



Norwegian University of
Science and Technology

SIDE-CHANNEL ATTACKS 2: SYMMETRIC KEY CRYPTO

TTM4205 – Lecture 11

Tjerand Silde

18.09.2026

Contents

Introduction

Recall: AES

Power Analysis

Trace Alignment

Cache Attacks

Countermeasures

Summary

Contents

Introduction

Recall: AES

Power Analysis

Trace Alignment

Cache Attacks

Countermeasures

Summary

Reference Material

These slides are based on:

- ▶ The referred papers in the slides
- ▶ JPA: parts of chapter 4
- ▶ DW: parts of chapters 4 and 13
- ▶ HHH: parts of chapters 8 to 12

Where This Fits

- ▶ **Lecture 9** said *what* leaks and named the attack families
- ▶ **Lecture 10** set up the hardware that lets you measure it
- ▶ **Today** we attack AES: power analysis, cache attacks, and the defenses
- ▶ **Lecture 12** does the same for RSA and elliptic curves



Black Box Crypto

We design the security of a cryptographic scheme to follow Kerckhoffs's principle: if everything about the scheme, except for the key, is known, then the scheme should be secure.

We analyze the scheme mathematically as black-box algorithms that take some (public or secret) input and give some (public or secret) output, and prove it secure concerning the algorithm description and the public data.

However, security depends on your model. In practice, it matters how these algorithms are implemented and what kind of information the *physical* system leaks about the inner workings of the algorithm computing on secret data.



Leakage

- ▶ The time it takes to compute...
- ▶ The power usage while computing...
- ▶ The electromagnetic radiation...
- ▶ The temperature variation...
- ▶ The memory pattern accessed...
- ▶ The sounds your laptop makes...
- ▶ The clock frequency the processor picks...

Attack Categories

- ▶ Remote vs physical attacks
- ▶ Software vs hardware attacks
- ▶ Passive vs active attacks
- ▶ Invasive vs non-invasive attacks

Contents

Introduction

Recall: AES

Power Analysis

Trace Alignment

Cache Attacks

Countermeasures

Summary

Recall: AES

- ▶ AES is a symmetric key encryption scheme
- ▶ AES is a substitution-permutation network
- ▶ AES-128: uses 10 rounds and 128-bit keys
- ▶ Works on a 4×4 column order array of 16 bytes
- ▶ Long messages are divided into 16 byte blocks
- ▶ Some modes of operation: ECB, CTR, GCM, etc.

The State

first column = first 4 bytes

$a_{0,0}$	$a_{0,1}$	$a_{0,2}$	$a_{0,3}$
$a_{1,0}$	$a_{1,1}$	$a_{1,2}$	$a_{1,3}$
$a_{2,0}$	$a_{2,1}$	$a_{2,2}$	$a_{2,3}$
$a_{3,0}$	$a_{3,1}$	$a_{3,2}$	$a_{3,3}$

Figure: The 16 input bytes fill the state *column by column*:
byte i goes to row $i \bmod 4$, column $\lfloor i/4 \rfloor$.

AES-128 Round Structure

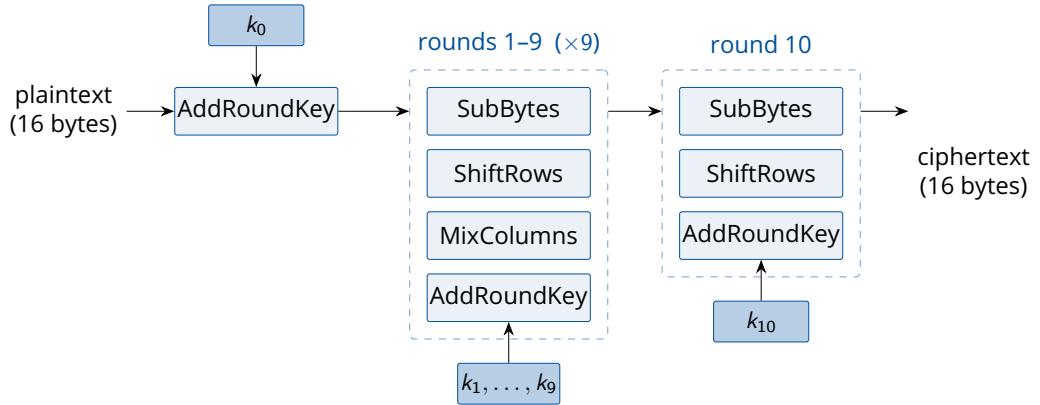


Figure: AES-128 high-level structure.

