Embedded Systems Security
Hardware architecture/microarchitecture attacks

Prof. Stefano Di Carlo

# Introduction to AES cache-collision timing attacks

Lecture credits:
Gaspard Michel: gaspard.michel01@gmail.com
Thomas Rio: thomasrio0506@gmail.com
Marius Helin: hln.marius@gmail.com

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.

Highly based on Cache-timing attacks on AES by Daniel J. Bernstein.

Credits for the preparation of this material go to:

- Gaspard Michel: gaspard.michel01@gmail.com
- Thomas Rio: thomasrio0506@gmail.com
- Marius Helin: hln.marius@gmail.com



# Objectives

The goal of this lab is to understand how some software implementations of the AES encryption algorithm are vulnerable. More precisely: how to recover a 128-bit AES encryption key by measuring the elapsed time for encryption on many samples.

# Description of the attack

Consider:

- $k[i]$ : the i-th byte of the key
- $n[i]$ : the i-th byte of the input message
- $K_i$ : an integer between 0 and 255

Since AES is an algorithm with several loops, most attacks targeting it focus on one of these loops. In our case, we will use the first one. The mechanism, that will have an influence on the execution time and give us information about the key, is the memory search.

So, there is a correlation between the execution time of the whole algorithm and the time of this search in the table. So, by having information on the execution time and the knowledge of n[i], we can discover k[i]. We can do this for all the bytes of the key. In our case, we look for the value n[i] for which the encryption time is the highest.

Indeed, for the first round we have:

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

By repeating the same operation on the same processor and the same software with a known key, we can find the value of K_i. And so, we have for all k[i] composing the key:

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

Our experiment is quite simplified. To remove a maximum of random factors lengthening our experiment, we directly ask the server to provide us with the execution time of AES for a sent packet. We could have done an implementation taking the round-trip time of a packet, but this would have added several random factors. These uncertainties would have forced us to perform longer experiments to obtain the same results.

# Setup of the lab environment

## Required hardware.

- A computer.
- Raspberry Pi 4 with its micro-SD card (and a micro SD ↔ USB adapter if needed);
- A direct connection from the computer to the Raspberry Pi 4, through a RJ45 cable, is strongly recommended for low latency.

## Required software.

### On the embedded board

If it's not already done, you must install a Linux distribution on the Raspberry Pi. We strongly advise you to install Raspberry Pi OS 64-bit if you use a Raspberry Pi 4. Other versions of the board or OS can be used, but a few changes must be made (see the README.md on GitHub for more details, short explanation: the 'ready to use' code uses a specific ARM 64-bit register).

A simple and efficient solution for this step is to use the Pi Imager software directly. Many very well done tutorials exist on the internet, this one for example is convenient: https://projects.raspberrypi.org/en/projects/raspberry-pi-setting-up/2.

Once the OS of the board is installed and well configured, one must add the necessary code for the experiment. If the Pi is connected to the internet, a quick and easy way to do it is to use git and clone the repository associated with this lab.

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

You may need to run the following command before being able to compile the code:

```
$ sudo apt-get install libssl-dev
```

One last thing to do on the embedded board, before starting the lab, is to note the IPv4 address associated with the Ethernet port of the Raspberry Pi 4. Indeed, we will need to put the computer on the same network as the target. Just type:

```
$ ip address
```

And note the address and the mask associated with eth0 (example: 192.168.0.32/24).

### On the computer

The first thing we can do is to change the IP address associated with the Ethernet port of the computer.

