Lecture 8: Timing attacks

- Homework 4 has been posted
- There is no lecture on Tuesday 2/18
- In-class test on Thursday 2/20, covering the following material:
  - All lectures up to (and including) this one
  - All Friday discussion sections
  - All concepts within the first 4 homework assignments
  - All required textbook reading
- Nicolas' discussion section tomorrow will be test preparation

Crypto = Scientific field at intersection of many disciplines

Algorithms

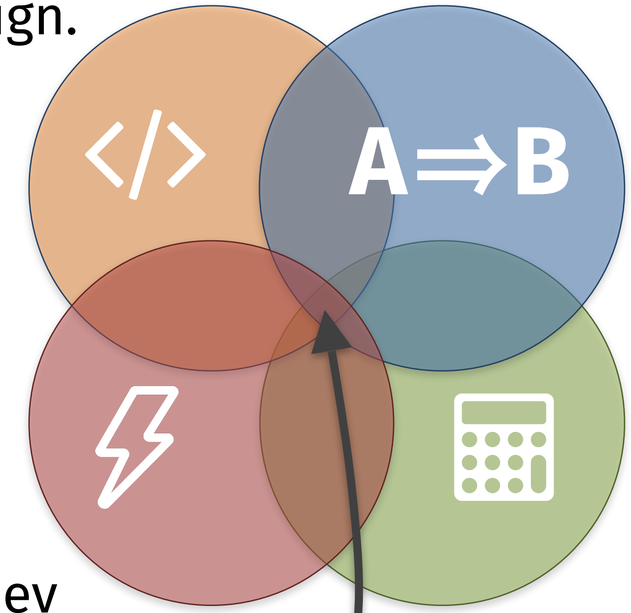
Known for cipher design.

Primarily found in European academia.

Complexity theory

Known for reductions.

Primarily found in American academia.



Engineering

Known for software dev and side channel attacks.

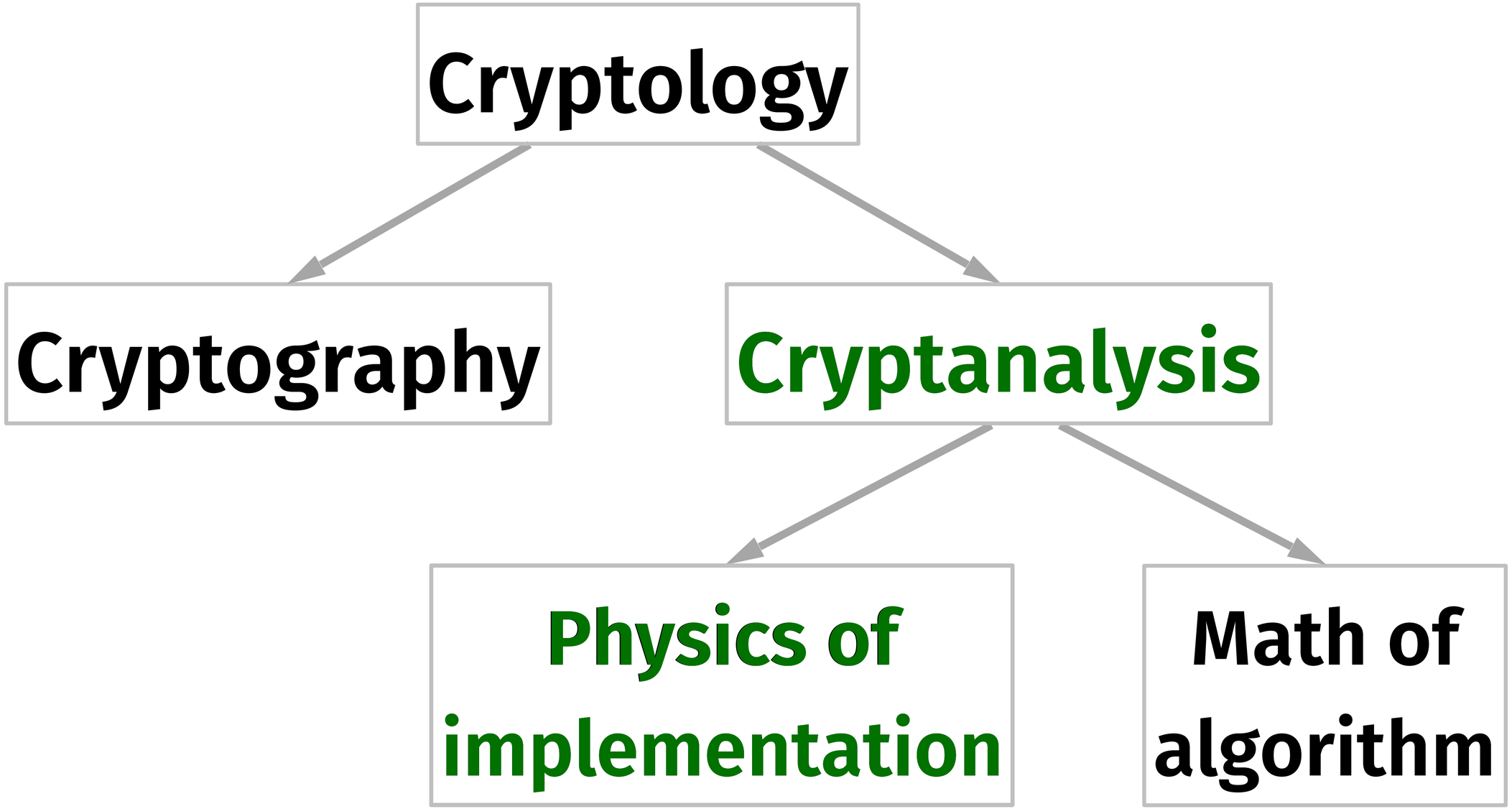
Primarily found in industry.

