Side-Channel Cryptanalysis

Joseph Bonneau
Security Group
jcb82@cl.cam.ac.uk
Rule 0: Attackers will *always* cheat

A CRYPTO NERD’S IMAGINATION:

His laptop’s encrypted. Let’s build a million-dollar cluster to crack it.

No good! It’s 4096-bit RSA!

Blast! Our evil plan is foiled!

WHAT WOULD ACTUALLY HAPPEN:

His laptop’s encrypted. Drug him and hit him with this $5 wrench until he tells us the password.

Got it.
What is side channel cryptanalysis?
Side Channels: whatever the designers ignored

\[ c = m^e \mod N \quad \text{Encryption} \quad T_0 = K_0 \oplus P_0 \quad \text{Ciphertext} \]
Side Channels: whatever the designers ignored

Key → Encryption → Ciphertext

Plaintext → Encryption → Ciphertext
Side Channels: whatever the designers ignored
Side Channels: whatever the designers ignored

- Key
- Plaintext
- Encryption
- Ciphertext

- Time
- Power
- Noise
- Heat
- EM radiation
Definition

Side-channel cryptanalysis is any attack on a cryptosystem requiring information emitted as a byproduct of the physical implementation.
Related attacks
White box cryptanalysis: nothing is hidden

Key

Cipher

Plaintext

Encryption

Ciphertext

int k[] = {0x1e, ...}

Debugger

Static analysis

Memory dumps
TEMPEST: EM signals containing full secrets

Key \rightarrow \text{Encryption} \rightarrow \text{Ciphertext} \rightarrow \text{Antenna}

Plaintext \rightarrow \text{Encryption} \rightarrow \text{Ciphertext}
Fault injection: inducing a telling error

- Key
- Plaintext

Encryption

Ciphertext
Hardware attacks: breaking the box open

Key → Plaintext → Encryption → Ciphertext
Covert channels: attack code running from within

Key \rightarrow Cipher \rightarrow Ciphertext

Plaintext \rightarrow Encryption

Ciphertext
The grandmother of all timing attacks
Insecure password checking routine

```c
int check_password(char * test, char * correct){
    return (strcmp(test, correct) == 0);
}
```
Insecure password checking routine

```c
int check_password(char * test, char * correct) {
    return (strcmp(test, correct) == 0);
}

int strcmp(char *s1, char *s2) {
    while (*s1 != '\0' && *s1 == *s2) {
        s1++;
        s2++;
    }

    return ((s1 < s2) ? -1 : (s1 > s2));
}
```
Insecure password checking routine

```c
int check_password(char * test, char * correct)
{
    return (strcmp(test, correct) == 0);
}
```

```c
int strcmp(char *s1, char *s2)
{
    while (*s1 != '\0' && *s1 == *s2) {
        s1++;
        s2++;
    }
    return ((s1 < s2) ? -1 : (s1 > s2));
}
```

$\approx n \cdot |A|$ queries
MAC timing attack against Xbox 360

C:\Documents and Settings\Administrator\Desktop\360\DGTool>DGTool.exe 1 Infectus_5759\4_1888_build2.bin

