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

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

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

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

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

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

- 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 = 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

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

- 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

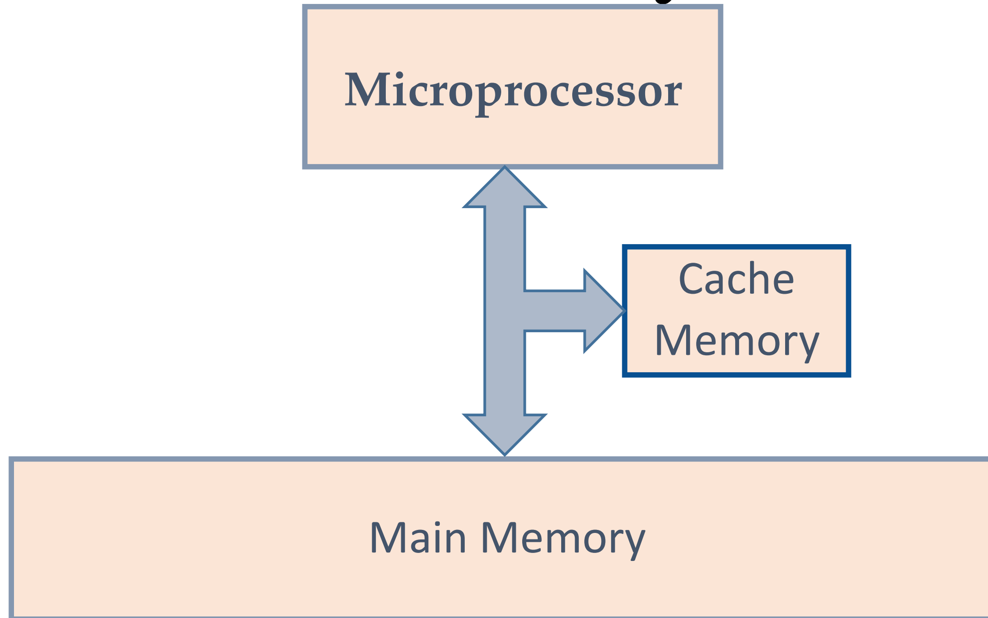
```
R0 = te0[b0]
R1 = te0[b4]
R2 = te0[b8]
R3 = te0[b12]
```

```
// 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

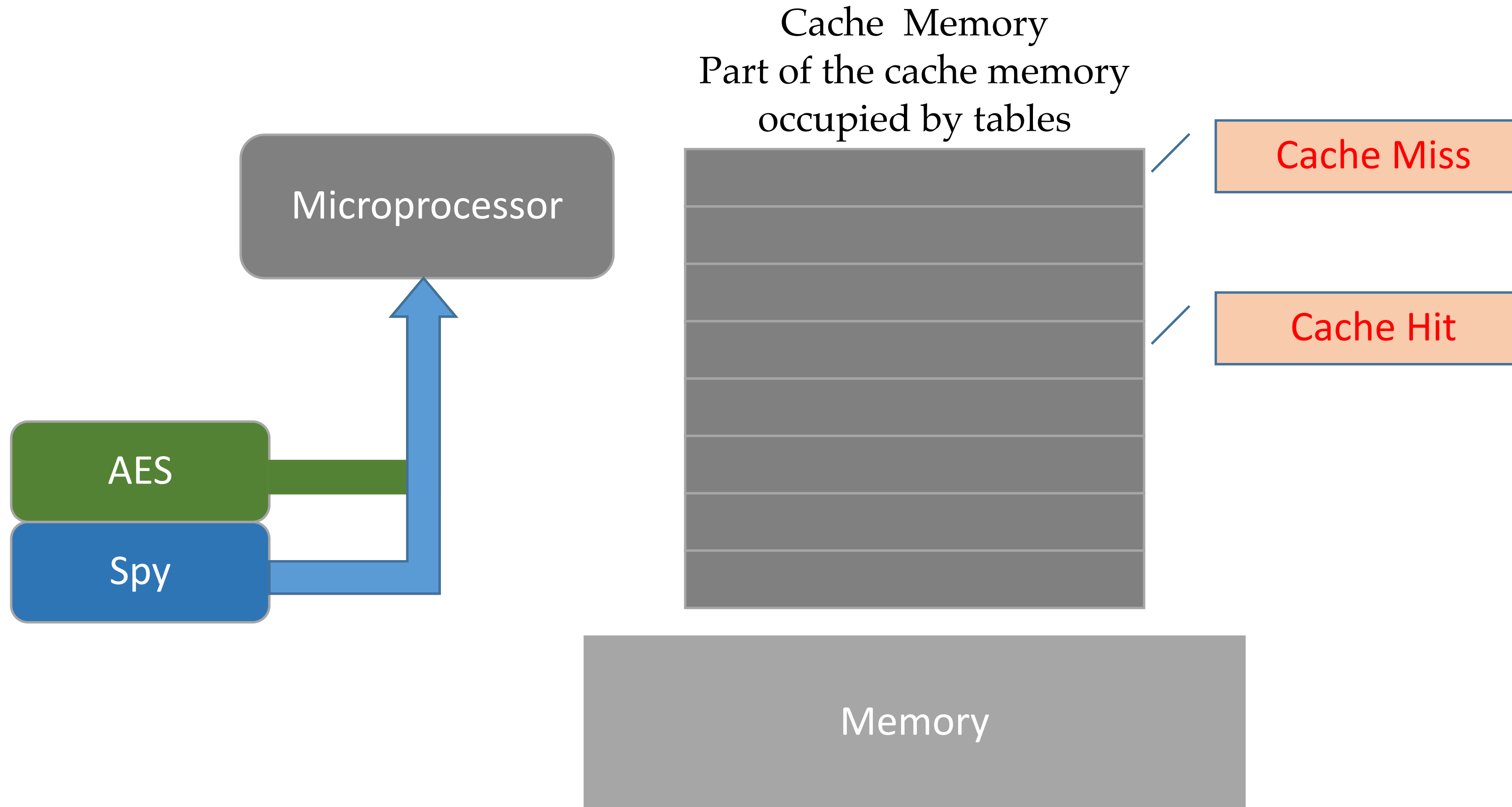


- If there is a *Cache Hit*
  - Access time is less
  - Power Consumption is less

- 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
- Idea: 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

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

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

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

