601.445/601.645: Practical Cryptographic Systems

Side Channel Attacks

Instructor: Matthew Green

Housekeeping:

- Grades for A1 back after class
- A2 ongoing
- Weekly HW assignment given out tonight

Current Events

2nd Death Near Seattle Adds to Signs Virus Is Spreading in U.S.

Officials see growing indications that the coronavirus has been spreading undetected for weeks. A cluster of cases at one nursing home have made Kirkland, Wash., a focus of concern.

IOTA Going Back Online in Ten Days

After a two-week network outage, the IOTA Foundation has released a fix to its wallet that will allow users to move their cryptocurrency to safety.

Current Events

Cryptology ePrint Archive: Report 2020/241

Separate Your Domains: NIST PQC KEMs, Oracle Cloning and Read-Only Indifferentiability

Mihir Bellare and Hannah Davis and Felix Günther

Abstract: It is convenient and common for schemes in the random oracle model to assume access to multiple random oracles (ROs), leaving to implementations the task (we call it oracle cloning) of constructing them
first part of the paper is a case study of oracle cloning in KEM submissions to the NIST Post-Quantum Cryptography standardization process. We give key-recovery attacks on some submissions arising from mistakes
find other submissions using oracle cloning methods whose validity is unclear. Motivated by this, the second part of the paper gives a theoretical treatment of oracle cloning. We give a definition of what is an "oracle c
what it means for such a method to "work," in a framework we call read-only indifferentiability, a simple variant of classical indifferentiability that yields security not only for usage in single-stage games but also in m
formalize domain separation, and specify and study many oracle cloning methods, including common domain-separating ones, giving some general results to justify (prove read-only indifferentiability of) certain class
not only able to validate the oracle cloning methods used in many of the unbroken NIST PQC KEMs, but also able to specify and validate oracle cloning methods that may be useful beyond that.

Category / Keywords: Post-Quantum Cryptography, NIST, Key Encapsulation, Public-Key Encryption, Random Oracles, Domain Separation, Indifferentiability, Composition

Current Events

- **Titan M chip in Android phones**

Here's what the key generation code for that looks like in Aegis. Excerpt from KeyStoreHandle.java:

```
1 KeyGenerator generator = KeyGenerator.getInstance(KeyProperties.KEY_ALGORITHM_AES,
2 generator.init(new KeyGenParameterSpec.Builder(id, KeyProperties.PURPOSE_ENCRYPT |
3         .setBlockModes(KeyProperties.BLOCK_MODE_GCM)
4         .setEncryptionPaddings(KeyProperties.ENCRYPTION_PADDING_NONE)
5         .setUserAuthenticationRequired(true)
6         .setRandomizedEncryptionRequired(true)
7         .setKeySize(CryptoUtils.CRYPTO_AEAD_KEY_SIZE * 8)
8         .build());
```

Current Events

```
diff --git a/app/src/main/java/com/beemdevelopment/aegis/crypto/KeyStoreHandle.ja
index cfa1e57..00cae52 100644
--- a/app/src/main/java/com/beemdevelopment/aegis/crypto/KeyStoreHandle.java
+++ b/app/src/main/java/com/beemdevelopment/aegis/crypto/KeyStoreHandle.java
@@ -57,6 +57,7 @@ public class KeyStoreHandle {
                     .setEncryptionPaddings(KeyProperties.ENCRYPTION_PADDING_NONE
                     .setUserAuthenticationRequired(true)
                     .setRandomizedEncryptionRequired(true)
+                    .setIsStrongBoxBacked(true)
                     .setKeySize(CryptoUtils.CRYPTO_AEAD_KEY_SIZE * 8)
                     .build());
```

To test this change, I went through Aegis' setup process, set a password and enabled biometric

Current Events

```
com.beemdevelopment.aegis.db.slots.SlotIntegrityException: javax.crypto.AEADBadTa
    at com.beemdevelopment.aegis.db.slots.Slot.getKey(Slot.java:57)
    ...
Caused by: javax.crypto.AEADBadTagException
    at android.security.keystore.AndroidKeyStoreCipherSpiBase.engineDoFinal(Andrc
    at javax.crypto.Cipher.doFinal(Cipher.java:2055)
    ...
Caused by: android.security.KeyStoreException: Signature/MAC verification failed
    at android.security.KeyStore.getKeyStoreException(KeyStore.java:1292)
    at android.security.keystore.KeyStoreCryptoOperationChunkedStreamer.doFinal(K
    at android.security.keystore.AndroidKeyStoreAuthenticatedAESCipherSpi$BufferA
    at android.security.keystore.AndroidKeyStoreCipherSpiBase.engineDoFinal(Andrc
    at javax.crypto.Cipher.doFinal(Cipher.java:2055)
    ...
```

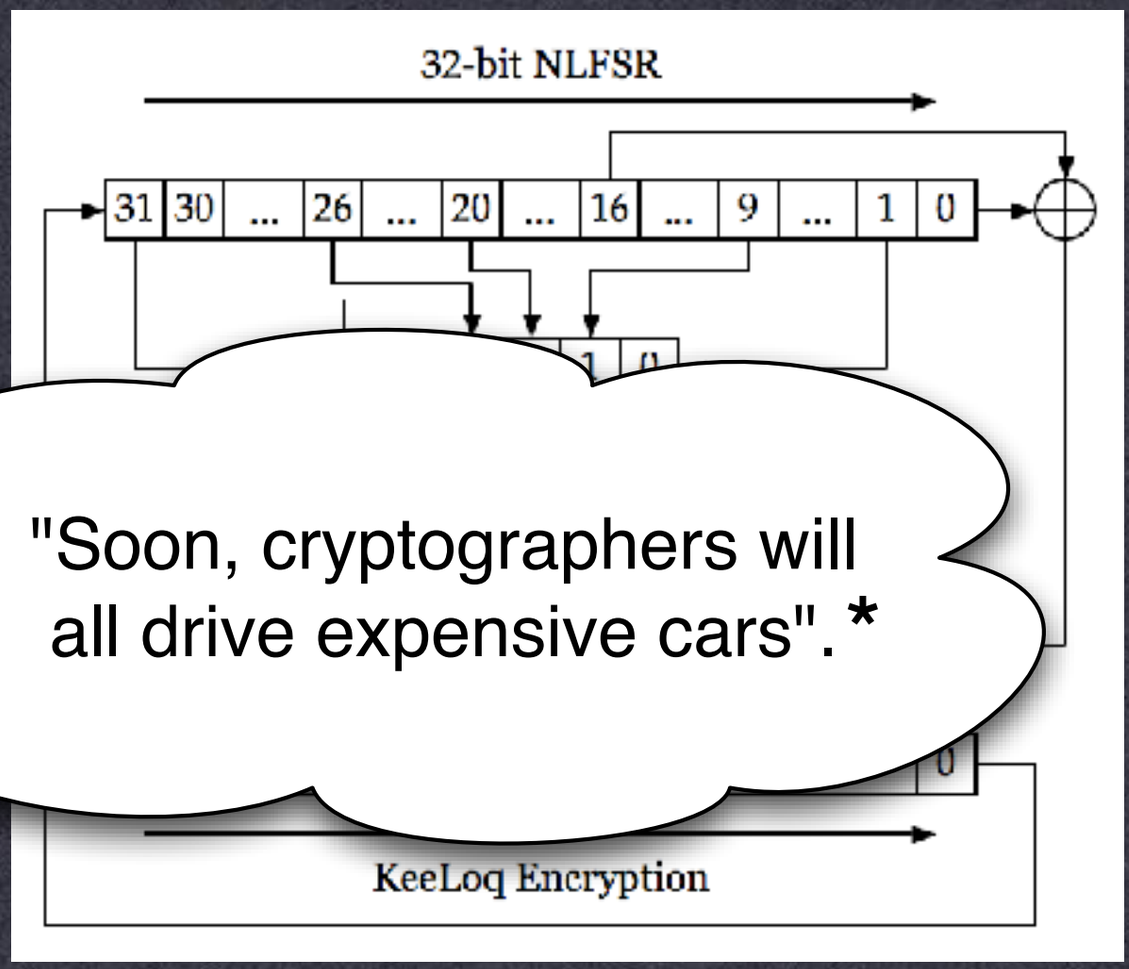
Current Events

```
plaintext: 746869732069732061207465737420737472696e67
ciphertext: d62a2349d993632dddabc30a4a2c8ab7ba2608c5f2
tag: fd6b38cba35ea579918f3b5ec1863e4b
nonce: f102f60a0ef39e310c5f9c4c
decrypted: 746869732069732061207465737420737472696e67

plaintext: 746869732069732061207465737420737472696e67
ciphertext: 0dd0adde0dd0adde0dd0adde0dd0adde585301005d
tag: 995d100cc5d068a83e7ecf13c49e92eb
nonce: cffb9148cb5154232c957ab7
```

