Implementation Security In Cryptography

Lecture 11: Entering the World of Attacks

Recap

- In the last lecture
  - Compact design of AES

Today

- Few more words about implementations
- Entering the world of attacks.

AES Once Again

- Well, we have seen how it is done in hardware..
- But what about software?
- A popular, extremely fast, yet terrible approach — T tables
  - Used quite a lot in OpenSSL
    - Not used anymore due to several **cache timing attacks**
- But table based secure implementations also do exist

AES T Tables

$$\begin{pmatrix} 2 & 1 & 1 & 3 \\ 3 & 2 & 1 & 1 \\ 1 & 3 & 2 & 1 \\ 1 & 1 & 3 & 2 \end{pmatrix} \begin{pmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \end{pmatrix} = \begin{pmatrix} s_0 \\ s_1 \\ s_2 \\ s_3 \end{pmatrix}$$

$$s_0 = 2c_0 \oplus 3c_1 \oplus 1c_2 \oplus 1c_3$$
$$s_1 = 1c_0 \oplus 2c_1 \oplus 3c_2 \oplus 1c_3$$
$$s_2 = 1c_0 \oplus 1c_1 \oplus 2c_2 \oplus 3c_3$$
$$s_3 = 3c_0 \oplus 1c_1 \oplus 1c_2 \oplus 2c_3$$

- Let us consider the MixColumns operation
- In software, each 32-bit AES column is a uint32 variable.

AES T Tables

$$\begin{pmatrix} 2 & 1 & 1 & 3 \\ 3 & 2 & 1 & 1 \\ 1 & 3 & 2 & 1 \\ 1 & 1 & 3 & 2 \end{pmatrix} \begin{pmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \end{pmatrix} = \begin{pmatrix} s_0 \\ s_1 \\ s_2 \\ s_3 \end{pmatrix}$$

- Let us consider the MixColumns operation
- In software, each 32-bit AES column is a uint32 variable.
- So we can store s in a uint32 variable.
- Here each s_i is 8-bit, so we can simply do concatenation to get 32-bits.

$$\begin{aligned} s_0 &= 2c_0 \oplus 3c_1 \oplus 1c_2 \oplus 1c_3 \\ s_1 &= 1c_0 \oplus 2c_1 \oplus 3c_2 \oplus 1c_3 \\ s_2 &= 1c_0 \oplus 1c_1 \oplus 2c_2 \oplus 3c_3 \\ s_3 &= 3c_0 \oplus 1c_1 \oplus 1c_2 \oplus 2c_3 \end{aligned}$$

```
s = s0 | s1 | s2 | s3
s = 2*c0 ^ 3*c1 ^ 1*c2 ^ 1*c3 |
    1*c0 ^ 2*c1 ^ 3*c2 ^ 1*c3 |
    1*c0 ^ 1*c1 ^ 2*c2 ^ 3*c3 |
    3*c0 ^ 1*c1 ^ 1*c2 ^ 2*c3
```

AES T Tables

$$\begin{pmatrix} 2 & 1 & 1 & 3 \\ 3 & 2 & 1 & 1 \\ 1 & 3 & 2 & 1 \\ 1 & 1 & 3 & 2 \end{pmatrix} \begin{pmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \end{pmatrix} = \begin{pmatrix} s_0 \\ s_1 \\ s_2 \\ s_3 \end{pmatrix}$$

```
s = s0 | s1 | s2 | s3
s = 2*c0 ^ 3*c1 ^ 1*c2 ^ 1*c3 |
    1*c0 ^ 2*c1 ^ 3*c2 ^ 1*c3 |
    1*c0 ^ 1*c1 ^ 2*c2 ^ 3*c3 |
    3*c0 ^ 1*c1 ^ 1*c2 ^ 2*c3
```

- Here each s_i is 8-bit, so we can simply do concatenation to get 32-bits.
- Now we can also rearrange the terms.
  - Why this is beneficial??

```
s = (2*c0 | 1*c0 | 1*c0 | 3*c0) ^
    (3*c1 | 2*c1 | 1*c1 | 1*c1) ^
    (1*c2 | 3*c2 | 2*c2 | 1*c2) ^
    (1*c3 | 1*c3 | 3*c3 | 2*c3)
```

AES T Tables

$$\begin{pmatrix} 2 & 1 & 1 & 3 \\ 3 & 2 & 1 & 1 \\ 1 & 3 & 2 & 1 \\ 1 & 1 & 3 & 2 \end{pmatrix} \begin{pmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \end{pmatrix} = \begin{pmatrix} s_0 \\ s_1 \\ s_2 \\ s_3 \end{pmatrix}$$

- Here each s_i is 8-bit, so we can simply do concatenation to get 32-bits.
- Now we can also rearrange the terms.
  - Why this is beneficial??
  - Observe that you can compute each term being XORed only from a c_i

```
s = s0 | s1 | s2 | s3

s = 2*c0 ^ 3*c1 ^ 1*c2 ^ 1*c3 |

    1*c0 ^ 2*c1 ^ 3*c2 ^ 1*c3 |

    1*c0 ^ 1*c1 ^ 2*c2 ^ 3*c3 |

    3*c0 ^ 1*c1 ^ 1*c2 ^ 2*c3
```

```
s = (2*c0 | 1*c0 | 1*c0 | 3*c0) ^

    (3*c1 | 2*c1 | 1*c1 | 1*c1) ^

    (1*c2 | 3*c2 | 2*c2 | 1*c2) ^

    (1*c3 | 1*c3 | 3*c3 | 2*c3)
```

AES T Tables

$$\begin{pmatrix} 2 & 1 & 1 & 3 \\ 3 & 2 & 1 & 1 \\ 1 & 3 & 2 & 1 \\ 1 & 1 & 3 & 2 \end{pmatrix} \begin{pmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \end{pmatrix} = \begin{pmatrix} s_0 \\ s_1 \\ s_2 \\ s_3 \end{pmatrix}$$

```
s = s0 | s1 | s2 | s3

s = 2*c0 ^ 3*c1 ^ 1*c2 ^ 1*c3 |

    1*c0 ^ 2*c1 ^ 3*c2 ^ 1*c3 |

    1*c0 ^ 1*c1 ^ 2*c2 ^ 3*c3 |

    3*c0 ^ 1*c1 ^ 1*c2 ^ 2*c3
```

```
s = (2*c0 | 1*c0 | 1*c0 | 3*c0) ^

    (3*c1 | 2*c1 | 1*c1 | 1*c1) ^

    (1*c2 | 3*c2 | 2*c2 | 1*c2) ^

    (1*c3 | 1*c3 | 3*c3 | 2*c3)
```

- Why this is beneficial??
  - Observe that you can compute each term being XORed only from a c_i
  - Since c_i is 8-bit, so we can compute all possible 256 values and store them in. A table.
  - Same can be done for all c_i
    - One table for each
  - **Catch**: Table lookup is much faster than a finite field operation.

AES T Tables

$$\begin{pmatrix} 2 & 1 & 1 & 3 \\ 3 & 2 & 1 & 1 \\ 1 & 3 & 2 & 1 \\ 1 & 1 & 3 & 2 \end{pmatrix} \begin{pmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \end{pmatrix} = \begin{pmatrix} s_0 \\ s_1 \\ s_2 \\ s_3 \end{pmatrix}$$

```
s = s0 | s1 | s2 | s3
s = 2*c0 ^ 3*c1 ^ 1*c2 ^ 1*c3 |
    1*c0 ^ 2*c1 ^ 3*c2 ^ 1*c3 |
    1*c0 ^ 1*c1 ^ 2*c2 ^ 3*c3 |
    3*c0 ^ 1*c1 ^ 1*c2 ^ 2*c3
```

- Why this is beneficial??
  - We denote these tables as te0, te1, te2, and te3.
  - The operation is as follows:

```
s = (2*c0 | 1*c0 | 1*c0 | 3*c0) ^
    (3*c1 | 2*c1 | 1*c1 | 1*c1) ^
    (1*c2 | 3*c2 | 2*c2 | 1*c2) ^
    (1*c3 | 1*c3 | 3*c3 | 2*c3)
```

```
te0[i] = 2*i | 1*i | 1*i | 3*i
te1[i] = 3*i | 2*i | 1*i | 1*i
te2[i] = 1*i | 3*i | 2*i | 1*i
te3[i] = 1*i | 1*i | 3*i | 2*i

s = te0[c0] ^ te1[c1] ^ te2[c2] ^ te3[c3]
```

It does not ends here…

$$\begin{pmatrix} 2 & 1 & 1 & 3 \\ 3 & 2 & 1 & 1 \\ 1 & 3 & 2 & 1 \\ 1 & 1 & 3 & 2 \end{pmatrix} \begin{pmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \end{pmatrix} = \begin{pmatrix} s_0 \\ s_1 \\ s_2 \\ s_3 \end{pmatrix}$$

```
s = s0 | s1 | s2 | s3

s = 2*c0 ^ 3*c1 ^ 1*c2 ^ 1*c3 |

    1*c0 ^ 2*c1 ^ 3*c2 ^ 1*c3 |

    1*c0 ^ 1*c1 ^ 2*c2 ^ 3*c3 |

    3*c0 ^ 1*c1 ^ 1*c2 ^ 2*c3
```

- Can we do better than this?
-

```
te0[i] = 2*i | 1*i | 1*i | 3*i
te1[i] = 3*i | 2*i | 1*i | 1*i
te2[i] = 1*i | 3*i | 2*i | 1*i
te3[i] = 1*i | 1*i | 3*i | 2*i


s = te0[c0] ^ te1[c1] ^ te2[c2] ^ te3[c3]
```

```
s = (2*c0 | 1*c0 | 1*c0 | 3*c0) ^
    (3*c1 | 2*c1 | 1*c1 | 1*c1) ^
    (1*c2 | 3*c2 | 2*c2 | 1*c2) ^
    (1*c3 | 1*c3 | 3*c3 | 2*c3)
```

It does not ends here...

```
s = (2*c0 | 1*c0 | 1*c0 | 3*c0) ^
    (3*c1 | 2*c1 | 1*c1 | 1*c1) ^
    (1*c2 | 3*c2 | 2*c2 | 1*c2) ^
    (1*c3 | 1*c3 | 3*c3 | 2*c3)
```

- Can we do better than this?
- Observe that: $c0|c1|c2|c3$ can be
  - `S[b0]|S[b5]|S[b10]|S[b15]` or, `S[b4]|S[b9]|S[b14]|S[b3]` or `S[b8]|S[b13]|S[b2]|S[b7]` `S[b12]|S[b1]|S[b6]|S[b11]`
  - So we can merge subtypes and shift rows in a table

```
R0 = (2*S[b0]  | 1*S[b0]  | 1*S[b0]  | 3*S[b0])  ^
     (3*S[b5]  | 2*S[b5]  | 1*S[b5]  | 1*S[b5])  ^
     (1*S[b10] | 3*S[b10] | 2*S[b10] | 1*S[b10]) ^
     (1*S[b15] | 1*S[b15] | 3*S[b15] | 2*S[b15])

R1 = (2*S[b4]  | 1*S[b4]  | 1*S[b4]  | 3*S[b4])  ^
     (3*S[b9]  | 2*S[b9]  | 1*S[b9]  | 1*S[b9])  ^
     (1*S[b14] | 3*S[b14] | 2*S[b14] | 1*S[b14]) ^
     (1*S[b3]  | 1*S[b3]  | 3*S[b3]  | 2*S[b3])

R2 = (2*S[b8]  | 1*S[b8]  | 1*S[b8]  | 3*S[b8])  ^
     (3*S[b13] | 2*S[b13] | 1*S[b13] | 1*S[b13]) ^
     (1*S[b2]  | 3*S[b2]  | 2*S[b2]  | 1*S[b2])  ^
     (1*S[b7]  | 1*S[b7]  | 3*S[b7]  | 2*S[b7])

R3 = (2*S[b12] | 1*S[b12] | 1*S[b12] | 3*S[b12]) ^
     (3*S[b1]  | 2*S[b1]  | 1*S[b1]  | 1*S[b1])  ^
     (1*S[b6]  | 3*S[b6]  | 2*S[b6]  | 1*S[b6])  ^
     (1*S[b11] | 1*S[b11] | 3*S[b11] | 2*S[b11])
```

It does not ends here…

- So finally

```
R0 = te0[b0]  ^ te1[b5]  ^ te2[b10] ^ te3[b15]
R1 = te0[b4]  ^ te1[b9]  ^ te2[b14] ^ te3[b3]
R2 = te0[b8]  ^ te1[b13] ^ te2[b2]  ^ te3[b7]
R3 = te0[b12] ^ te1[b1]  ^ te2[b6]  ^ te3[b11]
```

It doe

- So finall

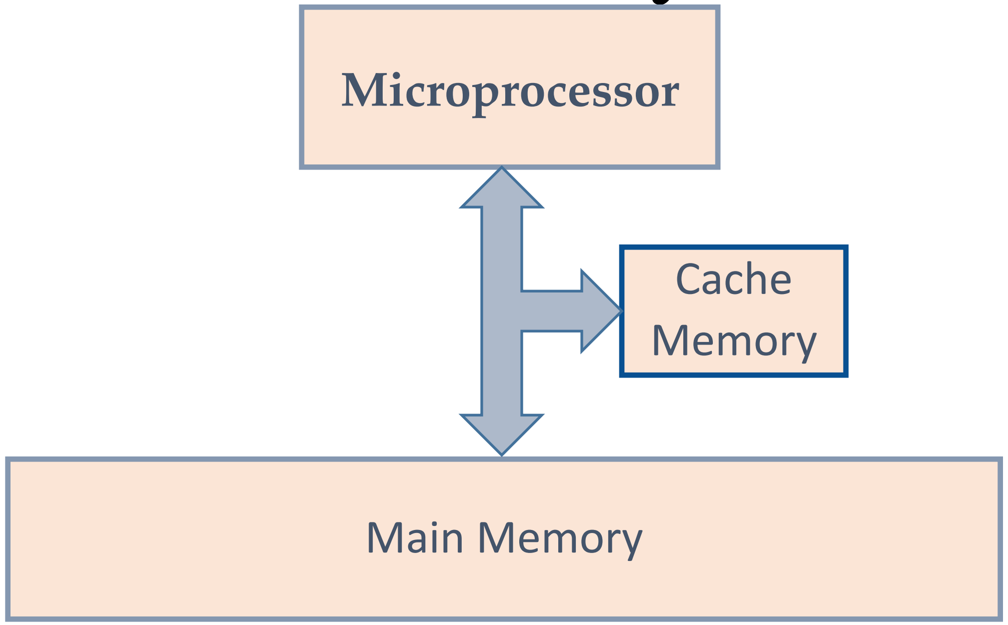
```
R0 = te0[b0]
R1 = te0[b4]
R2 = te0[b8]
R3 = te0[b12]
```

```
// Initialize the state, stored in s0, s1, s2 and s3
s0 := uint32(src[0])<<24 | uint32(src[1])<<16 | uint32(src[2])<<8 | uint32(src[3])
s1 := uint32(src[4])<<24 | uint32(src[5])<<16 | uint32(src[6])<<8 | uint32(src[7])
s2 := uint32(src[8])<<24 | uint32(src[9])<<16 | uint32(src[10])<<8 | uint32(src[11])
s3 := uint32(src[12])<<24 | uint32(src[13])<<16 | uint32(src[14])<<8 | uint32(src[15])

// Add the first round key to the state
s0 ^= xk[0]
s1 ^= xk[1]
s2 ^= xk[2]
s3 ^= xk[3]

for i:= 1; i < nr; i++ {
    // This performs SubBytes + ShiftRows + MixColumns + AddRoundKey
    tmp0 = te0[s0>>24] ^ te1[s1>>16&0xff] ^ te2[s2>>8&0xff] ^ te3[s3&0xff] ^ xk[4*i]
    tmp1 = te0[s1>>24] ^ te1[s2>>16&0xff] ^ te2[s3>>8&0xff] ^ te3[s0&0xff] ^ xk[4*i+1]
    tmp2 = te0[s2>>24] ^ te1[s3>>16&0xff] ^ te2[s0>>8&0xff] ^ te3[s1&0xff] ^ xk[4*i+2]
    tmp3 = te0[s3>>24] ^ te1[s0>>16&0xff] ^ te2[s1>>8&0xff] ^ te3[s2&0xff] ^ xk[4*i+3]

    s0, s1, s2, s3 = tmp0, tmp1, tmp2, tmp3
}
```

But As We Said…

- The tables are stored in cache memory of your system.
- AES accesses this table depending on the secret key values
- An adversary, who is able to measure the time for each encryption operation, and also using the same cache can do something so that it can recover the secret key!!!
- Such attacks are called cache timing attacks…