The Key Schedule

AES-128 needs *eleven* round keys of 16 bytes, so 176 bytes in total, all derived from the one 16-byte key. The schedule works on 4-byte words

$W_0, W_1, \dots, W_{43}$:

$$W_i = \begin{cases} \text{the } i\text{-th word of the key,} & 0 \leq i < 4, \\ W_{i-4} \oplus g(W_{i-1}), & i \equiv 0 \pmod{4}, \\ W_{i-4} \oplus W_{i-1}, & \text{otherwise,} \end{cases}$$

Here g rotates the four bytes left by one, applies the S-box to each, and XORs $Rcon_j = (x^{j-1}, 0, 0, 0)$ over $\mathbb{F}_{2^8}$. Round key k_r is $W_{4r}, \dots, W_{4r+3}$.

Why this matters: the recursion is *invertible*, so any one full round key runs backwards to the master key. Recovering 16 bytes of a single round key recovers the whole key.

AddRoundKey

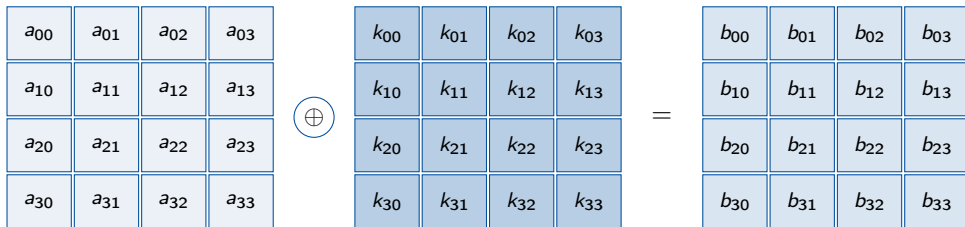


Figure: $b_{r,c} = a_{r,c} \oplus k_{r,c}$: one XOR per byte, which is a linear function.

SubBytes

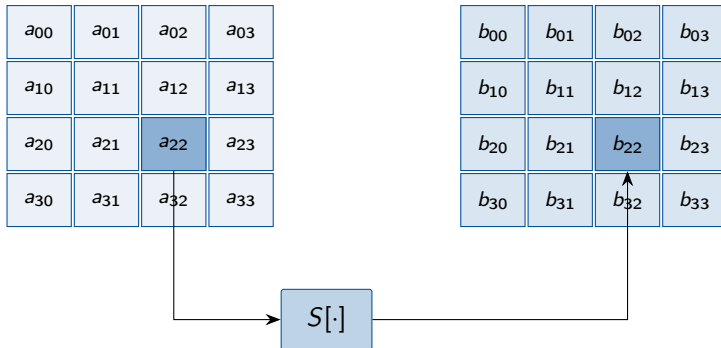


Figure: $b = S[a]$, byte by byte. The only non-linear step, where S can be represented as a look-up table. S can also be represented as an input-specific algebraic operation: computing the inverse in $\mathbb{F}_{2^8}$.

ShiftRows

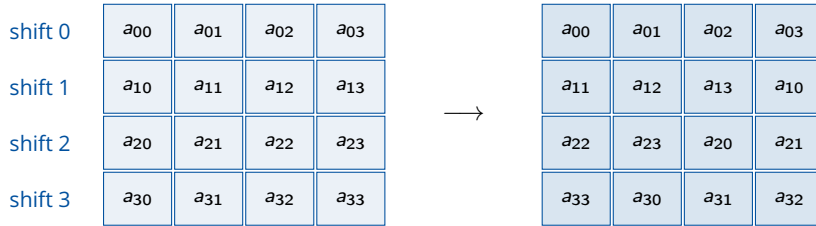


Figure: Row r is rotated left by r bytes based on location in the matrix.

MixColumns

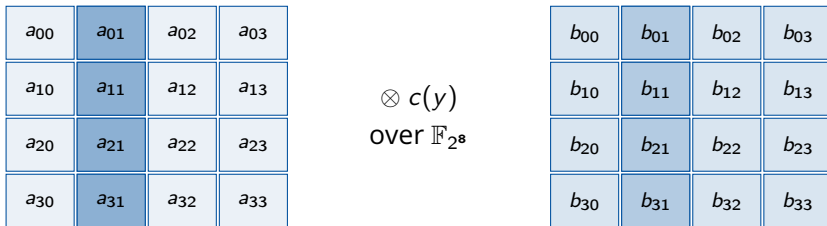


Figure: Each column is multiplied by a fixed polynomial over $\mathbb{F}_{2^8}$.

MixColumns: Which Polynomial?

There are *two* levels of polynomial here, and it helps to keep them apart.