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

- 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…

- 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:

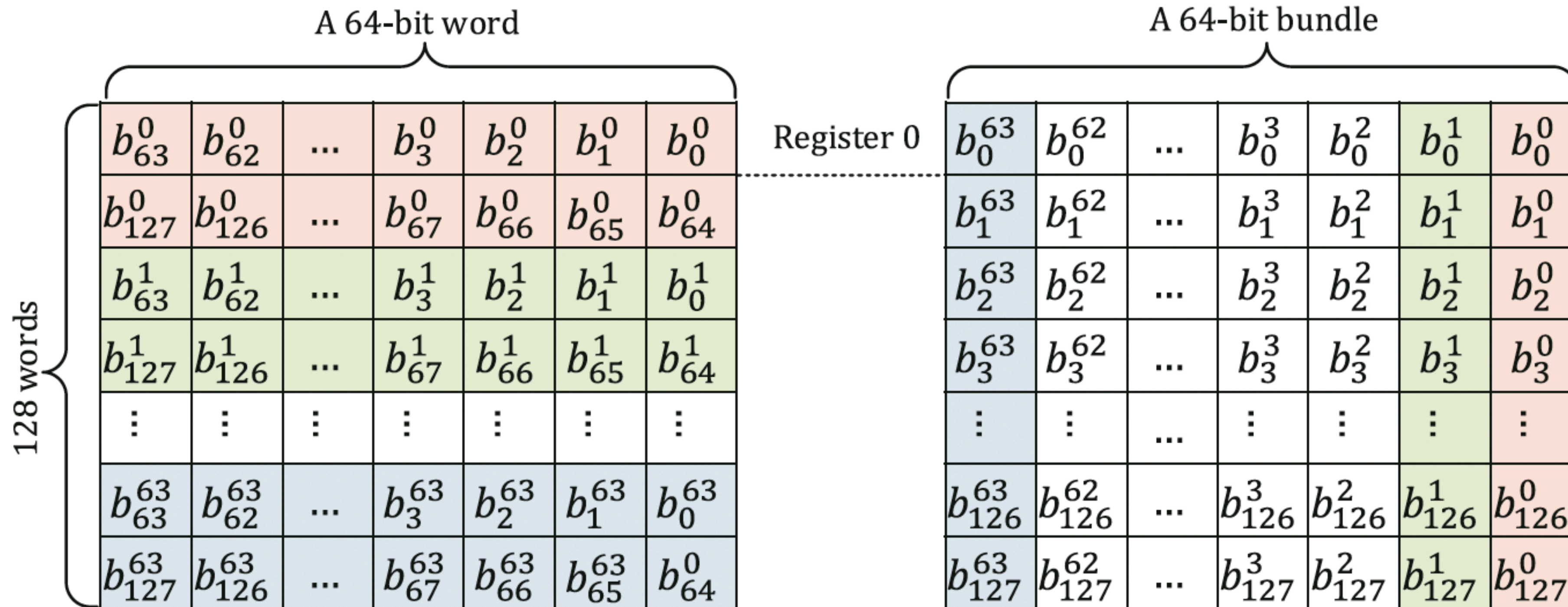
```
and t1, x1, x4
and t2, t1, x2
and t3, t1, x3
xor t4, t2, t3
```

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

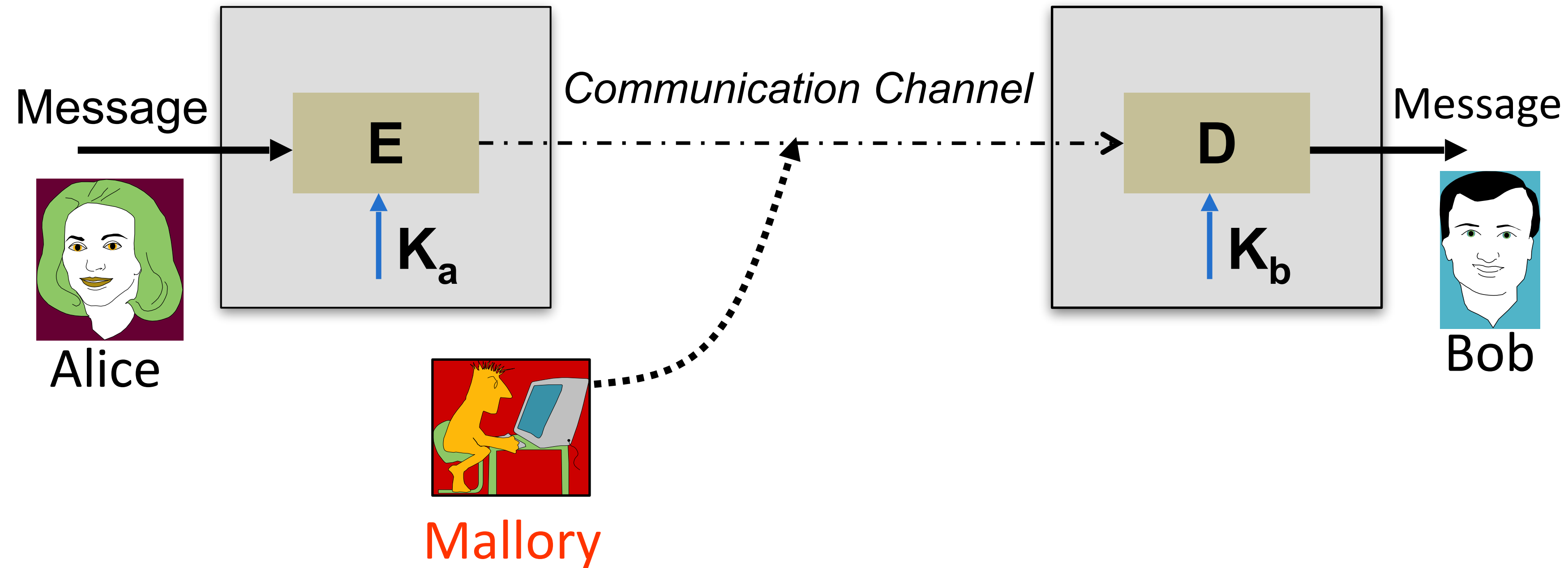


(a) Original storage matrix

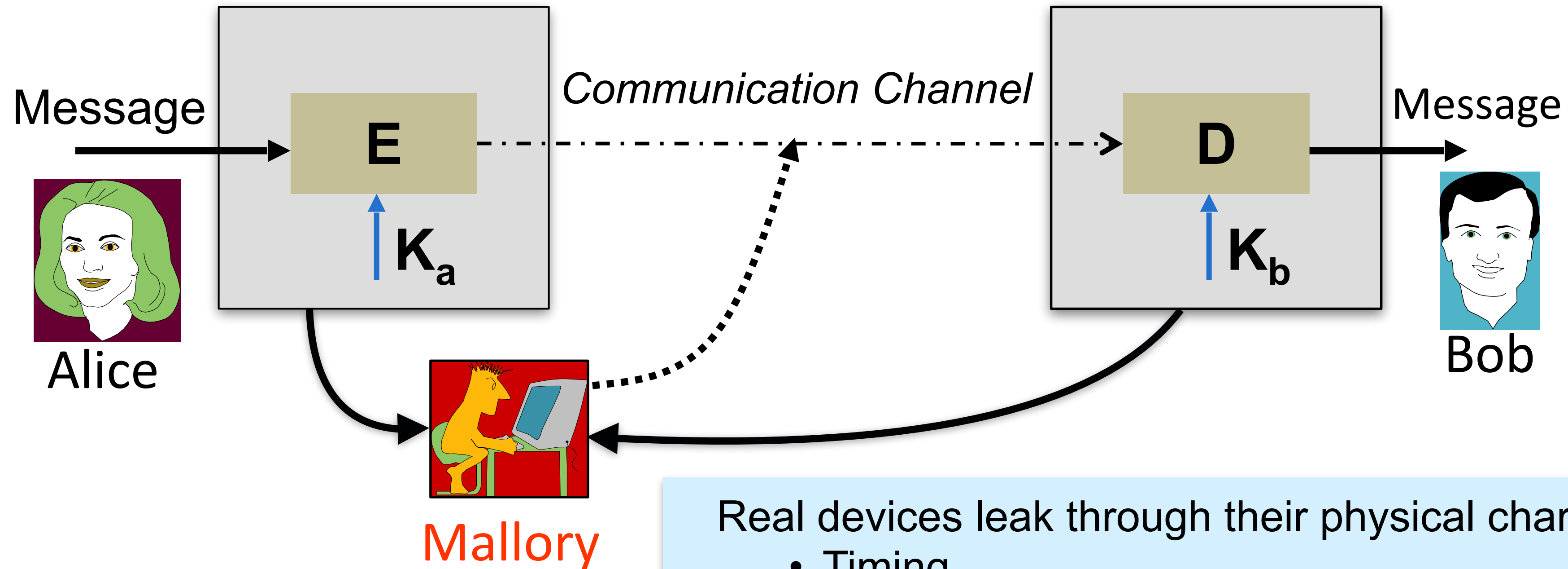
(b) Bit-slice storage matrix

Side Channel Attacks

Why do Cryptographers Need Engineers?