Attacks due to Memory Wall

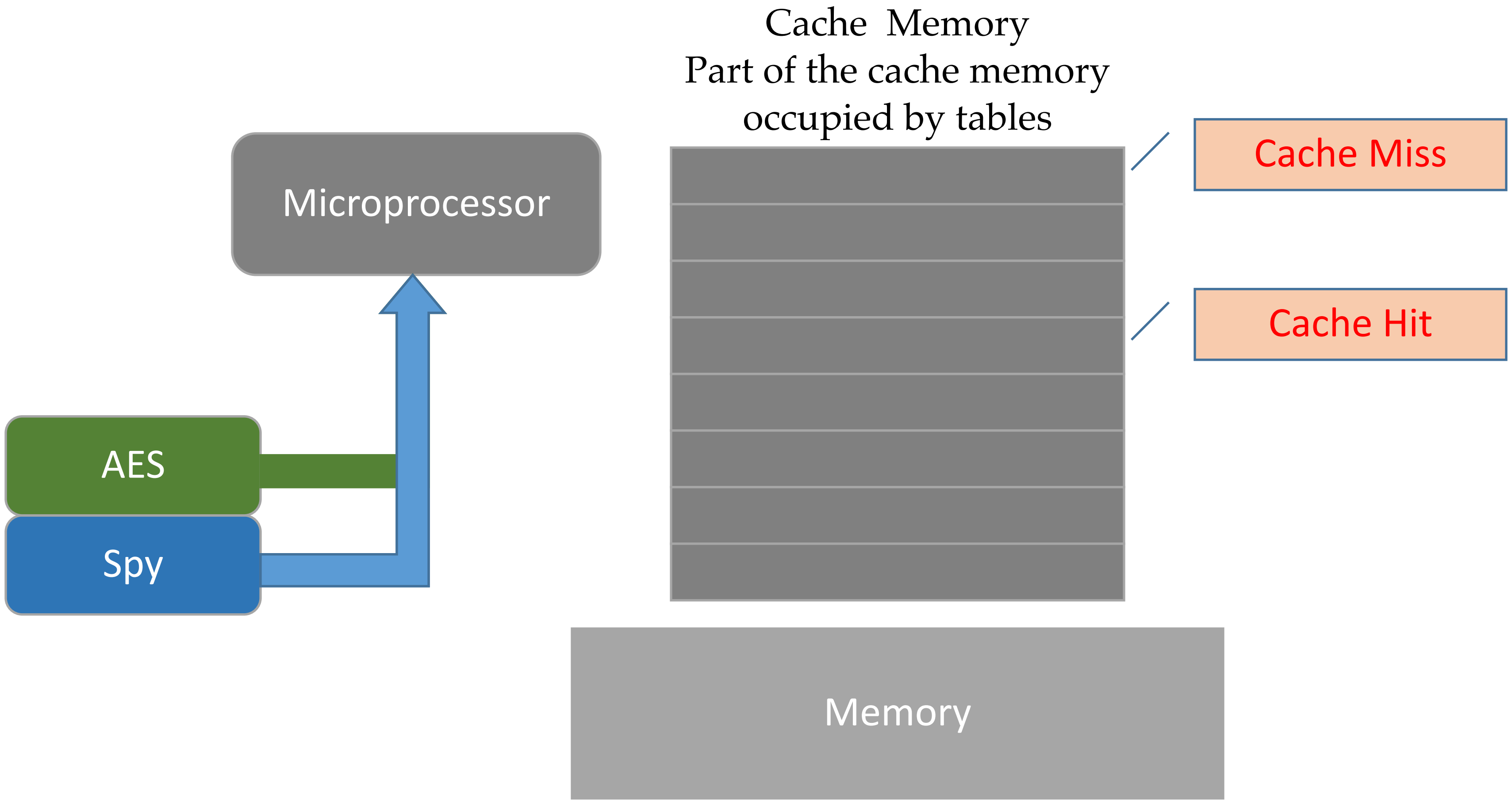


- If there is a *Cache Hit*
  - Access time is less
  - Power Consumption is less

- If there is a *Cache Miss*
  - Access time is more
  - Power Consumption is more

Timing Attacks due to Cache Memory

- Uses a spy program to determine cache behavior



Bitslicing

- Simply speaking, implement the software like hardware
- Results in constant-time crypto
- <u>Idea</u>: Let's say you are running on a 32-bit machine.
  - 32-bit registers
  - Logical AND, OR, NOT, XOR.
  - Also consider your block cipher in terms of these gates.

Bitslicing: A Simple Example

$$y_1 = x_1x_2x_4 + x_1x_3x_4 + x_1 + x_2x_3x_4 + x_2x_3 + x_3 + x_4 + 1$$

$$y_2 = x_1x_2x_4 + x_1x_3x_4 + x_1x_3 + x_1x_4 + x_1 + x_2 + x_3x_4 + 1$$

$$y_3 = x_1x_2x_4 + x_1x_2 + x_1x_3x_4 + x_1x_3 + x_1 + x_2x_3x_4 + x_3$$

$$y_4 = x_1 + x_2x_3 + x_2 + x_4$$

- Let us consider the first equation only.
  - It can be written as follows:

```
and t1, x1, x4
and t2, t1, x2
and t3, t1, x3
xor t4, t2, t3
```

And so on…

Bitslicing: A Simple Example

- Let us consider the first equation only.
  - It can be written as follows:

```
and t1, x1, x4
and t2, t1, x2
and t3, t1, x3
xor t4, t2, t3
```

And so on…

$$y_1 = x_1x_2x_4 + x_1x_3x_4$$
$$+ x_1 + x_2x_3x_4 + x_2x_3 + x_3 + x_4 + 1$$
$$y_2 = x_1x_2x_4 + x_1x_3x_4 + x_1x_3 + x_1x_4 +$$
$$x_1 + x_2 + x_3x_4 + 1$$
$$y_3 = x_1x_2x_4 + x_1x_2 + x_1x_3x_4 + x_1x_3 +$$
$$x_1 + x_2x_3x_4 + x_3$$
$$y_4 = x_1 + x_2x_3 + x_2 + x_4$$

- Now, each of t1, x1, x2,… are mapped to 32 bit registers; but actually they are processing 1-bit values
  - So, what to do?

Bitslicing: A Simple Example

- Let us consider the first equation only.
  - It can be written as follows:

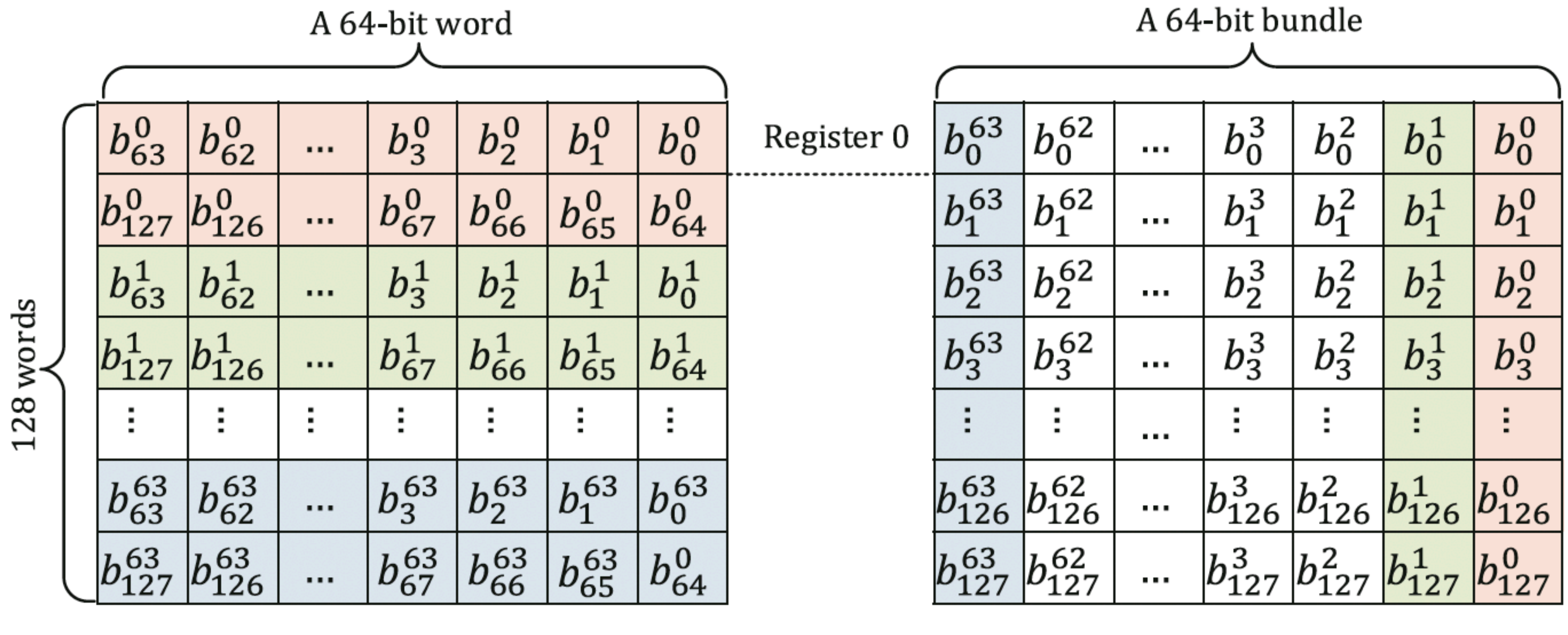
```
and t1, x1, x4
and t2, t1, x2
and t3, t1, x3
xor t4, t2, t3
```

And so on…

- Pack each register with independent values and process them using the same instruction.!!!
- Easiest case: you can encrypt 32 plaintext together

Bitslicing: A Simple Example

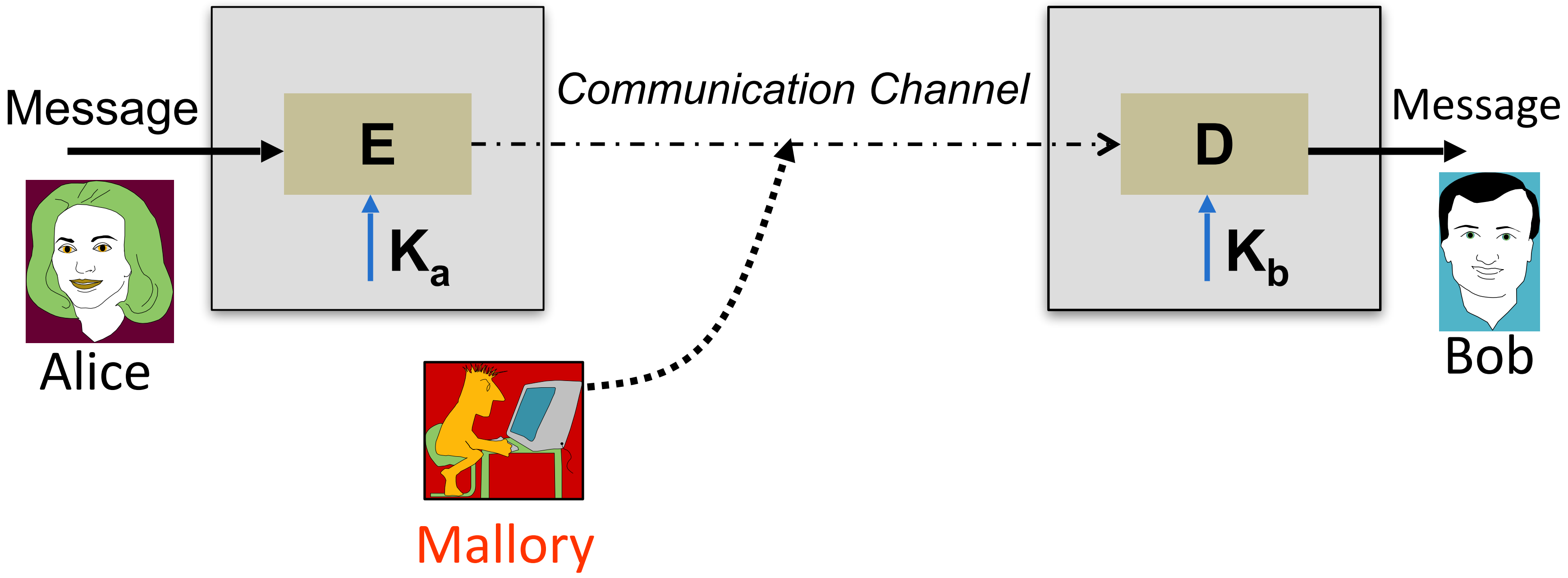
- Easiest case: you can encrypt 32 plaintext together
- You can parallelize S-Box computations for one plaintext



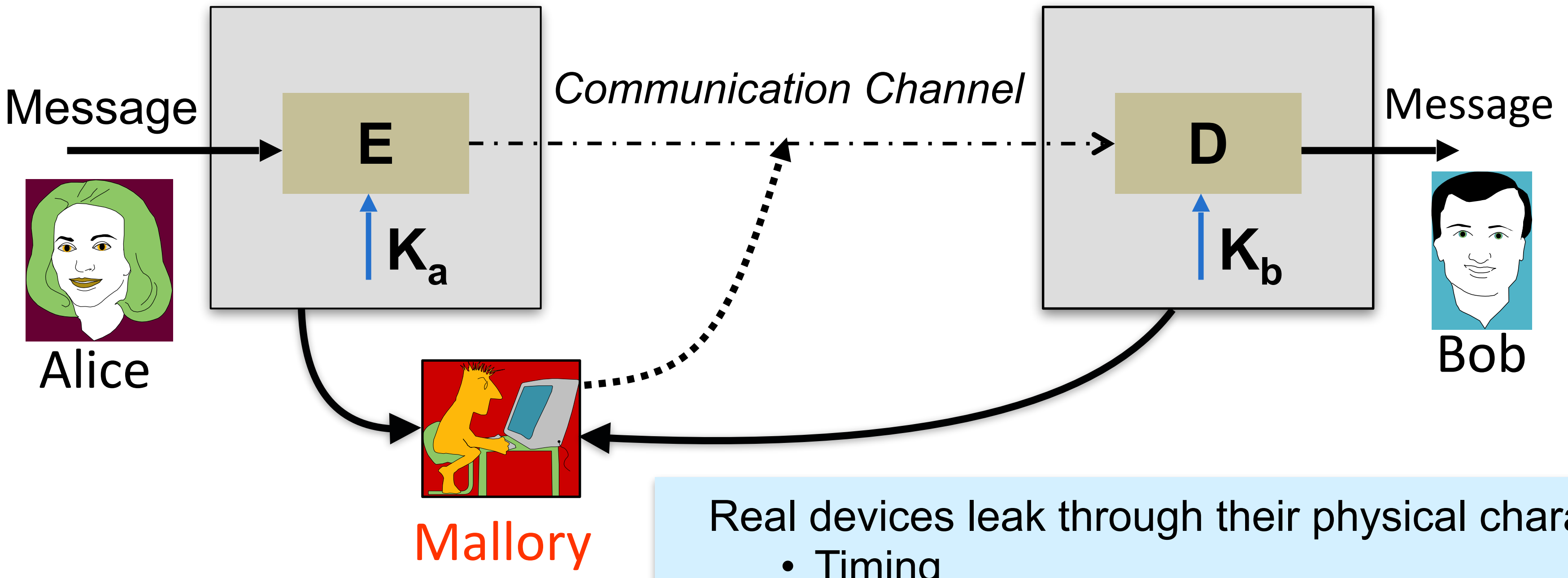
(a) Original storage matrix (b) Bit-slice storage matrix

Side Channel Attacks

Why do Cryptographers Need Engineers?



Cryptographic Security: Real World



Strong cryptographic algorithms are only the beginning

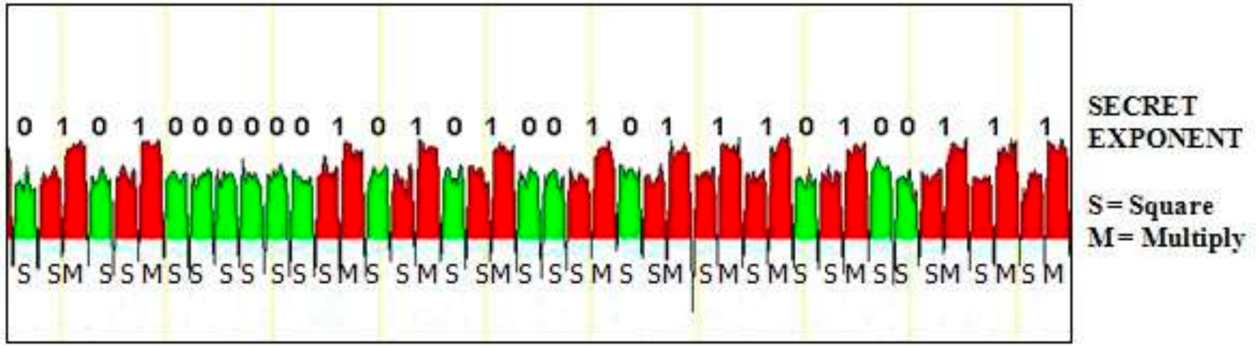
Real devices leak through their physical characteristics

- Timing
- Power consumption
- Electromagnetic Radiation
- Sound
- Faults

Analysis and mitigation of physical attacks are cryptographic as well as engineering problems

Side-Channel Attacks (SCA)

- The physical channels are correlated with the information being processed
- <u>Fundamental cause:</u> power consumption is correlated with switching of CMOS transistors (0->1, 1->0)
    - Typically it is assumed that power consumption is correlated with the Hamming Weight/ Distance.
- If some internal state is exposed, the secret key can be recovered in seconds.