A **byte** is already a polynomial. We work in $\mathbb{F}_{2^8} = \mathbb{F}_2[x]/(x^8 + x^4 + x^3 + x + 1)$, so the byte 0x02 is the element x , and 0x03 is $x + 1$.

A **column** of four such bytes is then read as a polynomial of degree less than 4 in a *second* variable y , with coefficients in $\mathbb{F}_{2^8}$:

$$a(y) = a_3 y^3 + a_2 y^2 + a_1 y + a_0.$$

MixColumns multiplies that column by the *fixed* polynomial $c(y) = 03 y^3 + 01 y^2 + 01 y + 02$, reduced modulo $y^4 + 1$.

MixColumns: The Same Thing as a Matrix

Multiplying by $c(y)$ modulo $y^4 + 1$ is a circulant matrix over $\mathbb{F}_{2^8}$:

$$\begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{pmatrix} = \begin{pmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{pmatrix}$$

- ▶ Only 01, 02 and 03 occur, so no general multiplier is needed
- ▶ $02 \cdot a$ is `xtime`: shift left one bit, XOR 0x1B on overflow, and
 $03 \cdot a = \text{xtime}(a) \oplus a$
- ▶ $y^4 + 1 = (y + 1)^4$ over $\mathbb{F}_2$, so this is not a field, but $c(y)$ is coprime to $y^4 + 1$ and hence invertible: the step can be undone
- ▶ One changed input byte changes all four output bytes: the diffusion AES needs

Example Code

```
1  def encrypt(key, plaintext):
2
3      # AddRoundKey for initial round
4      ciphertext = AddRoundKey(plaintext, key[0])
5
6      for i in range(1, rounds):
7          ciphertext = SubBytes(ciphertext)
8          ciphertext = ShiftRows(ciphertext)
9          ciphertext = MixColumns(ciphertext)
10         ciphertext = AddRoundKey(ciphertext, key[i])
11
12         # Final round (no MixColumns)
13         ciphertext = SubBytes(ciphertext)
14         ciphertext = ShiftRows(ciphertext)
15         ciphertext = AddRoundKey(ciphertext, key[rounds])
16
17     return ciphertext
```

Example Code: The Subroutines

```
1 def AddRoundKey(self, state, key):
2     return np.bitwise_xor(state, key)
3
4 def SubBytes(self, state):
5     return self.S_box[state] # table lookup, secret index
6
7 def ShiftRows(self, state):
8     return state.take(
9         (0, 1, 2, 3, 5, 6, 7, 4, 10, 11, 8, 9, 15, 12, 13, 14)
10    ).reshape(4, 4)
```

Example Code: MixColumns

```
1  def MixColumns(state):
2
3      def single_col(col):
4          b = (col << 1) ^ (0x11B & -(col >> 7)) # xtime in F(2^8)
5          col_mixed = [
6              b[0] ^ col[3] ^ col[2] ^ b[1] ^ col[1],
7              b[1] ^ col[0] ^ col[3] ^ b[2] ^ col[2],
8              b[2] ^ col[1] ^ col[0] ^ b[3] ^ col[3],
9              b[3] ^ col[2] ^ col[1] ^ b[0] ^ col[0],
10             ]
11         return col_mixed
12
13     for c in range(4):
14         state[:, c] = single_col(state[:, c])
15     return state
```

Weaknesses and Defenses

We have now seen a common way to implement AES and its subroutines. For each algorithm, try to point out potential information leakage and protection.

Q: Which step would *you* attack, and why?

Contents

Introduction

Recall: AES

Power Analysis

Trace Alignment

Cache Attacks

Countermeasures

Summary

Differential Power Analysis

Paul Kocher, Joshua Jaffe, and Benjamin Jun

Cryptography Research, Inc.
~~607 Market Street, 5th Floor~~
~~San Francisco, CA 94105, USA~~
<http://www.cryptography.com>
~~E-mail: {paul,josh,ben}@cryptography.com~~

Abstract. Cryptosystem designers frequently assume that secrets will be manipulated in closed, reliable computing environments. Unfortunately, actual computers and microchips leak information about the operations they process. This paper examines specific methods for analyzing power consumption measurements to find secret keys from tamper resistant devices. We also discuss approaches for building cryptosystems that can operate securely in existing hardware that leaks information.