Cryptographic Security: Real World



Strong cryptographic algorithms are only the beginning

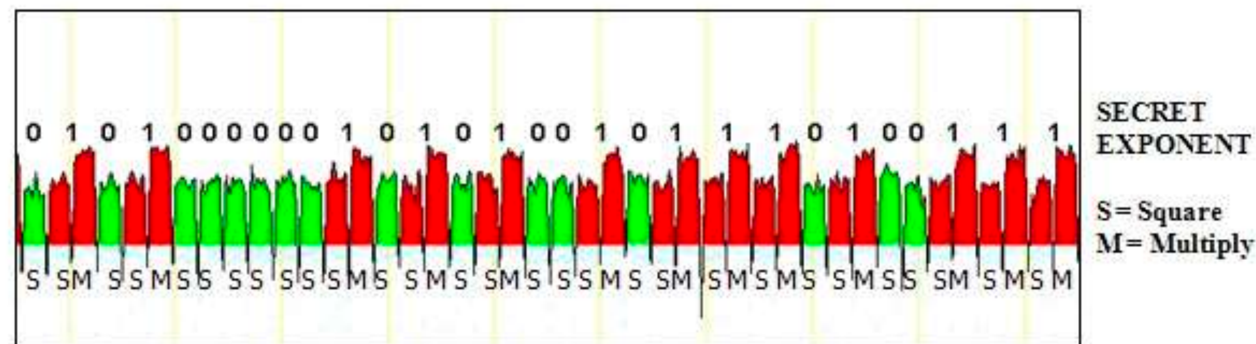
Real devices leak through their physical characteristics

- 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
- 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

Encryption

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

Decryption

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

