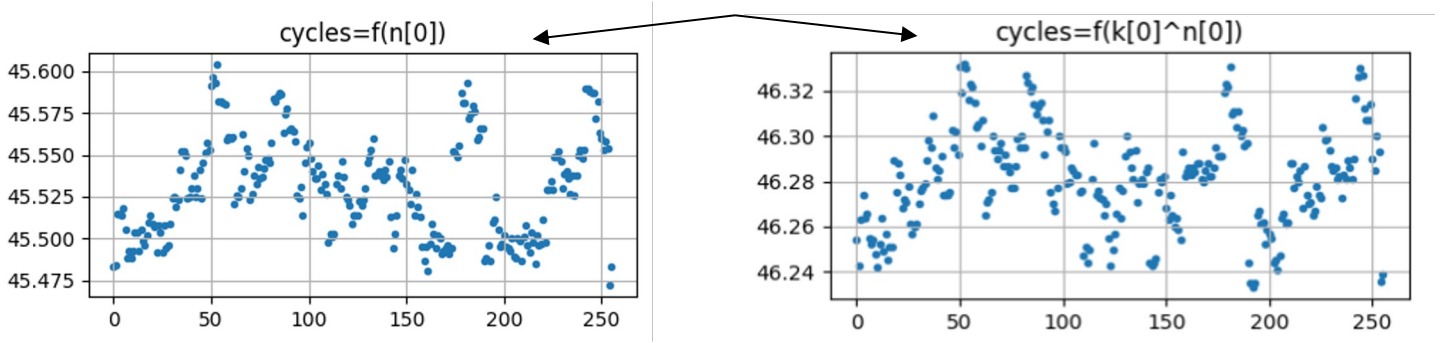
→ By using correlation over the whole range, from 0 to 255, instead of considering one point, taking advantage of the interdependence between all points.

Source: Daniel J. Bernstein. Cache-timing attacks on AES. Department of Mathematics, Statistics, and Computer Science. The University of Illinois at Chicago. 2005.

Main ideas

For the attack graph, let's replace n[0] with k[0]⊕n[0], with k[0] from the real private key.

Study: **Looks closer now !** Attack:

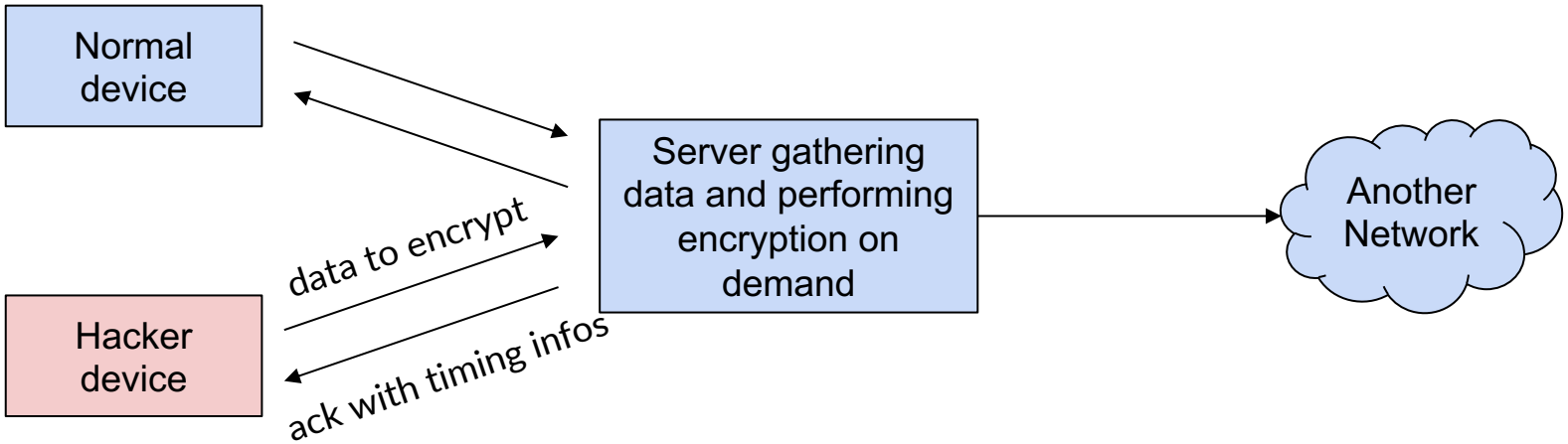


Without knowing the key, we must look for the k[0] giving us the best possible correlation ! We can then keep the best candidates and rank them by decreasing order of correlation.