Source: Internet

Encryption

Plaintext	$M < n$
Ciphertext	$C = M^e \pmod{n}$

Decryption

Ciphertext	C
Plaintext	$M = C^d \pmod{n}$

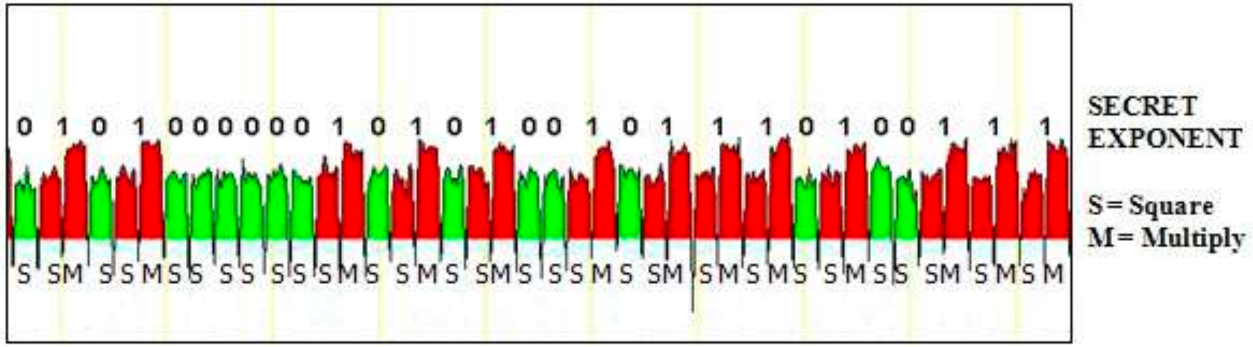
Square and Multiply Algorithm

- Goal: Compute $a^e \pmod{n}$

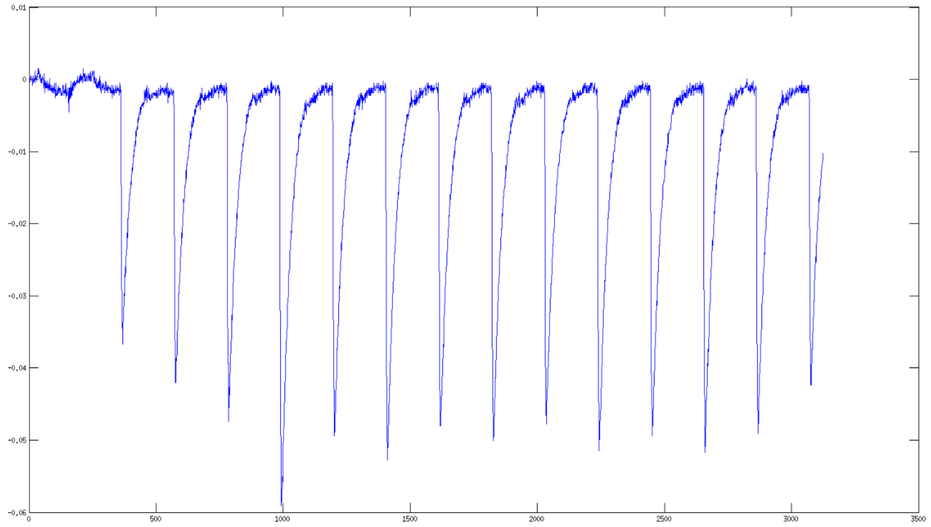
1. convert e to binary: $k_s k_{s-1} \ldots k_1 k_0$
2. b = 1;
3. for (i=s; i>=o; i--)
4. { b = b*b (mod n);
5. if (k_i == 1)
6. b = b * a (mod n)
7. }
8. return b;

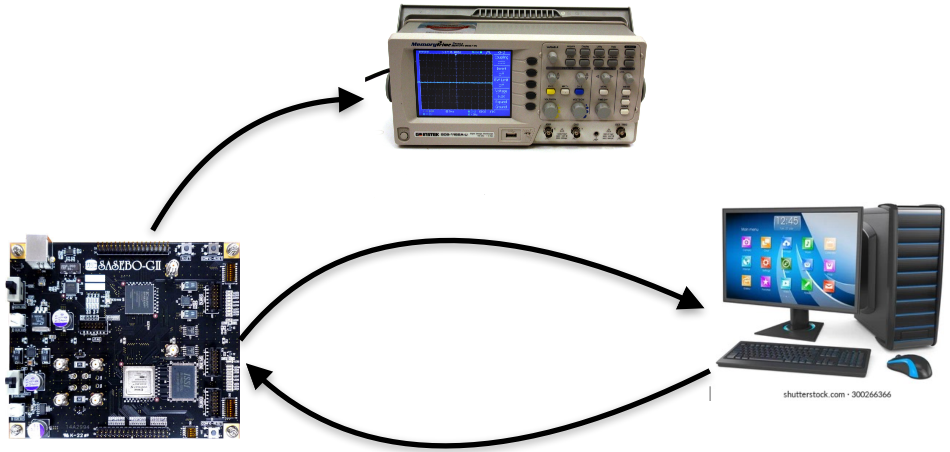
Side-Channel Attacks (SCA)

- The physical channels are correlated with the information being processed
- <u>Fundamental cause:</u> power consumption is correlated with switching of CMOS transistors (0->1, 1->0)
    - Typically it is assumed that power consumption is correlated with the Hamming Weight/ Distance.
- If some internal state is exposed, the secret key can be recovered in seconds.



Source: Internet

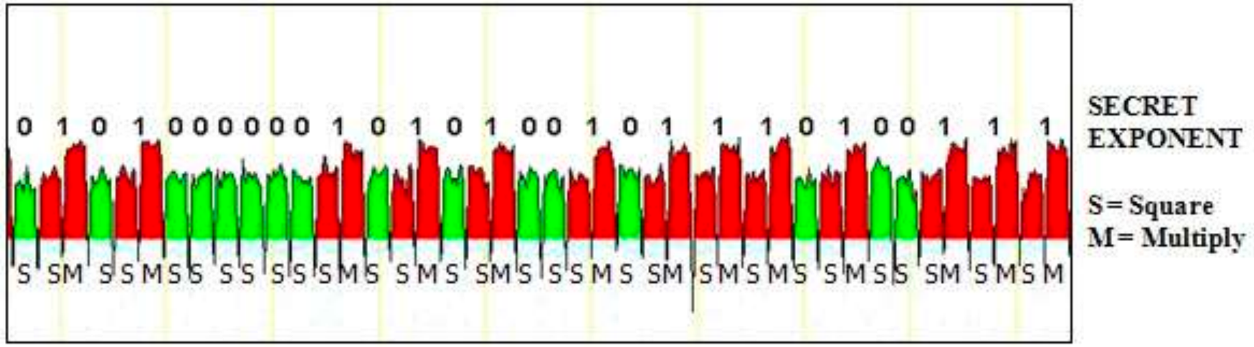




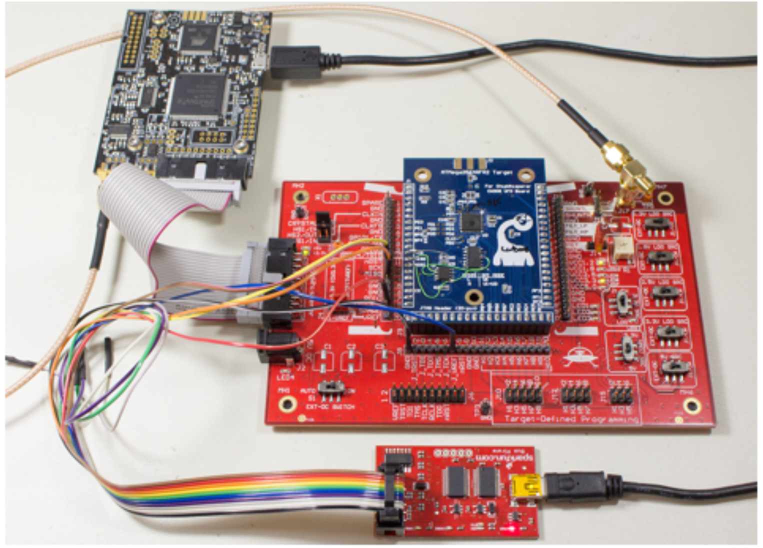
Source: Testbed for Side Channel analysis and security evaluation

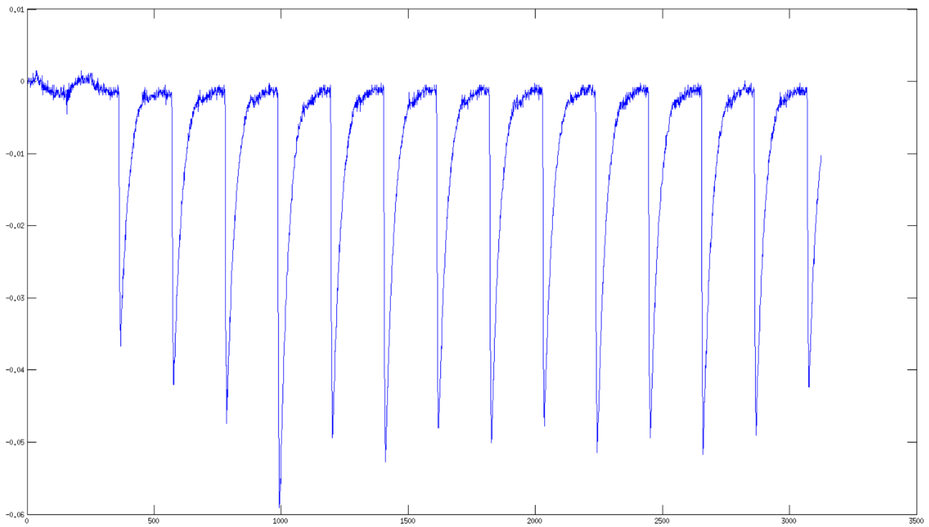
Side-Channel Attacks (SCA)

- The physical channels are correlated with the information being processed
- Fundamental cause: power consumption is correlated with switching of CMOS transistors (0->1, 1->0)
    - Typically it is assumed that power consumption is correlated with the Hamming Weight/ Distance.
- If some internal state is exposed, the secret key can be recovered in seconds.



Source: Internet





Source: Testbed for Side Channel analysis and security evaluation

Side-Channel Vs. Classical Cryptanalysis

- Cryptanalysis: Purely mathematical
  - Take example of AES
  - Cryptanalysis means, you only have access to plaintext, ciphertext — a lot of them
  - You have to
    - Find the key
    - Or, at least, show that it is distinguishable from uniform randomness

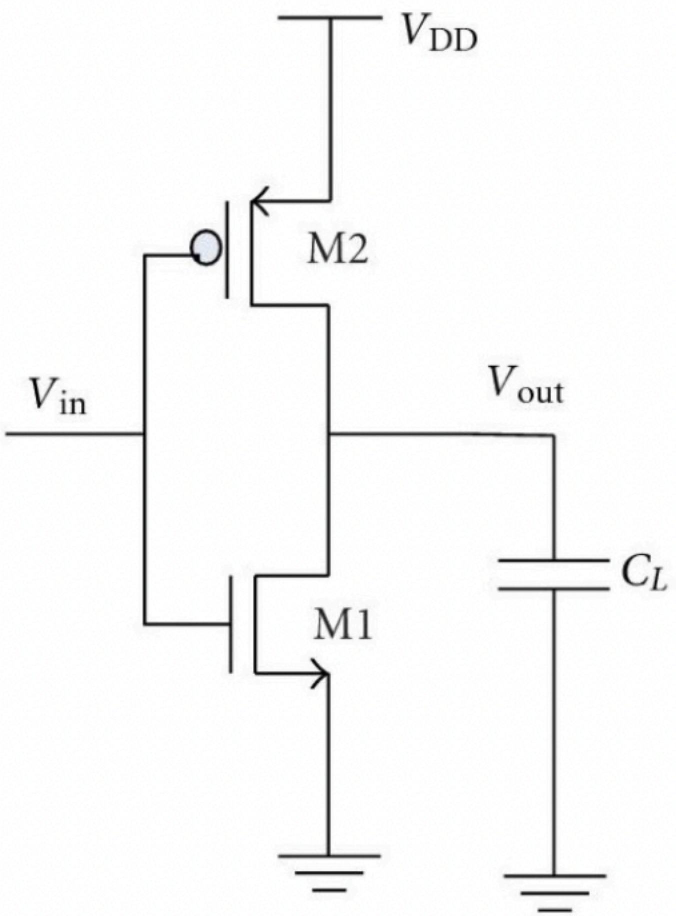
Side-Channel Vs. Classical Cryptanalysis

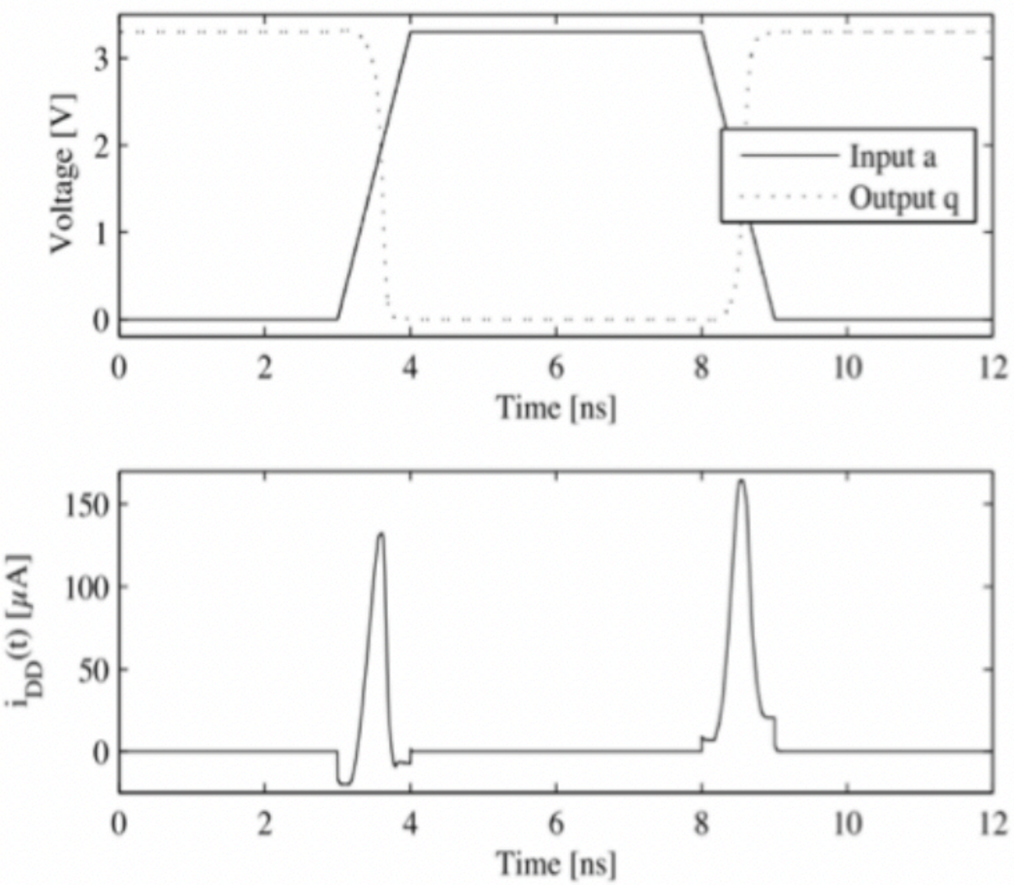
- <u>Cryptanalysis</u>: Purely mathematical
  - Take example of RSA/ECC/PQC
  - Cryptanalysis means, you only have access to plaintext, ciphertext — a lot of them
  - You have to
    - Find the key
      - Maybe you need to solve the underlying hard problem in some (mathematical) way.

Side-Channel Vs. Classical Cryptanalysis

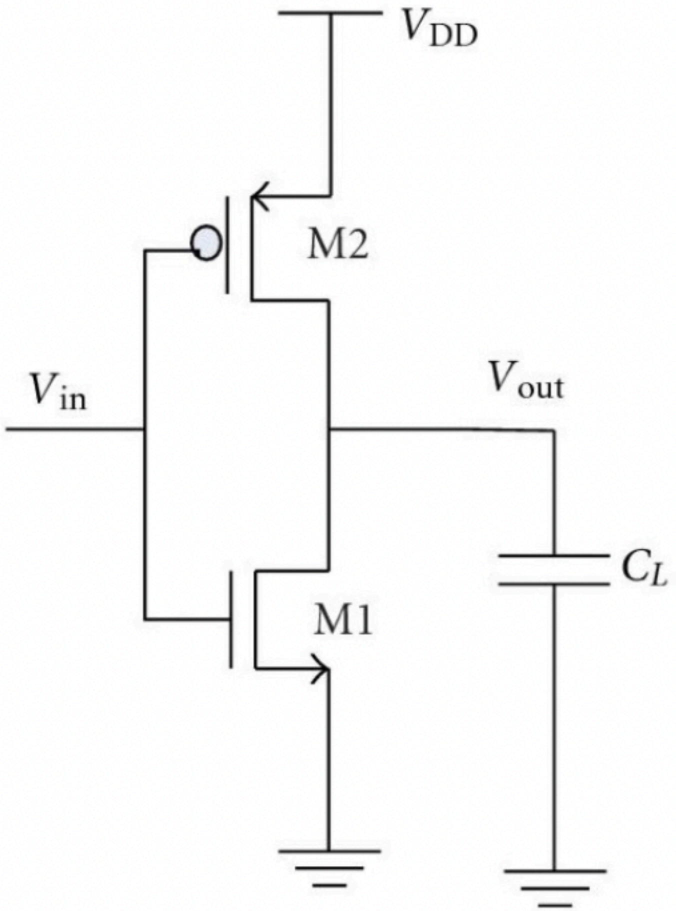
- Side-Channel Cryptanalysis: Mathematics + Physics + Statistics
  - The goal is mostly to recover key
    - But also signature forgery, confidentiality breach
  - Ranges beyond crypto…
    - Kernel information extraction
    - Unprivileged access
    - Neural network reverse engineering