Mathematics

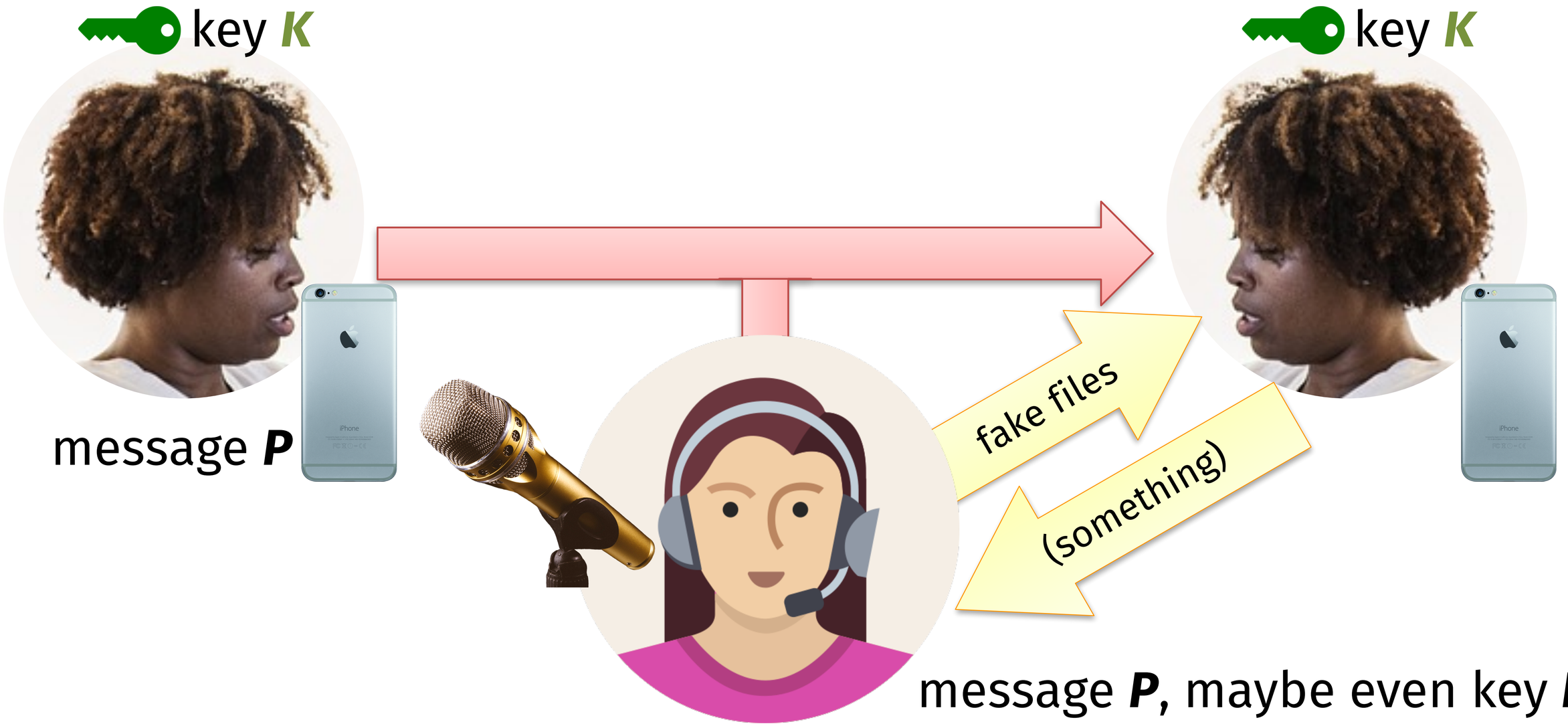
Known for cryptanalysis.

Primarily found in government.

This class

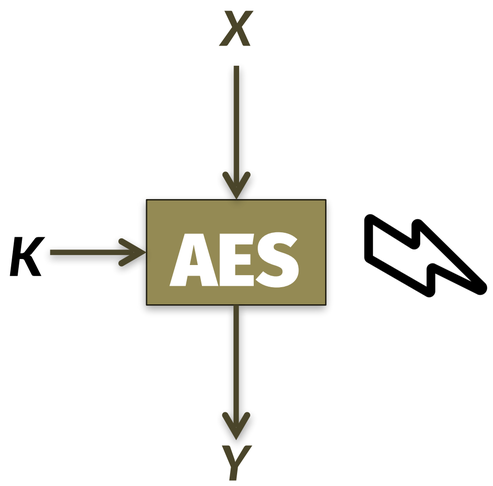

Cryptology
Cryptography
Cryptanalysis
Physics of implementation
Math of algorithm

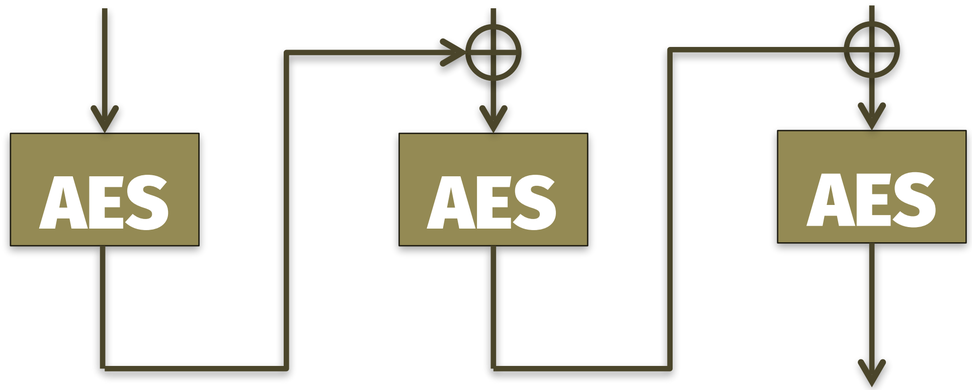
Breaking data confidentiality



Side channel attacks on crypto implementations

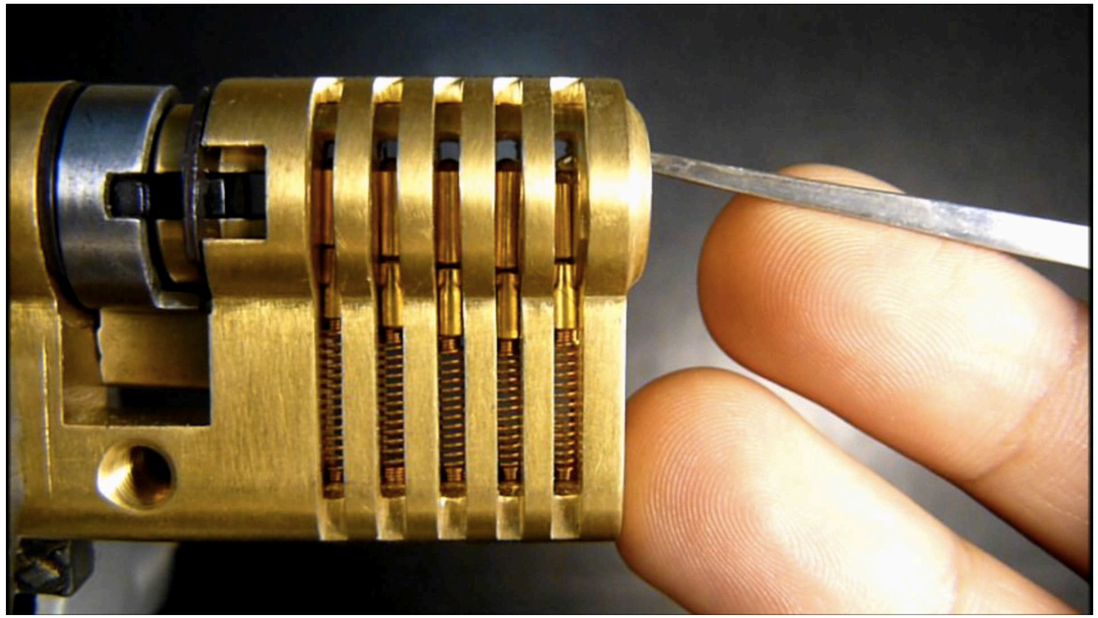
- Crypto security definitions ensure that the output is “harmless”
- But, crypto implementations can reveal more than its desired outputs! These *side channels* of information weren’t captured in our definitions
- Focus for this week: side channels on AES $\Rightarrow$ and thus any AES mode