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)



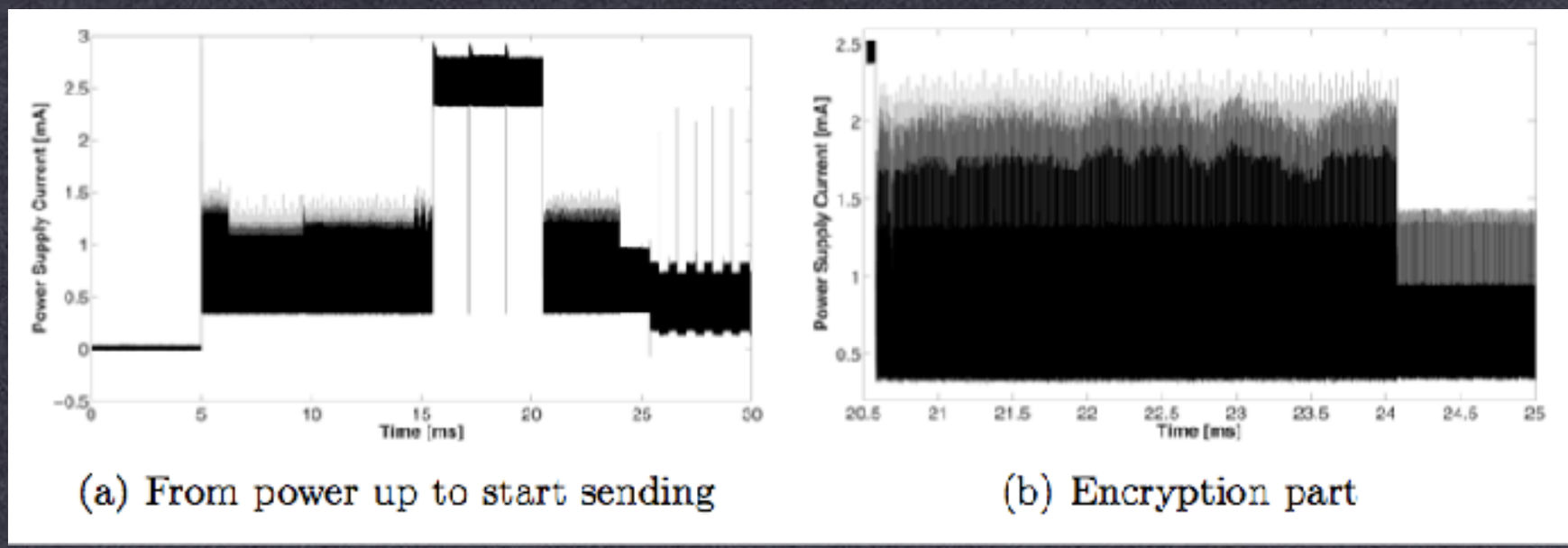
Public domain image from Wikipedia
* Source: Indesteege et al.: http://www.cosic.esat.kuleuven.be/keeloq/keeloq-slides.pdf

KeeLoq

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

KeyLoq

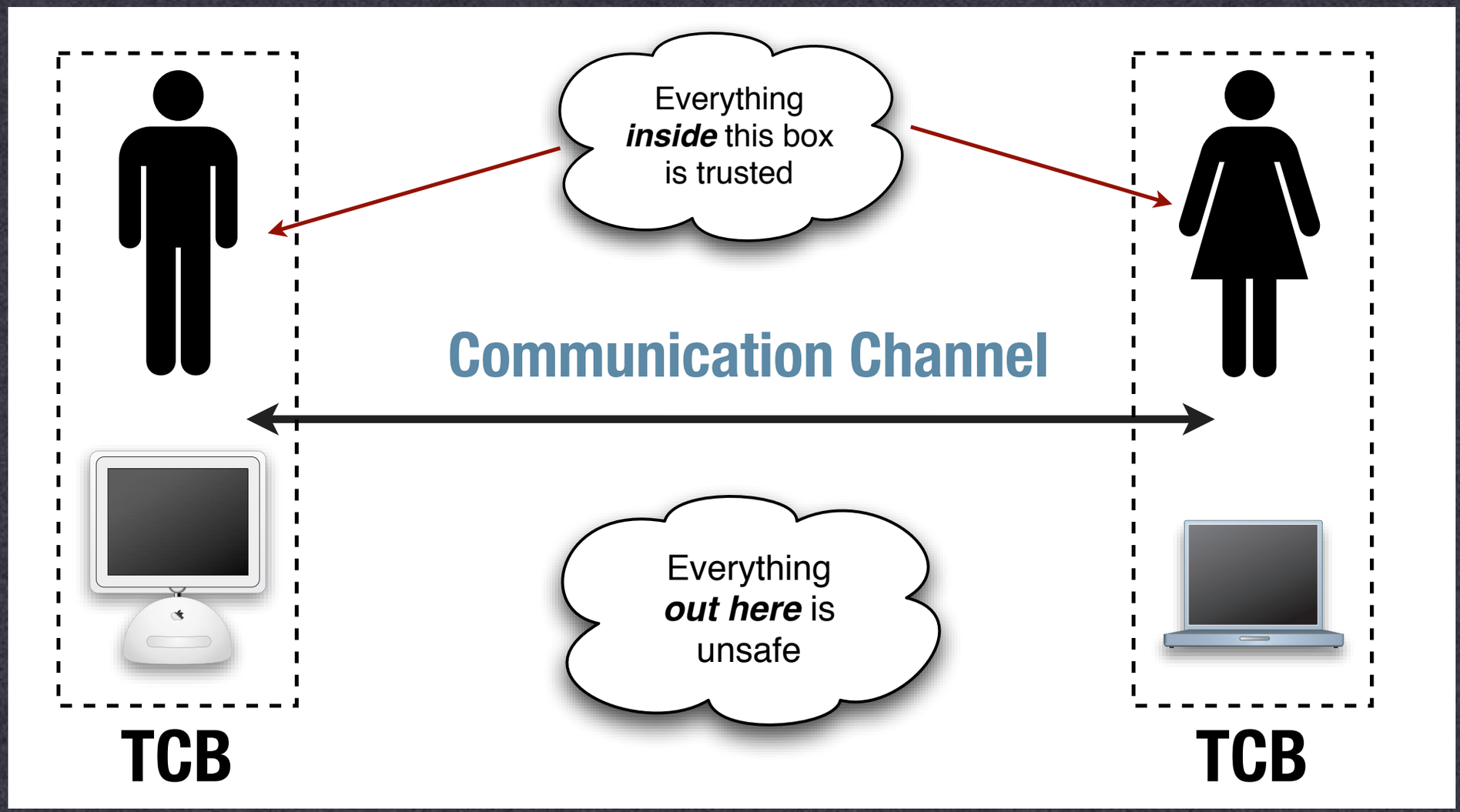
- CRYPTO 2008
  - Doesn't directly attack the cipher
  - Instead, measures how the cipher works in operation.
  - In this case, use power consumption



