Practical Cryptographic Systems

Side Channel Attacks

Instructor: Matthew Green

Housekeeping:

- A2 is out

News?

README.md

keypair

Generate a RSA PEM key pair from pure JS

build error downloads 358k/month

Usage

```
var keypair = require('keypair');

var pair = keypair();
console.log(pair);
```

outputs

```
$ node example.js
{ public: '-----BEGIN RSA PUBLIC KEY-----\r\nMIGJAoGBAM3CosR73CBNcJsLv5E90NsFt6qN1uziQ484gbOoule8leXH
  private: '-----BEGIN RSA PRIVATE KEY-----\r\nMIICXAIBAAKBgQDNwqLEe9wgTXCbC7+RPdDbBbeqjdbs4kOPOIGzqL
```

Tested Version

v1.0.3

Details

Issue 1: Poor random number generation (`GHSL-2021-1012`)

The library does not rely entirely on a platform provided CSPRNG, rather, it uses it's own counter-based CMAC approach. Where things go wrong is seeding the CMAC implementation with "true" random data in the function `defaultSeedFile`. In order to seed the AES-CMAC generator, the library will take two different approaches depending on the JavaScript execution environment. In a browser, the library will use `window.crypto.getRandomValues()`. However, in a nodeJS execution environment, the `window` object is not defined, so it goes down a much less secure solution, also of which has a bug in it.

It does look like the library tries to use node's CSPRNG when possible:

https://github.com/juliangruber/keypair/blob/87c62f255baa12c1ec4f98a91600f82af80be6db/index.js#L1016

Unfortunately, it looks like `crypto` is null because a variable was declared with the same name, and set to `null`:

https://github.com/juliangruber/keypair/blob/87c62f255baa12c1ec4f98a91600f82af80be6db/index.js#L759

So the node path is never taken.

However, when `window.crypto.getRandomValues()` is not available, a Lehmer LCG random number

Current Events

/index.js#L759

So the node path is never taken.

However, when `window.crypto.getRandomValues()` is not available, a Lehmer LCG random number generator is used to seed the CMAC counter, and the LCG is seeded with `Math.random`. While this is poor and would likely qualify in a security bug in itself, it does not explain the extreme frequency in which duplicate keys occur.

Main flaw

Main flaw

The output from the Lehmer LCG is encoded incorrectly. The specific line with the flaw is:

```
b.putByte(String.fromCharCode(next & 0xFF))
```

The definition of `putByte` is

```
util.ByteBuffer.prototype.putByte = function(b) {
  this.data += String.fromCharCode(b);
};
```

Simplified, this is `String.fromCharCode(String.fromCharCode(next & 0xFF))`. The double `String.fromCharCode` is almost certainly unintentional and the source of weak seeding. Unfortunately, this does not result in an error. Rather, it results most of the buffer containing zeros. An example generated buffer:

(Note: truncated for brevity)

```
\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
\x00\x00\x00\x00\x04\x00\x00\x00....\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
```

Since we are masking with 0xFF, we can determine that 97% of the output from the LCG are

News?

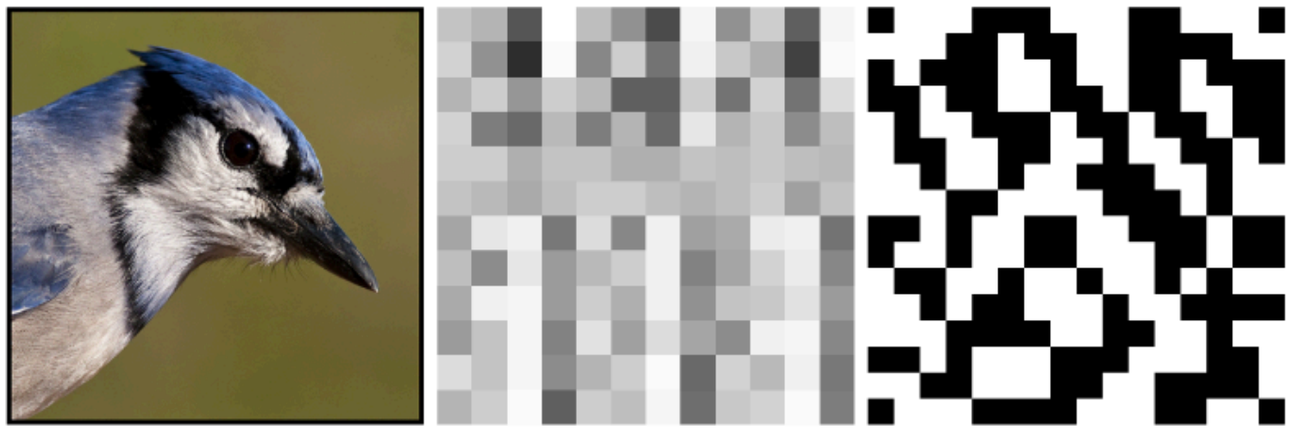


Figure 3: An image from ImageNet [34] and a visualization of its associated PhotoDNA (center) and PDQ hash (right).

News?

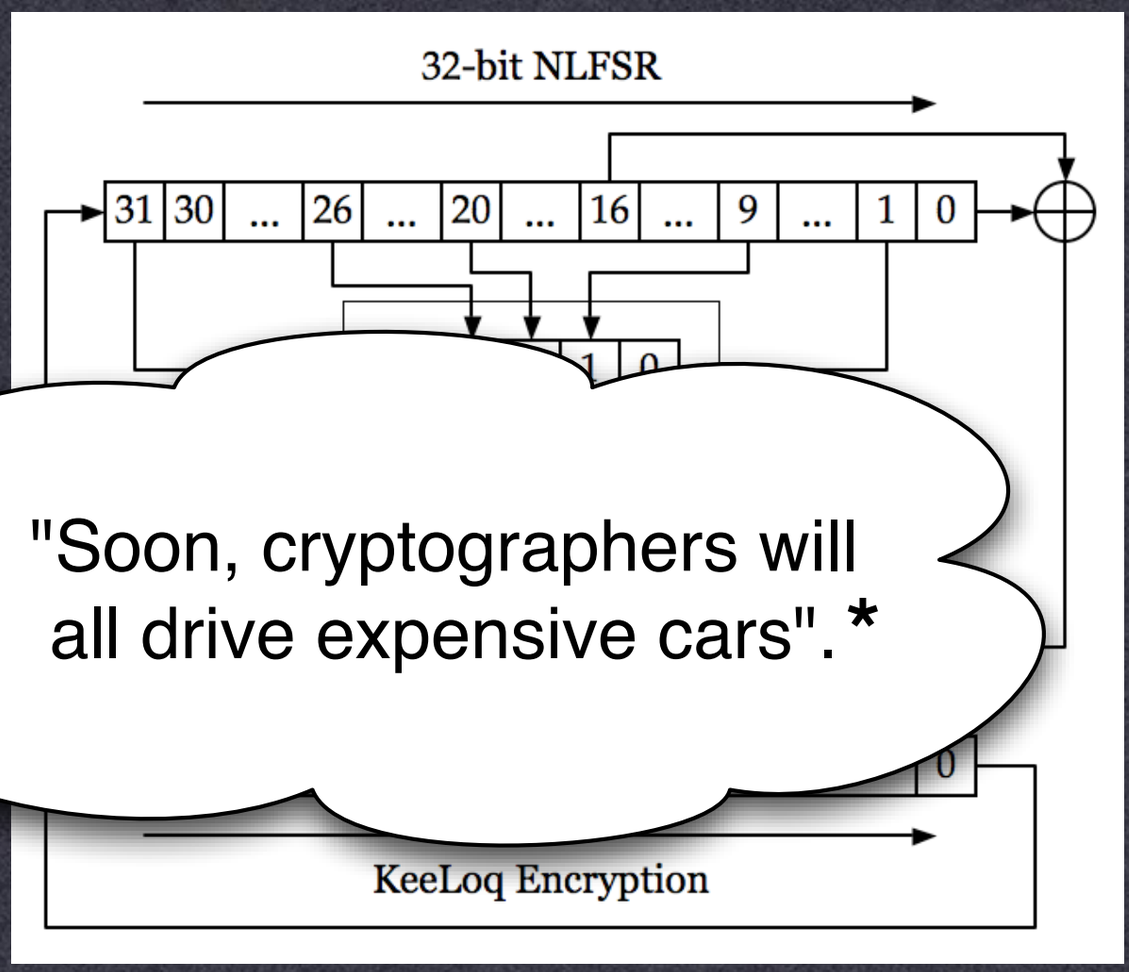
Figure 1: PhotoDNA targeted second preimage at $\Delta_J = 1720$