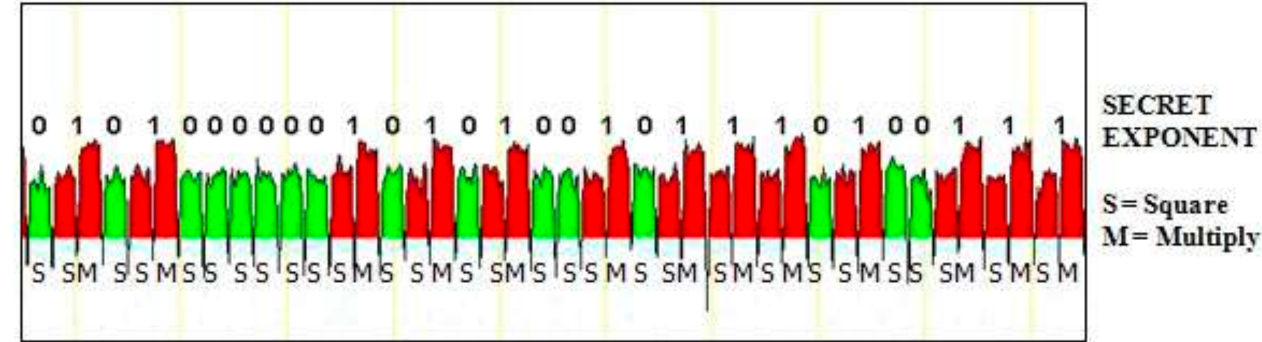
Square and Multiply Algorithm

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

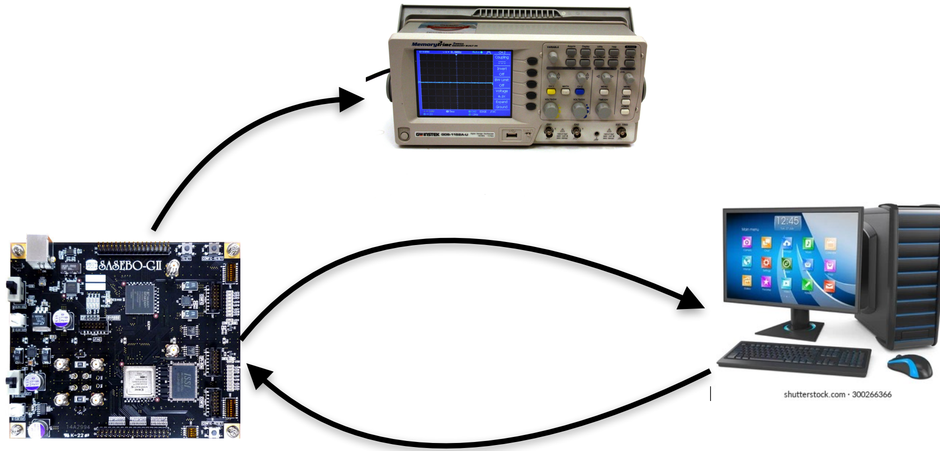
```
1. convert e to binary: k_s k_{s-1}...k_1 k_0
2. b = 1;
3. for (i=s; i>=o; i--)
4. { b = b*b (mod n);
5.   if (k_i == 1)
6.     b = b * a (mod n)
7. }
8. return b;
```

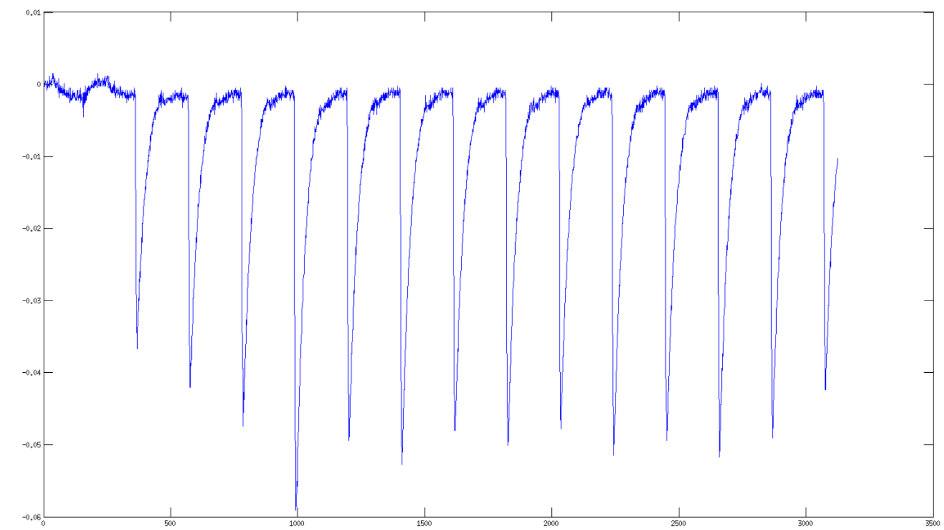