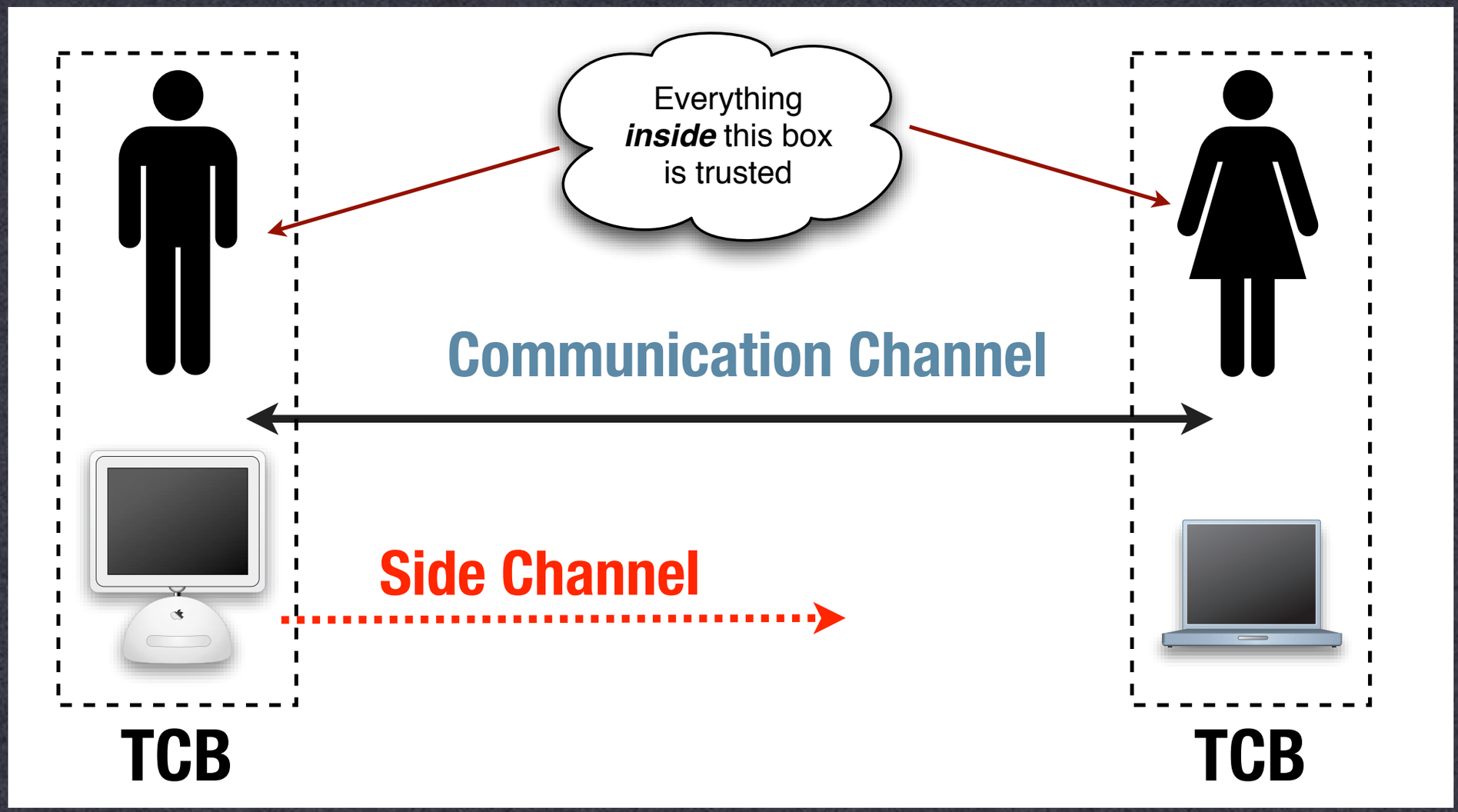
(a) From power up to start sending (b) Encryption part

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

Review

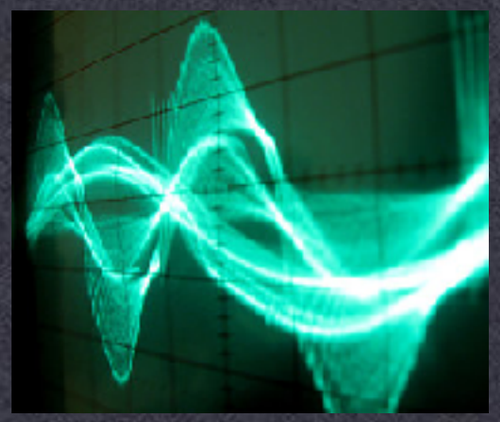

Everything
inside this box
is trusted
Communication Channel
Everything
out here is
unsafe
TCB
TCB

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

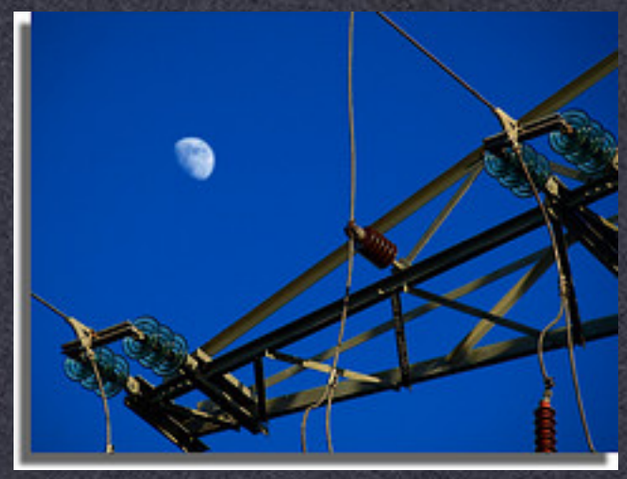


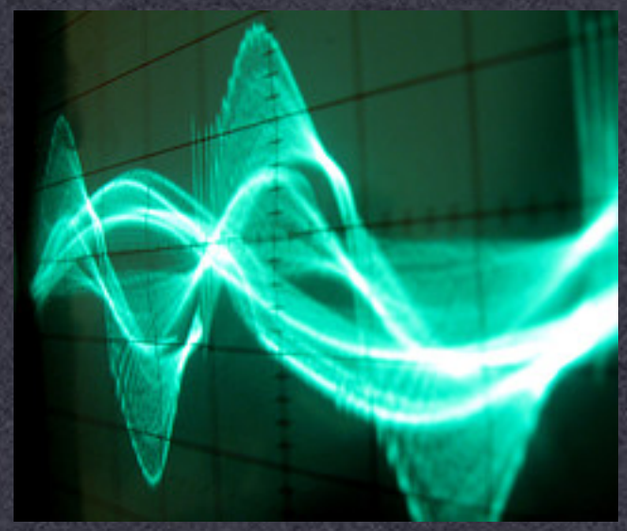
Side Channels

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

Common Examples

- Timing
- Power Consumption
- RF Emissions
- Audio





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

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

$$d = 10101010011101010110010010001001$$

```
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

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

$$d = 10101010011101010110010010001001$$