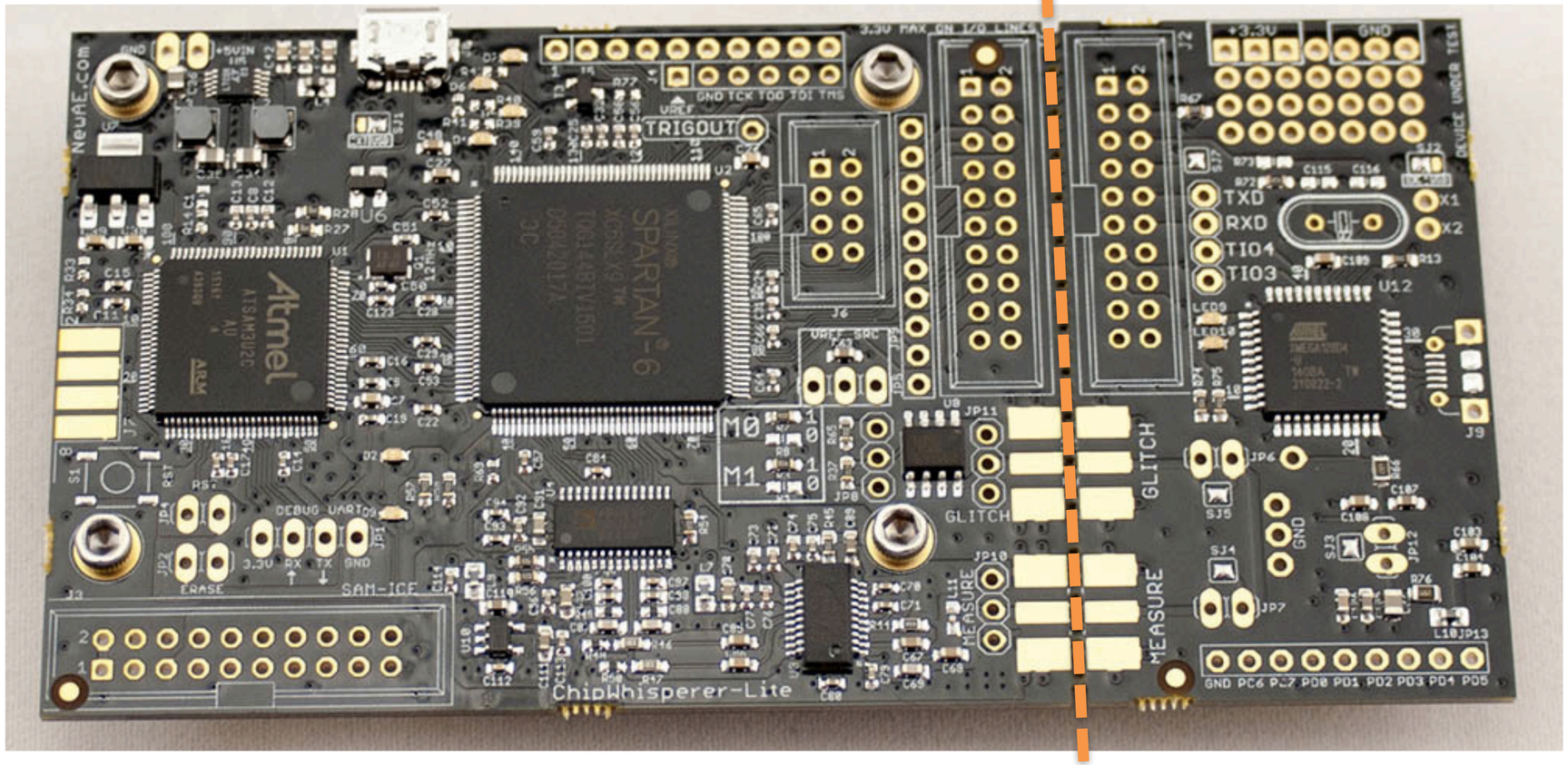


Divide and conquer

- Break 1 byte of the message or key at a time
- For each byte: guess all 256 values and check which works
- (Think: how you see crypto broken in any Hollywood movie)



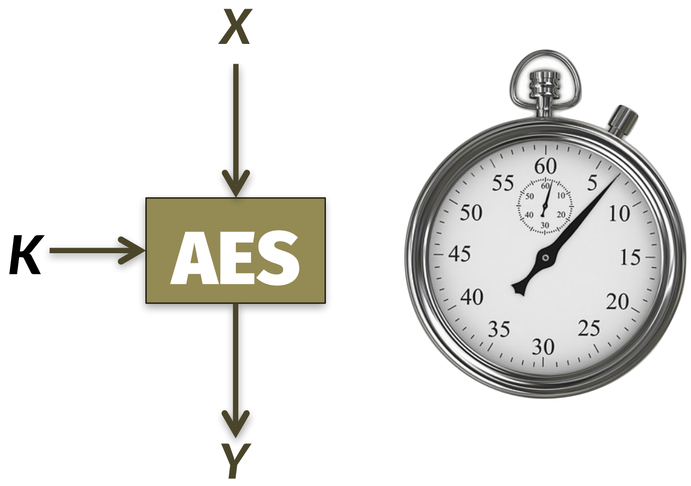
Last time: Power analysis of AES in hardware