Figure 2: PDQ targeted second preimage at $\Delta_J = 86$

KeeLoq

- Designed by Kuhn & Smit
  - Block cipher: 64-bit key, 32-bit block
  - 528 rounds
  - Mostly used for door opening
  - Direct attacks*:

A. 2^{16} data & 2^{51} operation
B. 2^{32} data & 2^{27} operations
C. Indesteege et al.:
(65 min to get data,
218 CPU days to process)



"Soon, cryptographers will all drive expensive cars".*

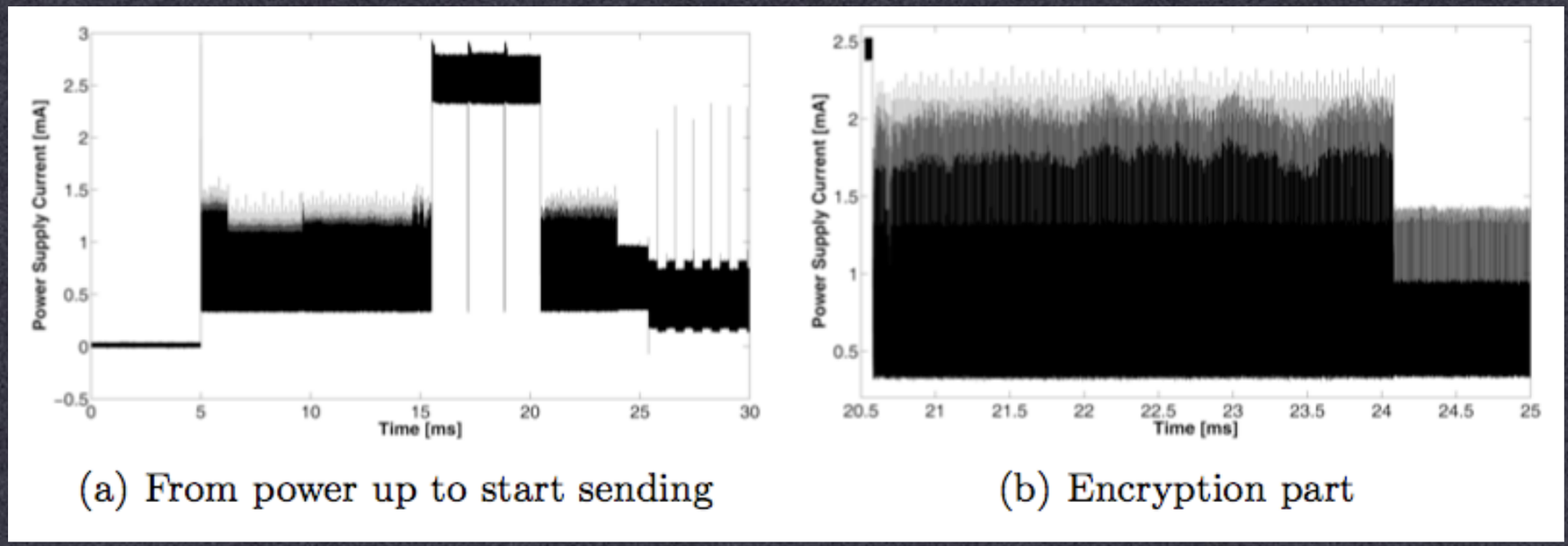
Public domain image from Wikipedia
* Source: Indesteege et al.: http://www.cosic.esat.kuleuven.be/keeloq/keeloq-slides.pdf

KeeLoq

- Attack CRYPTO 2008
  - Requires physical access to device
  - Recovers secret key after 10-60 reads
  - How?

KeyLoq

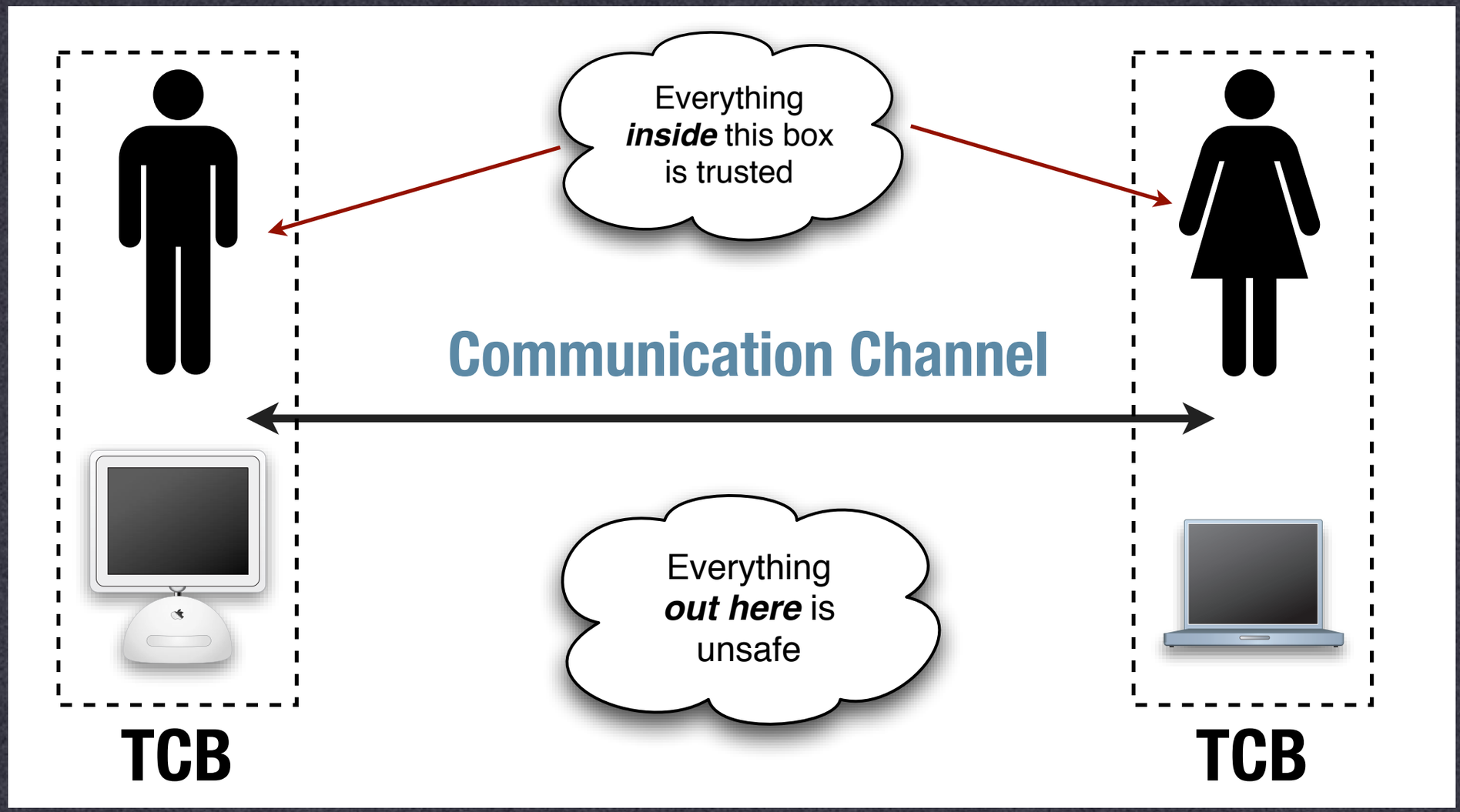
- Side channel attack
  - Doesn't directly attack the cipher
  - Instead, measures how the cipher works in operation.
  - In this case, use power consumption