```
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

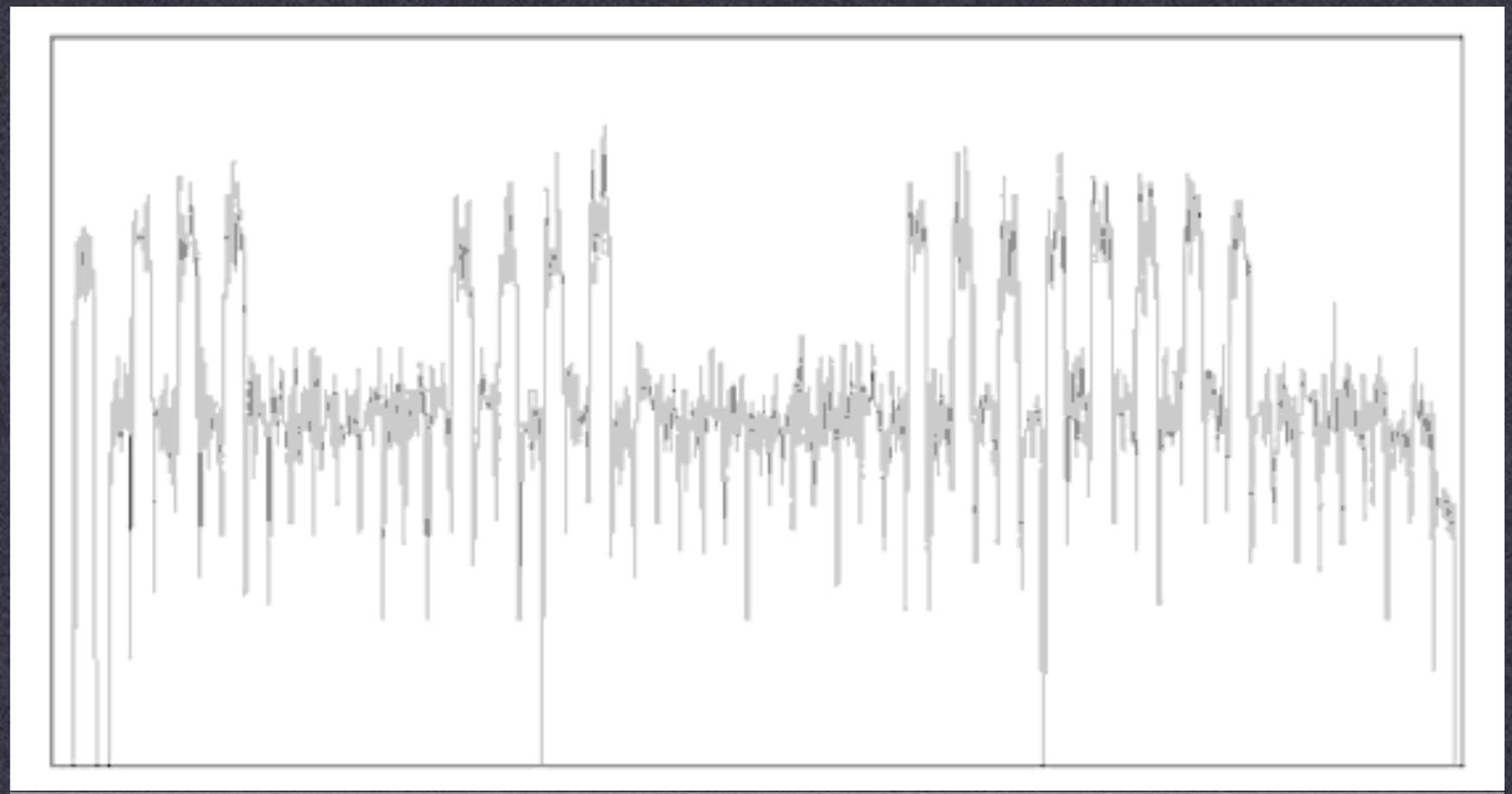
$$m = c^d \ mod \ N$$

$$d = 10101010011101010110010010001001$$

```
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;
}
```