Pairing Data 0x6DF3B8 01

Turn on your Xbox, press any key when the RRoD starts

H[0 00XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX] M 17821 A 17821 D 0 : 0 NEXT
H[0 01XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX] M 17817 A 17819 D -3 : 0 NEXT
H[0 02XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX] M 17822 A 17820 D 3 : 0 NEXT
...
H[0 1AXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX] M 17822 A 17819 D 3 : 0 NEXT
H[0 1BXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX] M 17831 A 17819 D 12 : 11 RPT
H[0 1BXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX] M 17830 A 17819 D 11 : 0 HIT!
H[1 1B00XXXXXXXXXXXXXXXXXXXXXXXXXXXXX] M 17830 A 17830 D 0 : 0 NEXT
H[1 1B01XXXXXXXXXXXXXXXXXXXXXXXXXXXXX] M 17829 A 17829 D -1 : 0 NEXT
...
H[1 1BFDXXXXXXXXXXXXXXXXXXXXXXXXXXXXX] M 17844 A 17830 D 14 : 8 RPT
H[1 1BFDXXXXXXXXXXXXXXXXXXXXXXXXXXXXX] M 17839 A 17839 D 9 : 0 HIT!
H[2 1BFD00XXXXXXXXXXXXXXXXXXXXXXXXXXX] M 17839 A 17838 D 1 : 0 NEXT
H[2 1BFD01XXXXXXXXXXXXXXXXXXXXXXXXXXX] M 17844 A 17841 D 4 : 0 NEXT
...
H[2 1BFD0FXXXXXXXXXXXXXXXXXXXXXXXXXXX] M 17856 A 17841 D 15 : 11 RPT
H[2 1BFD0FXXXXXXXXXXXXXXXXXXXXXXXXXXX] M 17851 A 17841 D 10 : 0 HIT!
...
H[15 1BFDF0625C214F67CD94DCA3FC47CA55] M 18014 A 17988 D 16 : 0 HIT!

Correct hash: 1BFDF0625C214F67CD94DCA3FC47CA55
Result: BOOT
MAC timing attack against Xbox 360

C:\Documents and Settings\Administrator\Desktop\360\DGTool>DGTool.exe 1 Infectus_5759#4_1888_build2.bin

Pairing Data 0x6DF3B8 01
Turn on your Xbox, press any key when the RRoD starts
H[0 00XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX] M 17821 A 17821 D 0 : 0 NEXT
H[0 01XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX] M 17817 A 17819 D -3 : 0 NEXT
H[0 02XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX] M 17822 A 17820 D 3 : 0 NEXT
... H[0 1AXXXXXXXXXXXXXXXXXXXXXXXXXXXXX] M 17822 A 17819 D 3 : 0 NEXT
H[0 1BXXXXXXXXXXXXXXXXXXXXXXXXXXXX] M 17831 A 17819 D 12 : 11 RPT
H[1 1B00XXXXXXXXXXXXXXXXXXXXXXXXXXXX] M 17830 A 17830 D 0 : 0 NEXT
H[1 1B01XXXXXXXXXXXXXXXXXXXXXXXXXXXX] M 17829 A 17829 D -1 : 0 NEXT
... H[1 1BFDF0XXXXXXXXXXXXXXXXXXXXXXXXXX] M 17844 A 17841 D 4 : 0 NEXT

Correct hash: 1BFDF0625C214F67CD94DCA3FC47CA55
Result: BOOT

≈ 2,048 queries
Attacks against RSA
RSA: The original public key algorithm
RSA: The original public key algorithm

Private Key: $p, q$ (random primes)
$\text{d} \equiv e^{-1} \pmod{\phi(N)}$ (exponent)

Public Key: $N = p \cdot q$ (modulus)
$e$ (exponent)

Encryption: $c = m^e \pmod{N}$

Verification: $m = c^d \pmod{N}$

Signing: $s = m^d \pmod{N}$

Verification: $m = s^e \pmod{N}$
RSA is implemented via square-and-multiply

\[ b^{65553} \equiv b^{0x10011} \pmod{N} \]
\[ \equiv b^{(10000000000010001)} \pmod{N} \]
\[ \equiv b^{65536} \cdot b^{16} \cdot b^{1} \pmod{N} \]

\( n \) digit exponentiation requires \( \log n \) squares, up to \( \log n \) multiplications
RSA is implemented via square-and-multiply

```python
def power(b, e, N):
    result = 1

    for i in range(len(e)-1, -1):
        result = square(result, N)
        if bit_set(e, i):
            result = mult(result, b, N)

    return result
```
Simple power analysis of RSA
(Paul Kocher et. al 1999)
Simple power analysis setup

Key → Plaintext → Encryption → Ciphertext
What does power consumption reveal?