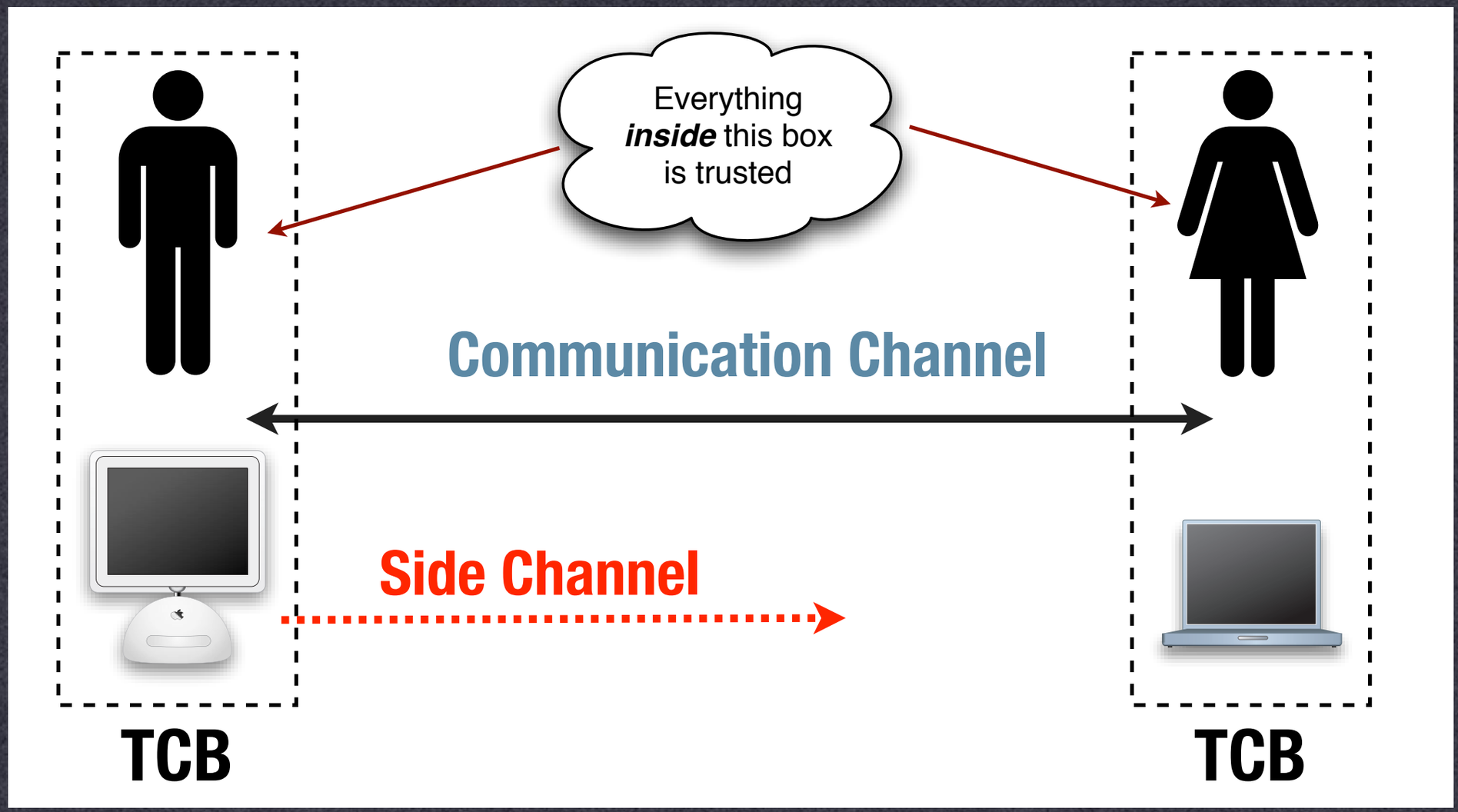
(a) From power up to start sending (b) Encryption part

Source: Eisenbarth et al., On the Power of Power Analysis in the Real World: A Complete Break of the KeeLoq Code Hopping Scheme

Review

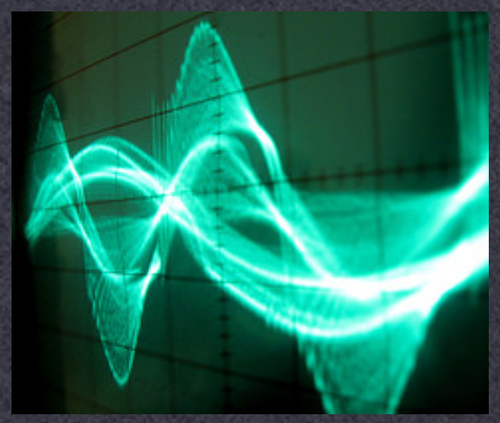

Everything
inside this box
is trusted
Communication Channel
Everything
out here is
unsafe
TCB
TCB

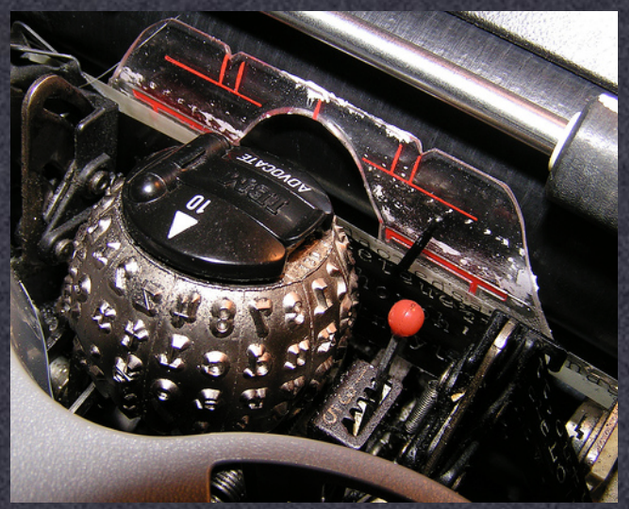
Side Channels



Side Channels

- Some history:
  - 1943: Bell engineer detects power spikes from encrypting teletype
  - 1960s: US monitors EM emissions from foreign cipher equipment
  - 1980s: USSR places eavesdropping device inside IBM Selectric typewriters



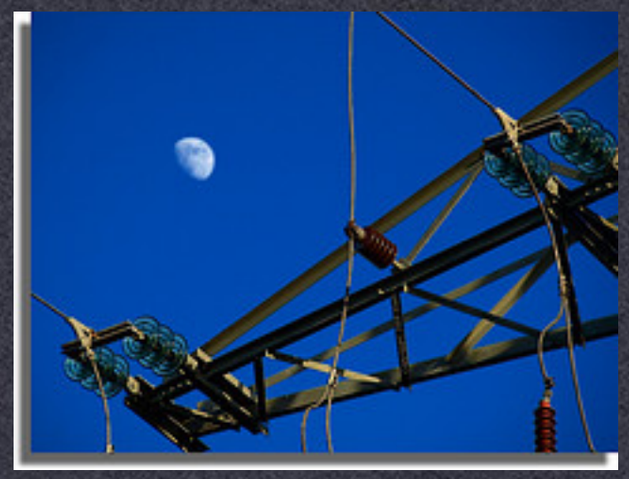


Side Channels

- Some history:
  - 1990s: Paul Kocher demonstrates timing attacks, power analysis attacks against RSA, Elgamal

Common Examples

- Timing
- Power Consumption
- RF Emissions
- Audio



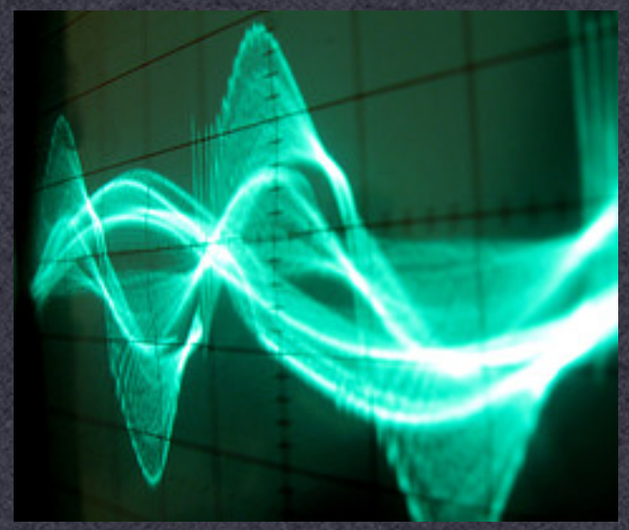


Photo Credits: Clock - Flickr user laffy4k, Waveform - Flickr user altemark, Power Line - Flickr user Kliefi. All used under a Creative Commons license.