Mallory: oscilloscope to measure power

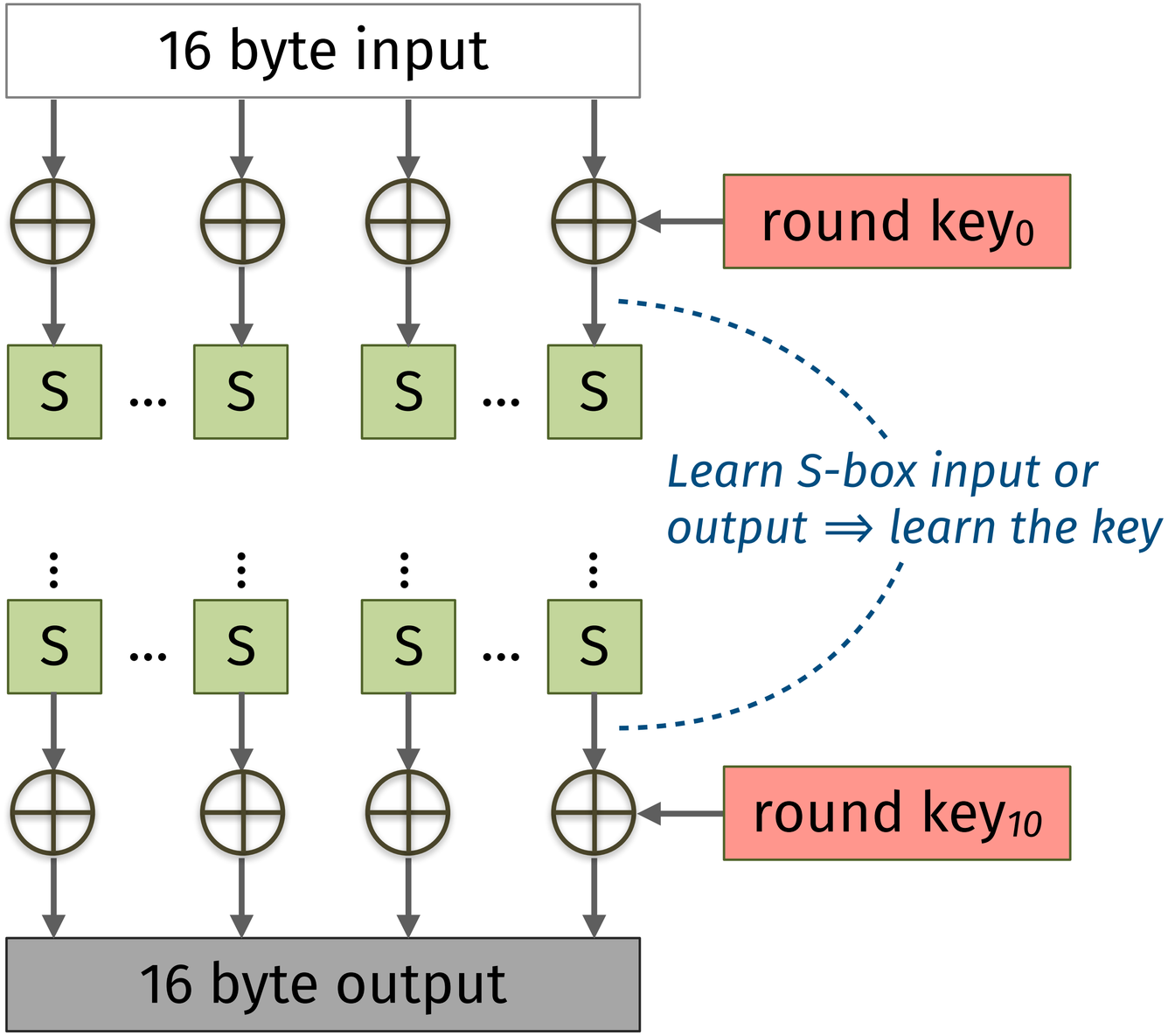
Alice: FPGA that runs AES

Today: Timing attacks on AES in software

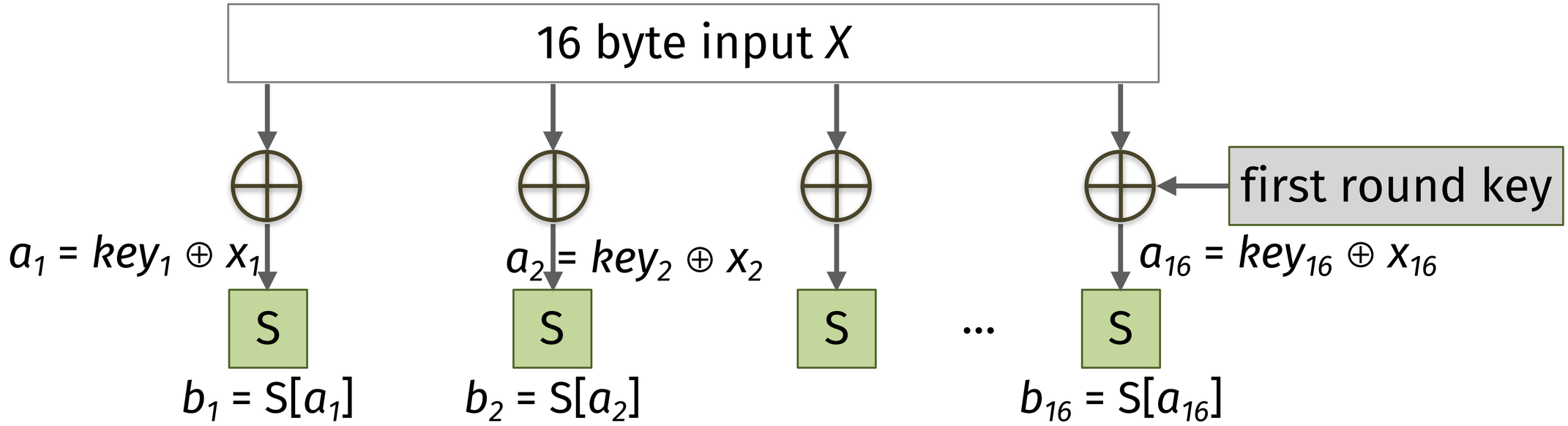


Question: what might affect the runtime of AES?

Answer: the S-box!
Let's look at simplified first and last rounds of AES

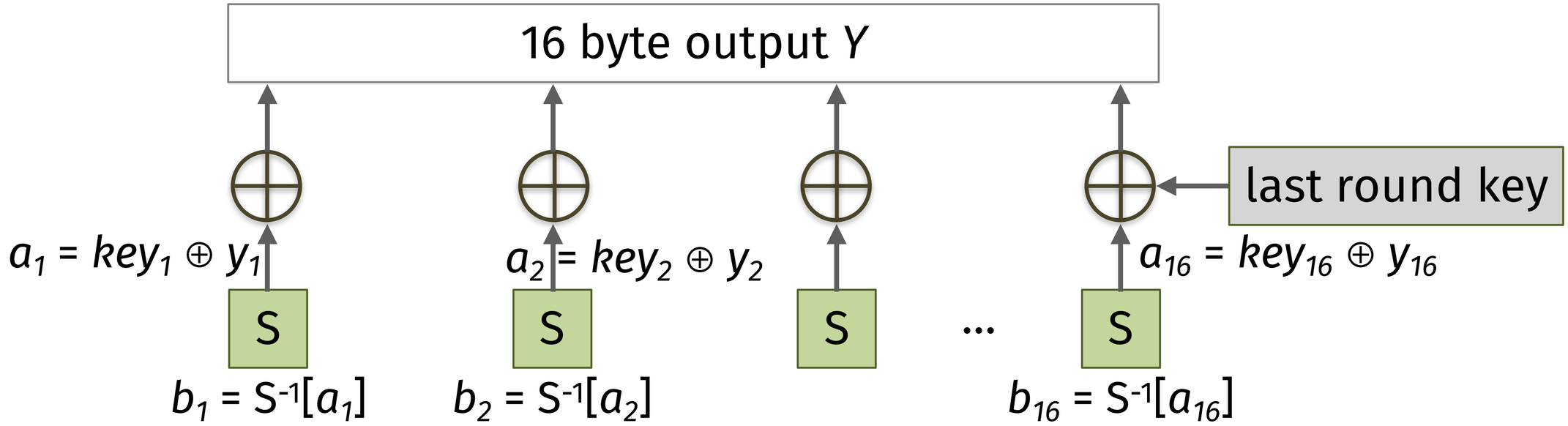


Simplified picture of first round of AES



- The first round of AES begins with xor of key followed by S-box lookups
- Let *A* and *B* denote the 16-byte state before/after the first round S-box
- Claim: if Eve knows input *X* along with *A* (or *B*), she can recover the secret key