Side-Channel Attacks (SCA)

- The physical channels are correlated with the information being processed
- 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

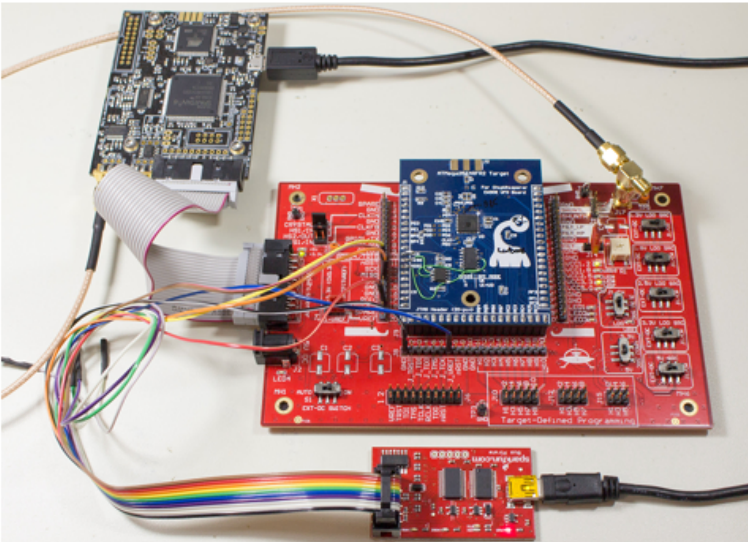


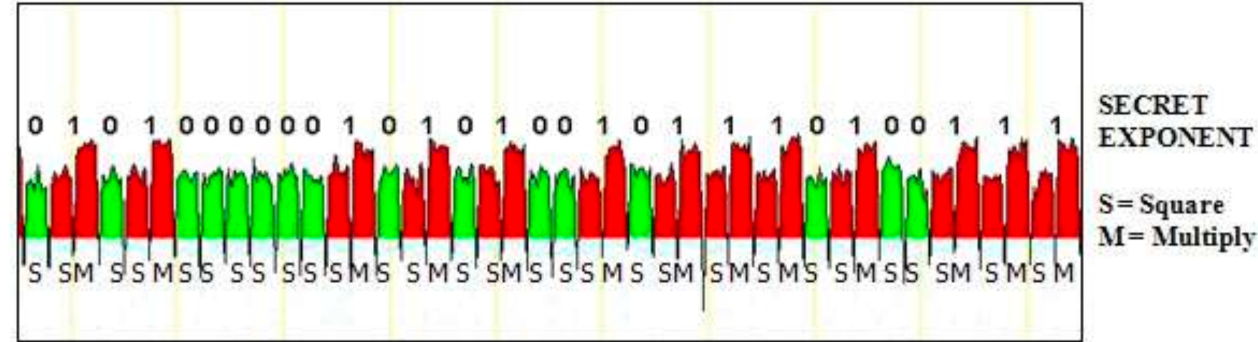


Source: Testbed for Side Channel analysis and security evaluation

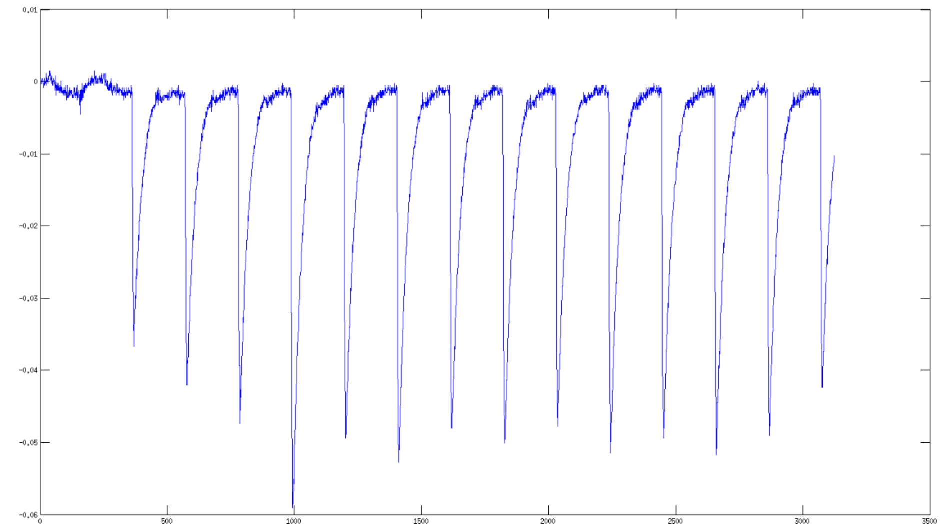
Side-Channel Attacks (SCA)

- 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