MAC verification

- How does one verify a MAC?

RSA Cryptosystem

$N = p \cdot q$

$e \cdot d \equiv 1 \; mod \; (p-1)(q-1)$

Encryption

$$c = m^e \; mod \; N$$

Decryption

$$m = c^d \; mod \; N$$

RSA Cryptosystem

$$N = p \cdot q$$

$$e \cdot d \equiv 1 \; mod \; (p-1)(q-1)$$

Encryption

$$c = m^e \; mod \; N$$

Decryption

$$m = c^d \; mod \; N$$

$$m = \underbrace{c * c * c * \cdots * c}_{\text{d times}} \; mod \; N$$

Modular Exponentiation

$$m = c^d \ mod \ N$$

$$d = 10101010011101010110010010001001$$

```
modpow(c, d, N) {

    result = 1;

    for (i = 1 to |d|) {
        if (the ith bit of d is 1) {
            result = (result * c) mod N;
        }

        c = c² mod N;
    }

    return result;
}
```

Modular Exponentiation

$$m = c^d \bmod N$$

$$d = 10101010011101010110010010001001$$

```
modpow(c, d, N) {

    result = 1;

    for (i = 1 to |d|) {
        if (the ith bit of d is 1) {
            result = (result * c) mod N;
        }

        c = c² mod N;
    }

    return result;
}
```

Modular Exponentiation

$$m = c^d \bmod N$$

$$d = 10101010011101010110010010001001$$

```
modpow(c, d, N) {

    result = 1;

    for (i = 1 to |d|) {
        if (the ith bit of d is 1) {
            result = (result * c) mod N;
        }

        c = c² mod N;
    }

    return result;
}
```

Modular Exponentiation

$$m = c^d \bmod N$$

$$d = 10101010011101010110010010001001$$

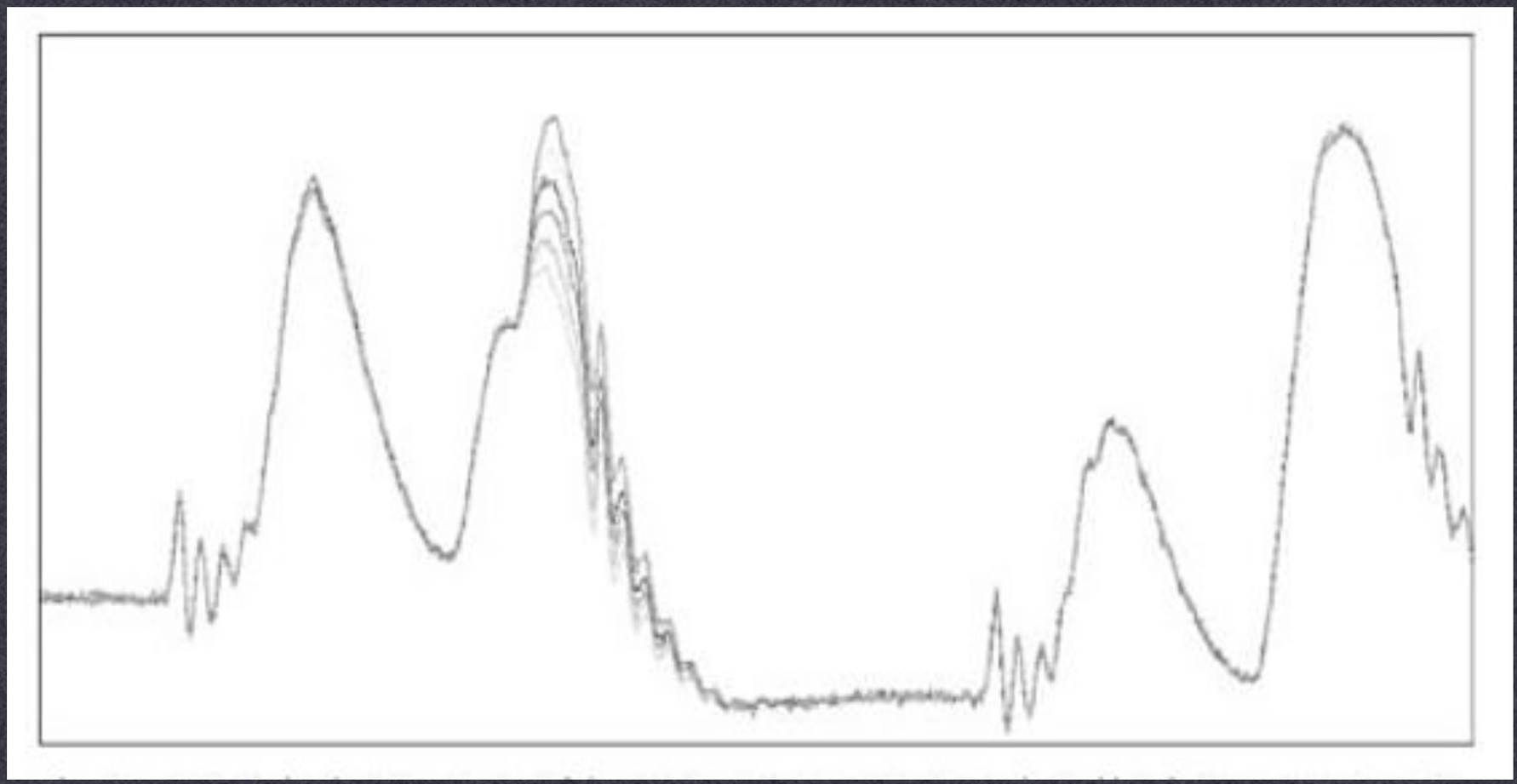
```
modpow(c, d, N) {

    result = 1;

    for (i = 1 to |d|) {
        if (the ith bit of d is 1) {
            result = (result * c) mod N;
        }

        c = c² mod N;
    }

    return result;
}
```