On Linux, the following command may be suitable for instance:

```
$ sudo ip address add 192.168.0.30/24 dev eth0
```

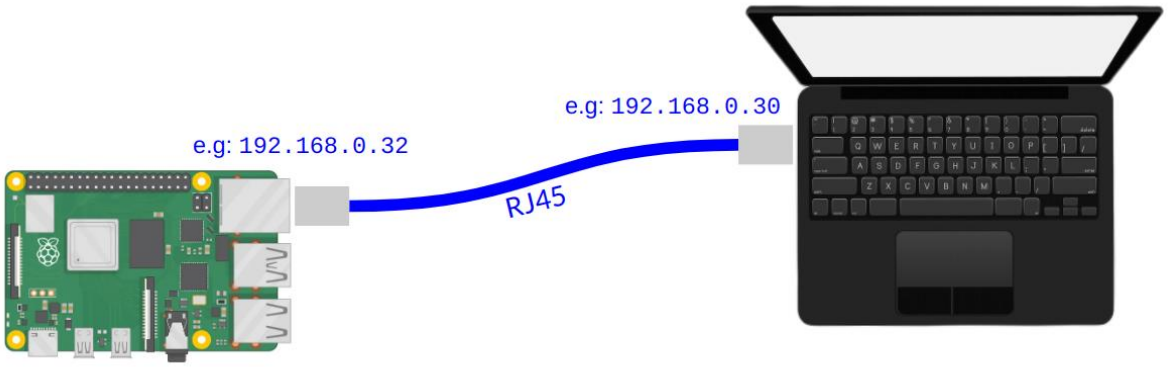
Depending on the interface's name and the network configuration. Moreover, you need to assign the environment variable TARGET_IP (used by the Makefile). Example: If the IP address of the Raspberry Pi is 192.168.0.32, just type:

```
$ export TARGET_IP=192.168.0.32
```

There's nothing else to do on the computer, just making sure the computer pings the target board.

## Wiring

We are almost ready to start the experience, we just have to connect the Raspberry Pi 4 to our computer!




# Preparation question

Question : Can you summarize, in a few words, the main principles of the attack created by Daniel J. Bernstein in his paper "Cache-timing attacks on AES"? (max. 5 lines)

## Implementation of the attack and exercises

Link of the GitHub repo to follow the instructions:
https://github.com/japanninja74/ES_Security_aes-cache-timing-attack-pi4

Q°1 (Step 1 on GitHub): Follow the procedure of step 1, until you get enough samples for the study phase. You would have to set up your environment, test the different commands from the included Makefile in the order shown in the README file. At the end of this step, you should plot the timing curves of your study phase.

Q°2 (Step 2 on GitHub): Now you can follow the instructions of step 2 from the README file. It shows how to proceed for the attack. Collect about the same number of samples as in the previous step.

Q°3 (Step 3 on GitHub)

a) Run the correlation command to show all the possible remaining candidates to find the key and show the known bits with the appropriate command. It will display the bits where the probability is high enough to be almost sure of the value.
b) Explain how the brute force algorithm written by Daniel J. Bernstein is more advanced than the one we used in the github repo.

# Students' self-evaluation

| | |
|---|---|
| Question 1: Study phase | …/1pt |
| Question 2: Attack phase | …/1pt |
| Question 3 a): Correlation and Known bits | …/1pt |
| Question 3 b): Brute force | …/2pt |
| total | …/5 |

# Appendix

## Appendix 1: AES summary

All the figures used in this appendix come from the course of J.M. Dutertre writen in 2011.

AES, for Advanced Encryption Standard, is a symmetric encryption algorithm defined as a standard. It is a widely used algorithm, considered secure. It works by encoding blocks of 128 bits and using keys of length 128, 192 or 256 bits.