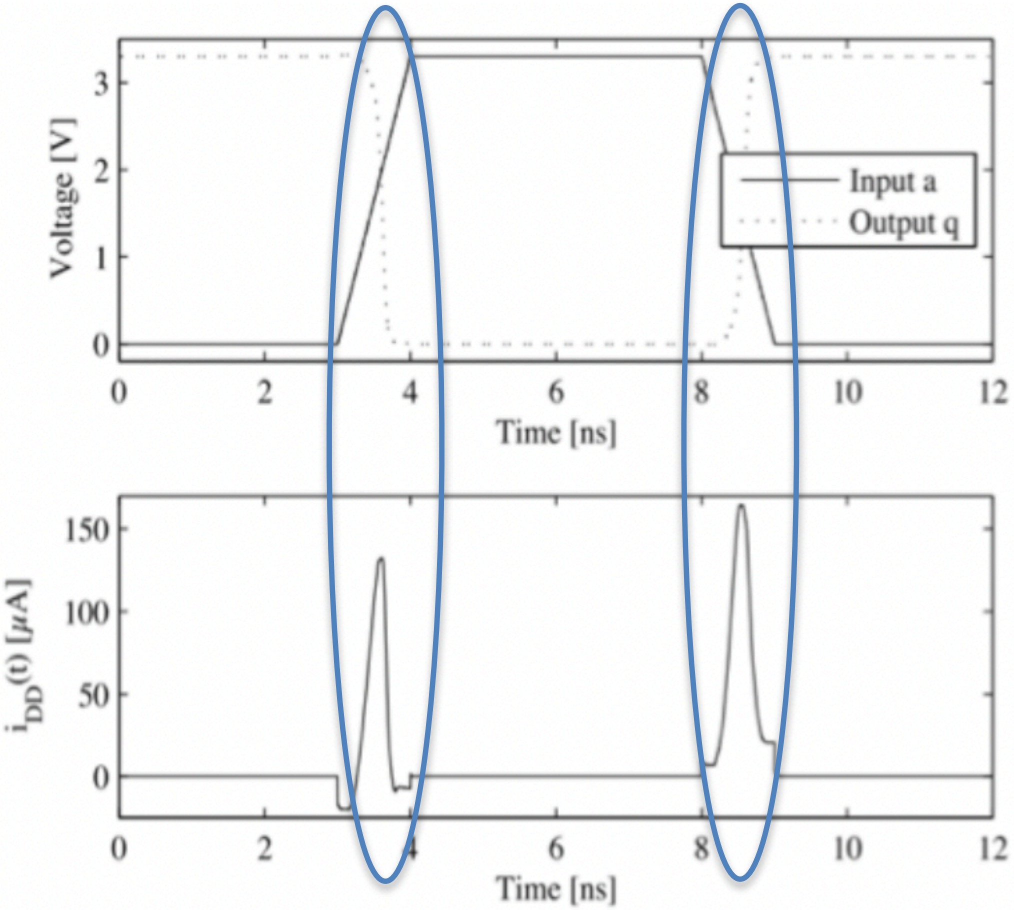
The Root Cause





The Root Cause

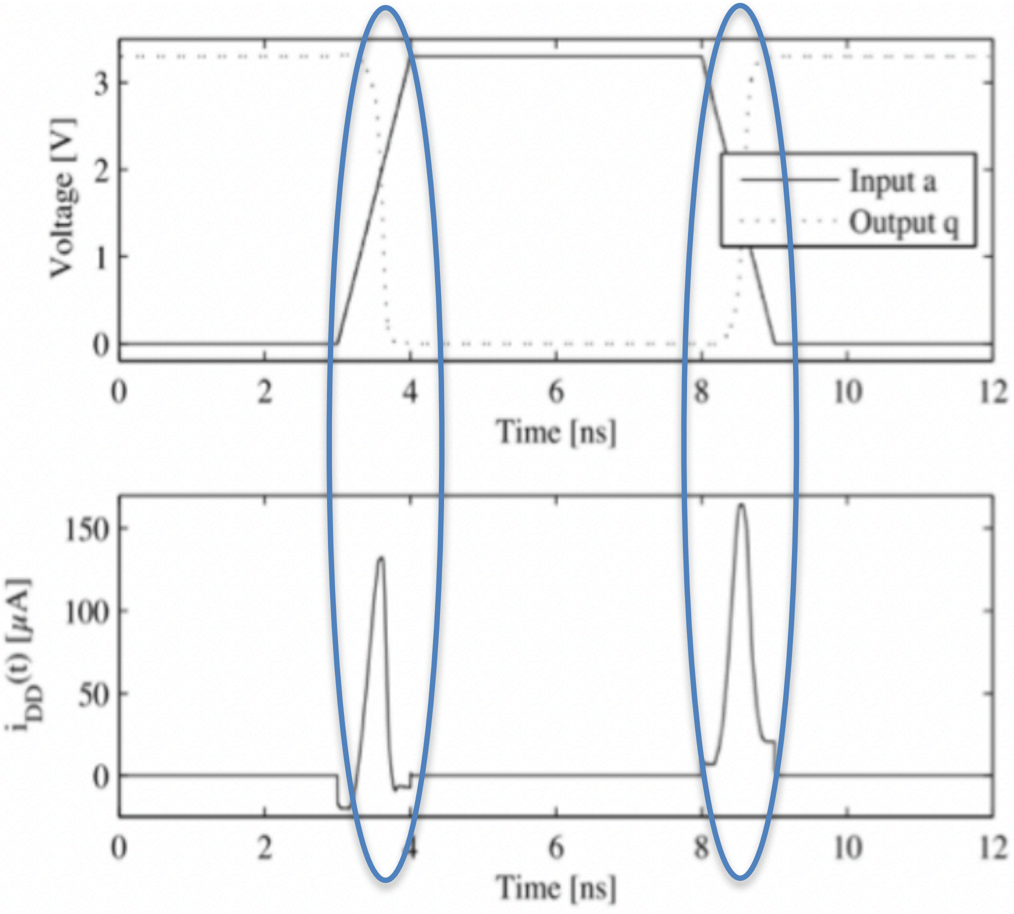




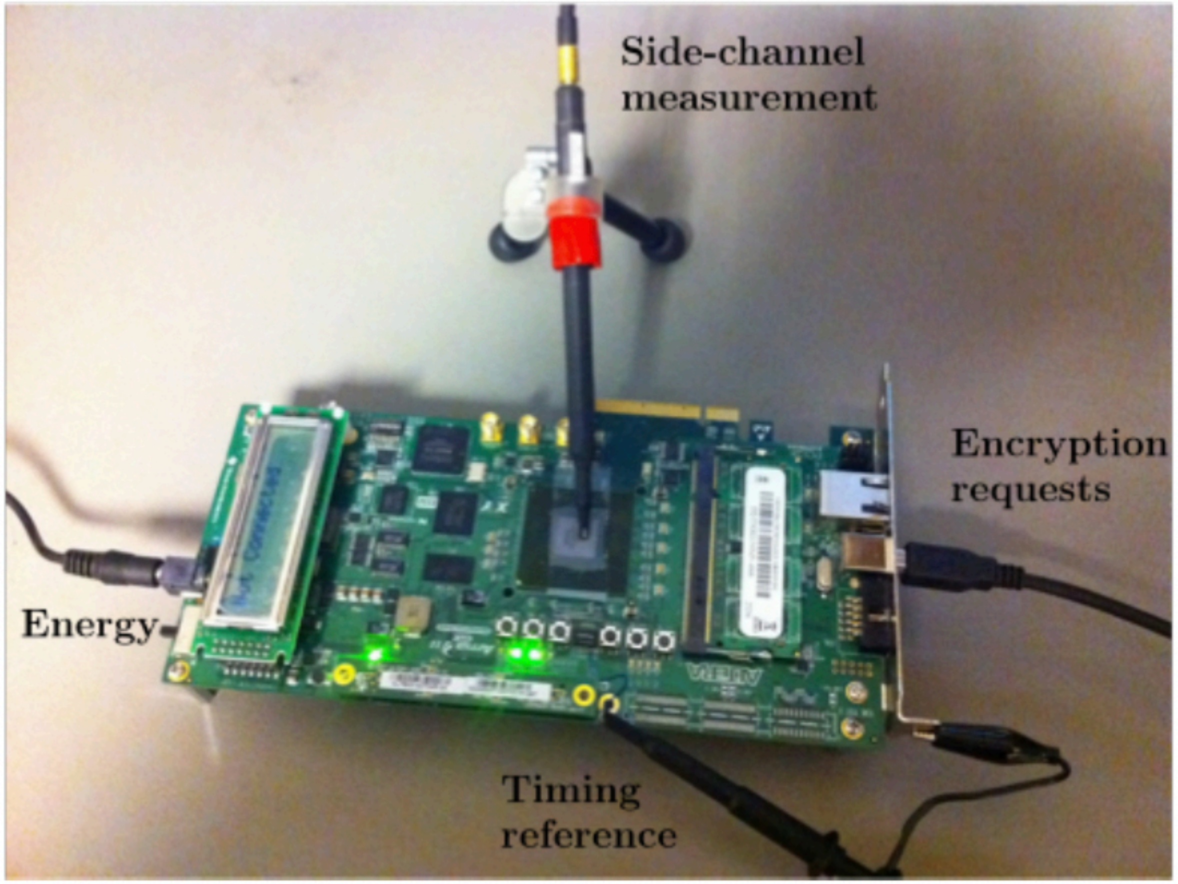
The Root Cause

What is exploited?

- The state change of a gate is proportional to the power dissipated.
- Think about a circuit with millions of gates.
- **How to measure**
  - Power dissipation can be measured by putting a resistor in series with Vdd or Vss and the true source/ground.
  - Roughly, 1 Ohm resistors work well for many microcontrollers, but it is highly target dependent
  - We actually measure current.
    - Differential probes.
  - **The best approach is to use a near-field H-probe an measure EM signal**
    - Less noisy than global power measurement



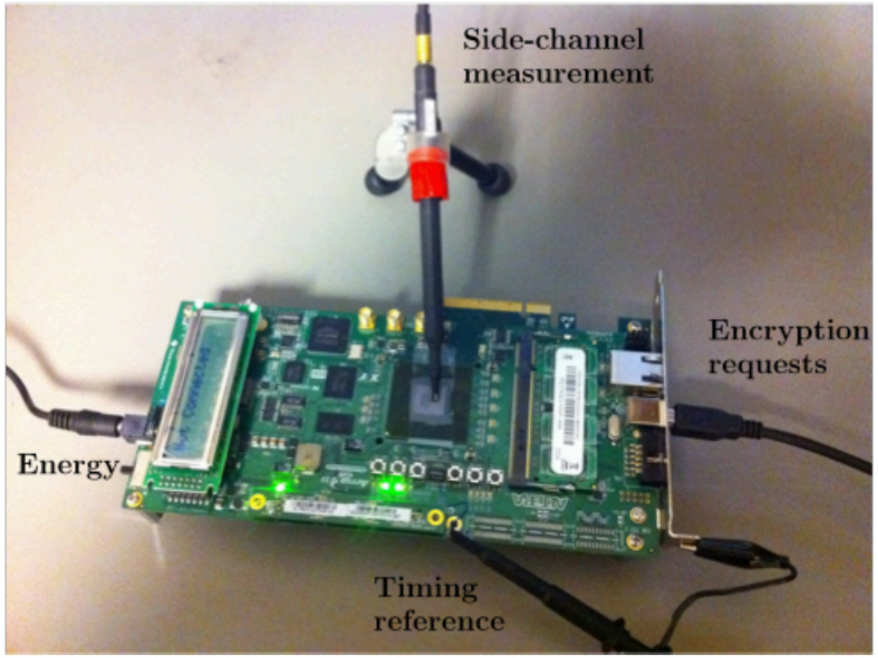
The Root Cause



What to “Measure”?

<u>The Crypto Running on a Microcontroller/ FPGA/ASIC</u>

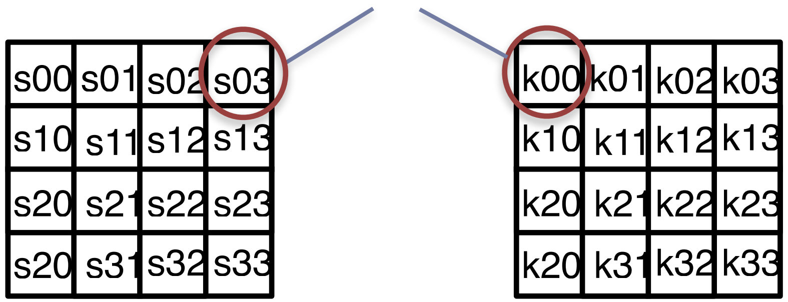
- End of the day everything is CMOS!!!
- Since power consumption is proportional to the switching activity, *so we can get some idea about the internal computation of the crypto*
    - **The crypto is no more black box**
- <u>In this talk we will be specifically focusing on symmetric key algorithms</u>
    - **AES**
- **<u>What do we mean by attacking AES?</u>**
    - **Finding out it’s secret key**



Looking Inside AES

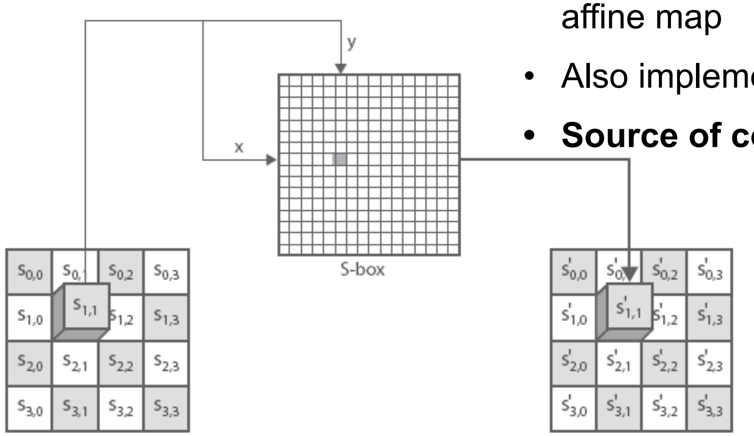
State

1 byte



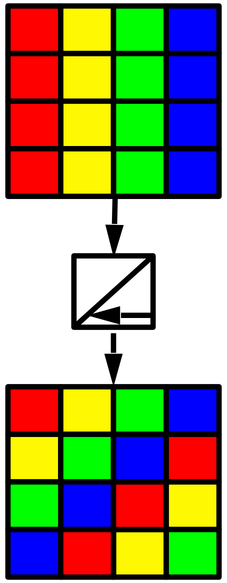
1. SubBytes

- Nonlinear Boolean Function
- Finite field inversion followed by affine map
- Also implemented as a table
- **Source of confusion**

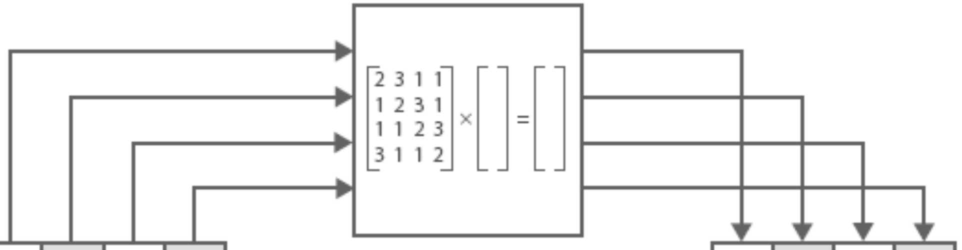


2. ShiftRows

- Linear Boolean Function
- Left circular shift of rows
- **Source of diffusion**

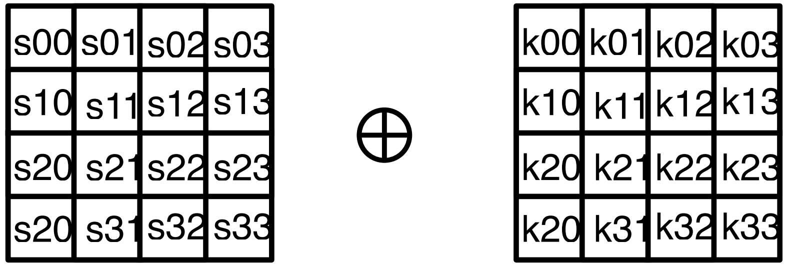


3. MixColumns



- Linear Boolean Function
- Multiplies each column by a constant matrix in GF(2^8)
- **Source of diffusion**

4. AddRoundKey

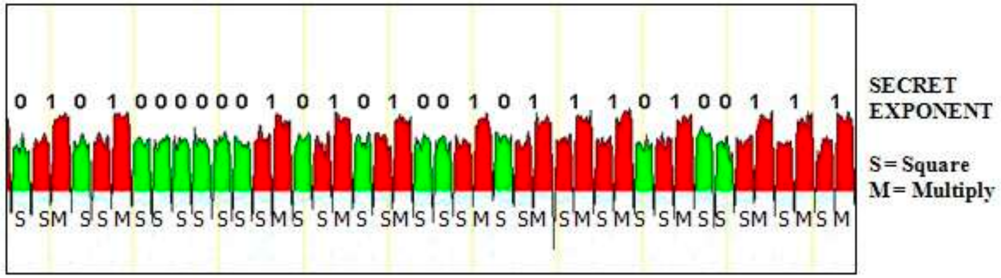


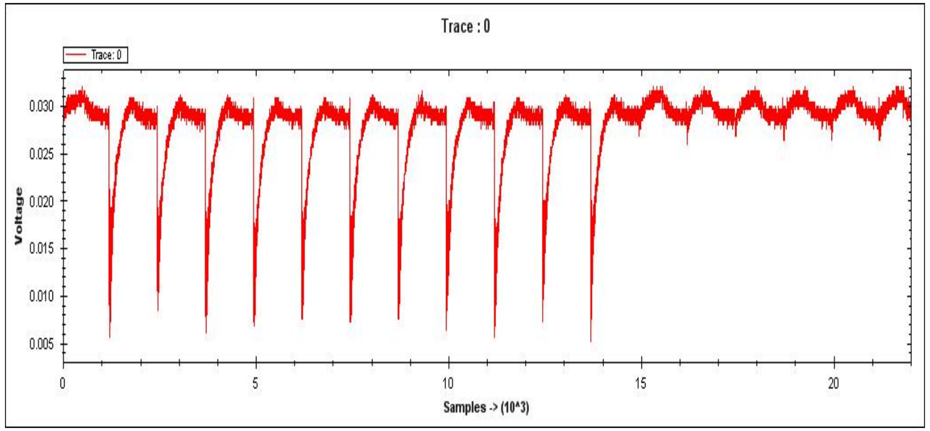
- Linear Boolean Function
- XOR the state with a round key