Simplified picture of last round of AES



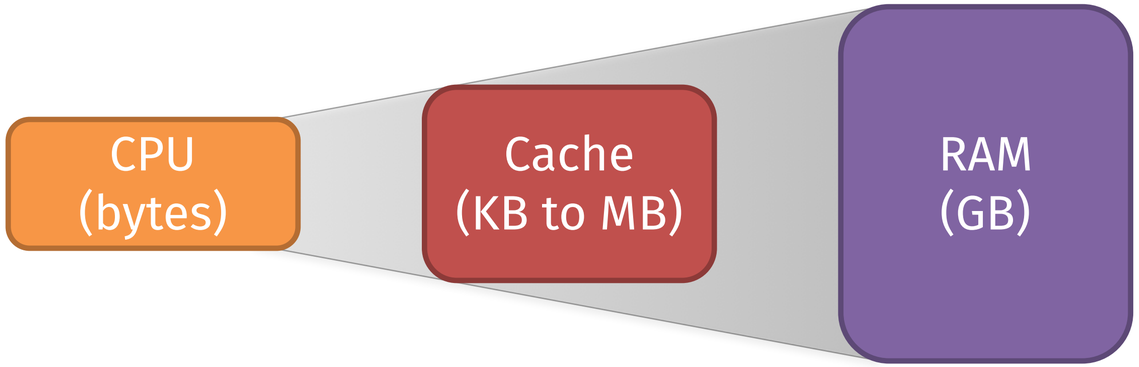
- Similar attack on last round: if Eve knows Y and A/B, can recover key
- There is a bijection between the first and last round keys of AES
- (Note: picture omits the final round ShiftRows, which permutes bytes)

AES code has table lookups

```
static const u8 Te4[256] = {
    0x63U, 0x7cU, 0x77U, 0x7bU, 0xf2U, 0x6bU, 0x6fU, 0xc5U,
    0x30U, 0x01U, 0x67U, 0x2bU, 0xfeU, 0xd7U, 0xabU, 0x76U,
    0xcaU, 0x82U, 0xc9U, 0x7dU, 0xfaU, 0x59U, 0x47U, 0xf0U,
    0xadU, 0xd4U, 0xa2U, 0xafU, 0x9cU, 0xa4U, 0x72U, 0xc0U,
    0xb7U, 0xfdU, 0x93U, 0x26U, 0x36U, 0x3fU, 0xf7U, 0xccU,
    0x34U, 0xa5U, 0xe5U, 0xf1U, 0x71U, 0xd8U, 0x31U, 0x15U,
    0x04U, 0xc7U, 0x23U, 0xc3U, 0x18U, 0x96U, 0x05U, 0x9aU,
    0x07U, 0x12U, 0x80U, 0xe2U, 0xebU, 0x27U, 0xb2U, 0x75U,
    0x09U, 0x83U, 0x2cU, 0x1aU, 0x1bU, 0x6eU, 0x5aU, 0xa0U,
    0x52U, 0x3bU, 0xd6U, 0xb3U, 0x29U, 0xe3U, 0x2fU, 0x84U,
    0x53U, 0xd1U, 0x00U, 0xedU, 0x20U, 0xfcU, 0xb1U, 0x5bU,
    0x6aU, 0xcbU, 0xbeU, 0x39U, 0x4aU, 0x4cU, 0x58U, 0xcfU,
    0xd0U, 0xefU, 0xaaU, 0xfbU, 0x43U, 0x4dU, 0x33U, 0x85U,
    0x45U, 0xf9U, 0x02U, 0x7fU, 0x50U, 0x3cU, 0x9fU, 0xa8U,
    0x51U, 0xa3U, 0x40U, 0x8fU, 0x92U, 0x9dU, 0x38U, 0xf5U,
    0xbcU, 0xb6U, 0xdaU, 0x21U, 0x10U, 0xffU, 0xf3U, 0xd2U,
    0xcdU, 0x0cU, 0x13U, 0xecU, 0x5fU, 0x97U, 0x44U, 0x17U,
    0xc4U, 0xa7U, 0x7eU, 0x3dU, 0x64U, 0x5dU, 0x19U, 0x73U,
    0x60U, 0x81U, 0x4fU, 0xdcU, 0x22U, 0x2aU, 0x90U, 0x88U,
    0x46U, 0xeeU, 0xb8U, 0x14U, 0xdeU, 0x5eU, 0x0bU, 0xdbU,
    0xe0U, 0x32U, 0x3aU, 0x0aU, 0x49U, 0x06U, 0x24U, 0x5cU,
    0xc2U, 0xd3U, 0xacU, 0x62U, 0x91U, 0x95U, 0xe4U, 0x79U,
    0xe7U, 0xc8U, 0x37U, 0x6dU, 0x8dU, 0xd5U, 0x4eU, 0xa9U,
    0x6cU, 0x56U, 0xf4U, 0xeaU, 0x65U, 0x7aU, 0xaeU, 0x08U,
    0xbaU, 0x78U, 0x25U, 0x2eU, 0x1cU, 0xa6U, 0xb4U, 0xc6U,
    0xe8U, 0xddU, 0x74U, 0x1fU, 0x4bU, 0xbdU, 0x8bU, 0x8aU,
    0x70U, 0x3eU, 0xb5U, 0x66U, 0x48U, 0x03U, 0xf6U, 0x0eU,
    0x61U, 0x35U, 0x57U, 0xb9U, 0x86U, 0xc1U, 0x1dU, 0x9eU,
    0xe1U, 0xf8U, 0x98U, 0x11U, 0x69U, 0xd9U, 0x8eU, 0x94U,
    0x9bU, 0x1eU, 0x87U, 0xe9U, 0xceU, 0x55U, 0x28U, 0xdfU,
    0x8cU, 0xa1U, 0x89U, 0x0dU, 0xbfU, 0xe6U, 0x42U, 0x68U,
    0x41U, 0x99U, 0x2dU, 0x0fU, 0xb0U, 0x54U, 0xbbU, 0x16U
```