Trace courtesy of Cryptography Research, Inc.
Each set exponent bit inserts a multiplication

def power(b, e, N):
    result = 1
    for i in range(len(e)-1, -1):
        result = square(result, N)
        if bit_set(e, i):
            result = mult(result, b, N)
    return result
Multiplies can often be visually detected

Trace courtesy of Cryptography Research, Inc.
Multiplies can often be visually detected

Trace courtesy of Cryptography Research, Inc.
Secret exponent can be easily read out

Trace courtesy of Cryptography Research, Inc.
Secret exponent can be easily read out

1 encryption

Trace courtesy of Cryptography Research, Inc.
Algorithmic patch: square and **always** multiply

```python
def power(b, e, N):
    result = 1
    b = mult(b, r, N)
    for i in range(len(e)-1, -1, -1):
        result = square(result, N)
        if bit_set(e, i):
            result = mult(result, b, N)
        else:
            result = mult(result, r, N)
    return mult(result, r_inverse, N)
```
Timing attack against RSA
(Paul Kocher 1996)
Timing attack setup

Key → Plaintext → Encryption → Ciphertext
def power(b, e, N):
    result = 1

    for i in range(len(e) - 1, -1):
        result = square(result, N)
        if bit_set(e, i):
            result = mult(result, b, N)

    return result
Timing of individual multiplies varies significantly

Kocher 1996
Need a model relating multiplication inputs to time

Example:

\[2345 \cdot 6789 \pmod{9997}\]

**Multiply:** \[2345 \cdot 6826 = 16006970\]

**Reduce:**

\[16006970 - 9997 \cdot 1 \cdot 1000 = 6009970 - 9997 \cdot 6 \cdot 100 = 11770 - 9997 \cdot 0 \cdot 10 = 11770 - 9997 \cdot 1 \cdot 1 = 1773\]
def power(b, e, N):
    result = 1

    for i in range(len(e)-1, -1, -1):
        result = square(result, N)
        if bit_set(e, i):
            result = mult(result, b, N)

    return result
Attack exponent one bit at a time

\[ T = \text{observed timing of entire algorithm} \]
\[ M = \text{model for time of one multiplication} \]

Bit n-1: always 1
Attack exponent one bit at a time

\[ T = \text{observed timing of entire algorithm} \]

\[ M = \text{model for time of one multiplication} \]

\[ \text{Bit n-2: } \quad \text{Is } T(\text{power}(r, e, N)) \propto \]

\[ M(\text{mult}(1, r, N)) + \]

\[ M(\text{square}(r, N)) + \]

\[ M(\text{mult}(r^2, r, N)) + \]

\[ M(\text{square}(r^3, N))? \]
Attack exponent one bit at a time

\[ T = \text{observed timing of entire algorithm} \]
\[ M = \text{model for time of one multiplication} \]

Bit n-2: Is \( T(\text{power}(r, e, N)) \propto \) \\
\[ M(\text{mult}(1, r, N)) + \]
\[ M(\text{square}(r, N)) + \]
\[ M(\text{mult}(r^2, r, N)) + \]
\[ M(\text{square}(r^3, N)) \]?
Attack exponent one bit at a time

\[ T = \text{observed timing of entire algorithm} \]
\[ M = \text{model for time of one multiplication} \]

Bit n-3: \( \log T(\text{power}(r, e, N)) \propto \]
\[ M(\text{mult}(1, r, N)) \cdot e^{[n-1]} + \]
\[ M(\text{square}(r, N)) + \]
\[ M(\text{mult}(r^{2 \cdot e^{[n-1:n-2]}}, r, N)) \cdot e^{[n-2]} + \]
\[ M(\text{square}(r^{e^{[n-1:n-2]}}, N)) \]
\[ M(\text{mult}(r^{2 \cdot e^{[n-1:n-2]}}, r, N)) + \]
\[ M(\text{square}(r^{e^{[n-1:n-2]|1}}, N)) \]
Attack exponent one bit at a time

\[ T = \text{observed timing of entire algorithm} \]

\[ M = \text{model for time of one multiplication} \]

Bit n-3: \( \text{Is } T(power(r, e, N)) \propto \)

\[ M(mult(1, r, N)) \cdot e[n-1] + \]

\[ M(square(r, N)) + \]

\[ M(mult(r^{2 \cdot e[n-1:n-1]} r, N)) \cdot e[n-2] + \]

\[ M(square(r^{e[n-1:n-2]} r, N)) \]

\[ M(mult(r^{2 \cdot e[n-1:n-2]} r, N)) + \]

\[ M(square(r^{e[n-1:n-2]} ||1 r, N)) ? \]
Attack exponent one bit at a time

\[ T = \text{observed timing of entire algorithm} \]
\[ M = \text{model for time of one multiplication} \]