Keywords: differential power analysis, DPA, SPA, cryptanalysis, DES

Figure: <https://paulkocher.com/doc/DifferentialPowerAnalysis.pdf>



Simple Power Analysis (on DES)

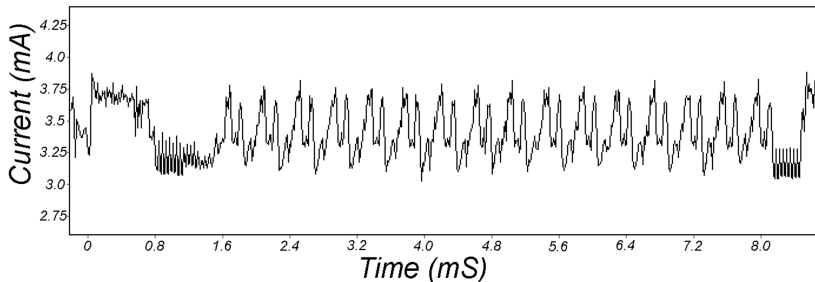


Figure 1: SPA trace showing an entire DES operation.

Figure: One DES operation. The 16 rounds are countable by eye.

<https://paulkocher.com/doc/DifferentialPowerAnalysis.pdf>

Detailed SPA (on DES)

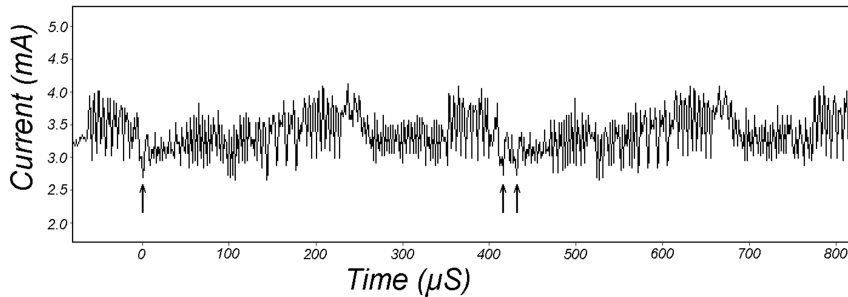


Figure 2: SPA trace showing DES rounds 2 and 3.

Figure: Zoomed to rounds 2 and 3. The arrows mark visible differences between the two. <https://paulkocher.com/doc/DifferentialPowerAnalysis.pdf>

Potential Weaknesses

Some information leaks directly:

- ▶ We can easily see how many rounds are computed
- ▶ We can easily see which operation is computed
- ▶ We can compare known traces with the first round

Based on this, we can compute the correlation between different keys, different messages, and their power consumption.

The Leakage Model

A gate burns power when a bit *changes*. So the power at a given instant grows with the number of ones being written, or with the number of bits that flipped.

We therefore model one sample of the trace as

$$P = \alpha \cdot \text{HW}(v) + \beta + \text{noise},$$

where $\text{HW}(v)$ is the Hamming weight of the intermediate value v .

We do not need α or β themselves. We only need the *ordering* to be preserved, because the attack looks for correlation, not for absolute values.

Divide and Conquer

Guessing the AES key means 2^{128} tries. Guessing *one byte* of it means 256. The attack rests on an intermediate value that depends on one key byte only.

That value is the first-round S-box output $v_j = S[p_j \oplus k_j]$, where p_j is a known plaintext byte and k_j the unknown key byte.

So 16 bytes $\times$ 256 candidates = 4096 hypotheses, instead of 2^{128} .

The S-box is the target on purpose: it is non-linear, so a wrong guess of k_j gives a prediction that is *uncorrelated*, not merely slightly off.

Correlation

Statistical analysis via the Pearson correlation coefficient ρ

Let X be the predicted leakage under one key hypothesis, and Y the measured samples at one point in time. Then

$$\rho = \frac{\text{Cov}(X, Y)}{\sqrt{\text{Var}(X) \cdot \text{Var}(Y)}} = \frac{\mathbb{E}[(X - \mu_X)(Y - \mu_Y)]}{\sigma_X \sigma_Y},$$

estimated from n traces as

$$r = \frac{\sum_i (x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_i (x_i - \bar{x})^2} \sqrt{\sum_i (y_i - \bar{y})^2}}.$$

The Attack in Four Lines

1. Capture n traces with known, random plaintexts
2. For each key byte j and each candidate $k \in \{0, \dots, 255\}$, predict $\text{HW}(S[p_j \oplus k])$ for every trace
3. Correlate that prediction against the measured value in every sample
4. The candidate with the largest $|r|$ anywhere in the trace is the key byte

Repeat for all 16 bytes. Each byte is independent, so this parallelizes perfectly.