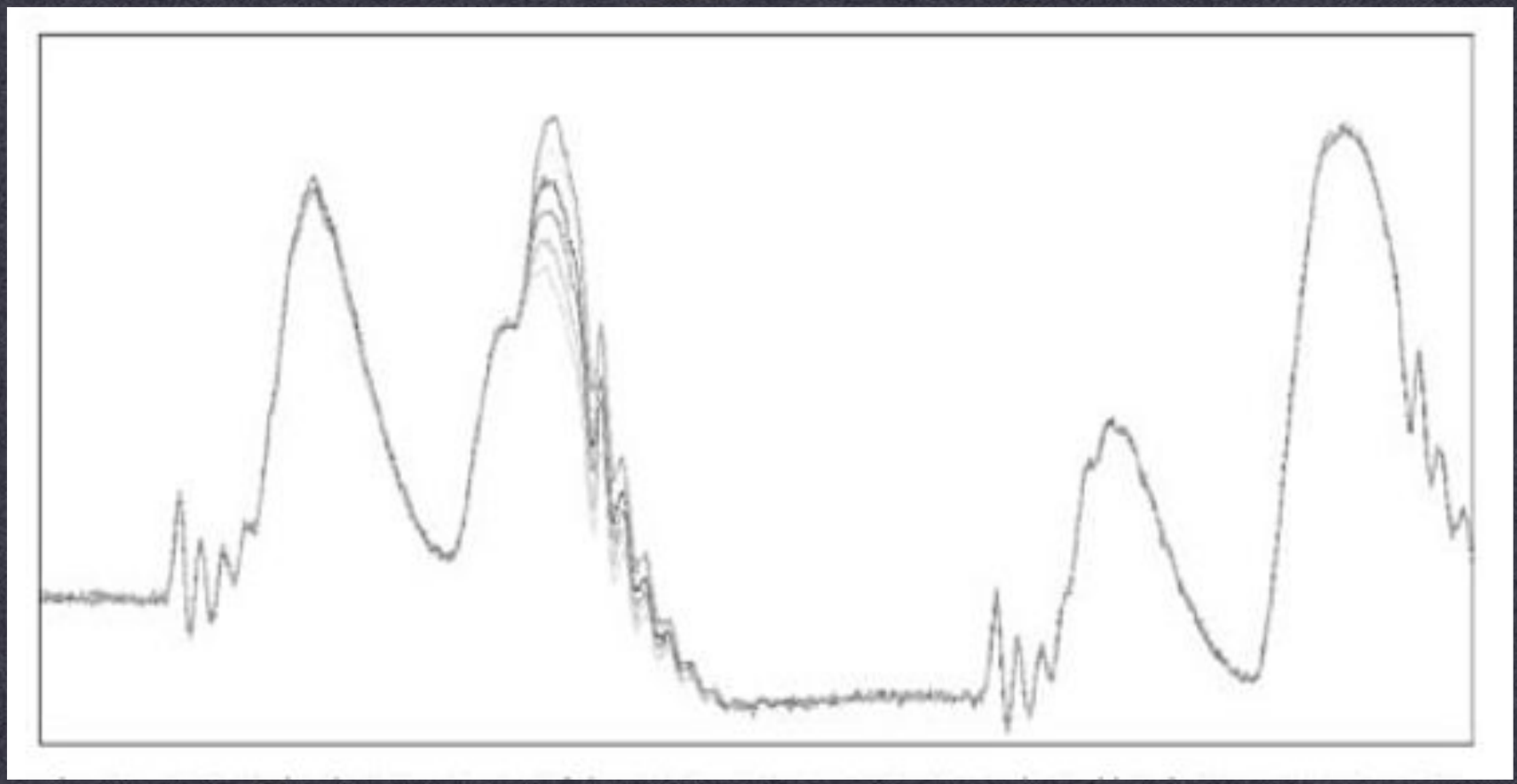
Expensive Operation

Simple Power Analysis



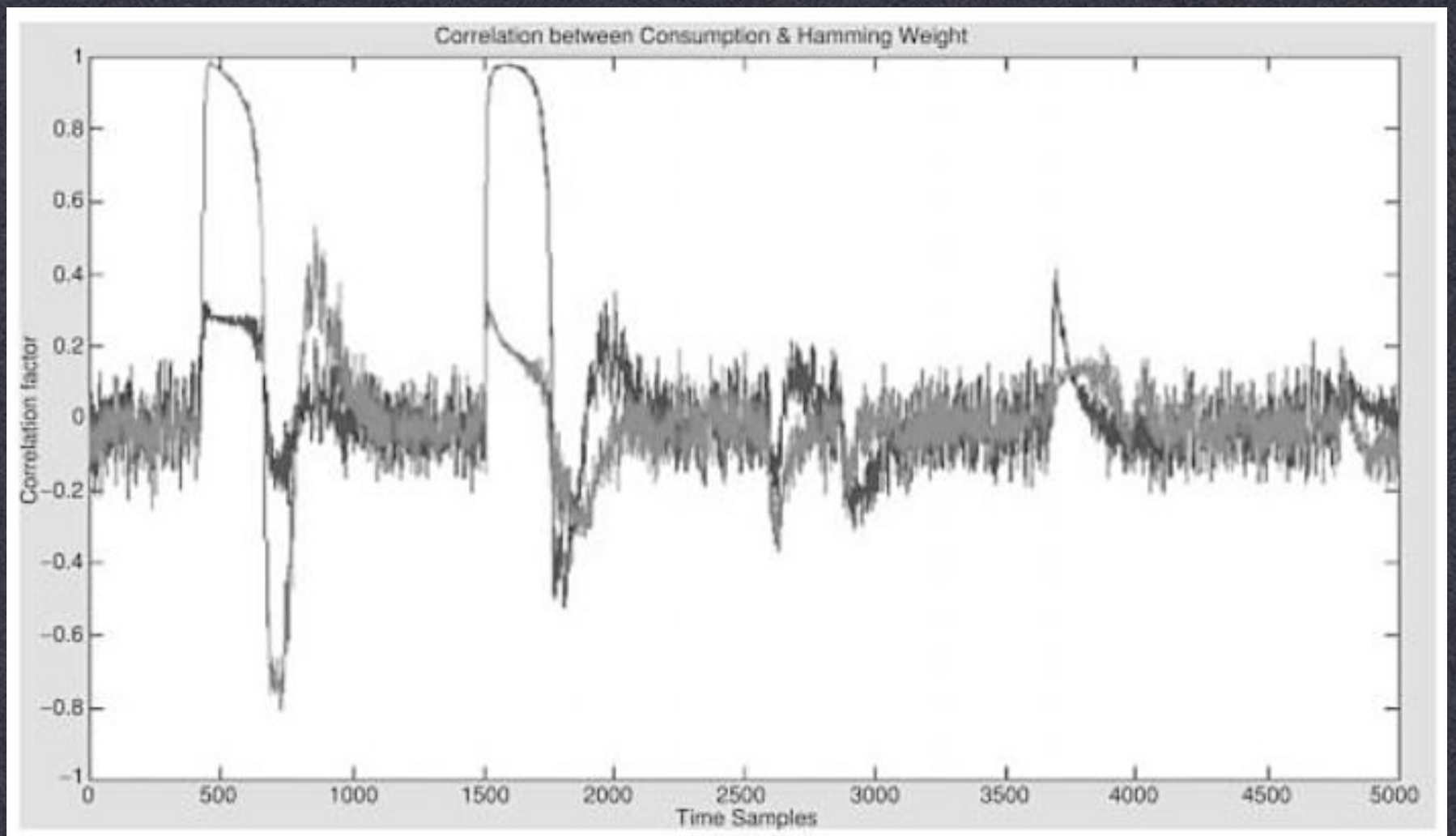
Source: Bidgoli, Handbook of Information Security