Bit \( n-i \): \[ \text{Is } T(\text{power}(r, e, N)) \propto M(\text{mult}(r^{e_{[n-1:n-i]}}, r, N)) + M(\text{square}(r^{e_{[n-1:n-i]}||1}, N))? \]
Attack exponent one bit at a time

\[ T = \text{observed timing of entire algorithm} \]
\[ M = \text{model for time of one multiplication} \]

\[ \text{Bit } n-i: \quad \text{Is } T(\text{power}(r, e, N)) \propto M(\text{mult}(r^{2 \cdot e \cdot [n-1:n-i]}, r, N) + M(\text{square}(r^e|n-1:n-i|, 1, N)) \approx 2,500 \text{ encryptions} \]
More complicated attacks work across a LAN

Boneh and Brumley, 2003
More complicated attacks work across a LAN.

Discontinuity when $g \mod q = 0$.

Discontinuity when $g \mod p = 0$.

$\approx 1,000,000$ encryptions.

Boneh and Brumley, 2003
Blinded RSA provides generic defense

Private Key: \( p, q \) (random primes)
\[
d \equiv e^{-1} \pmod{\varphi(N)} \quad \text{(exponent)}
\]

Public Key: \( N = p \cdot q \) (modulus)
\( e \) (exponent)

Signing: \( s = m^d \pmod{N} \)

Blind Signing: \( r_1 = r_0^e \pmod{N} \)
\[
s = r_0^{-1} (r_1 \cdot m)^d \pmod{N}
\]
Attacks against AES (Rijndael)
AES is cryptography's standard block cipher
AES is very complicated
AES is very complicated
AES is very complicated

<table>
<thead>
<tr>
<th>No change</th>
<th>a_{0,0}</th>
<th>a_{0,1}</th>
<th>a_{0,2}</th>
<th>a_{0,3}</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shift 1</td>
<td>a_{1,0}</td>
<td>a_{1,1}</td>
<td>a_{1,2}</td>
<td>a_{1,3}</td>
</tr>
<tr>
<td>Shift 2</td>
<td>a_{2,0}</td>
<td>a_{2,1}</td>
<td>a_{2,2}</td>
<td>a_{2,3}</td>
</tr>
<tr>
<td>Shift 3</td>
<td>a_{3,0}</td>
<td>a_{3,1}</td>
<td>a_{3,2}</td>
<td>a_{3,3}</td>
</tr>
</tbody>
</table>

ShiftRows:

<table>
<thead>
<tr>
<th>a_{0,0}</th>
<th>a_{0,1}</th>
<th>a_{0,2}</th>
<th>a_{0,3}</th>
</tr>
</thead>
<tbody>
<tr>
<td>a_{1,1}</td>
<td>a_{1,2}</td>
<td>a_{1,3}</td>
<td>a_{1,0}</td>
</tr>
<tr>
<td>a_{2,2}</td>
<td>a_{2,3}</td>
<td>a_{2,0}</td>
<td>a_{2,1}</td>
</tr>
<tr>
<td>a_{3,3}</td>
<td>a_{3,0}</td>
<td>a_{3,1}</td>
<td>a_{3,2}</td>
</tr>
</tbody>
</table>
AES is very complicated

\[ \begin{align*}
    a_{0,0} & \rightarrow a_{0,1} & \rightarrow a_{0,2} & \rightarrow a_{0,3} \\
    a_{1,0} & \rightarrow a_{1,1} & \rightarrow a_{1,2} & \rightarrow a_{1,3} \\
    a_{2,0} & \rightarrow a_{2,1} & \rightarrow a_{2,2} & \rightarrow a_{2,3} \\
    a_{3,0} & \rightarrow a_{3,1} & \rightarrow a_{3,2} & \rightarrow a_{3,3}
\end{align*} \]