Expensive Operation

Simple Power Analysis

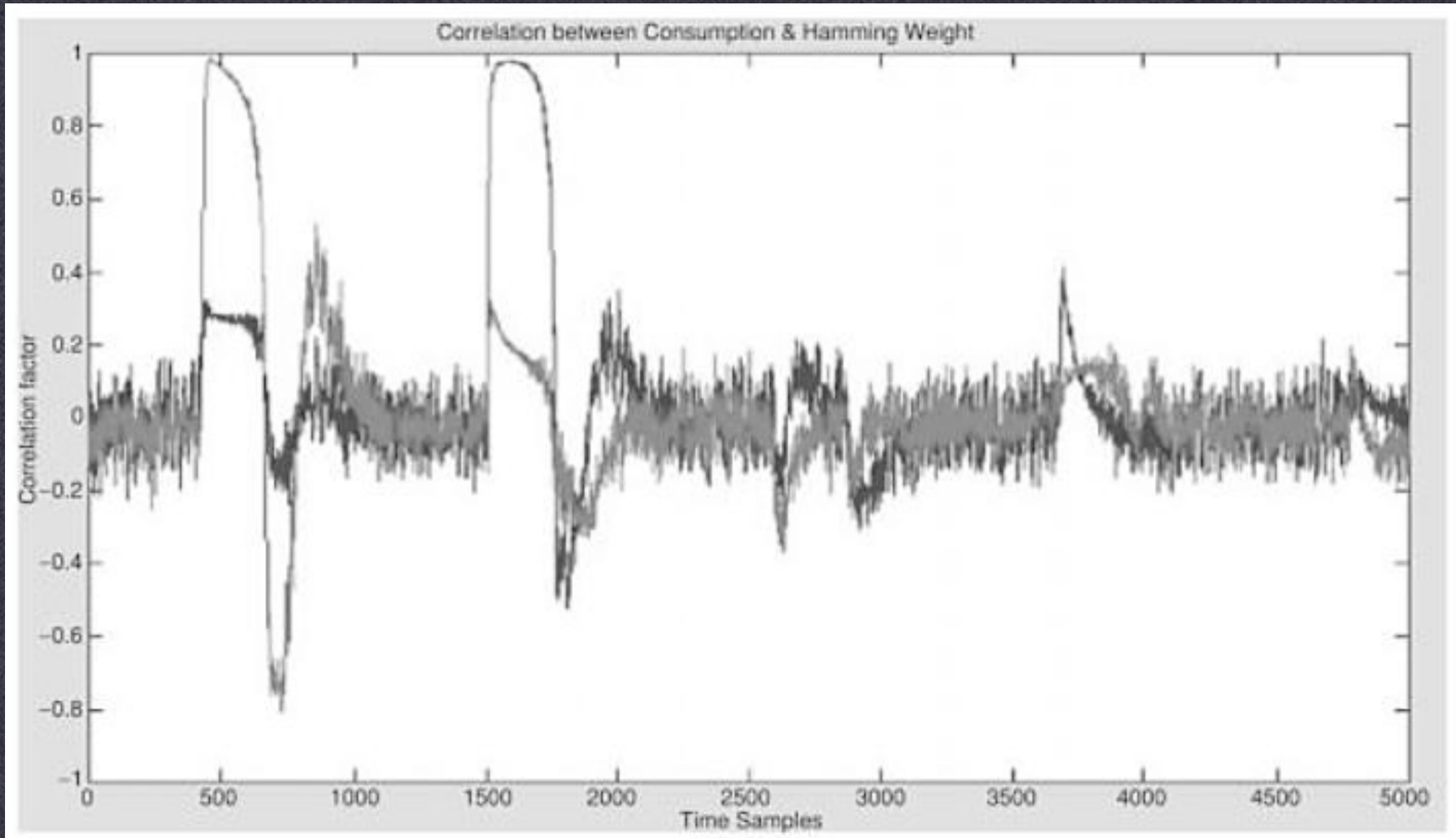
Source: Bidgoli, Handbook of Information Security

Differential Power Analysis



Source: Bidgoli, Handbook of Information Security

Differential Power Analysis



Source: Bidgoli, Handbook of Information Security

Modular Exponentiation

$$m = c^d \ mod \ N$$

$$d = 10101010011101010110010010001001$$

```
modpow(c, d, N) {

    result = 1;

    for (i = 1 to |d|) {
        c = c² mod N;

        if (the iᵗʰ bit of d is 1) {
            result = (result * c) mod N;
        }
    }

    return result;
}
```

Expensive Operation

Modular Exponentiation

$$m = c^d \ mod \ N$$

$$d = 10101010011101010110010010001001$$

```
modpow(c, d, N) {

    result = 1;

    for (i = 1 to |d|) {
        if (the ith bit of d is 1) {
            result = (result * c) mod N;
        }

        c = c² mod N;
    }

    return result;
}
```

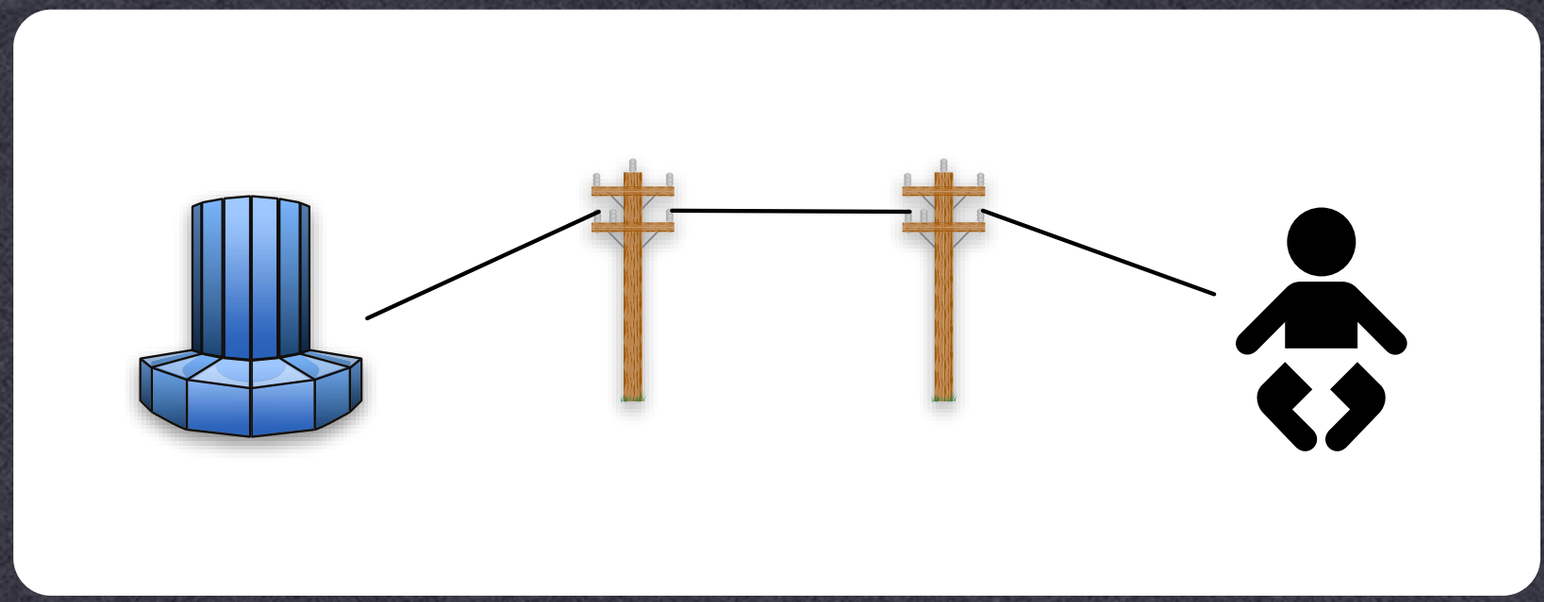
Expensive Operation

Kocher's Timing Attack

- Assume that for some values of (a * b), multiplication takes unusually long
- Given the first b bits, compute intermediate value
  "result" up to that point
- If the next bit of d = 1, then calc is (result * c)
- If this is expected to be slow,
  but response is fast then the next bit of d != 1

Remote Timing Attacks

- Boneh & Brumley
  - Remote attack on RSA-CRT as implemented in OpenSSL
  - Optimization, uses knowledge of p,q



CRT

- If one knows the remainder of division by several integers (p, q) then one can compute the remainder of division by products (p*q)
  — assuming p, q are co-prime
- Why does this help us?

Solutions

- Quantization:
  - All operations take the same time
  - Hard to do without sacrificing performance
- Blinding:
  - Prevents attacker from selecting ciphertext (that is processed with the secret key)