Key Candidates

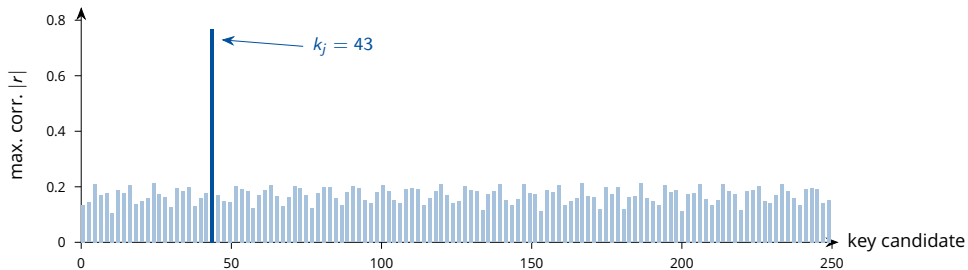


Figure: Maximum correlation over the whole trace, for each of the 256 candidates for one key byte. One candidate stands well clear of the noise floor.

DPA and CPA

DPA (Kocher, Jaffe, Jun, 1999) splits the traces into two piles according to one predicted *bit*, averages each pile, and subtracts. A wrong key guess splits the traces essentially at random, so the difference is flat.

CPA (Brier, Clavier, Olivier, 2004) uses the whole Hamming weight instead of one bit, and scores with the correlation coefficient above. It needs fewer traces and gives a cleaner peak.

Both are *non-profiled*: no training device, only the target. A *profiled* template attack first characterizes an identical device and can then succeed with a handful of traces.

How Many Traces?

Measurement noise dilutes the correlation. If the true correlation between prediction and leakage is ρ , the number of traces needed to see it above the noise is approximately

$$n \approx 3 + \frac{2z_{\alpha}^2}{\rho^2}.$$

Here z_{α} is the *standard normal quantile*: how many standard deviations you need for the confidence you want. 1.64 gives 95%, 2.33 gives 99%, 3.72 gives 99.99%.

At $z_{\alpha} = 3.72$: $\rho = 0.5$ needs about 110 traces, $\rho = 0.1$ about 2 800, and $\rho = 0.01$ about 280 000.

So n grows as $1/\rho^2$. **No countermeasure in this lecture removes the leak; they all shrink ρ , and the attacker pays for that quadratically.**

Signal, Noise, and What Countermeasures Buy

- ▶ Write the trace as signal plus noise; the useful quantity is $\text{SNR} = \text{Var}(\text{signal})/\text{Var}(\text{noise})$
- ▶ **Adding noise** (random delays, dummy operations, a noise generator) lowers the SNR by a constant factor, so n grows by the same constant
- ▶ **Masking** of order d forces the attacker to combine $d + 1$ points, and the correlation of that combination is *exponentially* smaller in d
- ▶ **Removing the dependency** (bitslicing, AES-NI) sets the signal to zero, which is the only qualitative win

We report success with a *success rate* (how often the right key ranks first) or *guessing entropy* (the average rank of the right key), both as functions of n .



Template Attacks

Everything so far assumed we know nothing about the device beyond a leakage model. If we own an *identical* device, we can do much better.

Profiling. For every value v of the target intermediate, run the copy many times and estimate a multivariate normal distribution $\mathcal{N}(\mu_v, \Sigma_v)$ over a handful of points of interest. That pair is the *template* for v .

Attack. Given a target trace t , return the v maximizing the likelihood $\Pr[t \mid \mu_v, \Sigma_v]$.

This is optimal in the information-theoretic sense, and it can succeed on a *single* trace, which matters when the key is used once. The price is a device you control and enough profiling traces.

Potential Weaknesses

Usually the correlation is not this clean, and we need to analyze each of the underlying subroutines in more detail:

- ▶ Computation after AddRoundKey might leak the Hamming weight
- ▶ SubBytes is a non-linear operation (inverses in $\mathbb{F}_{2^8}$)
- ▶ MixColumns is a polynomial/matrix multiplication
- ▶ Algebraic operations are computed over $\mathbb{F}_{2^8}$

And the traces must be *aligned* before any of this works, which is an important task during the SCA lab.

Contents

Introduction

Recall: AES

Power Analysis

Trace Alignment

Cache Attacks

Countermeasures

Summary

Why Alignment Matters

CPA correlates *sample index by sample index*: it compares the prediction against sample j of every trace. That only makes sense if sample j is the same operation in every trace.

If the traces are shifted relative to each other, the target operation lands on a different index each time, the correlation at every index averages over unrelated activity, and the attack simply fails.

In the SCA Lab the firmware does this to you on purpose: the `ADD_JITTER` option inserts a loop that runs 0 to 15 times depending on the first plaintext byte, so the shift is both data-dependent and different in every trace.