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

        c = c² 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

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;
}
```

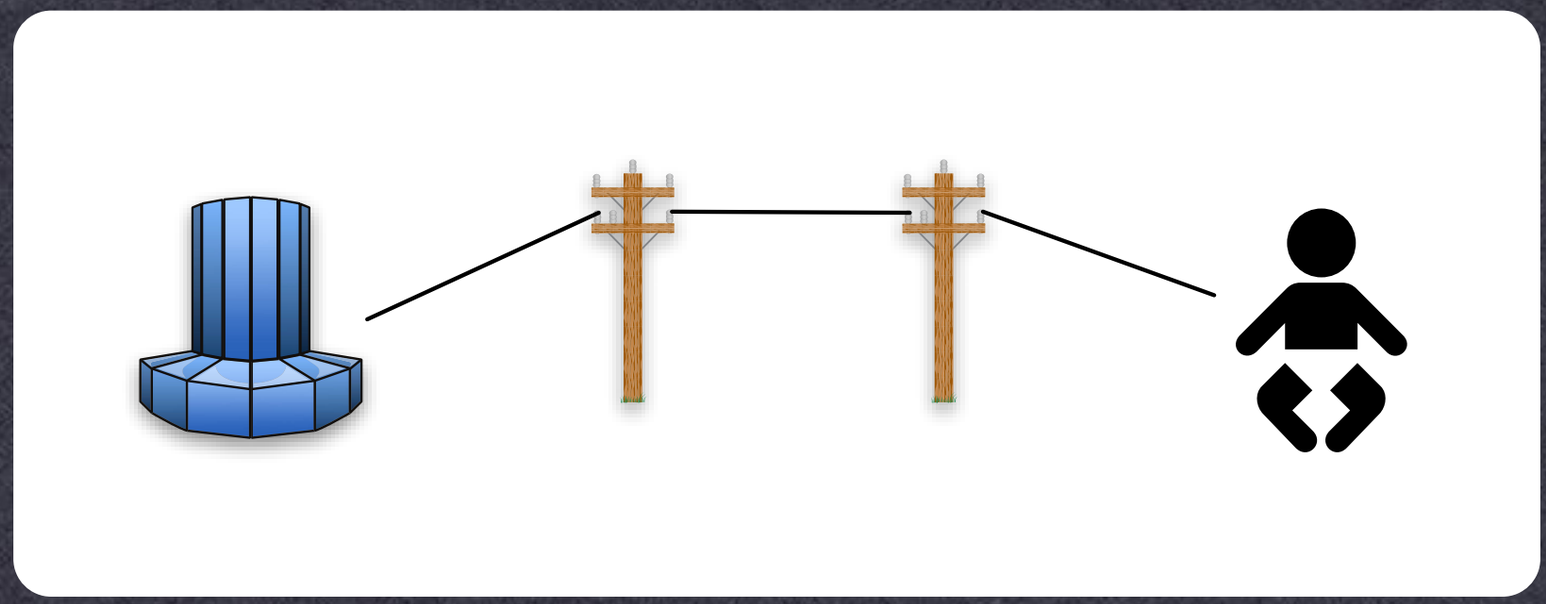
Expensive Operation

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)**

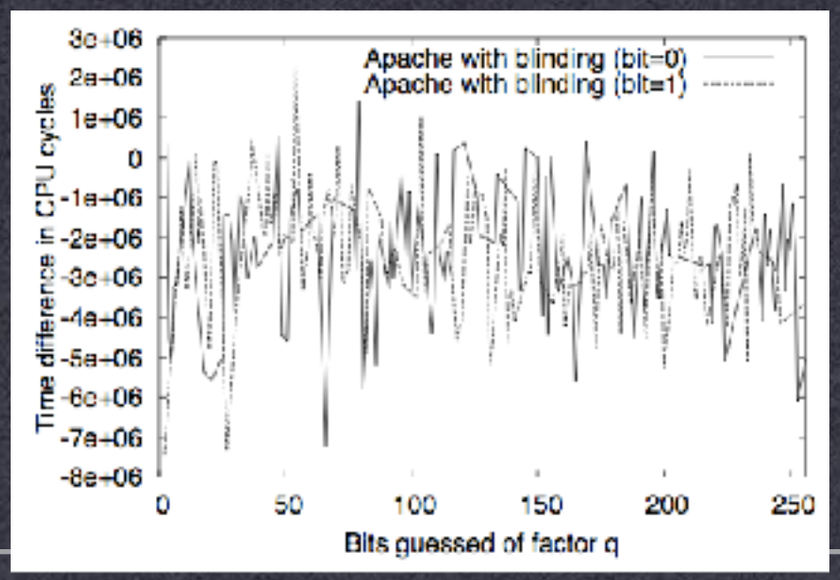
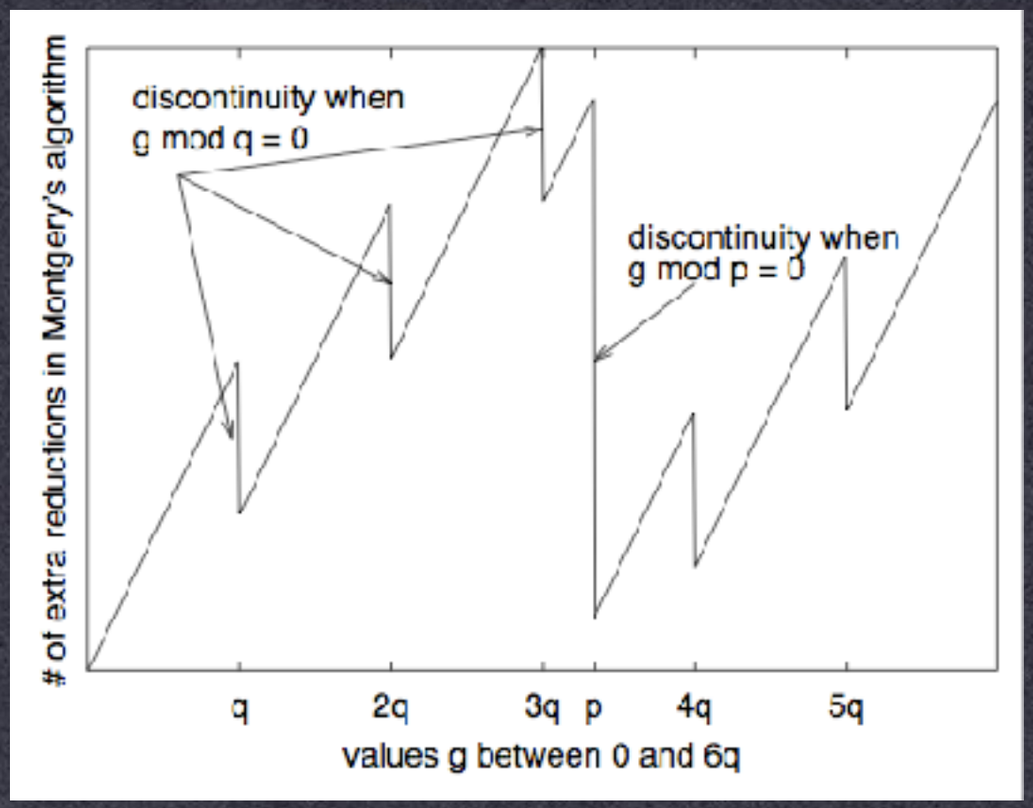
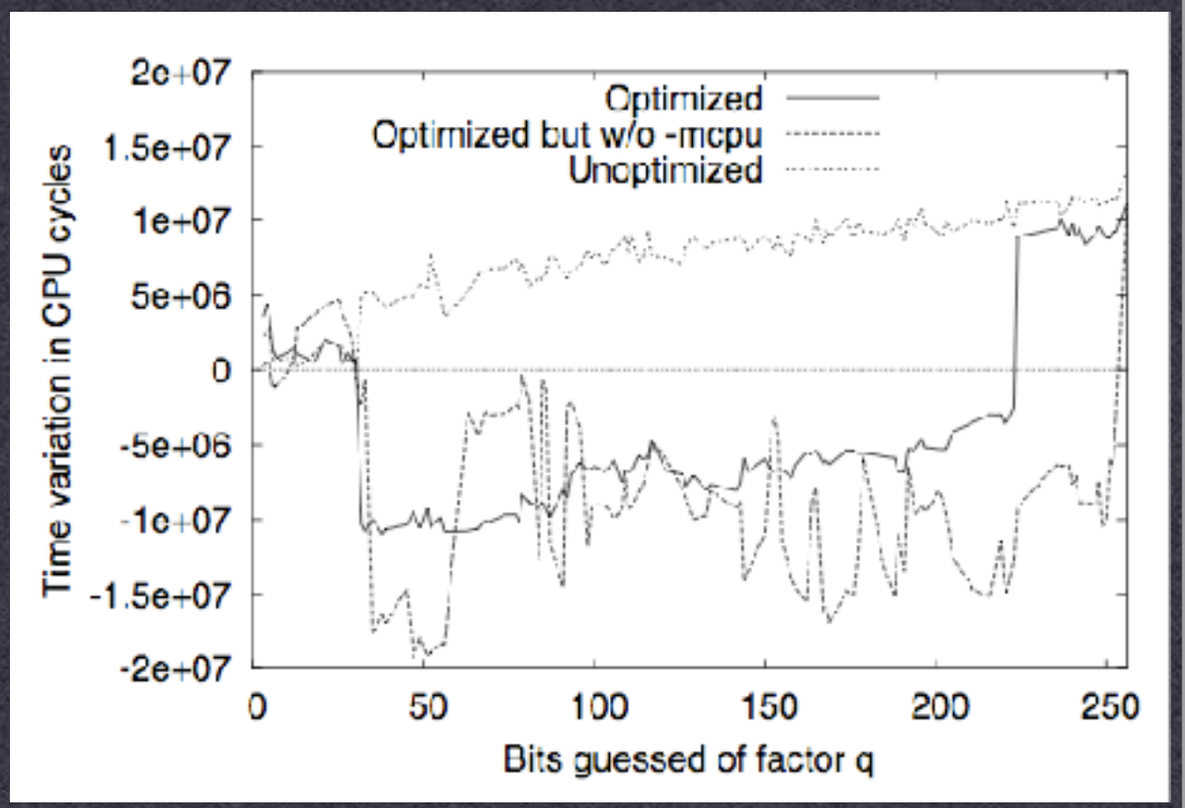


Image Source: Boneh, Brumley: Remote Timing Attacks are Practical

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

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

$$d = 10101010011101010110010010001001$$

- 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