Source: github.com/openssl/openssl/blob/master/crypto/aes/aes_core.c

Computer caching



Computers *cache* recently-accessed data, assuming that if you wanted it before, then you may want it again

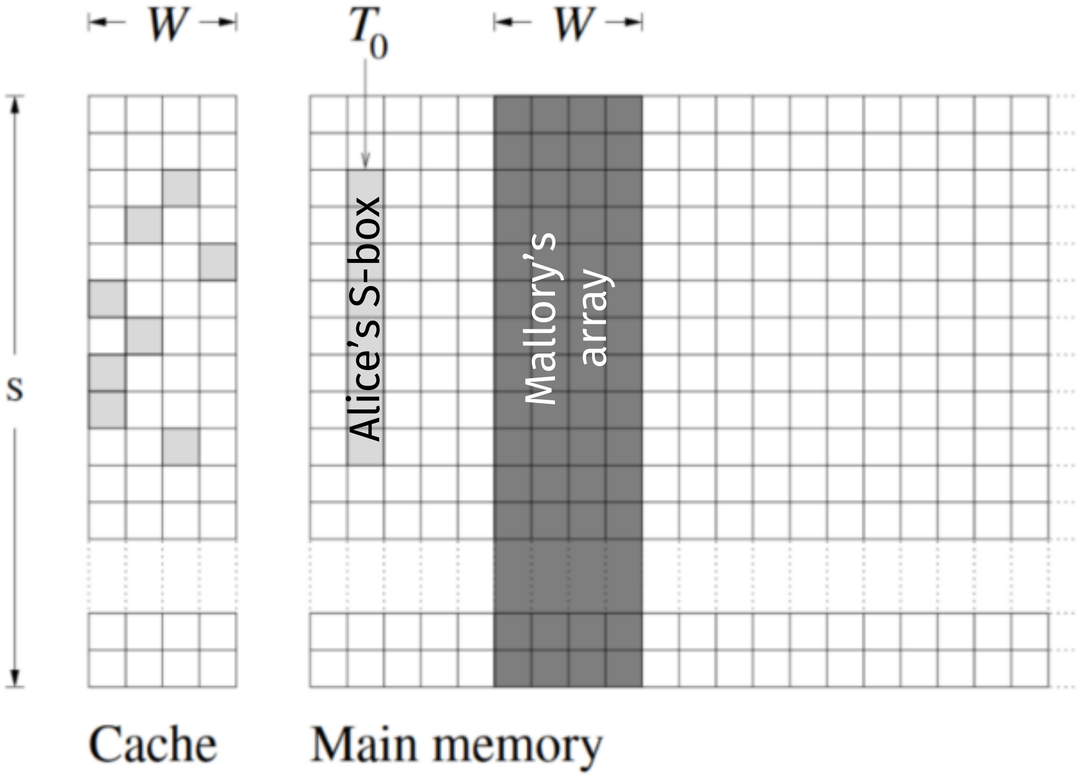
- Response of array lookup depends upon whether the value is already in cache
- This, in turn, depends on whether you've already looked up this value in the past

Cache attack: Mallory co-resident with Alice

- For now, suppose Mallory has a presence on Alice's machine
  - Co-located VMs on the cloud
  - Unprivileged account on a multi-tenant Unix machine with full-disk encryption
  - Unprivileged application that can write files to a phone with disk encryption
- Cache is shared between all tenants on a machine
- Ergo, Mallory can influence the state of Alice's cache!
  [Osvik, Shamir, Tromer 2006]

How the cache works

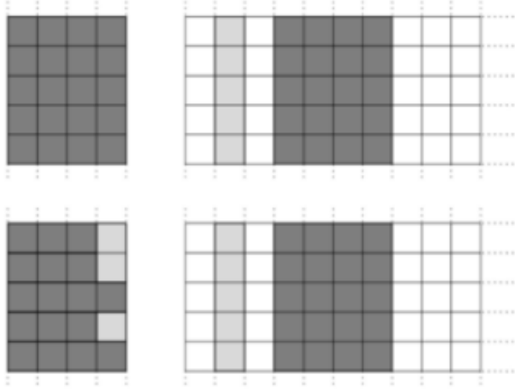
- Fixed mapping between locations in memory & cache
- When victim Alice runs AES, selected portions of the S-box move into cache
- If Mallory controls a large region of memory (~size of cache), she can fill in the cache with her own contents



Prime + Probe attack

Algorithm:

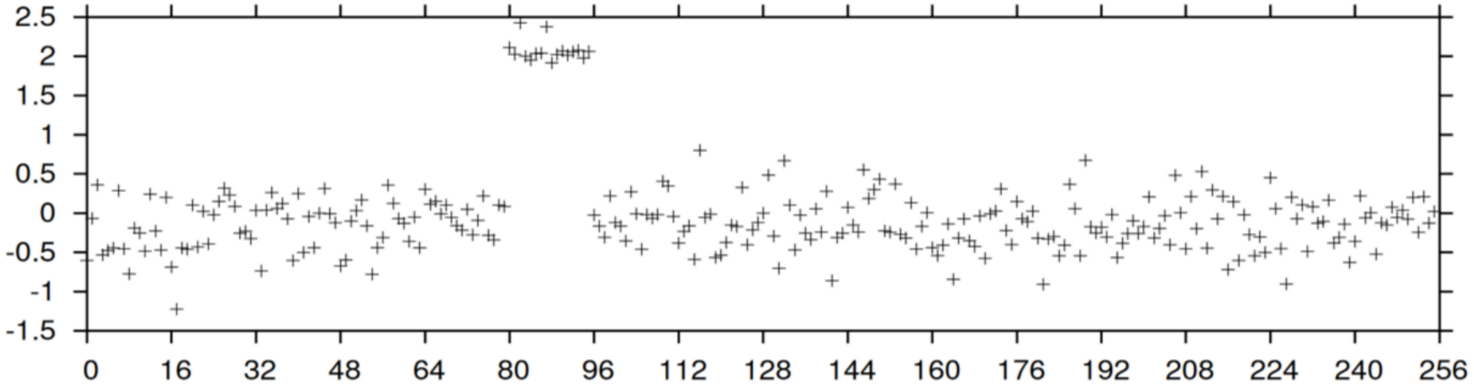
1. Mallory fills the cache with a large array A
2. Wait for Alice to execute an AES encipher/decipher operation
3. Mallory re-reads array A, records time to retrieve each byte



Upshot: AES evicted some lines of Mallory's cache based upon secret key

Strength: Find key byte with ~800 samples over 65ms

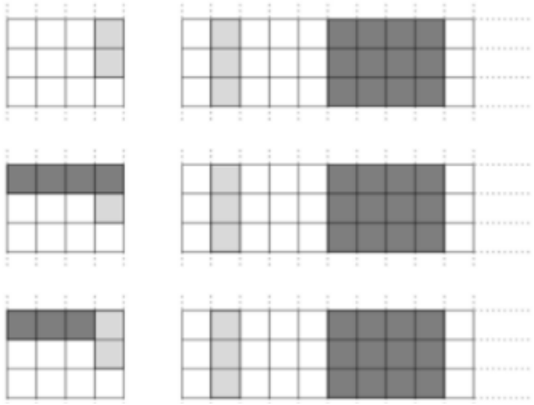
Countermeasure: check for scans of large arrays?