Aligned and Misaligned

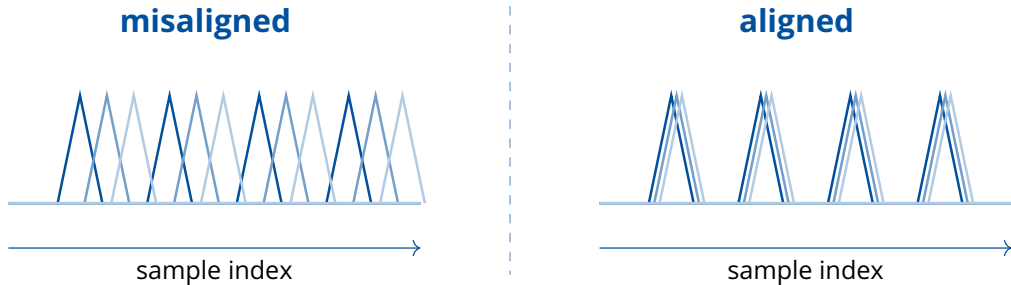


Figure: Three overlaid traces. On the left the same operation falls on a different index in each trace, so averaging across traces destroys it. On the right the peaks coincide and the statistic has something to find.

Sum of Absolute Differences

Pick a *reference* window r of length L from one trace, somewhere with a recognisable shape. For every other trace t , slide the window over a search range and pick the shift s minimizing

$$\text{SAD}(s) = \sum_{i=0}^{L-1} |t[i+s] - r[i]|.$$

Then cut every trace at its own best s . Cheap, easy to implement, and enough when the disturbance is a *constant offset*.

The Husky can even do this in hardware as a trigger, matching a 512-sample pattern, so the capture starts at the right place to begin with.

Dynamic Time Warping

SAD assumes the trace was *shifted*. Random delays and data-dependent loops *stretch* it: different parts move by different amounts, and no single s fixes all of them.

DTW instead finds a monotone, non-uniform matching between the two time axes, minimizing the total distance along the path, so it can compress one region and stretch another in the same trace.

Exact DTW costs $O(L^2)$; the lab uses an approximation with a *radius* r bounding how far the path may stray from the diagonal, giving roughly $O(L \cdot r)$.

Too slow? Fewer traces, smaller radius, or fewer samples via `scope.adc.samples`.

Contents

Introduction

Recall: AES

Power Analysis

Trace Alignment

Cache Attacks

Countermeasures

Summary

Sharing a Processor

Nothing so far needed physical access. Two processes on one machine already share caches, branch predictors, and execution ports, and the hardware makes all of them faster by *remembering what was used recently*.

Memory protection, sandboxing, and virtualization separate the *contents* of the two processes. They do not separate the *timing* of the shared hardware.

So an unprivileged process can watch which memory addresses a victim touches, without reading a single byte of the victim's memory.

Cache Attack Primitives

- ▶ **Evict+Time:** evict a cache line, run the victim, and see whether it got slower
- ▶ **Prime+Probe:** fill the cache with your own data, let the victim run, then time your own accesses to see which lines it evicted
- ▶ **Flush+Reload:** flush a line of *shared* memory, wait, then time reloading it; a fast reload means the victim used it

All three learn the same thing: the victim's memory *access pattern*. That is fatal for any implementation that indexes a table with a secret.

Flush+Reload

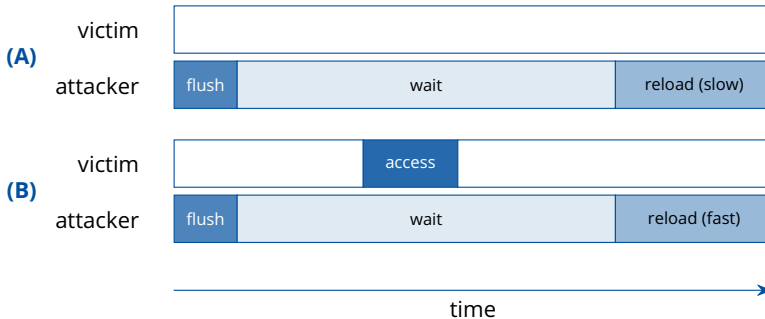


Figure: The attacker flushes a shared cache line, waits, and times the reload. A *fast* reload means the victim put the line back, so the attacker learns that the victim touched that address.