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

$$d = 10101010011101010110010010001001$$

- 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

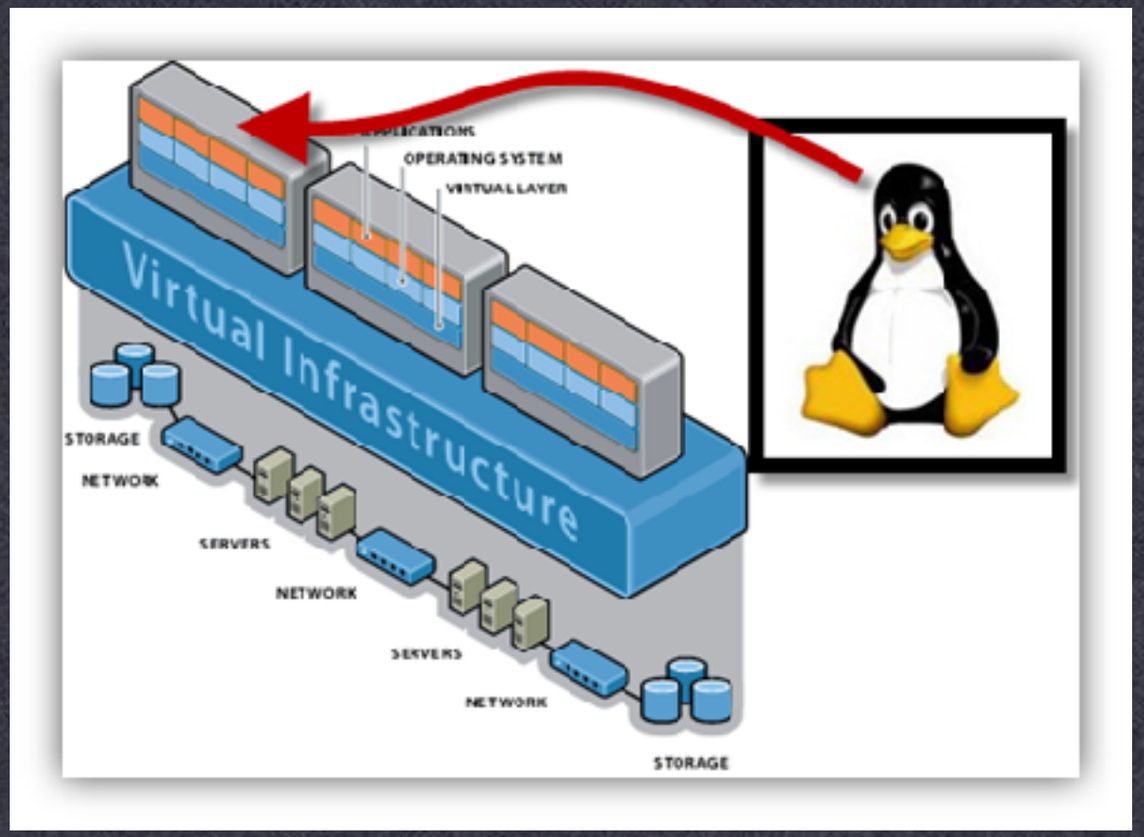
$$m = c^d \bmod N$$

$$d = 10101010011101010110010001001001$$

- 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_

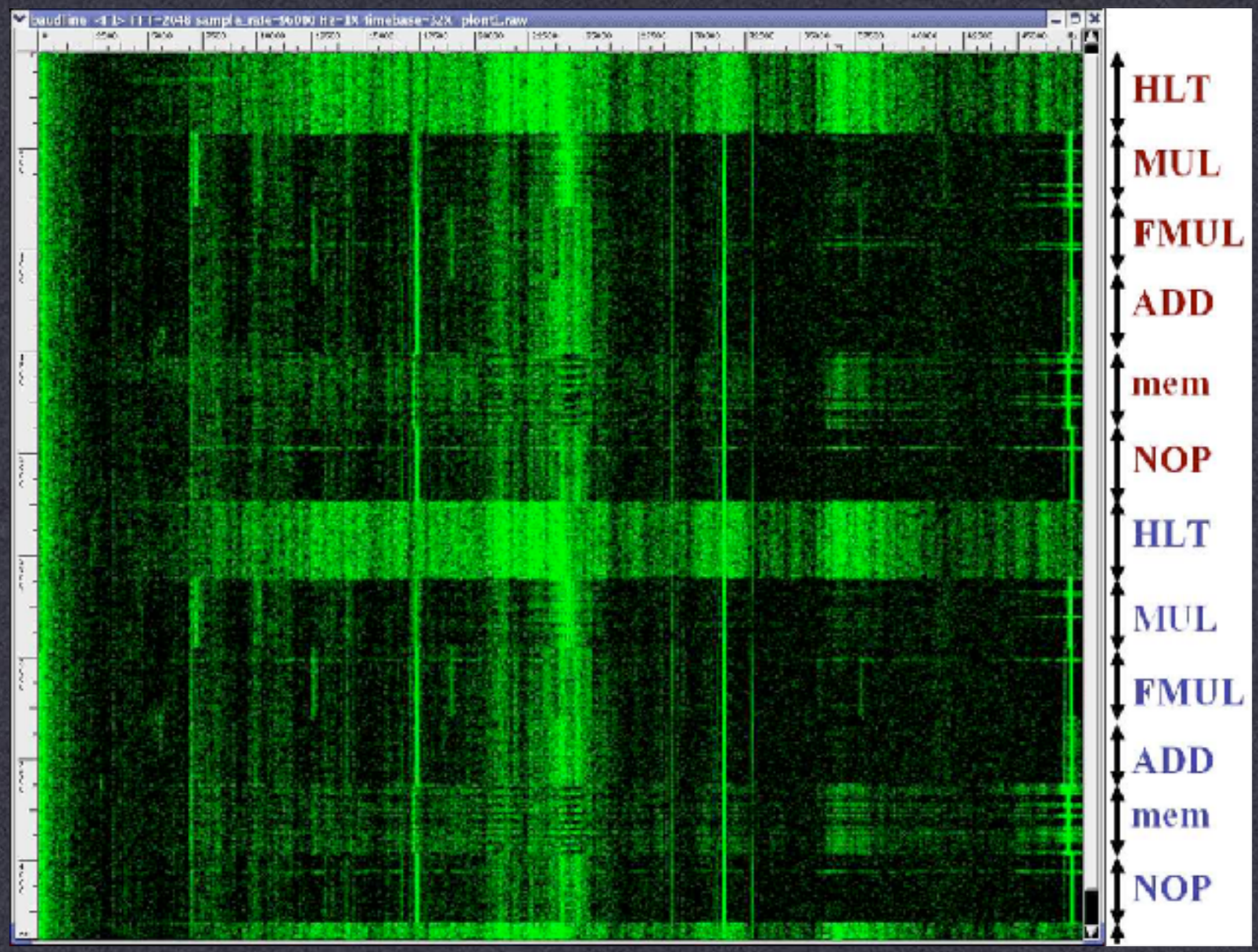
```
1  int SSLDecodeRSAKeyExchange(keyExchange, ctx){
2    keyRef = ctx->signingPrivKeyRef;
3    src  = keyExchange.data;
4    localKeyModulusLen = keyExchange.length;
5    ... // addtional inilization code omitted
6
7    err = sslRsaDecrypt(keyRef, src,
       localKeyModulusLen,
8      ctx->preMasterSecret.data,
       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