Evict + Time attack

Algorithm:

1. Mallory creates a large array A but does not read it yet
2. Trigger an AES encipher/decipher with known input *x*/output *y*
3. Mallory reads a few bytes of array A
4. Trigger another AES encipher/decipher with the same *x*/y



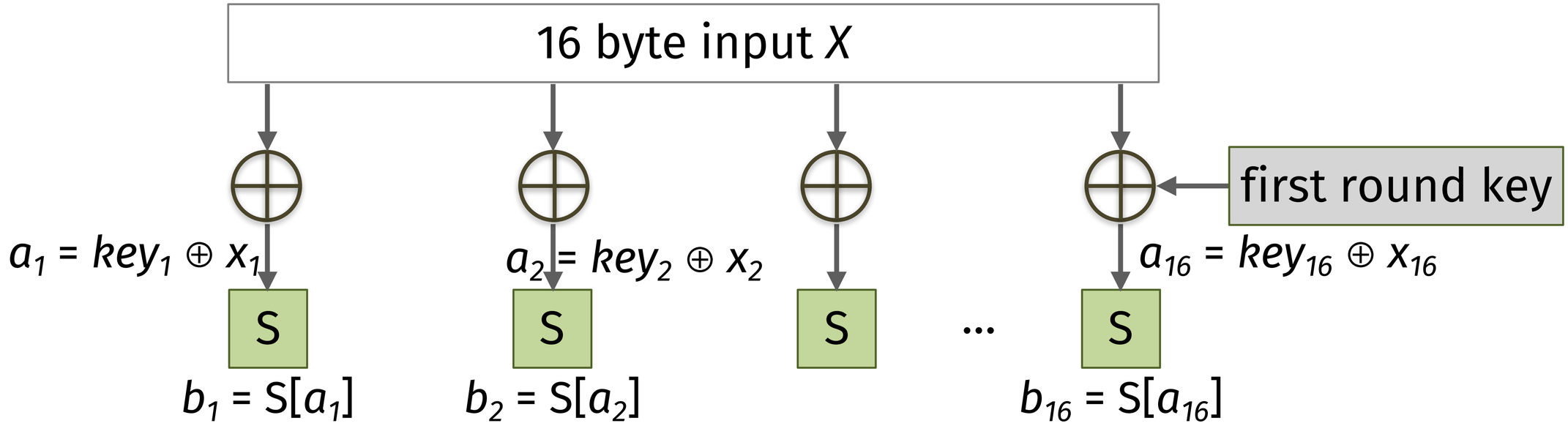
Upshot: 2nd AES is slower iff Mallory evicted the right cacheline

Strength: Find key byte with ~50k samples over ~30s, without ever reading a really large array

Timing attack: Mallory observes Alice over network

- Suppose Alice kicks Mallory off of her machine
  - Mallory cannot tamper with Alice's cache
  - Mallory doesn't get to observe Alice's cache directly
- Still, timing information may be viewable remotely!
  - Mallory can observe response times to Alice's TLS packets over the internet
  - Mallory can use this info to find Alice's key (albeit with many more samples)

Timing of first round table lookups



- In the first round, AES code makes 16 S-box table lookups
- If $a_1 == a_2$, then $S[a_2]$ table lookup will be *fast* since the value is in cache
- In general, 1st round running time $\propto$ # of distinct intermediate values

How can Mallory exploit speed differences?

- Let's say Mallory tells Alice to encipher an input with $x_1 = 01$, $x_2 = 02$
- Suppose for now that Mallory magically learns that a cache hit occurs in the first two S-box lookups
- Then, Mallory knows that

$$a_1 = a_2$$

$$key_1 \oplus x_1 = key_2 \oplus x_2$$

$$key_1 \oplus key_2 = x_1 \oplus x_2 = 03$$

How can Mallory find Alice's key?

- Even knowing $key_1 \oplus key_2 = x_1 \oplus x_2 = 03$ doesn't tell you key_1 or key_2
- What if *all* 16 input bytes caused collisions?
- Then Mallory can also compute $key_1 \oplus key_3$, $key_1 \oplus key_4$, ..., $key_1 \oplus key_{16}$
- I claim that Mallory has effectively learned 120 of the 128 bits of key!
  - There are 256 choices for key_1
  - Each choice gives a unique remaining option for key_2, key_3, ..., key_{16}
- Brute-force the rest if you have an (x, y) pair

Making the attack more realistic

Simplifying assumptions so far

- 1 input $\boldsymbol{x} \rightarrow$ many cache collisions
- Can tell which bytes of $\boldsymbol{a}$ collide
- Timing measurement corresponds precisely to first round runtime, which is exactly proportional to # of $\boldsymbol{a}$ collisions

How to remove these assumptions

- View time for many colliding $\boldsymbol{x}$ (stronger *signal*)
- Vary $\boldsymbol{x}$ samples only in certain locations (more precise *signal*)
- Collect even more samples to overcome *noise*

Tactic 1: Collect more samples

- Strategy
  - Don’t assume the existence of a single “magical” ***x*** with many collisions
  - Instead, simply try many possible ***x***
- If ***x*** is chosen randomly, then the probability that:
  - a given pair of bytes (e.g., bytes 1 and 2) collide = 1/256
  - byte 1 collides with some other byte ≈ 1/16
  - there exists a collision = $1 - (256 \text{ choose } 16)/(256^{16}) \approx 1 - 10^{-14}$
- Just as before, each collision yields a constraint on the *key*
- Sample enough ***x*** until we observe 15 independent constraints

Tactic 2: Strategically vary *x*

- Mallory needs to (1) observe a collision and (2) know *where* it occurs
- We can determine which bytes collide by fixing part of *x*
- Example: take average timing over several inputs with
  - $x_1 = 0$, $x_2 = 0$, and the other 14 bytes randomly chosen
  - $x_1 = 0$, $x_2 = 1$, and the other 14 bytes randomly chosen
  - …
  - $x_1 = 0$, $x_2 = 255$, and the other 14 bytes randomly chosen
- For whichever bucket is consistently faster, $x_1 \oplus x_2 = key_1 \oplus key_2$