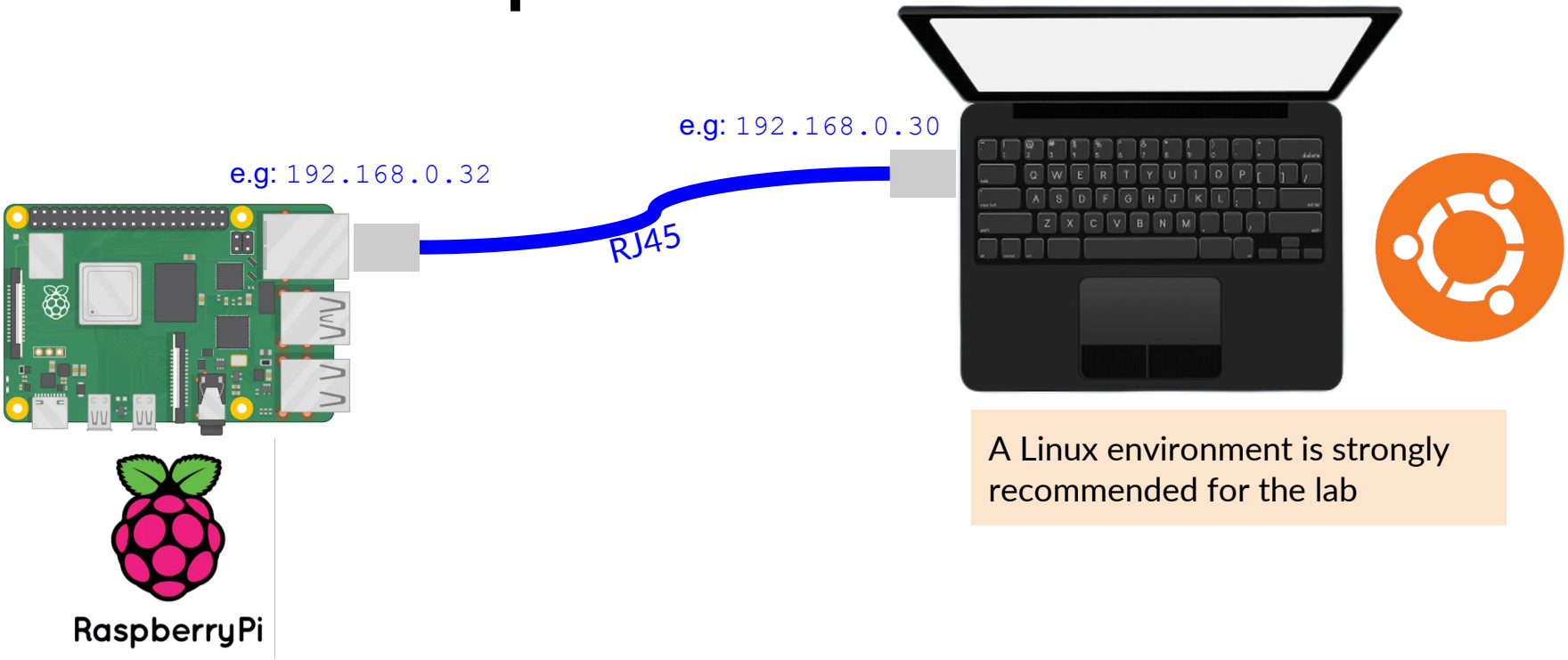
Source: Daniel J. Bernstein. Cache-timing attacks on AES. Department of Mathematics, Statistics, and Computer Science. The University of Illinois at Chicago. 2005.

Scenario

Example: A remote terminal unit has to get data from some devices, encrypt it and send it to another “master” network. Devices can know when their data has been received and sent. This allows the devices to know if they should send again their data.



Setup



A Linux environment is strongly recommended for the lab

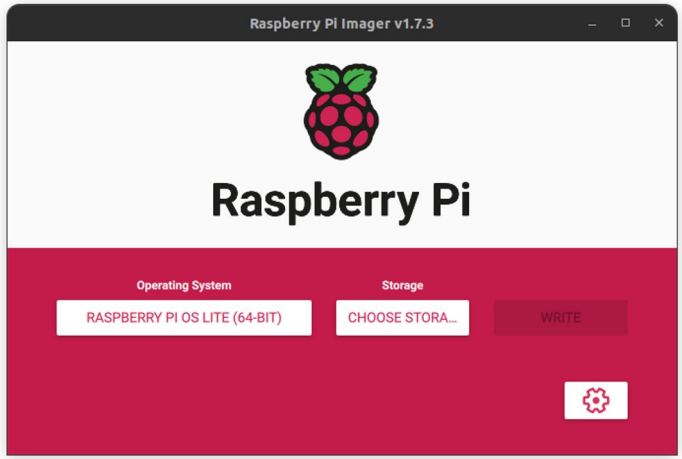
Preparation

Laptop

```
$ git clone https://github.com/japanninja74/ES_Security_aes-cache-timing-attack-pi4
```

Raspberry Pi 4

Set up a Raspberry Pi 4 with Raspberry Pi OS **64-bit**. Other versions of the board or OS can be used, but a few changes have to be done (see the README.md on GitHub for more details).