Cache Attacks and Countermeasures: the Case of AES

2005-08-14

Dag Arne Osvik¹, Adi Shamir² and Eran Tromer²

¹ `dag.arne@osvik.no`

² Department of Computer Science and Applied Mathematics,
Weizmann Institute of Science, Rehovot 76100, Israel
`{adi.shamir, eran.tromer}@weizmann.ac.il`

Figure: <https://eprint.iacr.org/2005/271.pdf>



The Case of AES

Fast software AES used to be table-driven: each round is a handful of lookups into precomputed T-tables, indexed by plaintext bytes XORed with key bytes. The index *is* the secret.

Osvik, Shamir, and Tromer recover the full key of Linux `dm-crypt` after 800 *writes to the partition, taking 65 milliseconds*, from an unprivileged process on the same CPU.

They also give a variant that needs neither plaintexts nor ciphertexts, only the effect of the victim on the cache.

A cache line holds several table entries, so one observation does not give the whole index: it gives its top bits. Enough observations pin down the key.

Cache-timing attacks on AES

Daniel J. Bernstein *

Department of Mathematics, Statistics, and Computer Science (M/C 249)
The University of Illinois at Chicago
Chicago, IL 60607-7045
djb@cr.yp.to

Figure: <https://cr.yp.to/antiforgery/cachetiming-20050414.pdf>

Potential Weaknesses

- ▶ NIST when standardizing SubBytes in AES:
“Table lookup: not vulnerable to timing attacks”
- ▶ Several finalists in the competition were secure, but Rijndael was fastest and this was important
- ▶ Flush+Reload attacks on the cache leak the secret indices of the SubBytes lookup table

The design was evaluated as an algorithm. The leak belongs to the implementation, and in 1999 nobody was scoring that.

Potential Defenses

We must ensure one of the following:

- ▶ Avoid memory access, or
- ▶ Always read all entries, or
- ▶ Disable cache-sharing

The last is impractical and affects general performance. Reading all entries costs one full table scan per lookup. That leaves *avoiding the table*, which is the next section.

Contents

Introduction

Recall: AES

Power Analysis

Trace Alignment

Cache Attacks

Countermeasures

Summary

A Fast New DES Implementation in Software

Eli Biham

Computer Science Department
Technion – Israel Institute of Technology
Haifa 32000, Israel

Email: biham@cs.technion.ac.il

WWW: <http://www.cs.technion.ac.il/~biham/>

Figure: https://link.springer.com/chapter/10.1007/3-540-68697-5_20



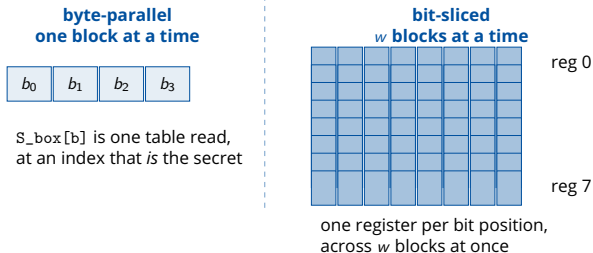
Bitslicing

Technique to avoid side-channel analysis:

- ▶ Work over bits, not bytes in $\mathbb{F}_{2^8}$
- ▶ Only use OR, AND, XOR, NAND, etc.
- ▶ Execute operations on vectors
- ▶ Is slower per block, but constant time
- ▶ Need a circuit for table lookup
- ▶ Integrated in hardware AES

We can combine this with randomized masking.

Bitslicing: The Idea



→ the S-box becomes a *circuit* of AND and XOR gates: no table, no branch, no secret address, and w blocks in the time of one

Figure: Transpose the data: each register holds one bit position across many blocks, so a table lookup becomes Boolean logic.

Bitslicing in Practice

The S-box circuit everyone uses comes from **Boyar and Peralta** (2009), who searched for a small AND/XOR circuit computing inversion in $\mathbb{F}_{2^8}$.

Käsper and Schwabe (CHES 2009) bitsliced AES into SSE2 registers: 128 bits per register means *eight* AES instances in parallel, giving constant-time AES-CTR and AES-GCM about twice as fast as the best table-based code.

So constant time is not automatically slower. It is slower only when the platform does not give you wide registers or AES instructions.

<https://eprint.iacr.org/2009/191> <https://eprint.iacr.org/2009/129>



One Library, Five AES Implementations

Implementation	Constant time?	CBC enc	CBC dec	CTR
big (table-based)	no	161.4	177.0	169.9
small (compact)	no	39.1	22.5	38.5
ct (32-bit bitslice)	yes	28.2	43.0	53.9
ct64 (64-bit bitslice)	yes	27.4	77.1	90.9
x86ni (AES-NI)	yes	679.8	2369.3	2420.7

BearSSL, megabytes per second, AES-128 on a 3.1 GHz Ivy Bridge. In CTR mode the portable constant-time code costs about $1.9\times$ the table-based code, while the hardware instructions are $14\times$ *faster* than it and constant time as well.