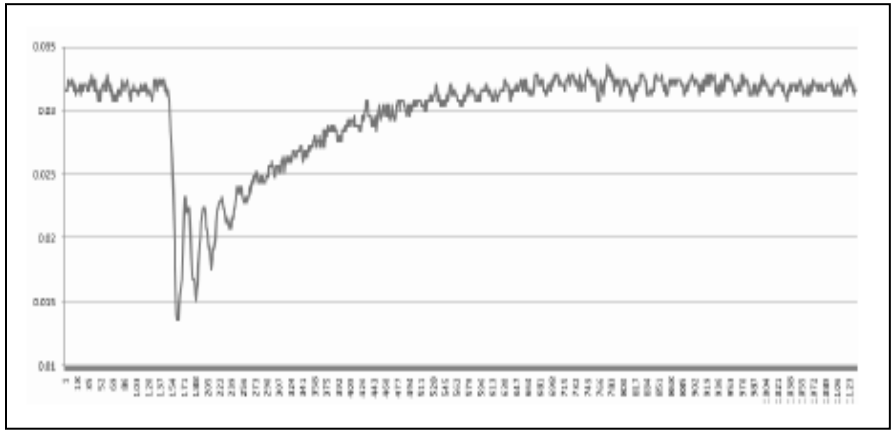
Through the Looking Glass

<u>There can be two kinds of power analysis attacks:</u>

- **Simple power analysis (SPA)**: Exploits the operation dependence of power consumption
  - Remember the RSA example from the beginning…
- **Differential Power Analysis (DPA):** Exploits the data dependence of power consumption
  - We well see now for AES
  - **Fact: there is no secret dependent operation in AES, everything is uniform.**



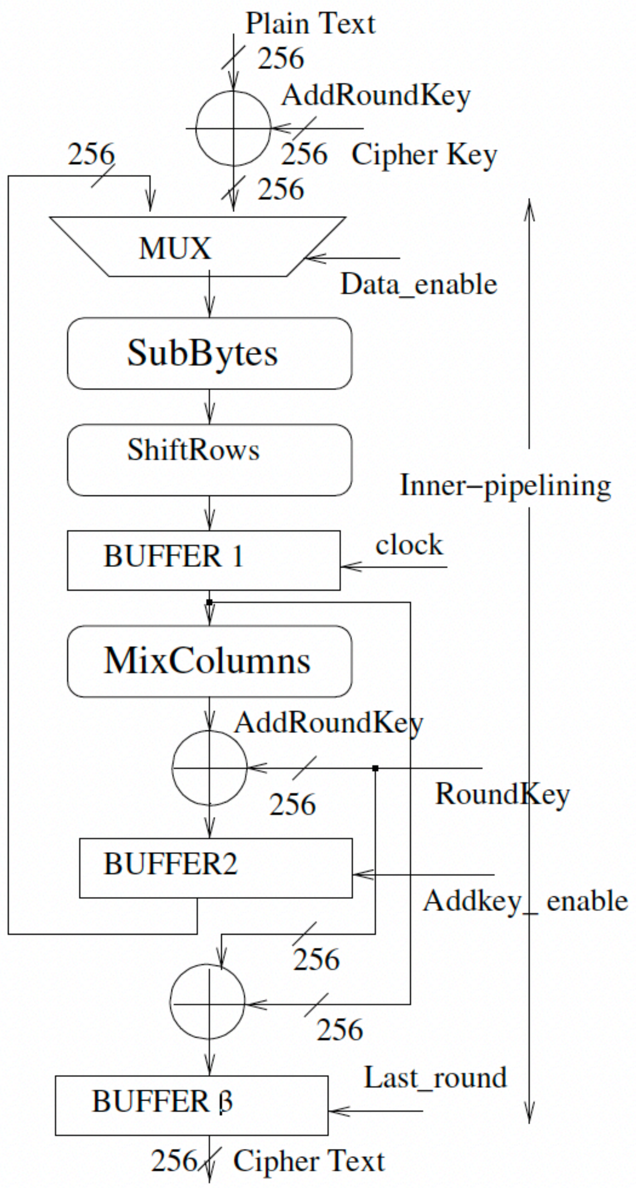




Differential Power Analysis

- **Power Trace**: A set of power consumptions across a cryptographic process
    - 1 millisecond operation sampled at 5MHz yield a trace with 5000 points.
- **Leakage Model**: Hypothetical model relating the leakage with the internal states of the target algorithm.
    - For AES the internal state is a 128-bit value.
    - **Hamming Weight Model:** The power consumption is proportional to the Hamming weight (count of 1's) of the state.
    - **Hamming Distance Model**: The power consumption is proportional to the Hamming distance between the state in two consecutive clock cycles. — FPGA/ASICs
    - More complex models are possible…
    - Used to simulate leakage and also in some attacks.

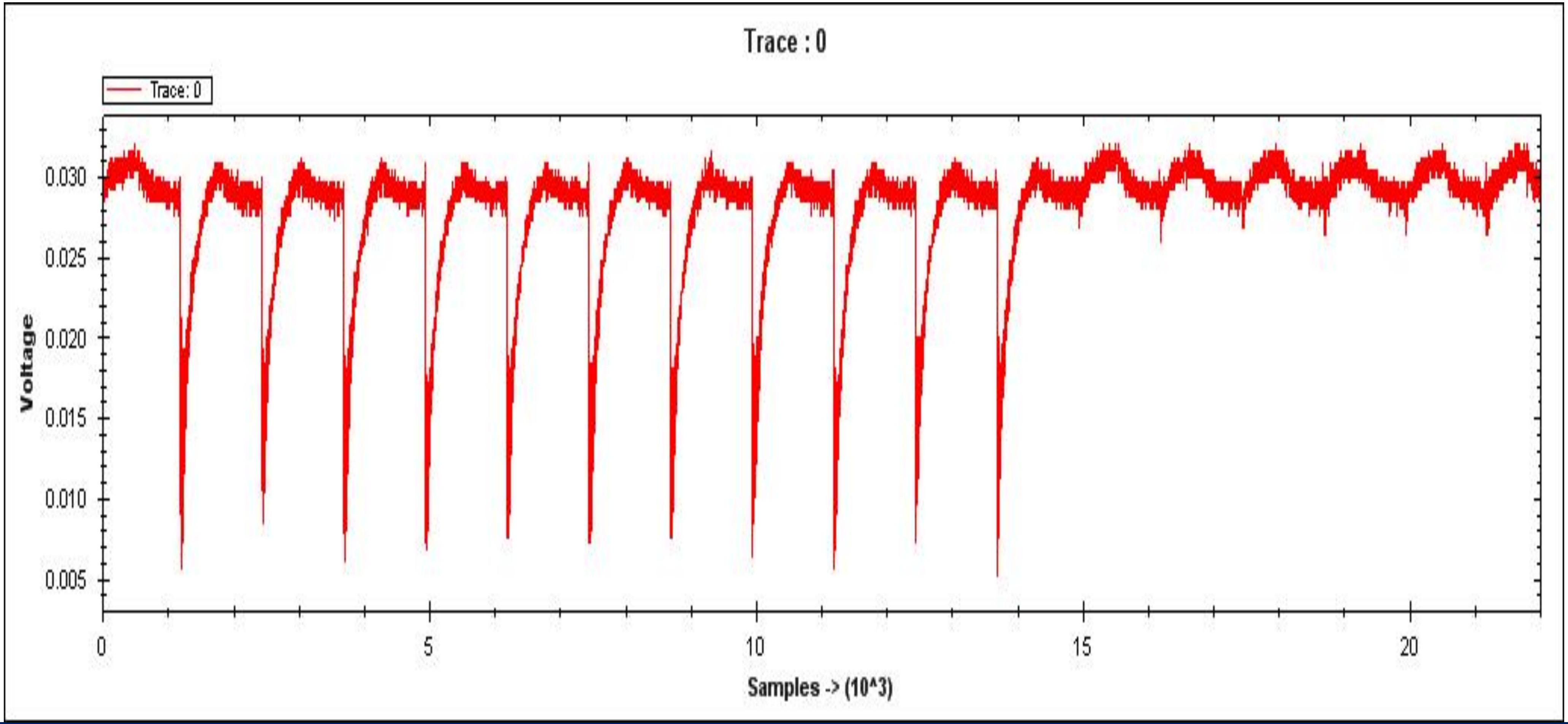
Differential Power Analysis



- Common hardware implementation of a block cipher
- We consider the state of the circuit at time instance *t* — you can consider it as one time point in the x-axis of the trace.
- Let this state be v_t
- Hamming weight is number of 1's in v_t.
- Hamming distance is HW($v_t \oplus v_{t-1}$)..
  - Why?

Differential Power Analysis

- **Power Trace**: A set of power consumptions across a cryptographic process
    - 1 millisecond operation sampled at 5MHz yield a trace with 5000 points.
-



Differential Power Analysis: The Idea

s	HW(s)	Target bit (LSB)
0000	0	0
0001	1	1
0010	1	0
0011	2	1
0100	1	0
0101	2	1
0110	2	0
0111	3	1
1000	1	0
1001	2	1
1010	2	0
1011	3	1
1100	2	0
1101	3	1
1110	3	0
1111	4	1

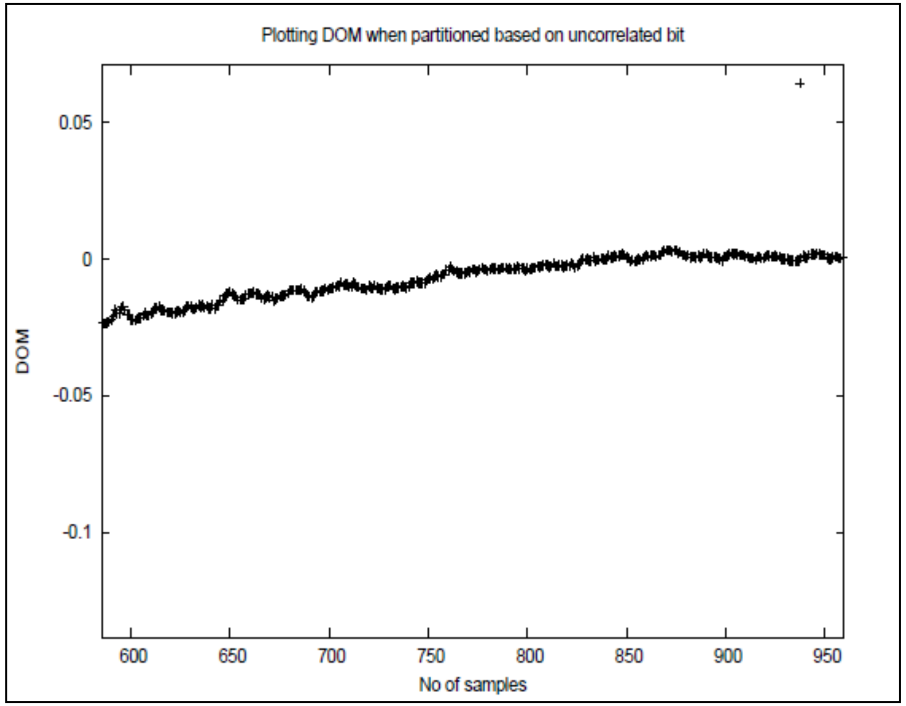
- Assume power leakage follows Hamming Weight.
- Divide the HW(s) into two bins:
  - 0 bin: when LSB is 0
  - 1 bin: when LSB is 1

Differential Power Analysis: The Idea

s	HW(s)	Target bit (LSB)
0000	0	0
0001	1	1
0010	1	0
0011	2	1
0100	1	0
0101	2	1
0110	2	0
0111	3	1
1000	1	0
1001	2	1
1010	2	0
1011	3	1
1100	2	0
1101	3	1
1110	3	0
1111	4	1

- Assume power leakage follows Hamming Weight.
- Divide the HW(s) into two bins:
  - 0 bin: when LSB is 0
  - 1 bin: when LSB is 1
- Difference-of-Mean (DoM)=20/8-12/8=1

When the Partitioning is Random



- Parititioning done by bits simulated using *rand* function in C.
- Observe the DoM is close to 0, as expected!
- **Note**: Instead of computing the difference, we can use some statistical hypothesis test, such as t-test.
- The hypothesis will be — whether the two trace distributions are the same or different
- For a random uncorrelated bit, the two distributions are the same.

- <u>Moral of the story</u>: If the bin partitioning is based on a bit from the actual state, there will be significant difference in the mean values of the bins. This is because, the traces are data dependent.

Attacking AES

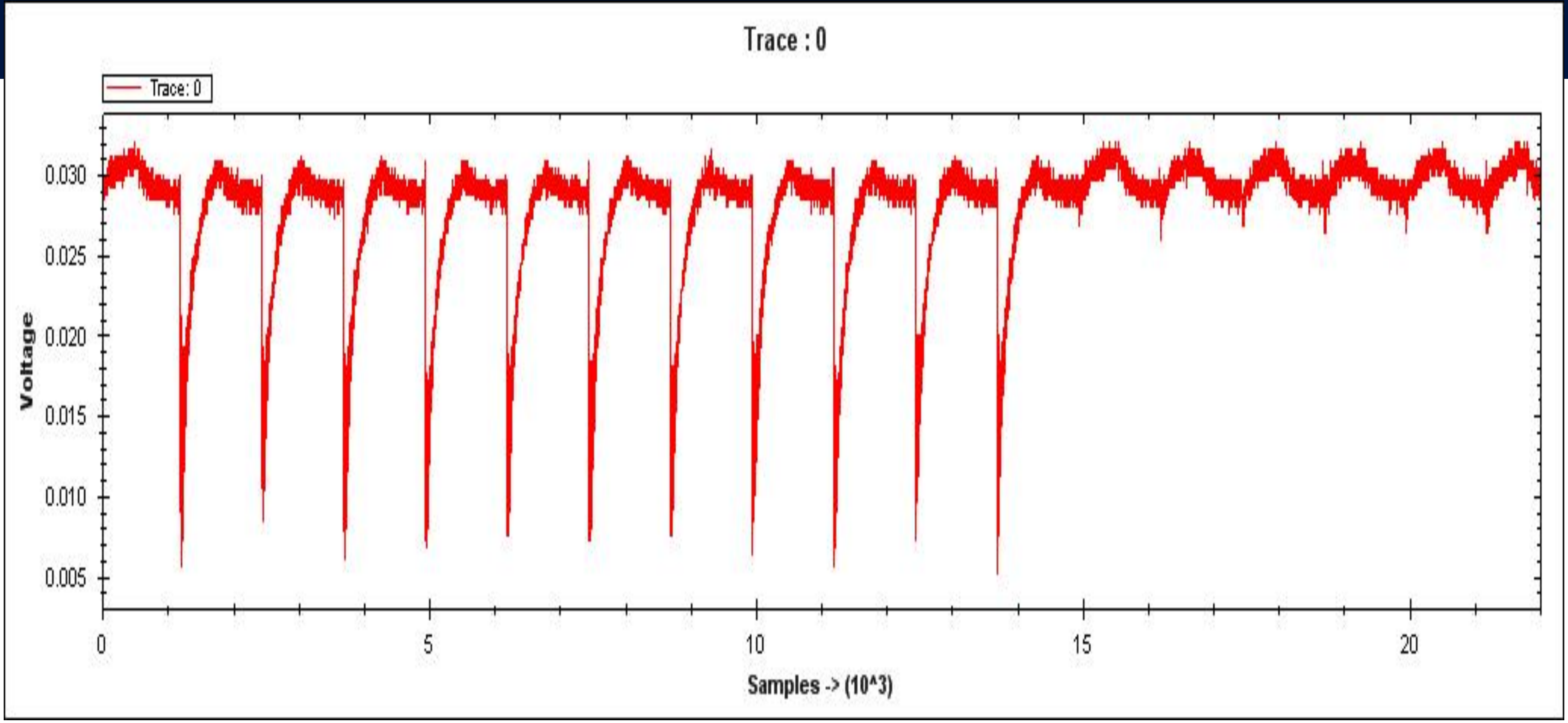
<u>A very very important point</u>

- We haven’t seen AES key schedule in detail — but it is has somewhat similar operations as the rounds — has SBoxes, shifts XORs etc.
- **Important**: key schedule is invertible
  - That is, if you recover one round key , you can recover all, and the master key

Attacking AES

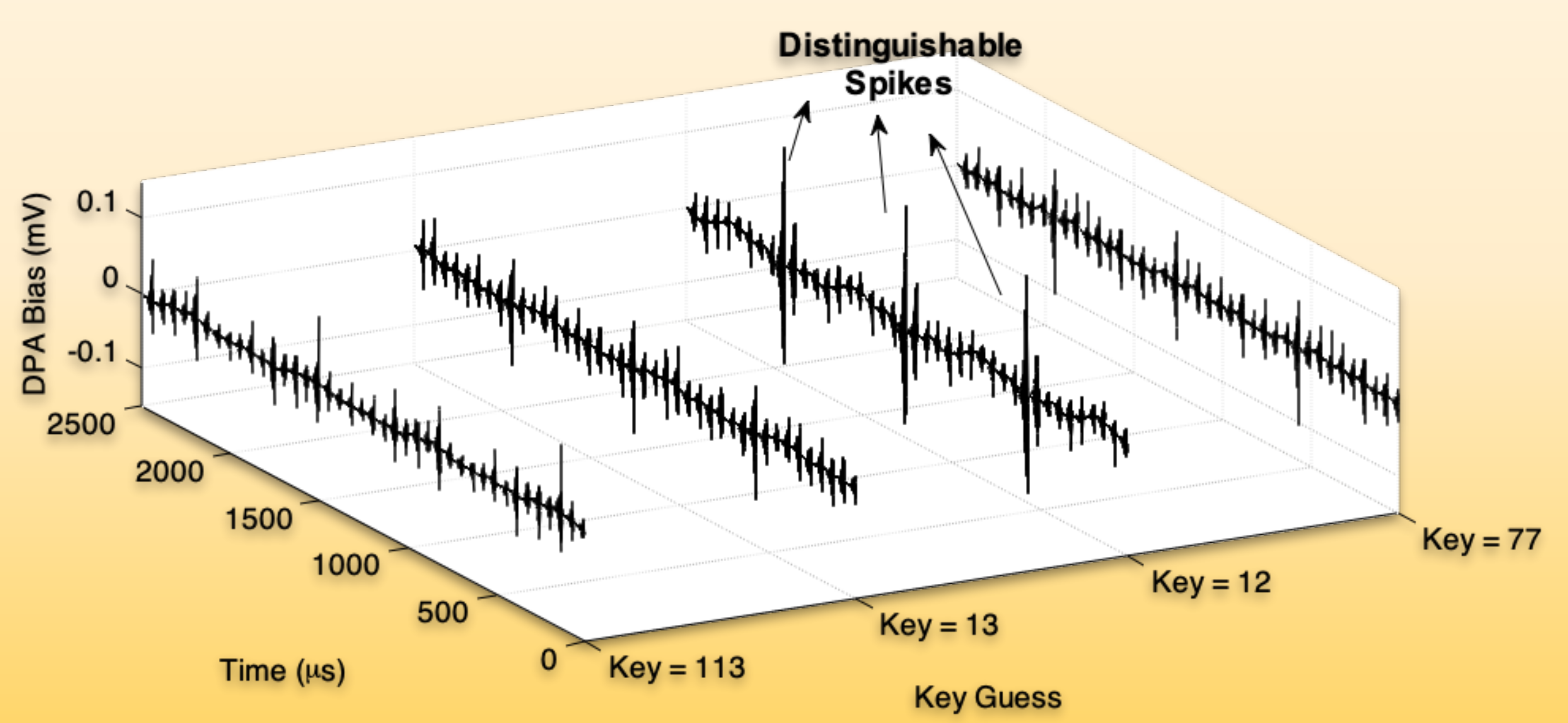
1. **Measurement**:
   - Make power consumption measurement of about 1000 AES operations, 100000 data points/trace,
   - Save (Ciphertext$_i$, trace_i)
2. **Attack**:
   - Target an S-Box in the last round (say the *j*-th S-Box)
     A. Guess a key for an S-box of last round (8 bit key, so total 256 guesses possible)
     B. Partially decrypt one byte of each ciphertext with the guessed key till the input of the last round S-Box. That is compute: $S_j = S^{-1}(C_j \oplus k_j^g)$
     C. Divide the traces into 2 groups based on the LSB of S_j
     D. Calculate the **average trace** of each group
     E. Calculate the difference of two **average traces**
     F. **Correct key guess → spikes in the differential curve**
   - Repeat A-F for other S-boxes