Implementation

Fundamental code comes from Daniel J. Bernstein's paper.
Some adaptations for the Raspberry Pi 4 and some helpful scripts were added.

Attacker

Client-side:

- **ciphertext.c**: gets one AES output for which we know the key
- **study.c**: sends a lot of random packets to encrypt to the victim and record elapsed time
- **correlate.c**: post-processing correlation to identify the best candidates for each key byte
- **search.c**: performs brute-force attack on correlation results

Victim

Server-side:

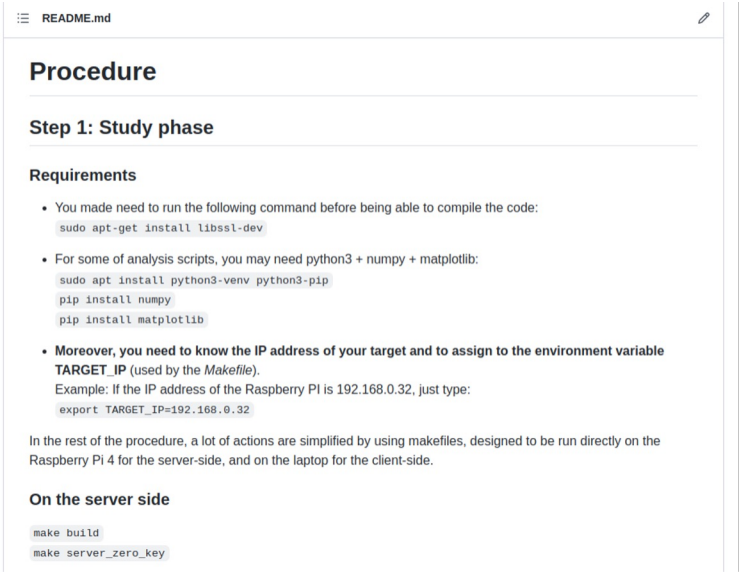
- **server.c**: performs "on-demand" encryption on received packets

Procedure

Main steps:

1. Study phase
2. Attack phase
3. Analysis
4. Brute force

Each step is described on the README.md on GitHub. The repository provides some useful scripts and shortcuts (through Makefile) to simplify the whole process.



README.md

Procedure

Step 1: Study phase

Requirements

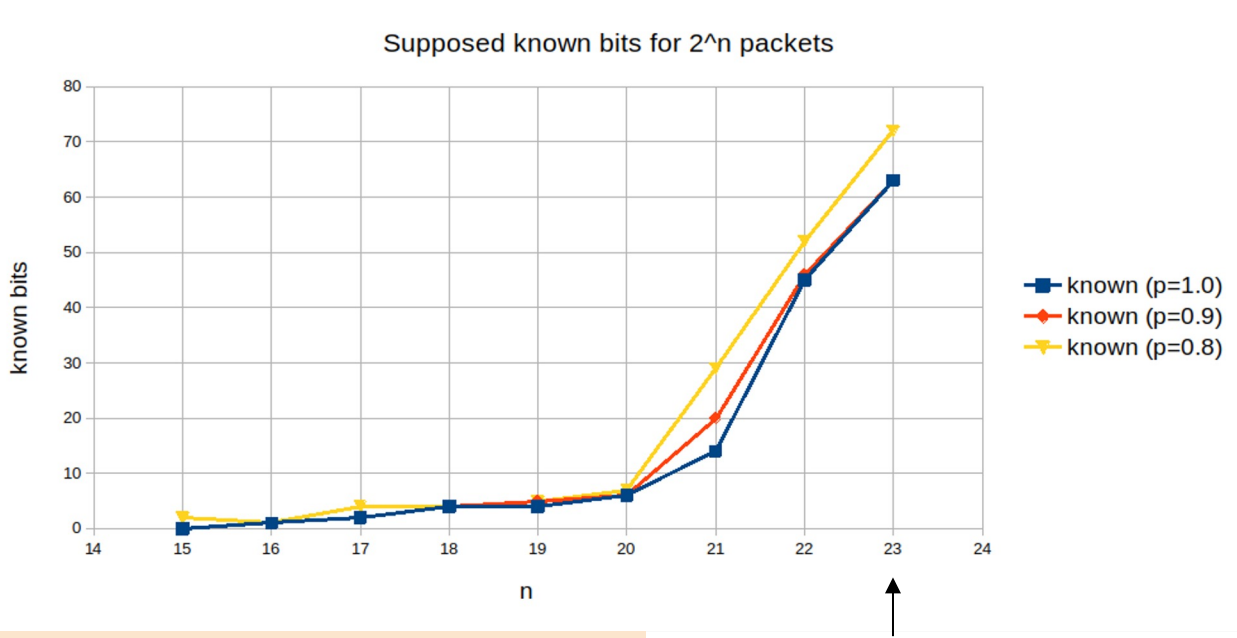
- You made need to run the following command before being able to compile the code:
  `sudo apt-get install libssl-dev`
- For some of analysis scripts, you may need python3 + numpy + matplotlib:
  `sudo apt install python3-venv python3-pip`
  `pip install numpy`
  `pip install matplotlib`
- **Moreover, you need to know the IP address of your target and to assign to the environment variable TARGET_IP** (used by the *Makefile*).
  Example: If the IP address of the Raspberry PI is 192.168.0.32, just type:
  `export TARGET_IP=192.168.0.32`