<https://www.bearssl.org/constanttime.html#aes>

Provably Secure Higher-Order Masking of AES[★]

Matthieu Rivain¹ and Emmanuel Prouff²

¹ CryptoExperts

`matthieu.rivain@cryptoexperts.com`

² Oberthur Technologies

`e.prouff@oberthur.com`

Figure: <https://eprint.iacr.org/2010/441.pdf>



AES Masking

- ▶ d -th order masking: split the secret into $d + 1$ shares
- ▶ linear operations are easy, non-linear ones are not
- ▶ AddRoundKey, ShiftRows and MixColumns are linear
- ▶ SubBytes is not linear: it requires extra work
- ▶ an attacker must now combine $d + 1$ points of the trace
- ▶ the number of traces needed grows exponentially in d

Masking a Linear Operation

Write the secret byte as $a = a_0 \oplus a_1 \oplus \dots \oplus a_d$, with $a_1, \dots, a_d$ drawn uniformly at random and a_0 chosen to make the sum come out right.

Any d of the shares are then uniformly distributed and independent of a : a first-order attacker, who sees one intermediate value at a time, learns nothing.

For a linear f this is free:

$$f(a) = f(a_0) \oplus f(a_1) \oplus \dots \oplus f(a_d),$$

so we simply run f on each share. ShiftRows and MixColumns cost nothing extra.

Masking AND (Ishai, Sahai, Wagner)

Let $c = a \cdot b$ for bits a and b , split as $(a_i)_{0 \leq i \leq d}$ and $(b_i)_{0 \leq i \leq d}$. Compute a $(d + 1)$ -tuple (c_i) with $\bigoplus_i c_i = c$:

1. For every $0 \leq i < j \leq d$, pick a random bit $r_{i,j}$
2. For every $0 \leq i < j \leq d$, set $r_{j,i} = (r_{i,j} \oplus a_i b_j) \oplus a_j b_i$
3. For every $0 \leq i \leq d$, set $c_i = a_i b_i \oplus \bigoplus_{j \neq i} r_{i,j}$

Masking AND: Completeness

$$\begin{aligned}\bigoplus_i c_i &= \bigoplus_i \left(a_i b_i \oplus \bigoplus_{j \neq i} r_{i,j} \right) = \bigoplus_i a_i b_i \oplus \bigoplus_{j > i} r_{i,j} \oplus \bigoplus_{j < i} (r_{j,i} \oplus a_i b_j \oplus a_j b_i) \\ &= \bigoplus_i a_i b_i \oplus \bigoplus_{j < i} (a_i b_j \oplus a_j b_i) = \left(\bigoplus_i a_i \right) \left(\bigoplus_i b_i \right).\end{aligned}$$

The $r_{i,j}$ cancel in pairs, and what remains is every cross term $a_i b_j$ exactly once.

The Price of Masking

Method	Reference	cycles	RAM (B)	ROM (B)
No masking	unprotected	$3 \cdot 10^3$	32	1150
First order, $d = 1$	Rivain-Prouff	$129 \cdot 10^3$	73	3153
Second order, $d = 2$	Rivain-Prouff	$271 \cdot 10^3$	79	3845
Third order, $d = 3$	Rivain-Prouff	$470 \cdot 10^3$	103	4648

One AES-128 block on an 8-bit AVR. First-order masking already costs about $43\times$ the running time, and the cost grows roughly as d^2 .

Figures from Table 2 of <https://eprint.iacr.org/2010/441.pdf>



Figure: <https://www.bearssl.org/constanttime.html#aes>

Contents

Introduction

Recall: AES

Power Analysis

Trace Alignment

Cache Attacks

Countermeasures

Summary

Summary

Protecting secret key computations is difficult. We need to:

- ▶ avoid lookup tables
- ▶ use constant time operations
- ▶ vectorize operations
- ▶ use randomness/masking

What to Remember

- ▶ The attack target is an intermediate value depending on *one* key byte
- ▶ A leakage model turns a key guess into a prediction you can correlate
- ▶ Non-linearity is what separates the right guess from the wrong ones
- ▶ A secret table index leaks through the cache, with no physical access at all
- ▶ Bitslicing and AES-NI remove the table; masking removes the correlation
- ▶ Every countermeasure costs, and masking costs roughly d^2

Next lecture: the same questions for RSA and elliptic curves, where the secret controls the *sequence* of operations rather than a table index.



Questions?