Attacking AES



CT[0]	T[0][0]	T[0][1]	…	T[0][m]
CT[1]	T[1][0]	T[1][1]	…	T[1][m]
CT[2]	T[2][0]	T[2][1]	…	T[2][m]
…	…	…	…	…
CT[n]	T[n][0]	T[n][1]	…	T[n][m]

Attacking AES



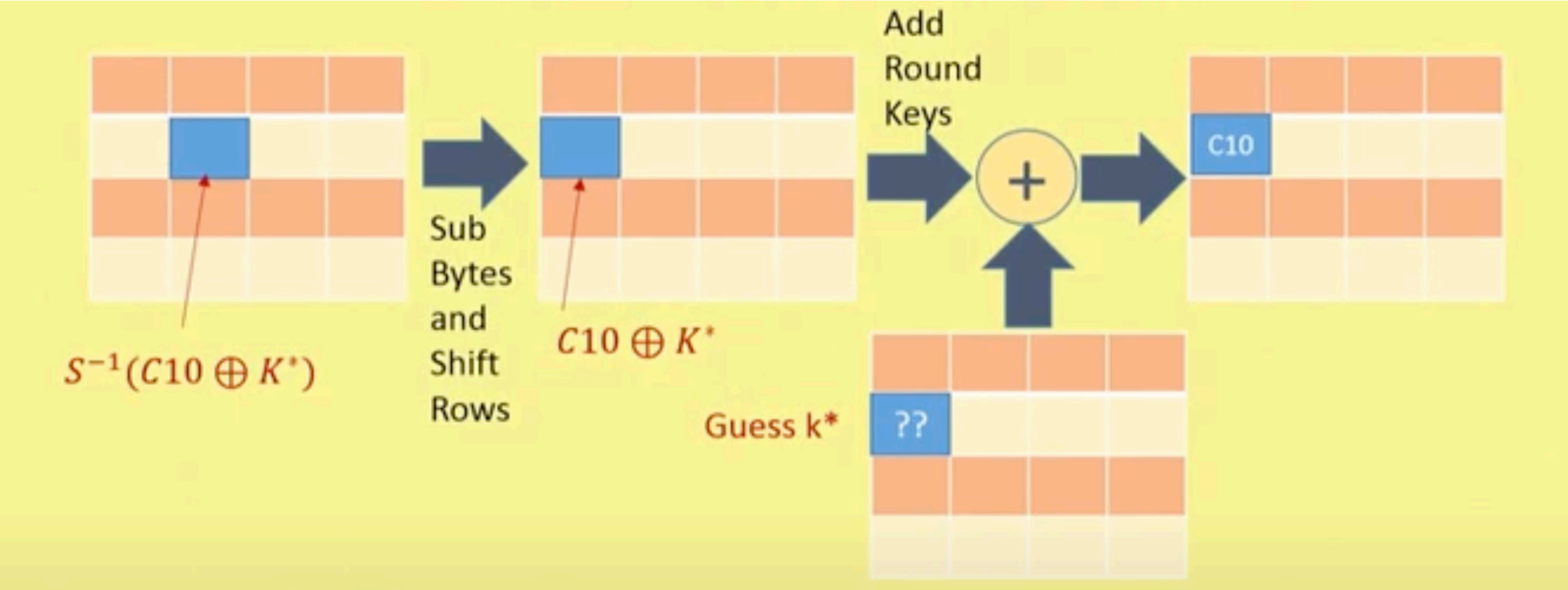
SBOX – 11 BIT – 8 TRACE COUNT = 15,000, FPGA implementation

Attacking AES

- **DPA selection function**: D(C,b,k^g) is defined as computing the value of the
  - b^{th} output bit, depending upon
    - C: Ciphertext
    - k^g is the guessed key for the S-Box
- In the attack, $D = S^{-1}(C_j \oplus k_j^g)$
- If k^g is a wrong guess then *b* is correctly evaluated only for half of the ciphertexts (randomly).
  - **Thus for large number of points, the difference between average traces is close to 0**
  - **In other words, distribution of both the bins will be the same**
- But if k^g is a correct guess, then *b* is correctly evaluated for all the ciphertexts.

Principle: If K_s is wrongly guessed, D behaves like a random guess. Thus for a large number of sample points, Δ[1..k] tends to zero. But if its correct, the differential will be non-zero and show spikes when D is correlated with the value being processed.

Attacking AES



$$D(C10, b = 0, K10) = S^{-1}(C10 \oplus K^*)\,|_{b=0}$$

Attacking AES

- **Differential Trace**: It is a *m* sample trace denoted as Δ_D, where,

$$\Delta_D[j] = \frac{\sum_{i=0}^{n-1} D(C_i, b, K_g)T[i][j]}{\sum_{i=0}^{n-1} D(C_i, b, K_g)} - \frac{\sum_{i=0}^{n-1} (1 - D(C_i, b, K_g))T[i][j]}{\sum_{i=0}^{n-1} (1 - D(C_i, b, K_g))}$$

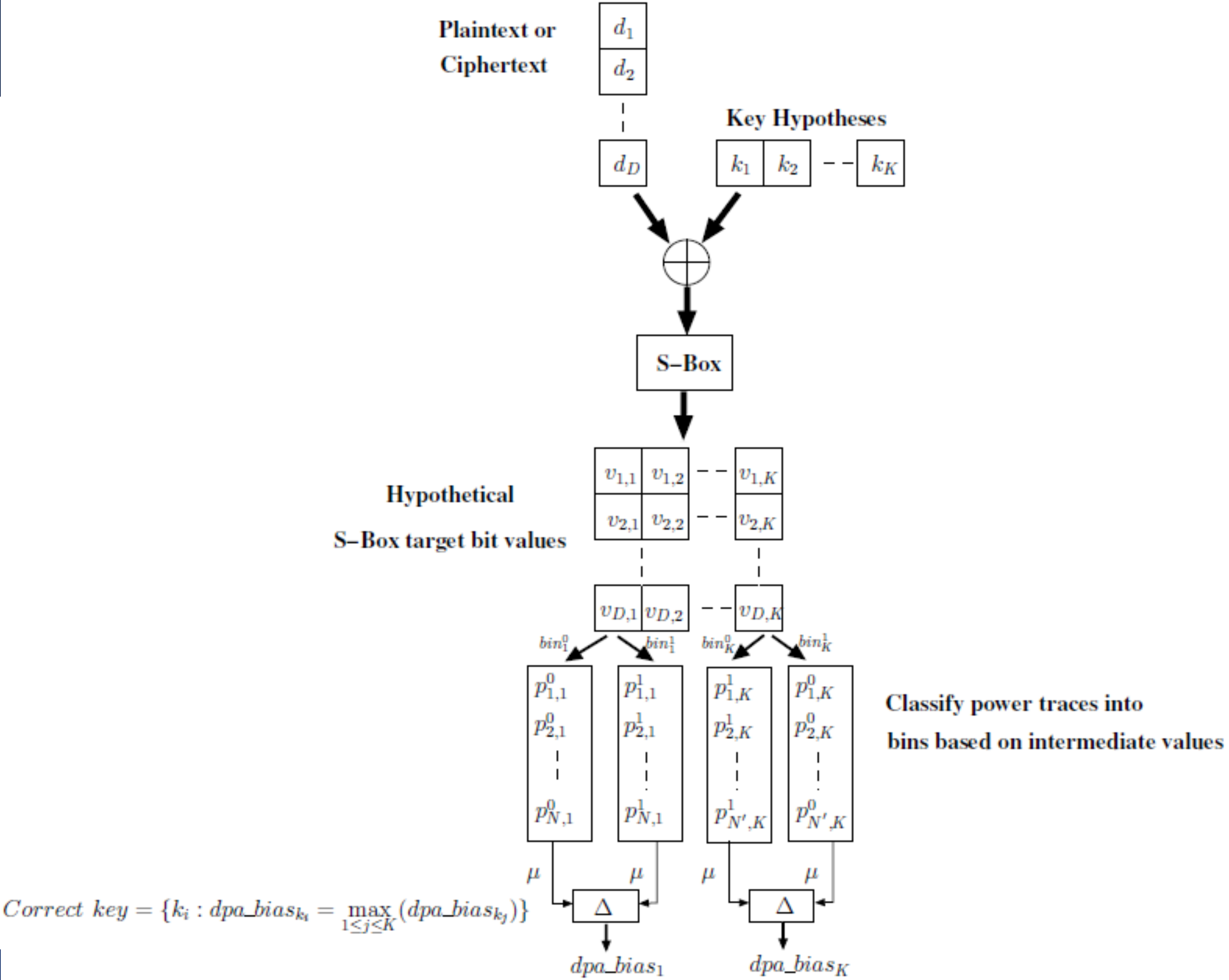
- **Note**: C_i is a particular byte of the *i*-th ciphertext.

Attacking AES

- **Why does the attack work?**
  - It's not only the data dependency. But it also depends on the mathematics of AES
  - **DPA selection function**: D(C,b,k^g) is defined as computing the value of the
    - *b*th output bit, depending upon
      - C: Ciphertext
      - k^g is the guessed key for the S-Box
  - In the attack, $D = S^{-1}(C_j \oplus k_j^g)$
  - If k^g is a wrong guess then *b* is correctly evaluated only for half of the ciphertexts (randomly).
    - **Thus for large number of points, the difference between average traces is close to 0**
    - **In other words, distribution of both the bins will be the same**
  - But if k^g is a correct guess, then *b* is correctly evaluated for all the ciphertexts.
  - **Note**: The non-linearity of the S-Boxes play an important role here.

Principle: If K_s is wrongly guessed, D behaves like a random guess. Thus for a large number of sample points, Δ[1..k] tends to zero. But if its correct, the differential will be non-zero and show spikes when D is correlated with the value being processed.

Attacking AES



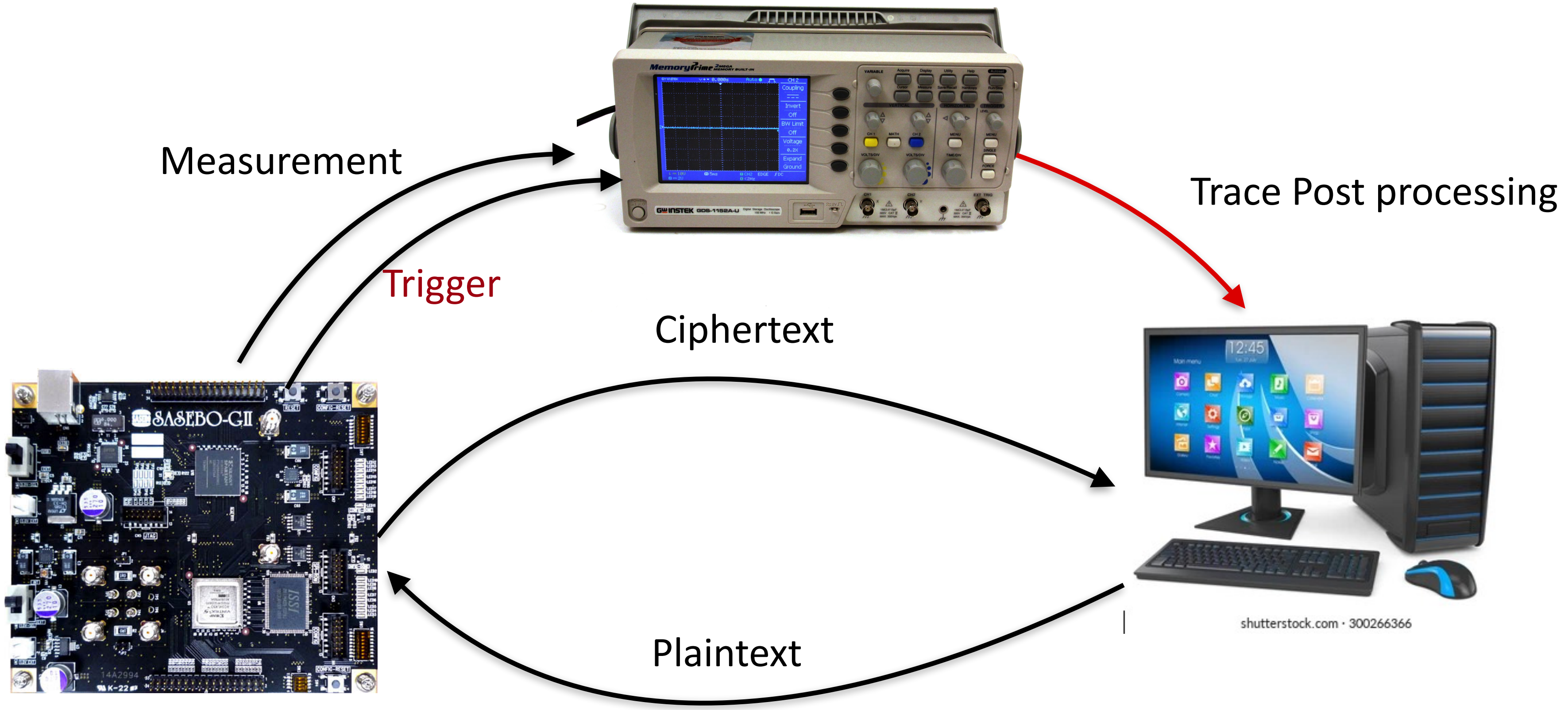
Attacking AES: Attack Complexity

- **What is the attack complexity**
  - Say you are given n number of traces
  - How much further computation you need to perform?

Attacking AES: Attack Complexity

- **Observe**: We are targeting the last round of AES and attacking each S-Box separately.
  - Divide and Conquer
  - Attack complexity: always 2^8
  - Trace complexity: depends
- **Observe**: The attack don’t need plaintext knowledge

Leakage Models



Measurement
Trigger
Trace Post processing
Ciphertext
Plaintext
SASEBO-GII
GW INSTEK
shutterstock.com · 300266366

Differential Power Analysis

- **Power Trace**: A set of power consumptions across a cryptographic process
    - 1 millisecond operation sampled at 5MHz yield a trace with 5000 points.
- **Measurement:**
  - Usually requires to adhere a signal bandwidth and frequency.
    - We do quite high frequency sampling — e.g. for a 65 Mhz microcontroller or 150 Mhz FPGA implementation, we do sampling at 5 GS/sec.
  - Number of samples is an important point. More samples results in better attack
  - **Trigger**: An important constraint
    - Without reliable trigger, the traces will be **misaligned.**—attack will be difficult
    - Most of the practical implementations does not provide a very reliable trigger
      - So, some realignment of traces are needed
        - Today, DL algorithms are great for this purpose — does automatic realignment.