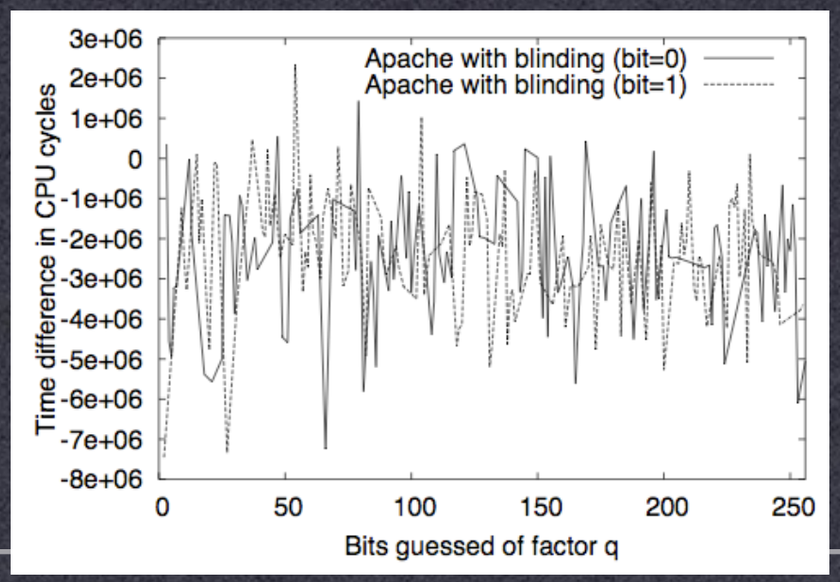
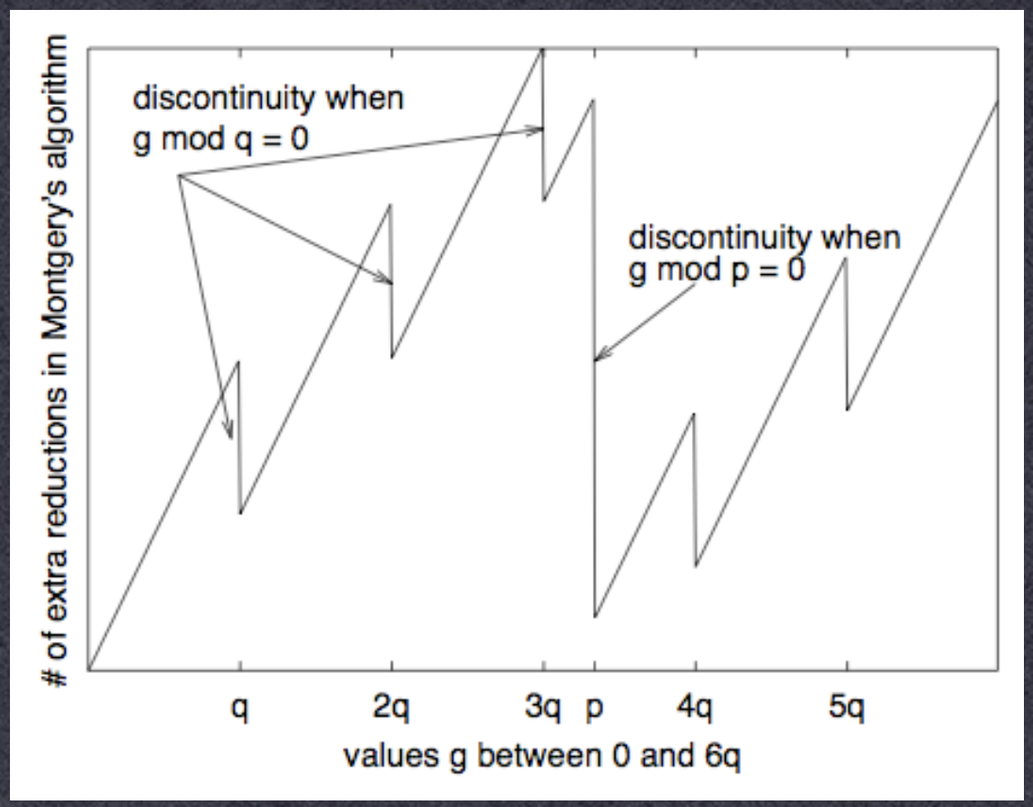
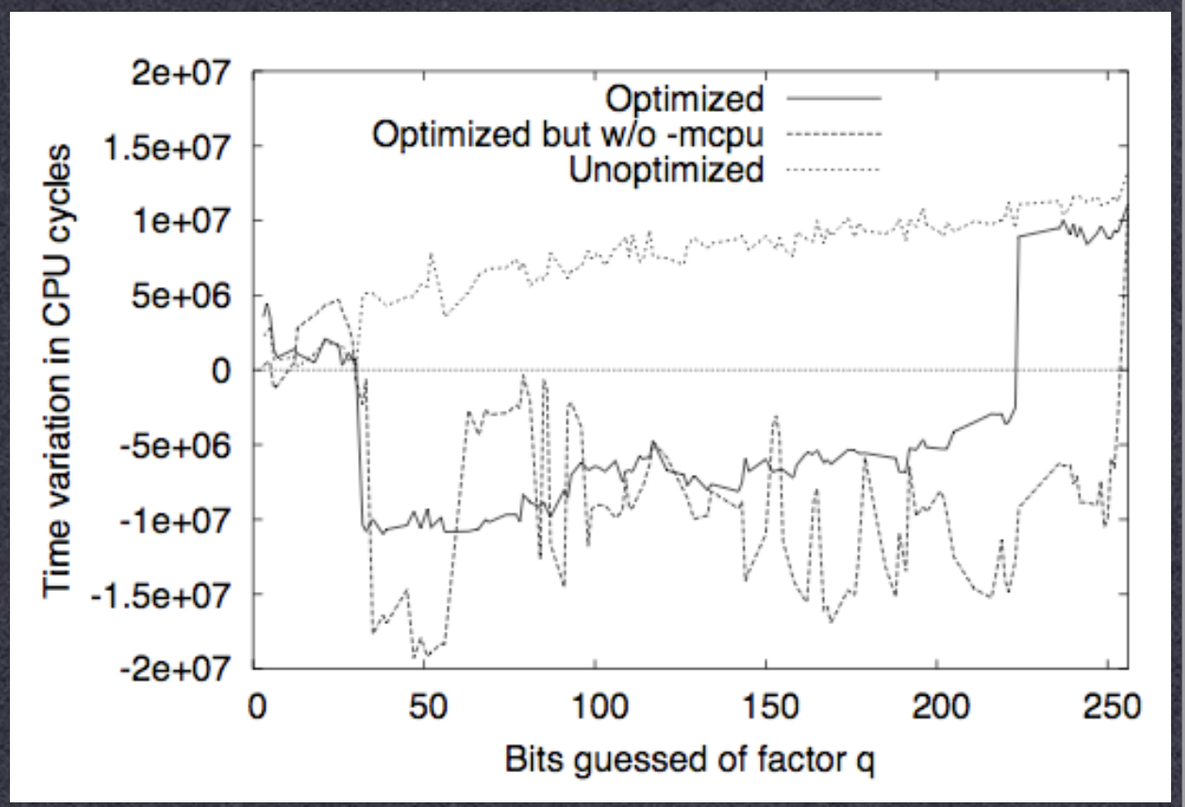


Image Source: Boneh, Brumley: Remote Timing Attacks are Practical

Remote Timing Attacks

- **Boneh & Brumley**
  - **By repeating the timing measurements, they were able to extract a secret key after several million samples**





Source: Boneh, Brumley: Remote Timing Attacks are Practical

Windowed Exponentiation

$$m = c^d \bmod N$$

$$d = 10101010011101010110010010001001$$

- Handles the input in larger “chunks”
  - Fixed or variable sized
  - Can speed up by pre-computing some values

-e.g., c^2, c^3, ..., c^16

Windowed Exponentiation

$$m = c^d \bmod N$$

$$d = 10101010011101010110010\,0100\,1001$$

- Handles the input in larger “chunks”
  - Fixed or variable sized
  - Can speed up by pre-computing some values
    - e.g., c^2, c^3, ..., c^16