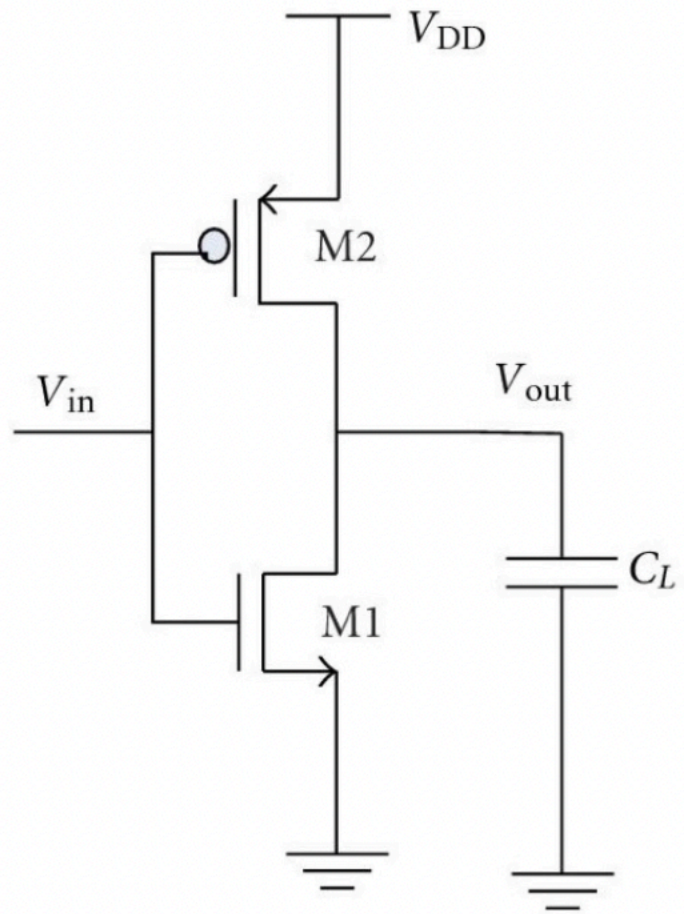
Side-Channel Vs. Classical Cryptanalysis

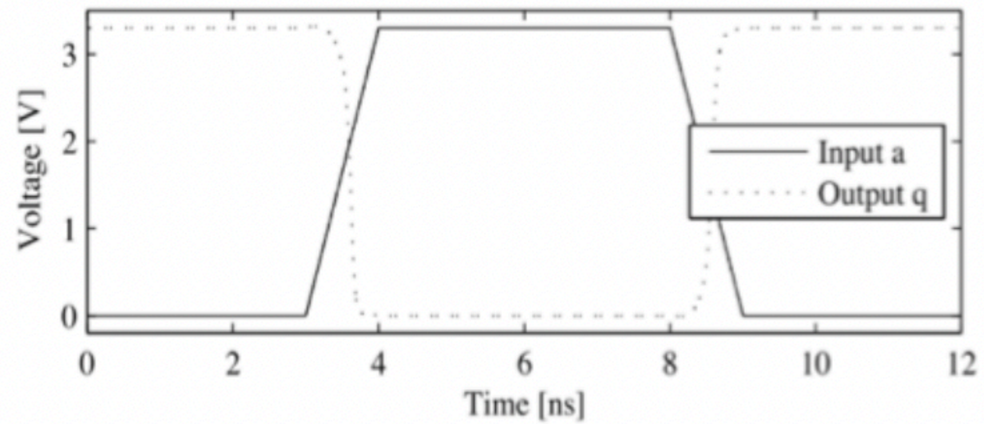
- Cryptanalysis: Purely mathematical
  - Take example of RSA/ECC/PQC
  - Cryptanalysis means, you only have access to plaintext, ciphertext — a lot of them
  - You have to
    - Find the key
      - Maybe you need to solve the underlying hard problem in some (mathematical) way.