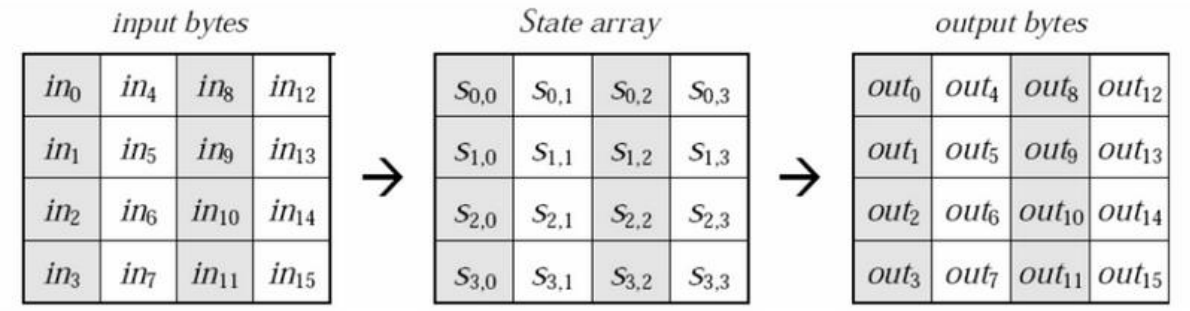
In our case, we will focus on AES-128, the version with a 128-bit key.

### Convention

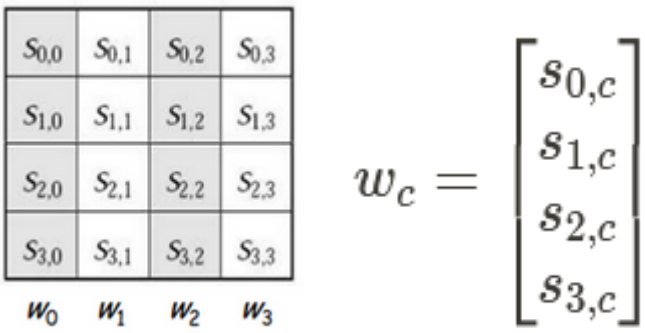
To best describe the operation of the algorithm, we define the following convention for the numbering of bits and bytes:

| Input bit sequence | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | … |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Byte number | 0 | | | | | | | | 1 | | | | | | | | 2 | | | | | | | | … |
| Bit numbers in byte | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | … |

We model the 16 bytes (128 bits) block by a 4x4 matrix as follows:



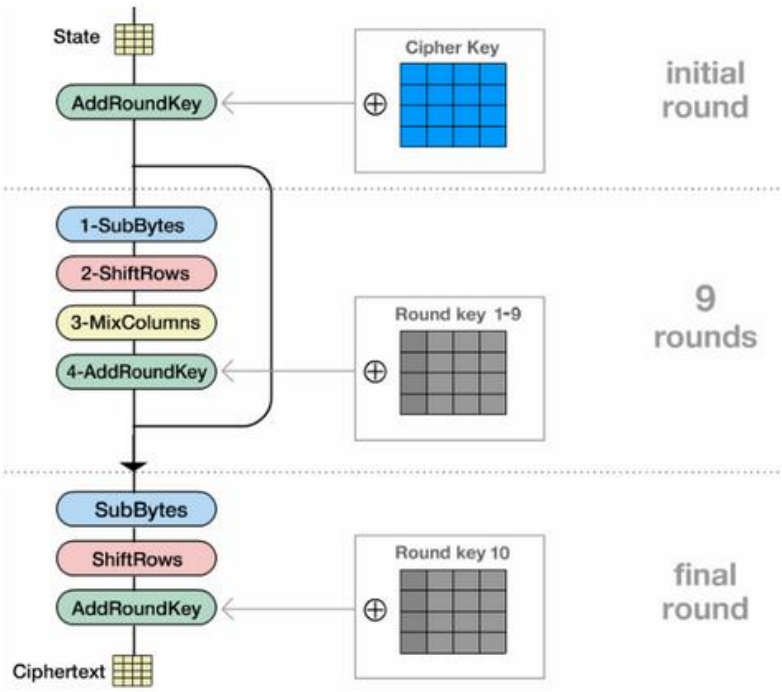

The matrix is also divided into 4 words of 4 bytes (32 bits) in the following way:




**General view**

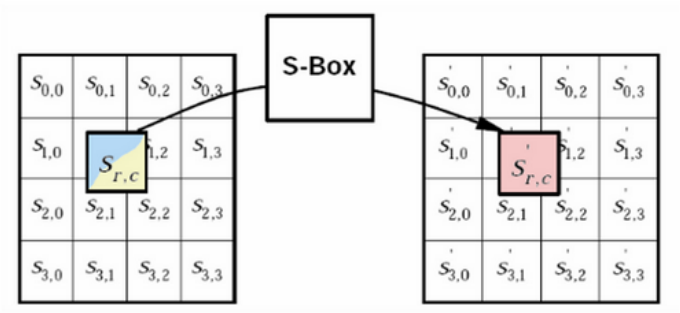
The algorithm takes as input the plaintext block of 16 bytes and a key of 16 bytes and outputs a ciphertext block of 16 bytes.

The algorithm starts with an initialization round with the input key, then continues with 9 standard rounds where it performs several mechanisms detailed later. Finally, it performs a last round very similar to the previous 9.




**SubByte**

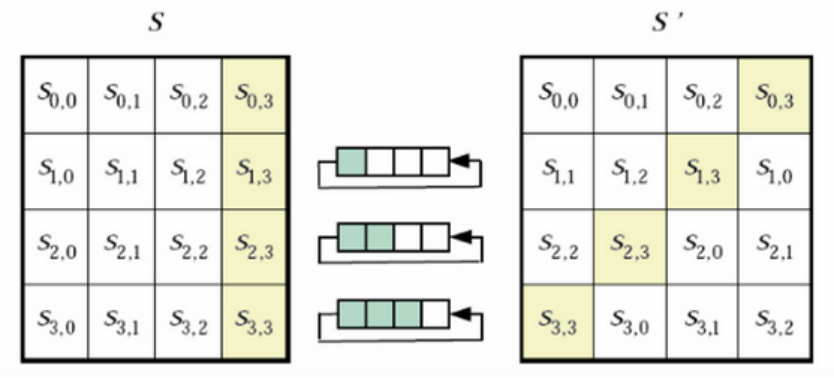
One byte is transformed into another from an array (substitution table, Sbox). It is a non-linear transformation applied byte to byte.




| 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 |

**ShiftRows**

A circular permutation of the bytes on a line is made according to the index of the line.




**MixColumns**

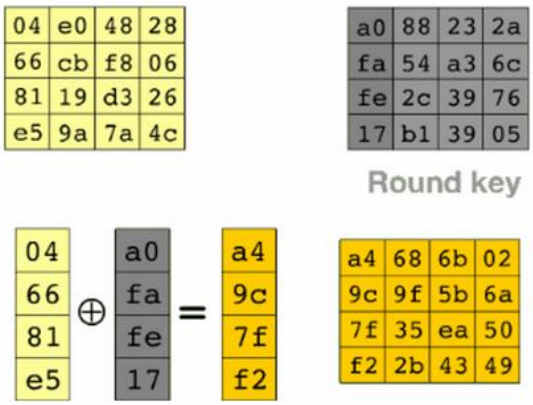
We perform a linear transformation on the columns. We consider the columns as polynomials, and we multiply them by $x^4 + 1$

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

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

**AddRoundKey**

Addition of the key to round to the considered state. An XOR is made between each of the bytes of the state and the round key.