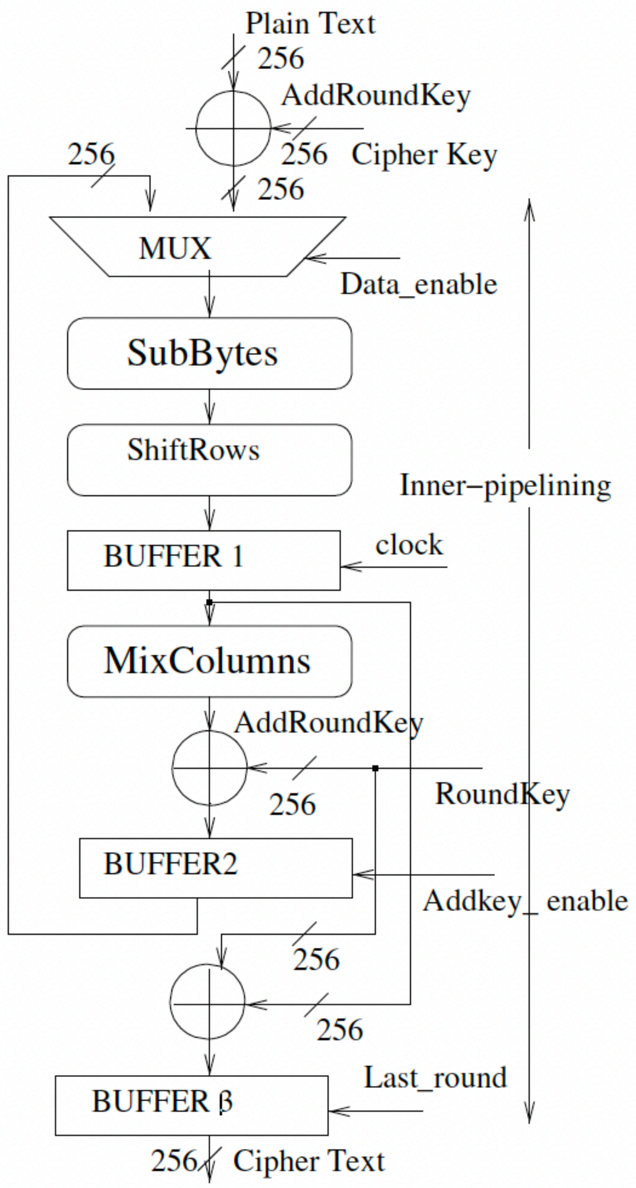
Attacking AES

1. **Measurement**:
   - Make power consumption measurement of about 1000 AES operations, 100000 data points/trace,
   - Save (Ciphertext$_i$, trace_i)
2. **Attack**:
   - Target an S-Box in the last round (say the *j*-th S-Box)
     A. Guess a key for an S-box of last round (8 bit key, so total 256 guesses possible)
     B. Partially decrypt one byte of each ciphertext with the guessed key till the input of the last round S-Box. That is compute: $S_j = S^{-1}(C_j \oplus k_j^g)$
     C. Divide the traces into 2 groups based on the LSB of S_j
     D. Calculate the **average trace** of each group
     E. Calculate the difference of two **average traces**
     F. **Correct key guess → spikes in the differential curve**
   - Repeat A-F for other S-boxes

Differential Power Analysis

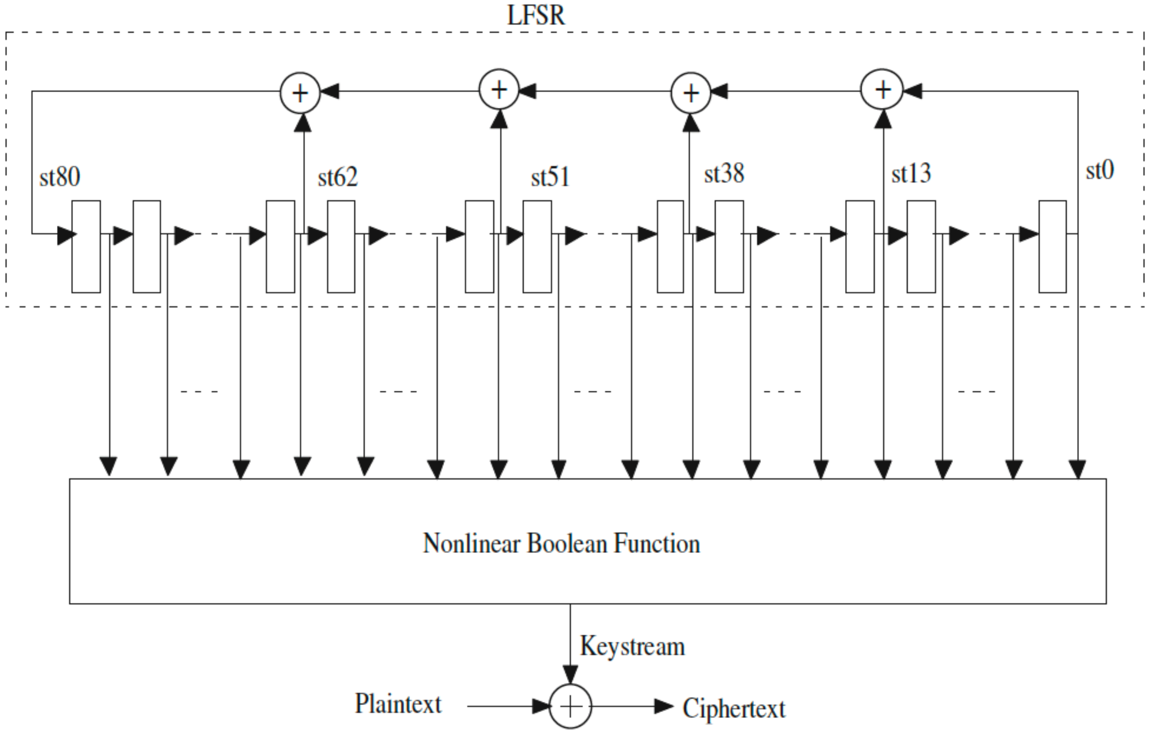
- **Power Trace**: A set of power consumptions across a cryptographic process
    - 1 millisecond operation sampled at 5MHz yield a trace with 5000 points.
- **Leakage Model**: Hypothetical model relating the leakage with the internal states of the target algorithm.
    - For AES the internal state is a 128-bit value.
    - **Hamming Weight Model:** The power consumption is proportional to the Hamming weight (count of 1's) of the state.
    - **Hamming Distance Model**: The power consumption is proportional to the Hamming distance between the state in two consecutive clock cycles. — FPGA/ASICs
    - More complex models are possible…
    - Used to simulate leakage and also in some attacks.

Differential Power Analysis

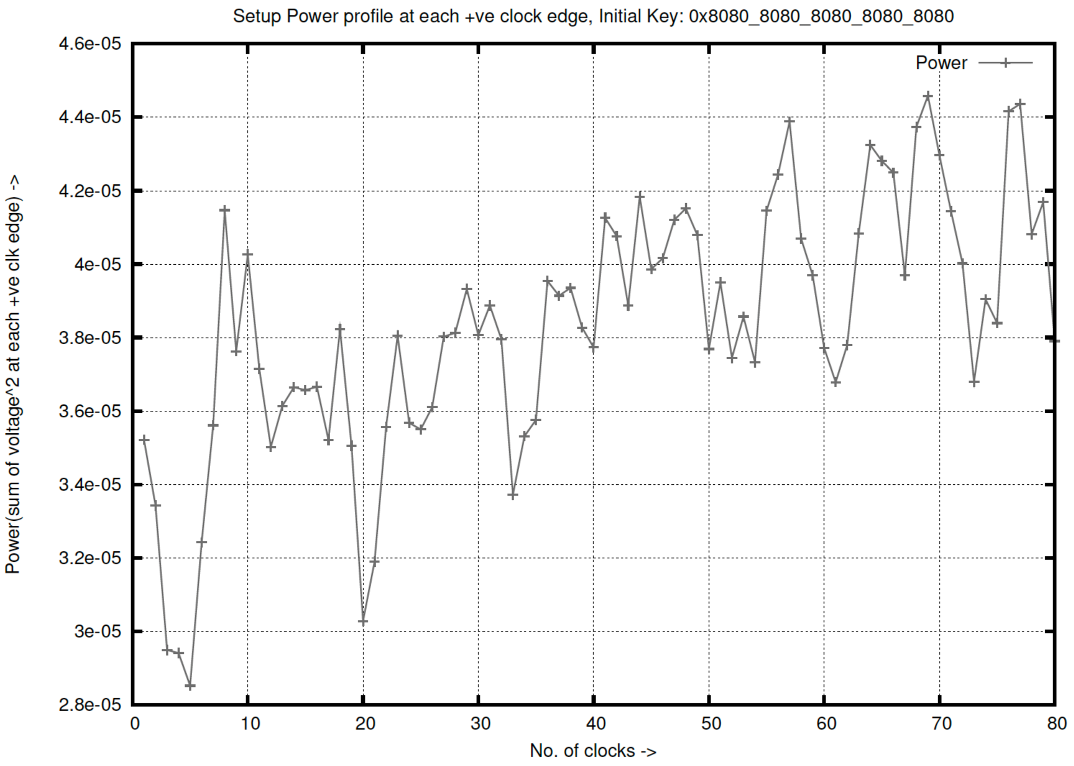


- Common hardware implementation of a block cipher
- We consider the state of the circuit at time instance *t* — you can consider it as one time point in the x-axis of the trace.
- Let this state be v_t
- Hamming weight is number of 1's in v_t.
- Hamming distance is HW($v_t \oplus v_{t-1}$)..
  - Why?
  - After all power consumption is a result of transition $v_{t-1} \rightarrow v_t$
  - So , HW is relatively inaccurate, but works well for software
    - Registers often start from a specific initialisation value.

Differential Power Analysis

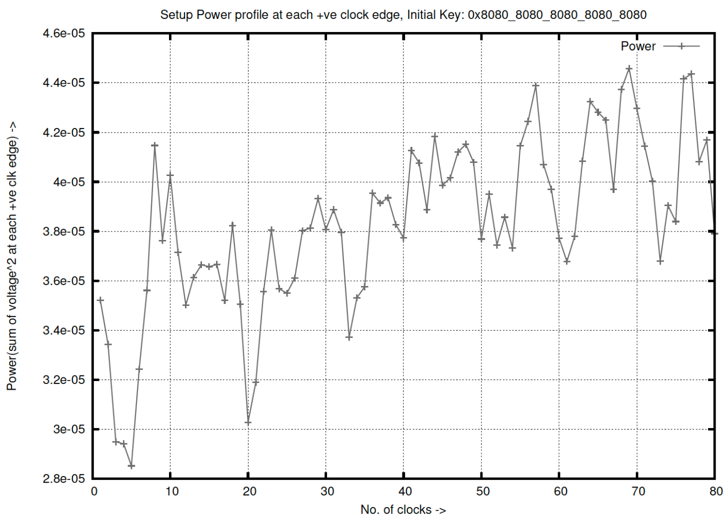


Linear Feedback Shift Register

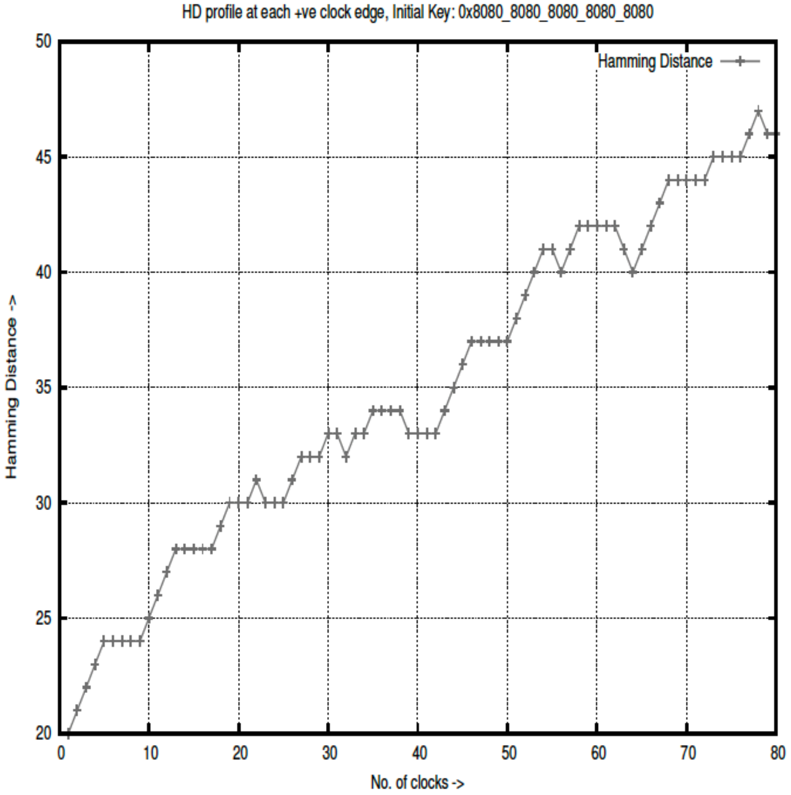


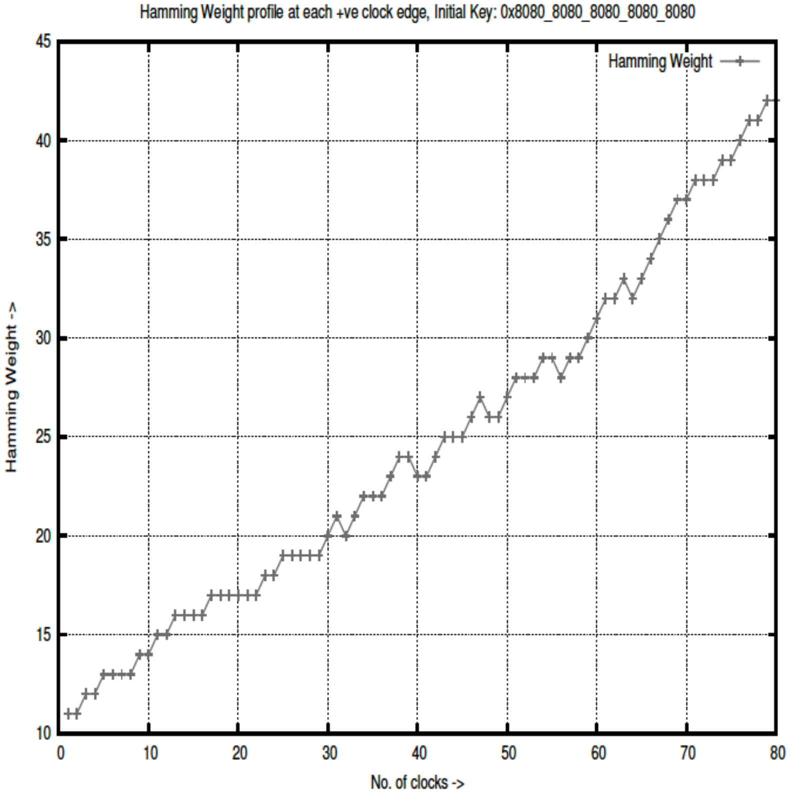
Actual Trace from an FPGA

Differential Power Analysis



Actual Trace from a





Actual Trace from an FPGA

Why do Gates Leak?

a	b	y	Energy
0→0	0→0	0→0	$E_{0\to0}$
0→0	0→1	0→0	$E_{0\to0}$
0→0	1→0	0→0	$E_{0\to0}$
0→0	1→1	0→0	$E_{0\to0}$
0→1	0→0	0→0	$E_{0\to0}$
0→1	0→1	0→1	$E_{0\to1}$
0→1	1→0	0→0	$E_{0\to0}$
0→1	1→1	0→1	$E_{0\to1}$
1→0	0→0	0→0	$E_{0\to0}$
1→0	0→1	0→0	$E_{0\to0}$
1→0	1→0	1→0	$E_{1\to0}$
1→0	1→1	1→0	$E_{1\to0}$
1→1	0→0	0→0	$E_{0\to0}$
1→1	0→1	0→1	$E_{0\to1}$
1→1	1→0	1→0	$E_{1\to0}$
1→1	1→1	1→1	$E_{1\to1}$

- Consider an AND gate $y = ab$
- 4 different energy levels due to transition of the gate $E_{0\to0}, E_{0\to1}, E_{1\to0}, E_{1\to1}$
- We estimate $E(q=0)$, and $E(q=1)$ which are the average energy levels when $q=0$, and $q=1$, respectively.
- $E(q=0) = \dfrac{9E_{0\to0} + 3E_{1\to0}}{12}$
- $E(q=1) = \dfrac{3E_{0\to1} + E_{1\to1}}{4}$
- So $E(q=0) \neq E(q=1)$

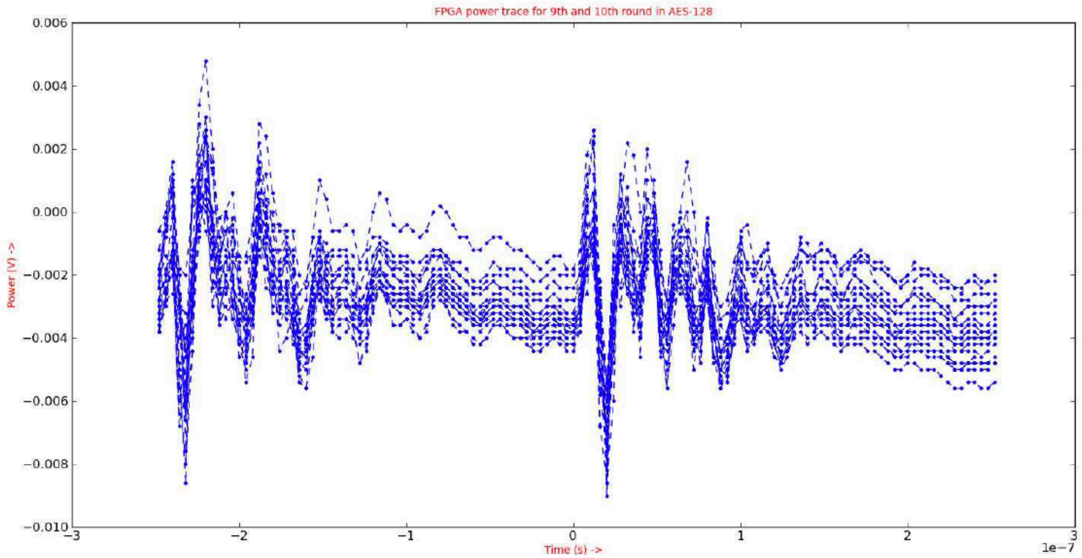
Make Some Noise

Measurement Noise

- **<u>Importance of high-quality measurement</u>**
  - Noise increases the required trace count.