In the rest of the procedure, a lot of actions are simplified by using makefiles, designed to be run directly on the Raspberry Pi 4 for the server-side, and on the laptop for the client-side.

On the server side

`make build`
`make server_zero_key`

Results

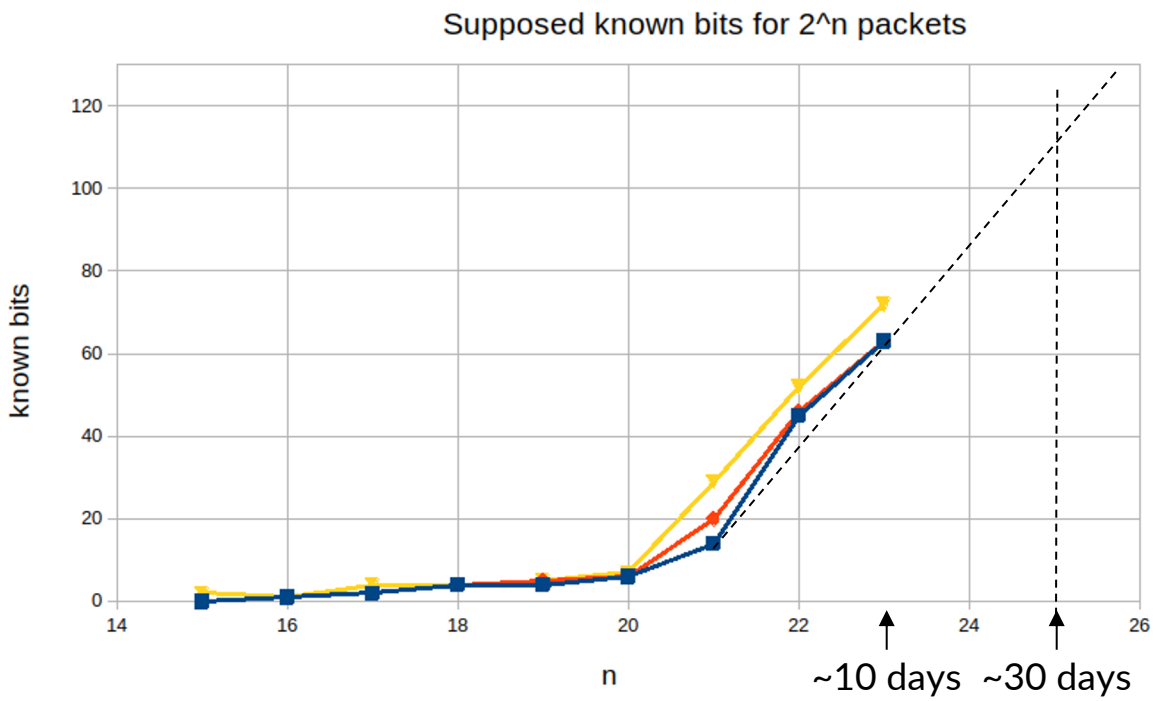


Example:

```
Key:   10100101 11010100 00011001 11111111 …
Guess: 1010010_ _1_1010_ ________ _111111_ …
```

It took about 10 days to collect 2^23 packets for the study AND attack phase on the Raspberry Pi 4

Extrapolation ?



In how many days would the number of candidates be reduced enough to make the key brute force feasible in less than an hour by an average laptop?

Here, a linear extrapolation is made ‘by hand’, just to have an idea of the order of magnitude

Potentials countermeasures

Potential countermeasures

- Disable cache
- Use an implementation aiming at maintaining constant timings
- Use a dedicated hardware cryptocore

Please attribute Creative Commons with a link to creativecommons.org