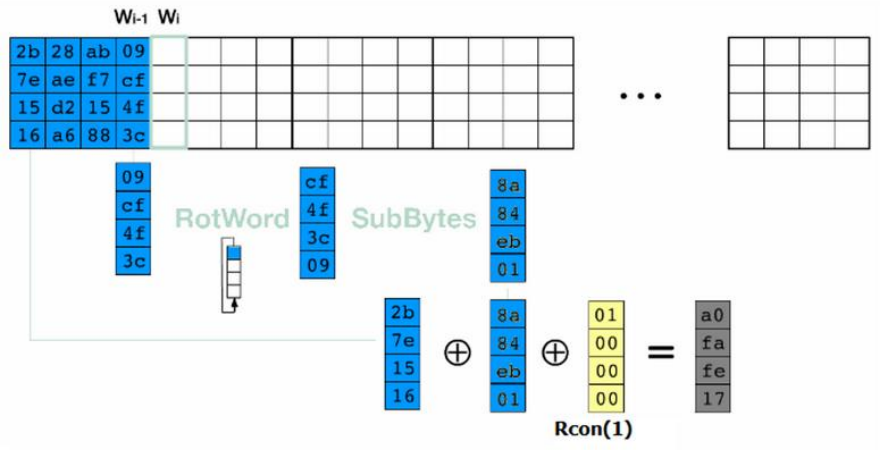



**Key Expansion**

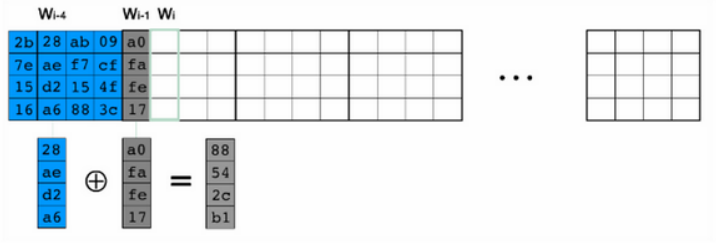
Rcon

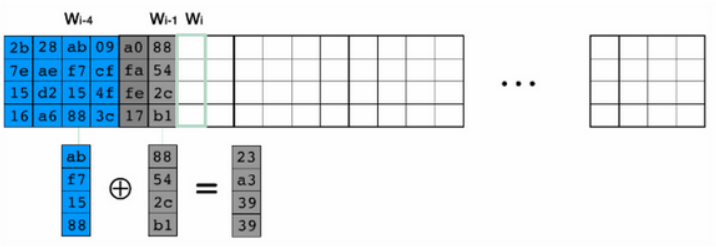
| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 01 | 02 | 04 | 08 | 10 | 20 | 40 | 80 | 1b | 36 |
| 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 |
| 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 |
| 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 |

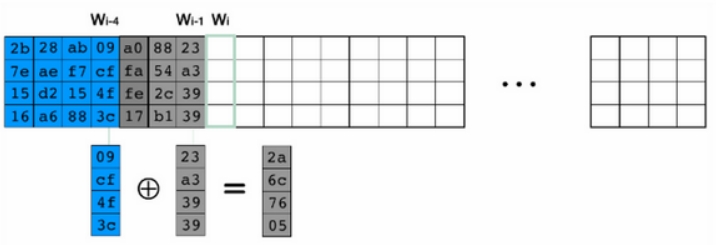
For the first column



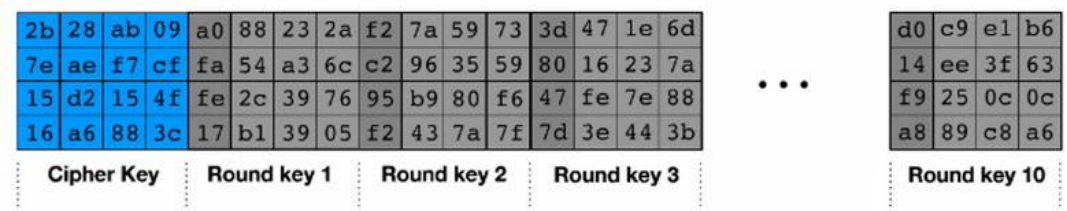

For the other columns




Result



## Appendix 2: Side-channel attack

A side-channel attack is a technique that allows an attacker to obtain sensitive information about a computer system or computational process by analyzing its external characteristics rather than attacking it directly. These external characteristics can include power consumption, the execution time of a process, the frequency of use of some instructions, electromagnetic emissions from a component, etc. Side-channel attacks are often used to bypass security measures put in place to protect sensitive data and can be very difficult to detect and counter.