\[ \text{MixColumns} \]

\[ \begin{align*}
    b_{0,0} & \rightarrow b_{0,1} & \rightarrow b_{0,2} & \rightarrow b_{0,3} \\
    b_{1,0} & \rightarrow b_{1,1} & \rightarrow b_{1,2} & \rightarrow b_{1,3} \\
    b_{2,0} & \rightarrow b_{2,1} & \rightarrow b_{2,2} & \rightarrow b_{2,3} \\
    b_{3,0} & \rightarrow b_{3,1} & \rightarrow b_{3,2} & \rightarrow b_{3,3}
\end{align*} \]

\[ \bigotimes c(x) \]
AES is very complicated
AES is designed for very efficient implementation

\[
\begin{align*}
  t_0 &= \text{Te0}[\text{s0} \gg 24] \uparrow \\
       &\quad \text{Te1}[\text{s1} \gg 16] \& 0xff \uparrow \\
       &\quad \text{Te2}[\text{s2} \gg 8] \& 0xff \uparrow \\
       &\quad \text{Te3}[\text{s3}] \& 0xff \uparrow \\
       &\quad \text{rk}[0]; \\
  t_1 &= \text{Te0}[\text{s1} \gg 24] \uparrow \\
       &\quad \text{Te1}[\text{s2} \gg 16] \& 0xff \uparrow \\
       &\quad \text{Te2}[\text{s3} \gg 8] \& 0xff \uparrow \\
       &\quad \text{Te3}[\text{s0}] \& 0xff \uparrow \\
       &\quad \text{rk}[1]; \\
  t_2 &= \text{Te0}[\text{s2} \gg 24] \uparrow \\
       &\quad \text{Te1}[\text{s3} \gg 16] \& 0xff \uparrow \\
       &\quad \text{Te2}[\text{s0} \gg 8] \& 0xff \uparrow \\
       &\quad \text{Te3}[\text{s1}] \& 0xff \uparrow \\
       &\quad \text{rk}[2]; \\
  t_3 &= \text{Te0}[\text{s3} \gg 24] \uparrow \\
       &\quad \text{Te1}[\text{s0} \gg 16] \& 0xff \uparrow \\
       &\quad \text{Te2}[\text{s1} \gg 8] \& 0xff \uparrow \\
       &\quad \text{Te3}[\text{s2}] \& 0xff \uparrow \\
       &\quad \text{rk}[3];
\end{align*}
\]
AES utilises large pre-computed lookup tables

```c
static const u32 Te0[256] = {
    0xc66363a5U, 0xf87c7c84U, 0xee777799U, 0xf67b7b8dU,
    0xfff2f20dU, 0xd66b6bbdU, 0xde6f6fb1U, 0x91c5c554U,
    0x60303050U, 0x02010103U, 0xce6767a9U, 0x562b2b7dU,
    0xe7fefe19U, 0xb5d7d762U, 0x4dababe6U, 0xec76769aU,
    ...  
    0x824141c3U, 0x299999b0U, 0x5a2d2d77U, 0x1e0f0f11U,
    0x7bb0b0cbU, 0xa85454fcU, 0x6dbbbbd6U, 0x2c16163aU,
};
```
Lookups into shared cache are vulnerable
Lookups into shared cache are vulnerable

First round: $T[P_i \oplus K_i]$
Lookups into shared cache are vulnerable

First round: $T[P_i \oplus K_i]$

Final round: $T[T^3[C_i \oplus K_i]]$
Simple power analysis of AES

(Bertoni et. Al, 2005; Bonneau 2006)
Cache hit/miss is very obvious in power trace

Bertoni et. al, 2005
Every miss yields many constraints

\[ P_0 \oplus K_0 \neq P_1 \oplus K_1 \]

\[ P_0 \oplus K_0 = P_1 \oplus K_1 \]
Every miss yields many constraints

Plaintext

Key XOR

Lookup

Miss?
\[
P_0 \oplus K_0 \neq P_1 \oplus K_1 \\
P_0 \oplus P_1 \neq K_0 \oplus K_1
\]