Windowed Exponentiation

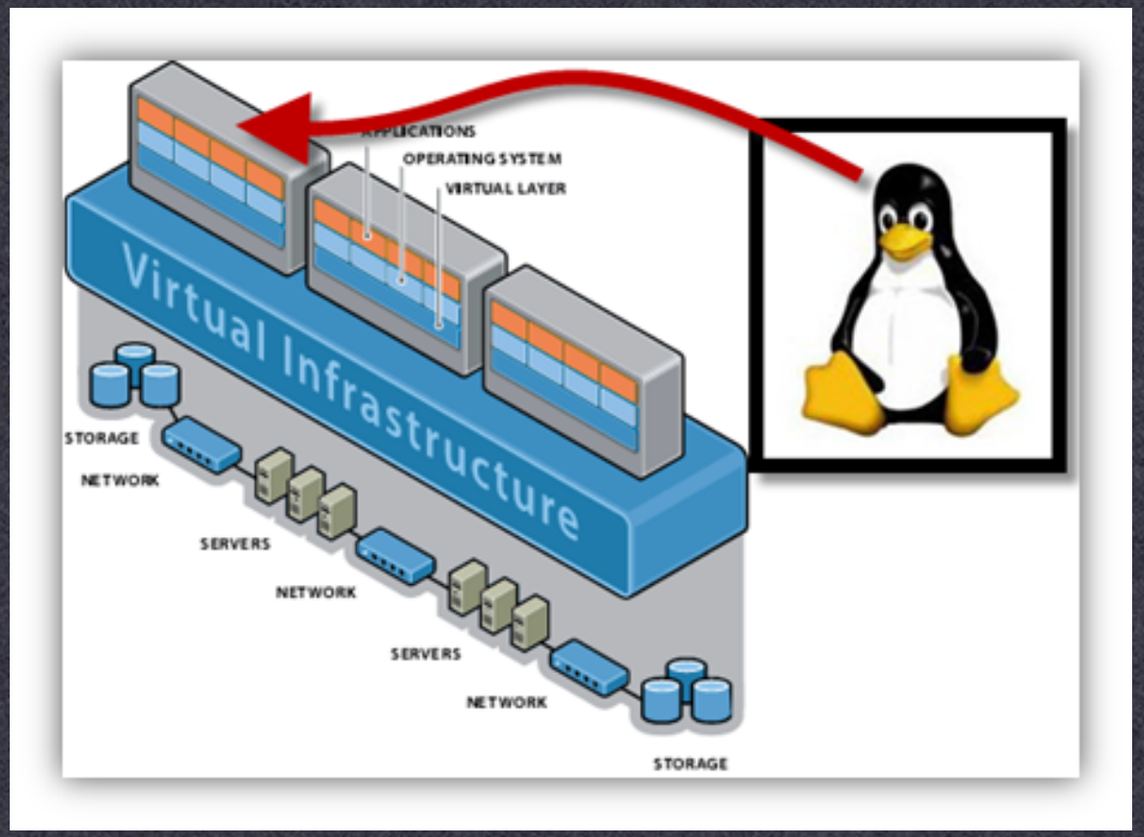
$$m = c^d \ mod \ N$$

$$d = 10101010011101010110010001001001$$

- Handles the input in larger “chunks”
  - Fixed or variable sized
  - Can speed up by pre-computing some values
    - e.g., c^2, c^3, ..., c^16

Cache Timing Attacks

- Observation
  - When we use pre-computed tables, this implies a memory access
  - This takes time
  - Unless the data is cached!



Hypervisor attacks

- **Observation**
  - **This applies to code too!**

SquareMult(x, e, N):
  let $e_n, \ldots, e_1$ be the bits of e
$y \leftarrow 1$
for $i = n$ down to 1 {
  $y \leftarrow$ Square(y) (S)
  $y \leftarrow$ ModReduce(y, N) (R)
  if $e_i = 1$ then {
    $y \leftarrow$ Mult(y, x) (M)
    $y \leftarrow$ ModReduce(y, N) (R)
  }
}
return y

The 9 Lives of Bleichenbacher's CAT: New Cache ATtacks on TLS Implementations

Eyal Ronen*, Robert Gillham†, Daniel Genkin‡, Adi Shamir¶, David Wong§, and Yuval Yarom†**

*Tel Aviv University, †University of Adelaide, ‡University of Michigan, ¶Weizmann Institute, §NCC Group, **Data61

Listing 2. Pseudocode of RSA_

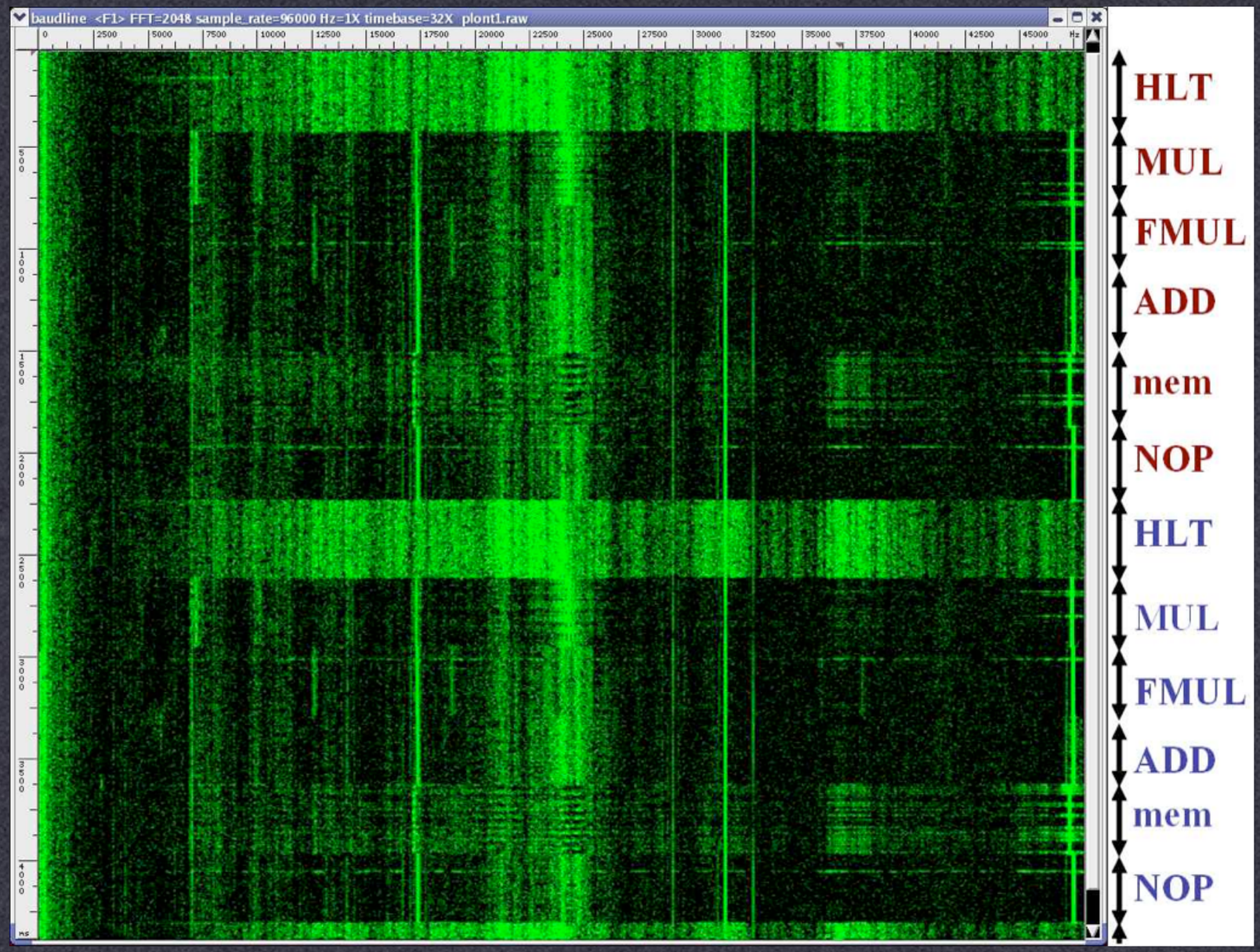
```
1  int SSLDecodeRSAKeyExchange(keyExchange, ctx){
2    keyRef = ctx->signingPrivKeyRef;
3    src  = keyExchange.data;
4    localKeyModulusLen = keyExchange.length;
5    ... // addtional inilization code omitted
6
7    err = sslRsaDecrypt(keyRef, src,
       localKeyModulusLen,
8      ctx->preMasterSecret.data,
       SSL_RSA_PREMASTER_SECRET_SIZE, &outputLen);
9    if(err != errSSLSuccess) {
10     /* possible Bleichenbacher attack */
11     sslLogNegotiateDebug("SSLDecodeRSAKeyExchange:
12                                 RSA decrypt fail");
13   } else if(outputLen !=
14                   SSL_RSA_PREMASTER_SECRET_SIZE) {
15     sslLogNegotiateDebug("SSLDecodeRSAKeyExchange:
       premaster secret size error");
16     // not passed back to caller
17     err = errSSLProtocol;
18   }
19   if(err == errSSLSuccess) {
20       ... // (omitted for brevity)
21   }
22   if(err != errSSLSuccess) {
23     ... // (omitted for brevity)
24     sslRand(&tmpBuf);
25   }
26   /* in any case, save premaster secret (good or
       bogus) and proceed */
27   return errSSLSuccess;
28 }
```

attack [69], as implemented in the Mastik toolkit [68].