In our case, we will try to attack the AES-128 encryption algorithm using the algorithm's runtime. There are many methods to attack AES-128 using a side-channel attack, most of them use the peculiarities of the cache memory to determine the key bits.

The key concept is the "cache hit". A cache hit occurs when a processor or other data processing element seeks to access data that is already in its cache, i.e., its temporary memory. This can happen when a processor is running a program and needs to access data that has already been used recently. Processors use their cache to speed up data access by temporarily storing the most frequently used data. When a processor needs to access a piece of data, it first checks if it's in its cache. If it is, that means that there is a cache hit and access to the data is much faster than if it had to be fetched from the main memory or the hard disk.

A cache hit will therefore modify the execution time of the program by shortening it. By studying enough execution of the program (and thus obtaining its execution time) we will be able to obtain information about the key to find it completely.

In the lab lecture, we present two attack methods to obtain an AES-128 key: Bernstein's attack and Bonneau's attack. We have only implemented the first attack and it is this one that will be used in the lab.

## Appendix 3: Bonneau's attack

We present here another form of attack using the same principle as Bernstein's attack. We have not implemented this attack. As for Bernstein's attack, the effect on the encryption time is done with a collision cache when reading the table.

Two types of attacks are presented in the paper.

We define:

- $T_i[x]$ : the value of the table i for the input x.

- $x_i^0$ : the result of $p_i \oplus k_i$
- $p_i$ : the plaintext at index ii

## First Round attack

This attack uses the first loop of AES.

In this first round, the bytes $x_0^0, x_4^0, x_8^0, x_{12}^0$ are used as entries for the table T_0. They form a "family" of bytes. There is also a family for T_1, T_2, T_3.

A cache hit occurs when 2 bytes of the same family are equal. This happens when:

- $p_i \oplus k_i = p_j \oplus k_j$
- Or $p_i \oplus p_j = k_i \oplus k_j$

The plaintexts satisfying this condition should have lower encryption times. The attack will therefore store information about the average execution time in the form of an array $t[i, j, p_i \oplus p_j]$ for all i, j of the same family.

If a low average encryption time occurs for $t[i, j, \Delta]$, the algorithm estimates that $k_i \oplus k_j = \Delta$. To determine these values, we perform a statistical test.

For each family, we obtain 6 equations. For T_0 we have:

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

These equations are the only information we get with this attack. We cannot discover the $log_2\ \delta$ weakest bits of each byte of the key. The attacker must therefore guess a complete byte of each family plus the $log_2\ \delta$ smallest bits. For $\delta = 8$ we obtain $4 \times (8 + 3.log_2\ \delta) = 68$.

This attack cannot be used in a real case.

## Final Round attack

In this attack, we use the last round of encryption. Since this last round does not have the *MixColumns* operation, we obtain:

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

So we have:

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

When $x_u^{10} = x_w^{10}$ a cache hit occurs during the search for table $T_4$. We get $T_4[x_u^{10}] = T_4[x_w^{10}] = \alpha$. And so:

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

The goal of this attack is to obtain the encryption times for each value $\Delta = c_i \oplus c_j$ that we will store in an array $t[i, j, \Delta]$. The objective is to find a value $\Delta'_{i,j}$ for each i, j such that $t[i, j, \Delta'_{i,j}] < \bar{t}$ where $\bar{t}$ is the average of the encryption time of all ciphertexts. This time will be lower than the average when $\Delta_{i,j} = k_i^{10} \oplus k_j^{10}$.

To find the last bytes of the extended key, we use the brute force technique on 256 values from the found equations.

From these values, we obtain the last 16 bytes of the extended key. From this value we can easily find the original key.

We obtain the following results:

| 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 | $2^{18.7}$ | $2^{15}$ |

# Bibliography

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

Robust Final-Round Cache-Trace Attacks Against AES. **Joseph Bonneau**. Computer Science Department. Stanford University. 2006.