Hit?
\[
P_0 \oplus K_0 \overset{?}{=} P_1 \oplus K_1 \\
P_0 \oplus P_1 \overset{?}{=} K_0 \oplus K_1
\]
Every miss yields many constraints

\[
P_0 \oplus P_2 \neq K_0 \oplus K_2 \land P_1 \oplus P_2 \neq K_1 \oplus K_2
\]
### Table of possible key byte differences refined

<table>
<thead>
<tr>
<th></th>
<th>K0</th>
<th>K1</th>
<th>K2</th>
<th>...</th>
<th>K15</th>
</tr>
</thead>
<tbody>
<tr>
<td>K0</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>K1</td>
<td>{27, e0}</td>
<td>{32, 45, 89}</td>
<td>{5f, f3}</td>
<td>{17, 64, 9c}</td>
<td>{42, d5}</td>
</tr>
<tr>
<td>K2</td>
<td>{35}</td>
<td>{23, 70, c4}</td>
<td>{86}</td>
<td>{0a, db}</td>
<td>{65}</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>K15</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>00</td>
</tr>
</tbody>
</table>
Table of possible key byte differences refined

<table>
<thead>
<tr>
<th></th>
<th>K0</th>
<th>K1</th>
<th>K2</th>
<th>...</th>
<th>K15</th>
</tr>
</thead>
<tbody>
<tr>
<td>K0</td>
<td>00</td>
<td>{27, e0}</td>
<td>{35}</td>
<td>{23,70,c}</td>
<td>{65}</td>
</tr>
<tr>
<td>K1</td>
<td></td>
<td>{32,45,89}</td>
<td>{5f,f3}</td>
<td></td>
<td>{0a,db}</td>
</tr>
<tr>
<td>K2</td>
<td></td>
<td></td>
<td>00</td>
<td>{86}</td>
<td>{17,64,9c}</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td>00</td>
<td>{42,d5}</td>
</tr>
<tr>
<td>K15</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>00</td>
</tr>
</tbody>
</table>
Cache observation attack
(Osvik et. al, 2006)
1) Attacker “primes” the cache with known data

```c
void * p = malloc(CACHE_SIZE);
while(i < CACHE_SIZE)
    p[i++]++;
```
1) Attacker “primes” the cache with known data

RAM

AES

Attacker

Cache

```c
void * p = malloc(CACHE_SIZE);
while(i < CACHE_SIZE)
    p[i++]+=;
```
2) Attacker triggers AES encryption

```c
void * p = malloc(CACHE_SIZE);

while(i < CACHE_SIZE)
    p[i++]++;

aes_encrypt(random_p());
```
3) AES loads some cache lines

```
void * p = malloc(CACHE_SIZE);
while(i < CACHE_SIZE)
    p[i++]++;
aes_encrypt(random_p());
```
4) Attacker can test which lines were touched

```
void * p = malloc(CACHE_SIZE);
while(i < CACHE_SIZE)
    p[i++]++;
aes_encrypt(random_p());
while(i < CACHE_SIZE)
    t[i++] = timed_read(p, i);
```
5) All untouched lines yield constraints

\[ P_0 \oplus K_0 \notin \{ \text{Untouched lines} \} \]
5) All untouched lines yield constraints

\[ K_0 \notin \{\text{Untouched lines } \oplus P_0\} \]
5) All untouched lines yield constraints

\[ K_0 \notin \{ \text{Untouched lines } \oplus P_0 \} \]

\( \approx 300 \) encryptions
Cache timing attack
(Bonneau and Mironov, 2006)
Observation: self-collisions lower encryption time

\[
P_i \oplus K_i \neq P_j \oplus K_j
\]
Observation: self-collisions lower encryption time

Plaintext

Key XOR

Lookup

\[ P_i \oplus K_i \not\equiv P_j \oplus K_j \]

\[ P_i \oplus P_j \equiv K_i \oplus K_j \]
Internal collisions cause most timing variation
Key byte differences ranked by average time

<table>
<thead>
<tr>
<th></th>
<th>K0</th>
<th>K1</th>
<th>K2</th>
<th>...</th>
<th>K15</th>
</tr>
</thead>
<tbody>
<tr>
<td>K0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>K1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>K2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>K15</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Key byte differences ranked by average time