- These fluctuations are due to electrical noise, caused due to power supply, clock generator, conduction and radiation emissions from the components

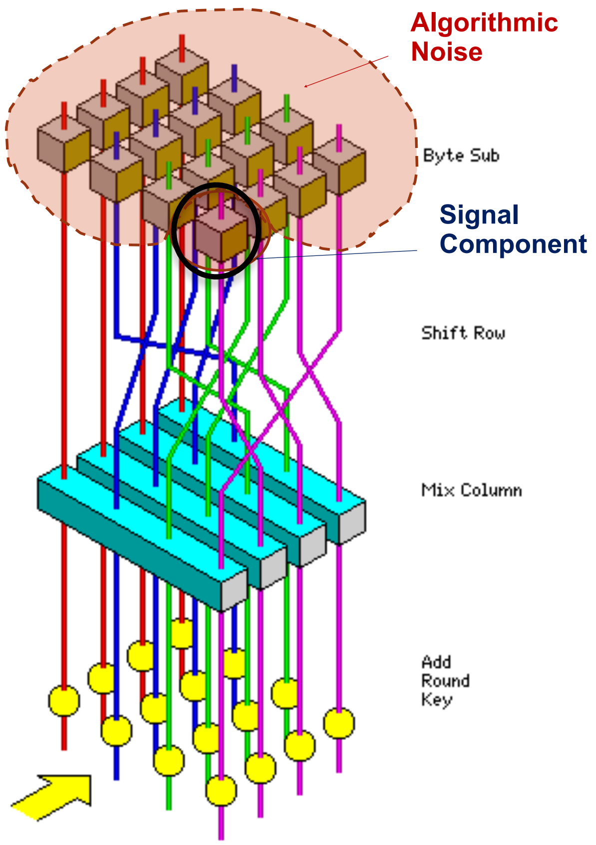
AES-128 Encryption with same plaintext and key, but resulting in different power traces.



Simple Power Model. *Let t denote the time, and $\mathcal{N}(t)$ be a normal distributed random variable which represents the noise components. Let $f(g,t)$ denote the power consumption of gate g at the time t. Then a simplified power model for the power consumption is the function*

$$P(t) = \sum_{g} f(g,t) + \mathcal{N}(t)$$

Algorithmic Noise



Fortunately, our statistical attack can handle both

- Algorithmic or switching noise occurs because of contributions of logic cells to the power consumption, which are not under attack.
- The power trace corresponds to the total power consumption of the circuit.
- However, in the attack we target only a small part (see black circle in fig.) to reveal a portion of the key.
- The power consumption from all the other parts (see blue shaded portion in fig.) form the algorithmic noise.
- This would be more in a parallel implementation, compared to a serialized implementation.

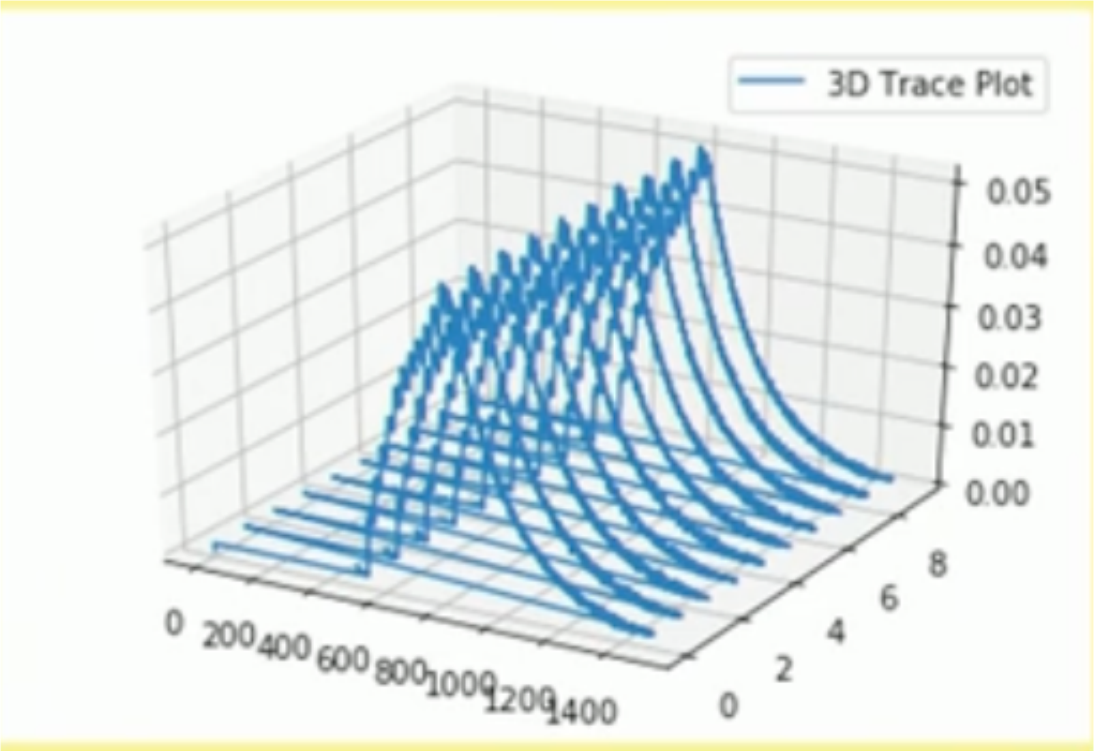
Statistical Analysis of Power Traces

- **Fact**: For fixed operation and fixed value, the power consumption is fixed. But there will be some additive noise.
- **Fortunately, this noise is Gaussian.**
  - Let us assume that the mean is μ and variance is σ^2
  - Noise mainly results in the variance…
- **Recall**: The gaussian probability distribution function

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\frac{(x-\mu)^2}{\sigma^2}} \qquad x \in \mathbb{R}$$

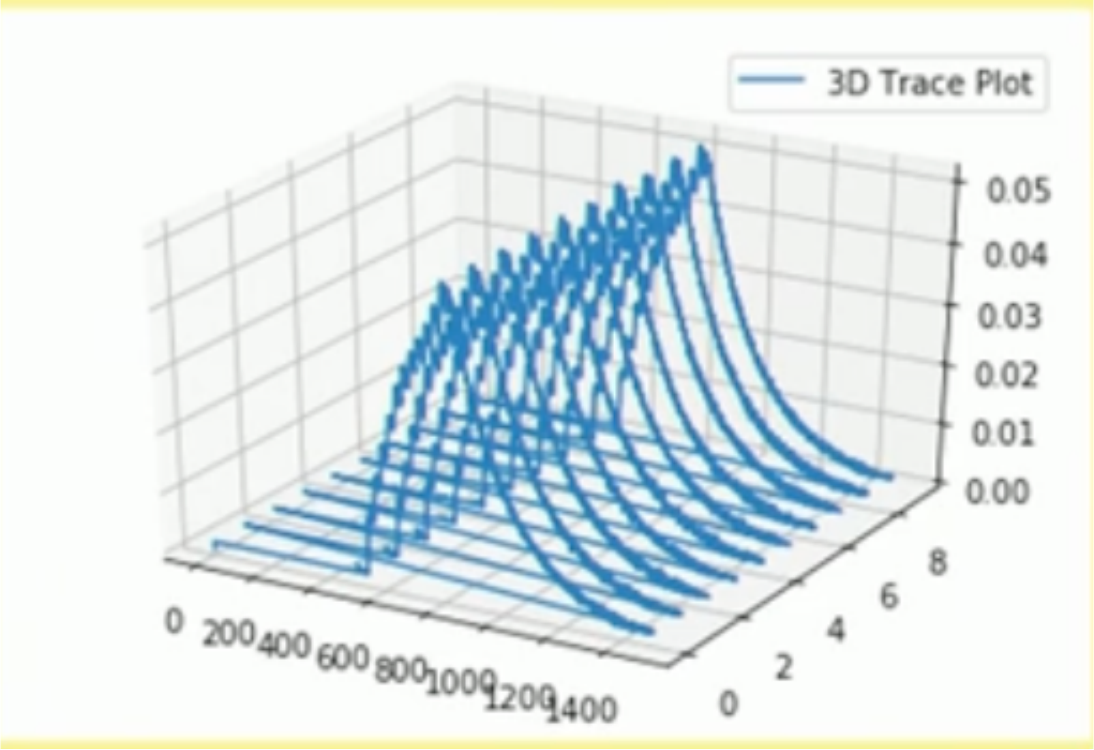
$$X \sim \mathcal{N}(\mu, \sigma)$$

- If $\mu = 0,\ \sigma = 1$, we call it a standard normal distribution.
- The CDF of standard normal is denoted as $\phi(x)$.



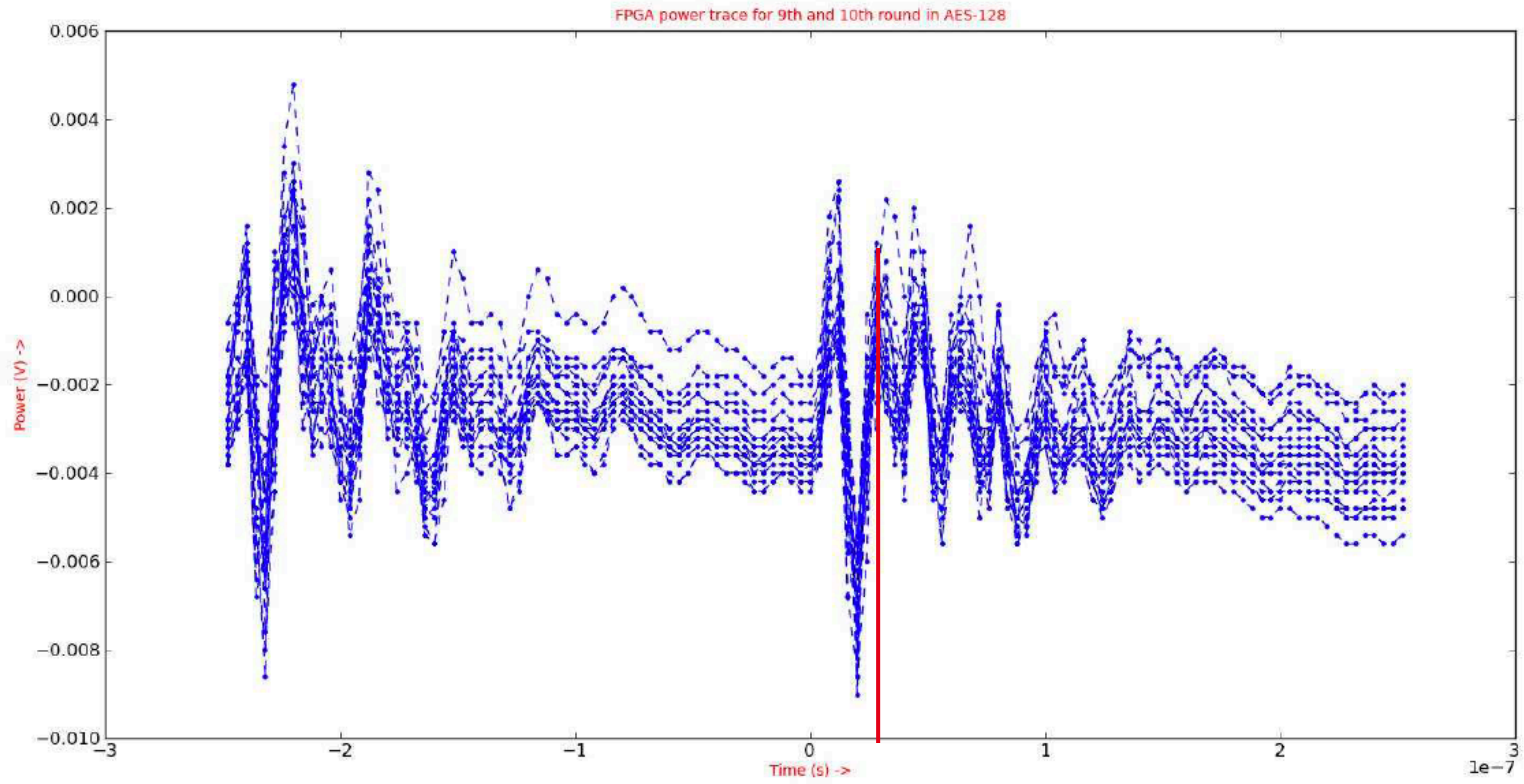
Statistical Analysis of Power Traces

- **Recall**: Sampling distribution
  - An *unbiased estimator* for μ is $\bar{x}$ — the sample mean
  - An unbiased estimator for σ^2 is s^2 — the sample variance (with n-1 correction factor, of course)
  - **Unbiased estimator**: The expected value of the estimator is equal to the true value of the parameter; e.g. $E(\bar{x}) = \mu$



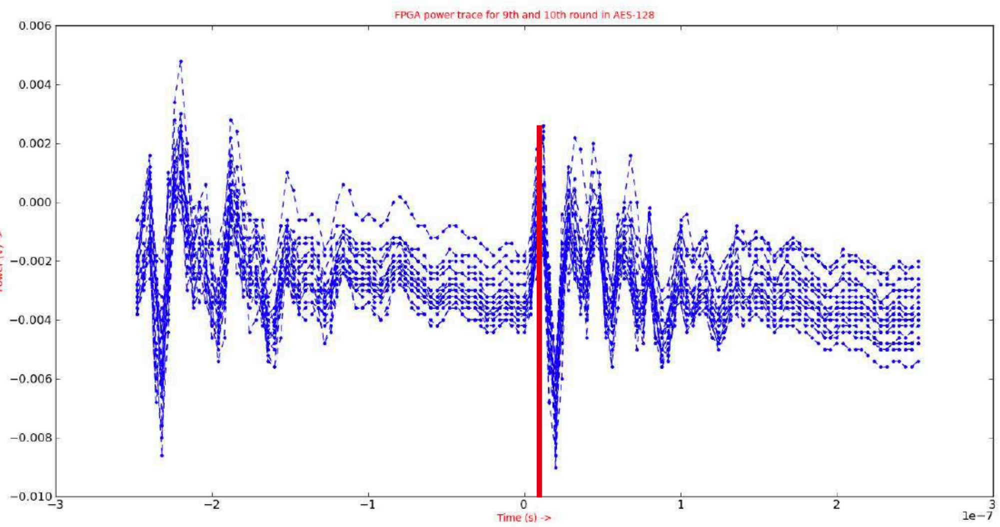
Statistical Analysis of Power Traces

-



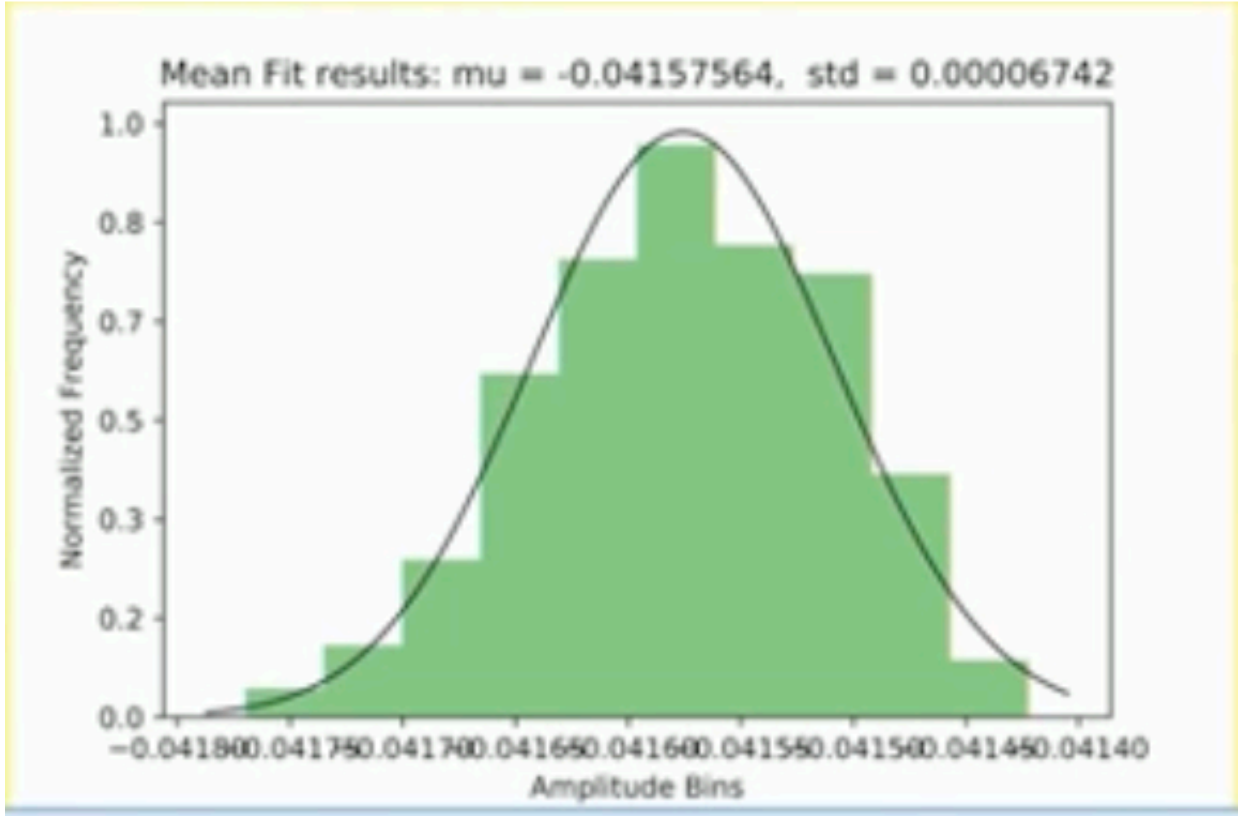
Statistical Analysis of Power Traces

- **A point in the power trace is represented as:**
  - $$P(t) = \sum_{g} f(g,t) + \mathcal{N}(\mu, \sigma, t)$$
- **POI (point of interest) in the trace:** The point(s) in the trace which are the best for distinguishing the correct key from the wrong one.



Simple Power Model. *Let t denote the time, and $\mathcal{N}(t)$ be a normal distributed random variable which represents the noise components. Let $f(g,t)$ denote the power consumption of gate g at the time t. Then a simplified power model for the power consumption is the function*

$$P(t) = \sum_{g} f(g,t) + \mathcal{N}(t)$$



Histogram of a point in power trace for same pt and key