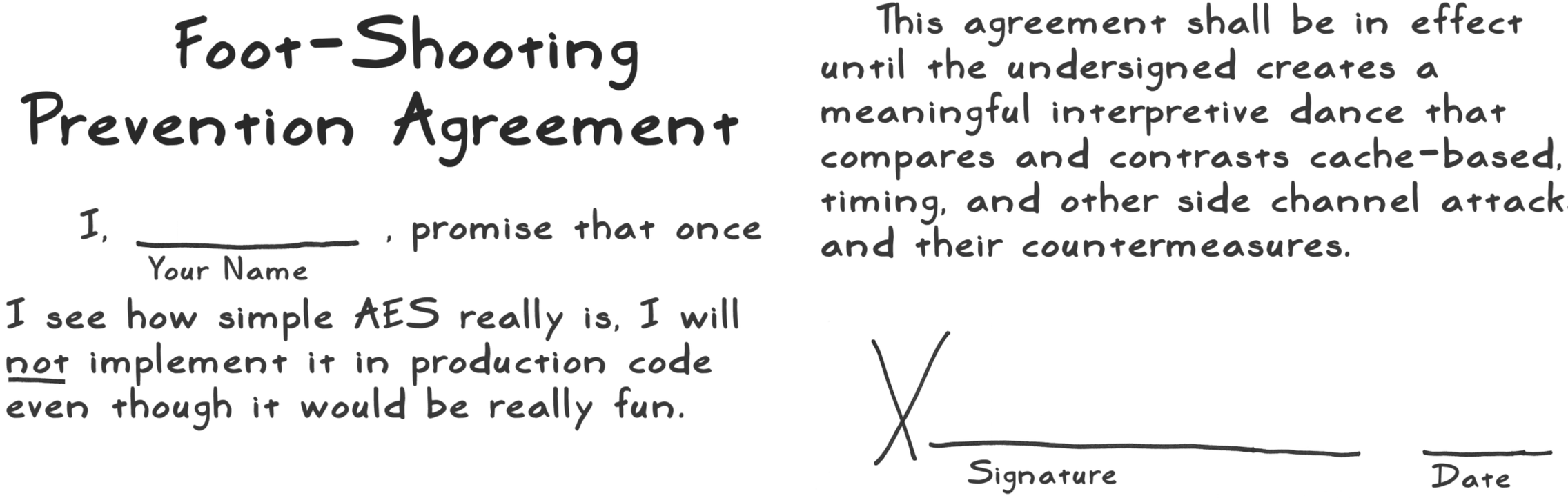
Tactic 3: Repeat to overcome noise

- We know that Time(AES) is smaller when $a_1 = a_2$ than when they differ
- Mallory's measurement of AES runtime depends on many other factors
  - Other bytes in the same cache line
  - Other bytes of the 1st round (or last round if Mallory knows the output y)
  - Other rounds
  - Network latency (if you're conducting this attack remotely)
- With enough samples, we can average over this noise!
- Bin running times by $x_1 \oplus x_2$, see which is smallest

Countermeasures to (cache) timing attacks

1. Don’t have table lookups
    - Hardware implementations of AES are not vulnerable
    - There exist other ciphers that are designed to avoid the need for table lookups (e.g., we will see later in the course that SHA-3 doesn’t have any)
2. Look up the entire table
    - Pre-load the entire S-box into the cache before beginning AES
    - Then the timing doesn’t depend on the particular values that you look up
    - Precarious because you might get interrupted in the middle of execution

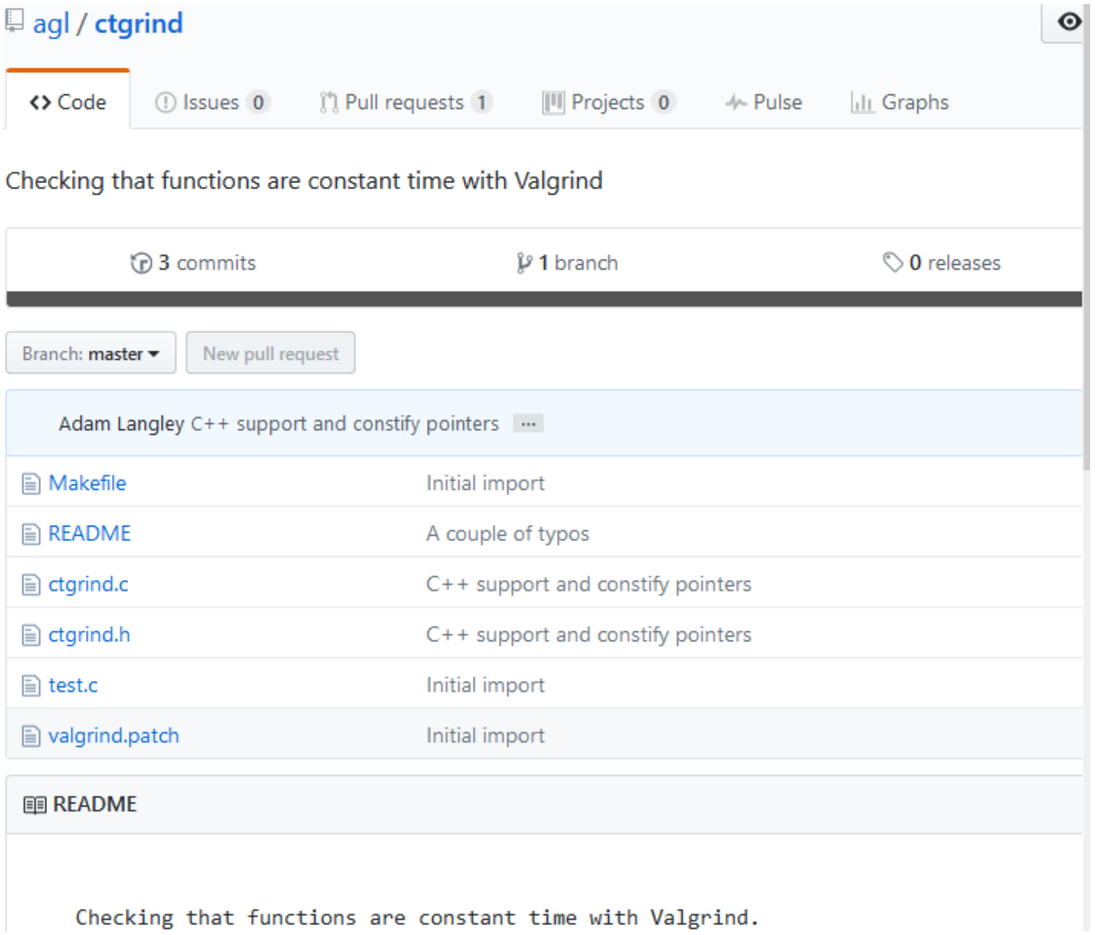
Side channels ⇒ difficult to implement crypto securely



Source:
moserware.com/2009/09/stick-figure-guide-to-advanced.html

What you should do

Validate code for timing independence



Source: github.com/agl/ctgrind

Use good *crypto* coding conventions

This page lists coding rules with for each a description of the problem addressed (with a concrete example of failure), and then one or more solutions (with example code snippets).

Contents [hide]

Source: cryptocoding.net/index.php/Coding_rules