Monitoring Locations. To reduce the likelihood of errors, we monitor both the call-site to RSAerr (Line 25 of Listing 2) and the code of the function RSAerr. Monitoring each of these locations may generate false positives, i.e. indicate access when the plaintext is PKCS #1 v1.5 conforming. The former results in false positives because the call to RSAerr shares the cache line with the surrounding code, that is always invoked. The latter results in false positives when unrelated code logs an error. By only predicting a non-conforming plaintext if *both* locations are accessed within a short interval, we reduce the likelihood of false positives. We note that this technique is very different to the approach of Genkin et al. [30] of monitoring two memory locations to reduce false negative errors due to a race between the victim and the attacker [6]. Unlike us, they assume access if *any* of the monitored locations is accessed.

Experimental Results. Overall, our technique achieves a

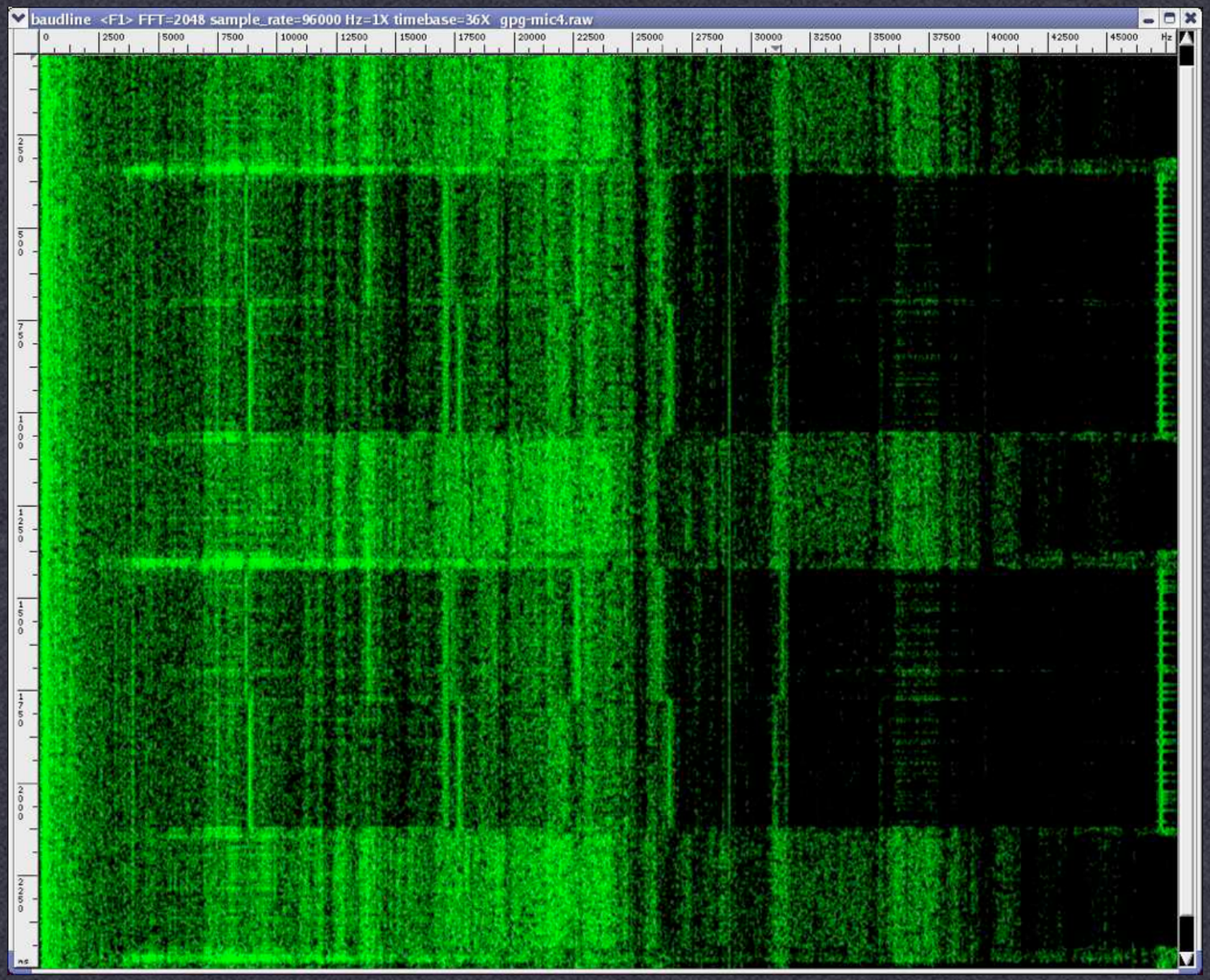
Acoustic Side Channels



Source: Tromer, Acoustic Cryptanalysis: http://people.csail.mit.edu/tromer/acoustic/

Acoustic Side-Channels

GPG RSA-CRT signature #1:



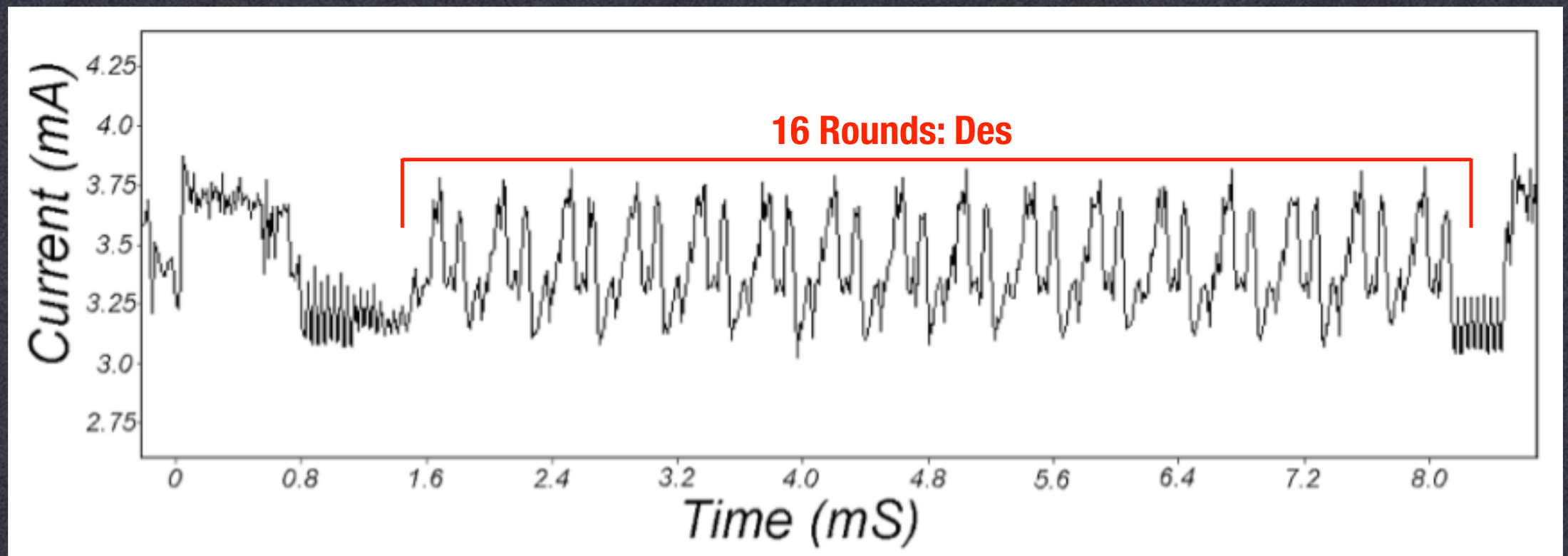
mod q

mod p

(repeated)

Source: Tromer, Acoustic Cryptanalysis: http://people.csail.mit.edu/tromer/acoustic/

Reverse Engineering



Source: Kocher et al. 1999

Back to KeyLoq

- **KeyLoq attack has an unhappy coda:**
  - **Turns out that all keys for a given manufacturer are derived from the same MK**

$$k = pad(ID, seed) \oplus MK$$

  - **Key derivation function based on XOR :(**
  - **By observing 2 interactions between key/car we can derive the MK and thus steal any car**