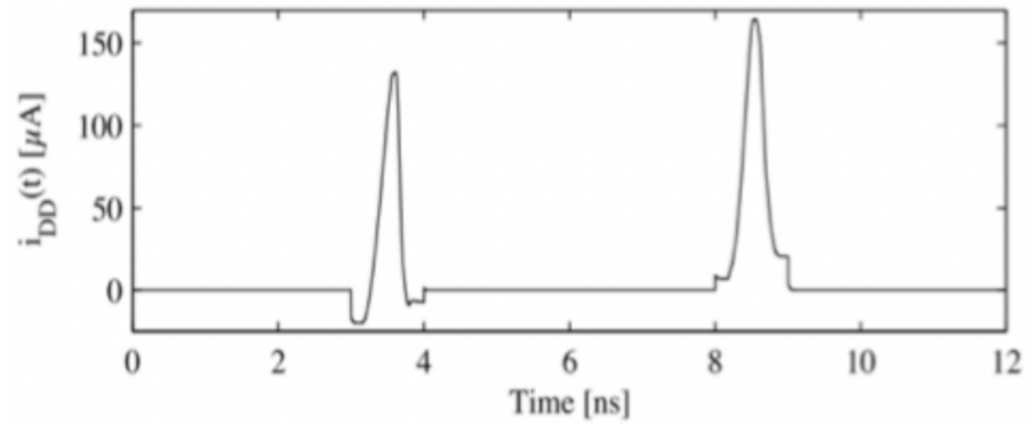
Side-Channel Vs. Classical Cryptanalysis

- 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

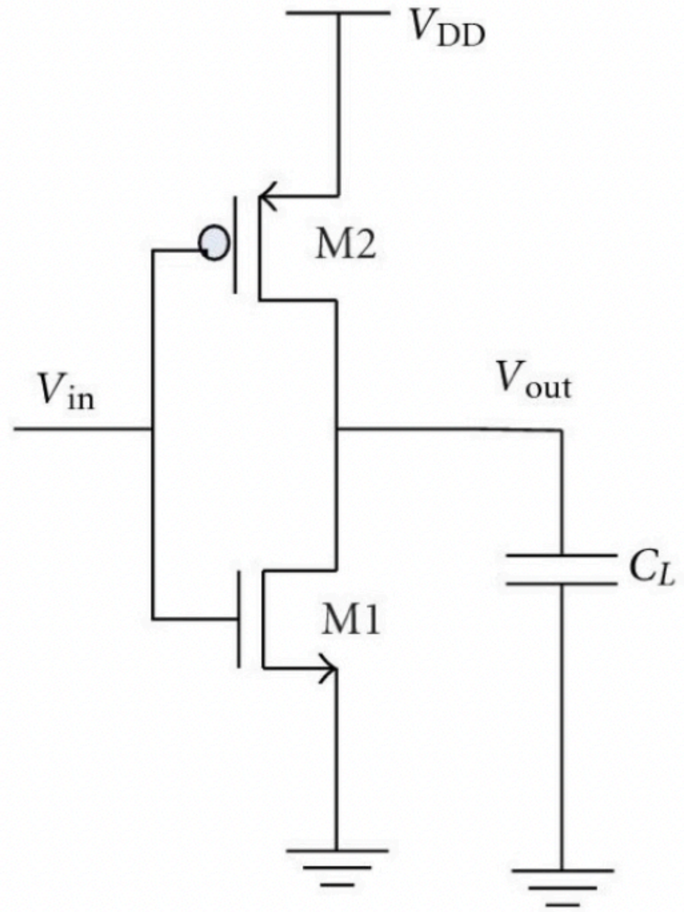
The Root Cause

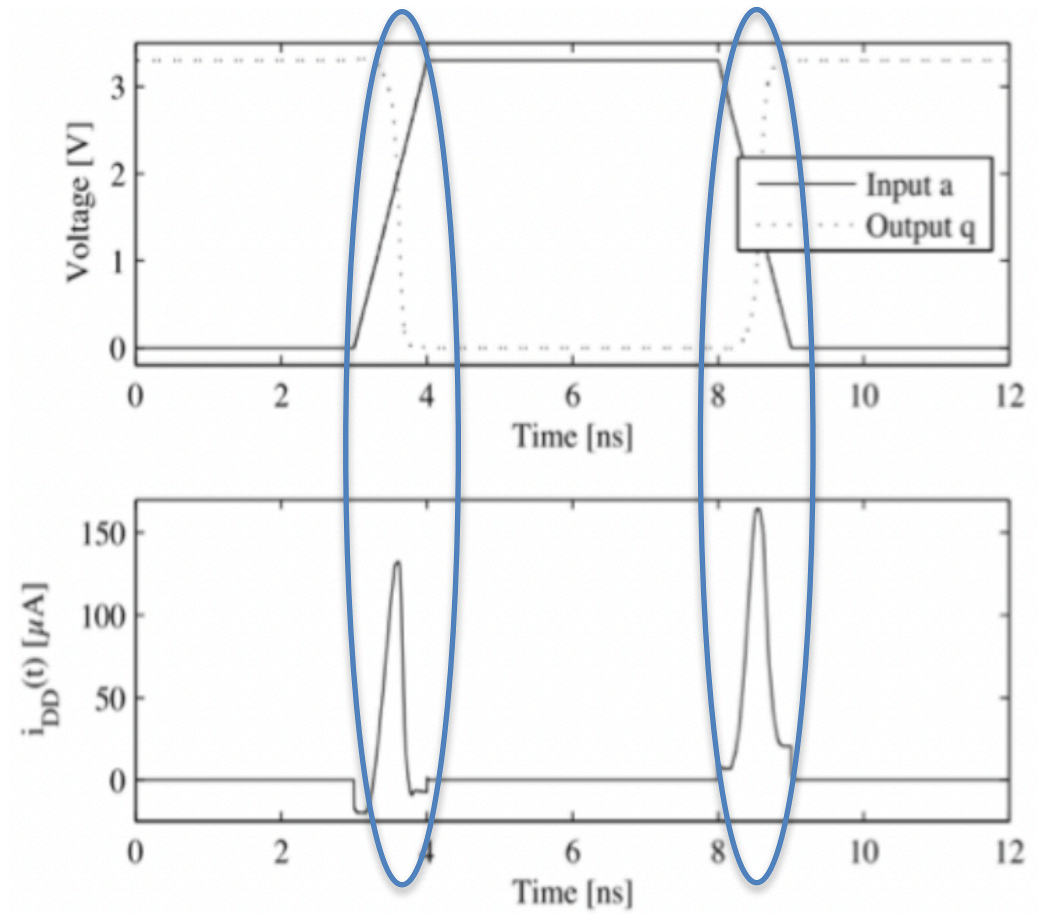






The Root Cause

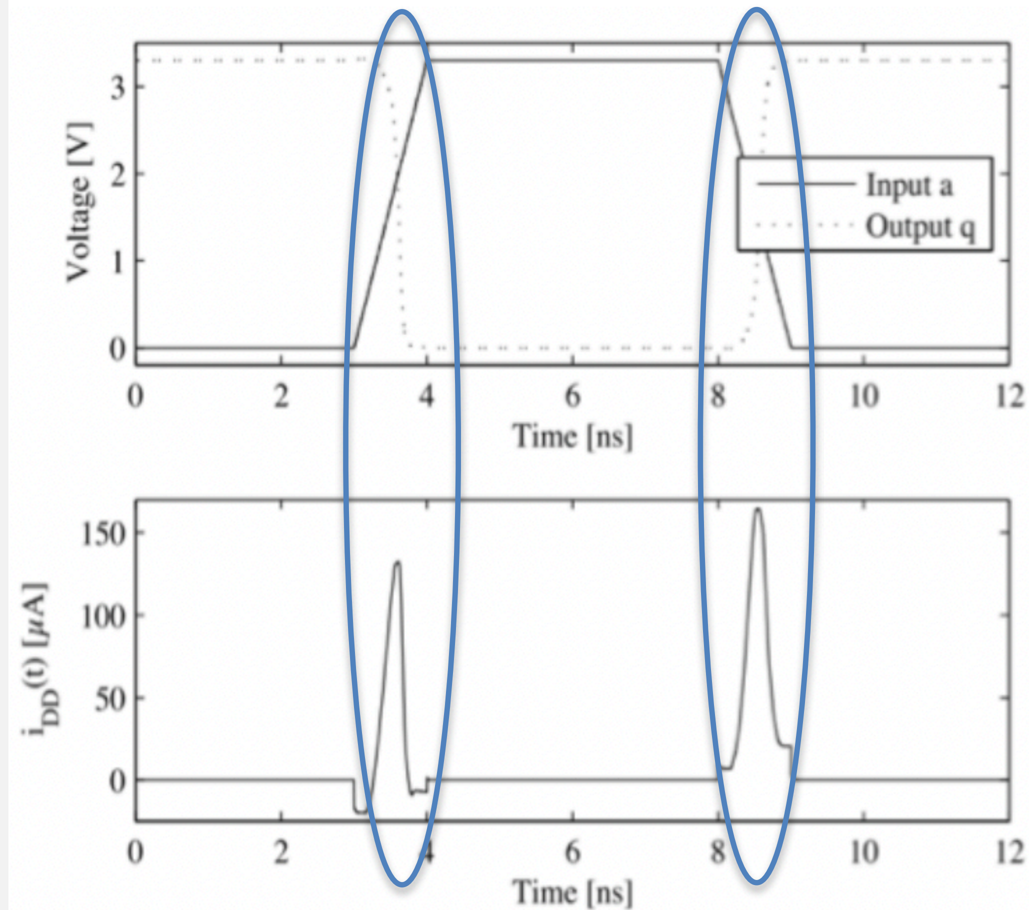




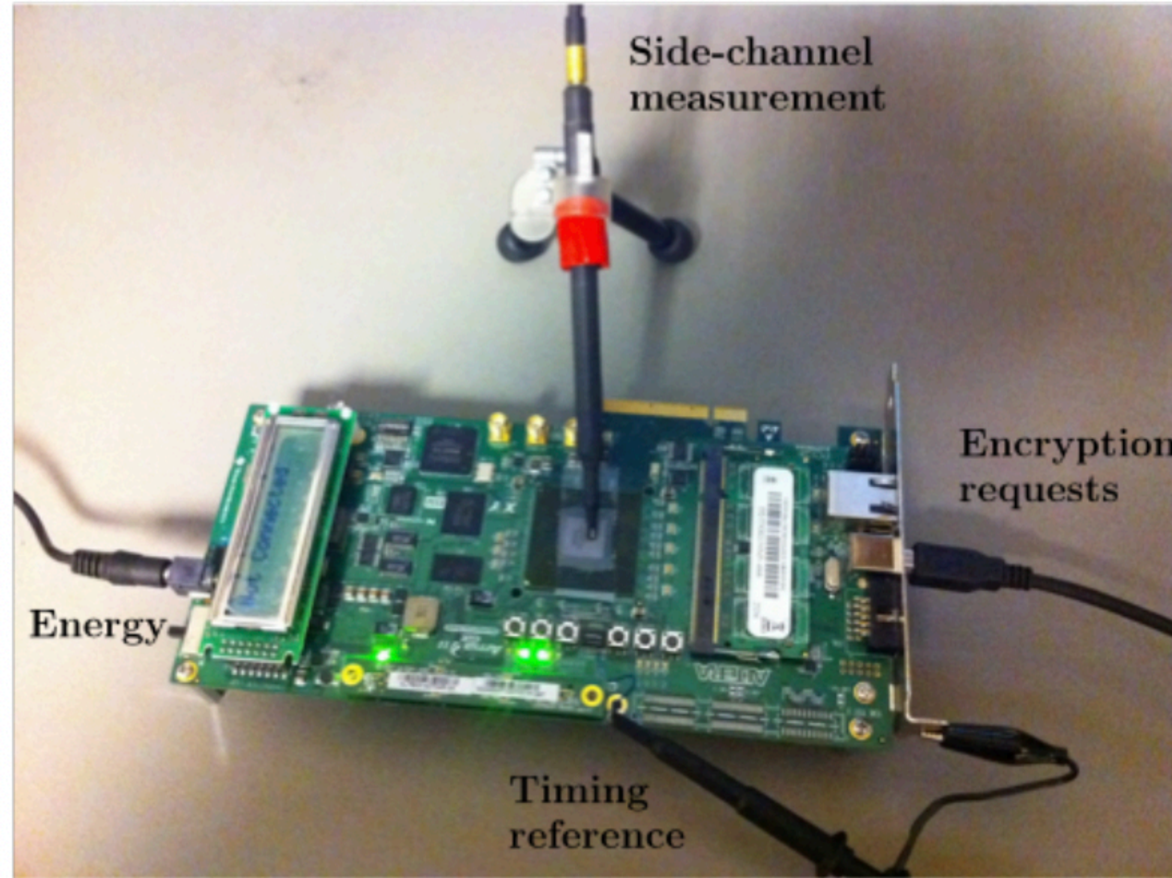
The Root Cause

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**