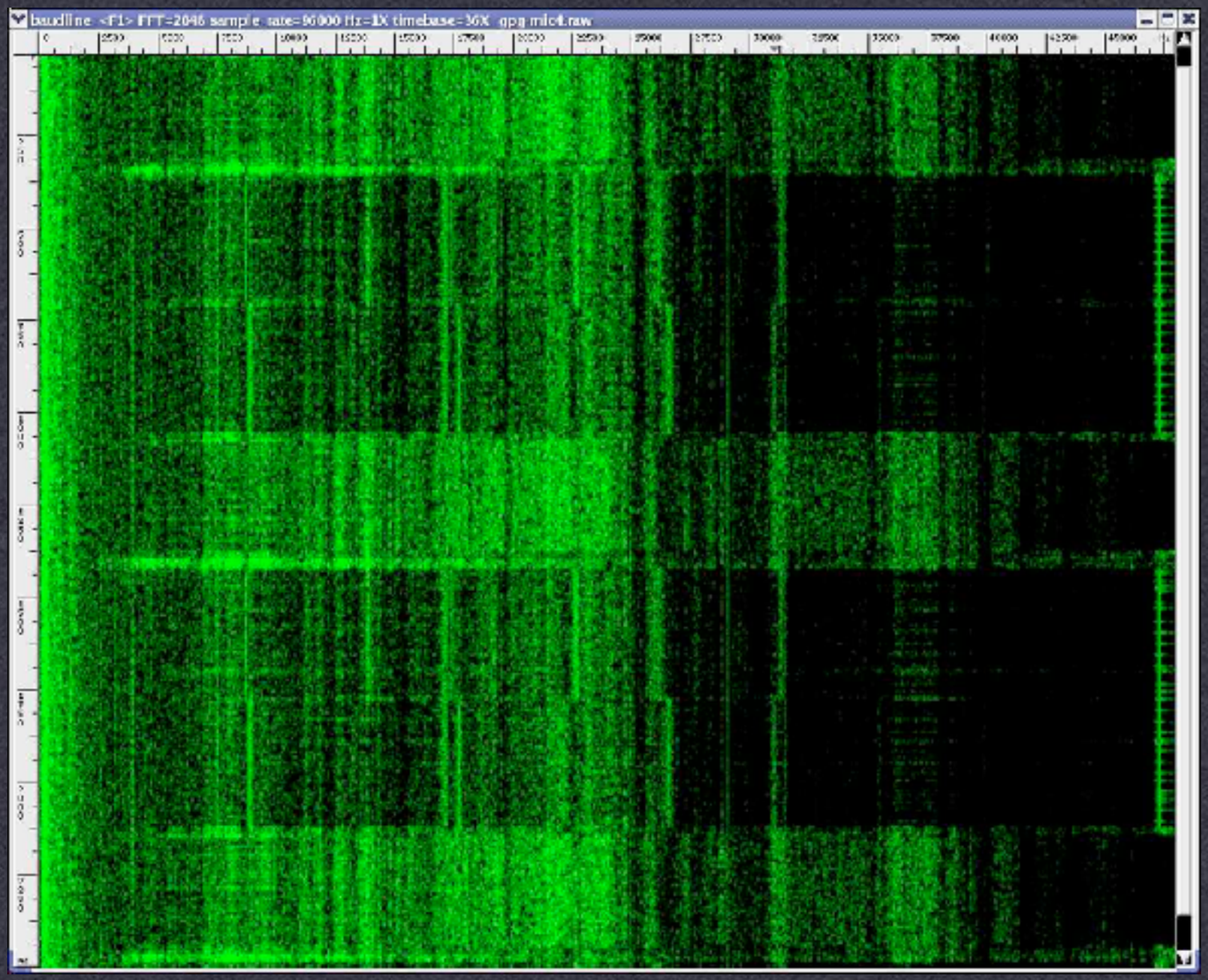
Acoustic Side Channels



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

Acoustic Side-Channels

GPG RSA-CRT signature #1:



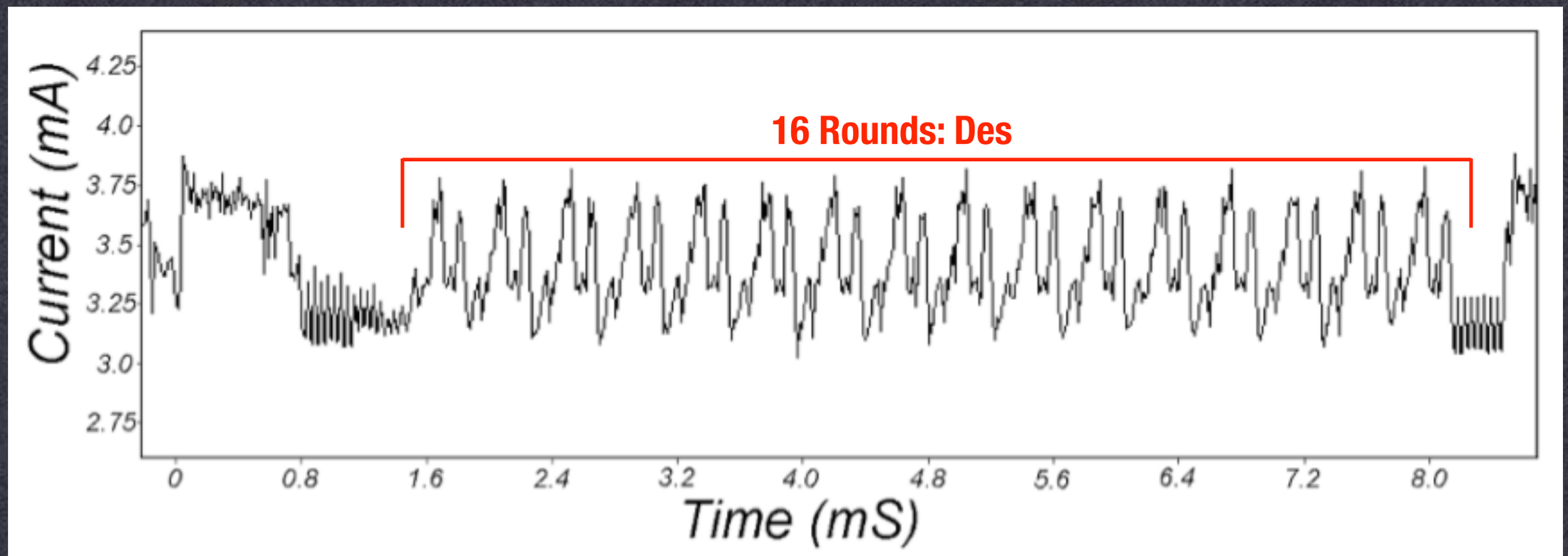
mod q

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**