<table>
<thead>
<tr>
<th></th>
<th>K0</th>
<th>K1</th>
<th>K2</th>
<th>...</th>
<th>K15</th>
</tr>
</thead>
<tbody>
<tr>
<td>K0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>K1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>K2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>K15</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

0)   f2 1024.32
1)   37 1036.71
2)   7a 1036.84
3)   26 1036.91
   ...  
255) a2 1038.42
### Key byte differences ranked by average time

<table>
<thead>
<tr>
<th></th>
<th>K0</th>
<th>K1</th>
<th>K2</th>
<th>...</th>
<th>K15</th>
</tr>
</thead>
<tbody>
<tr>
<td>K0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>K1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>K2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>K15</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Examples

- **0) f2 1024.32**
- **1) 37 1036.71**
- **2) 7a 1036.84**
- **3) 26 1036.91**
- **...**
- **255) a2 1038.42**

- **0) 5d 1025.61**
- **1) 10 1036.64**
- **2) 46 1036.79**
- **3) dc 1036.98**
- **...**
- **255) 03 1038.16**
Key byte differences ranked by average time

<table>
<thead>
<tr>
<th></th>
<th>K0</th>
<th>K1</th>
<th>K2</th>
<th>...</th>
<th>K15</th>
</tr>
</thead>
<tbody>
<tr>
<td>K0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>K1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>K2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>K15</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

≈ 100,000 encryptions

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0)</td>
<td>f2</td>
<td>1024.32</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1)</td>
<td>37</td>
<td>1036.71</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2)</td>
<td>7a</td>
<td>1036.84</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3)</td>
<td>26</td>
<td>1036.91</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>255)</td>
<td>a2</td>
<td>1038.42</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0)</td>
<td>5d</td>
<td>1025.61</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1)</td>
<td>10</td>
<td>1036.64</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2)</td>
<td>46</td>
<td>1036.79</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3)</td>
<td>dc</td>
<td>1036.98</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>255)</td>
<td>03</td>
<td>1038.16</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Final round is much better to attack

\[ C_i \oplus K_i = S[X] \]
\[ C_j \oplus K_j = S[Y] \]

\[ X = Y \Rightarrow C_i \oplus K_i = C_j \oplus K_j \]
\[ C_i \oplus C_j = K_i \oplus K_j \]
Final round is much better to attack

\[ C_i \oplus K_i = S[X] \]
\[ C_j \oplus K_j = S[Y] \]

\[ C_i \oplus C_j = K_i \oplus K_j \]

\[ \approx 32,000 \text{ encryptions} \]
Hardware countermeasures on the way

/*
AES-128 encryption sequence.
The data block is in xmm15.
Registers xmm0–xmm10 hold the round keys(from 0 to 10 in this order).
In the end, xmm15 holds the encryption result.
*/

pxor    xmm15, xmm0     // Input whitening
aesenc  xmm15, xmm1     // Round 1
aesenc  xmm15, xmm2     // Round 2
aesenc  xmm15, xmm3     // Round 3
aesenc  xmm15, xmm4     // Round 4
aesenc  xmm15, xmm5     // Round 5
aesenc  xmm15, xmm6     // Round 6
aesenc  xmm15, xmm7     // Round 7
aesenc  xmm15, xmm8     // Round 8
aesenc  xmm15, xmm9     // Round 9
aesenclast xmm15, xmm10 // Round 10

Courtesy of Intel
Differential power analysis
(Kocher et. al, 1999)
Simple power analysis ineffective

Trace courtesy of Cryptography Research, Inc.
Hardware implementations don't use cache

Plaintext

Key XOR

Lookup

Mix
Hardware implementations don't use cache

Plaintext

Key XOR

Lookup

Mix

\[ S[\text{P}_0 \oplus \text{K}_0] \]
Partition traces by some predicted intermediate bit

Guessing $K_0 = 00$, traces where high bit of $S[P_0 \oplus K_0]$ is set
Guessing $K_0 = 01$, traces where high bit of $S[P_0 \oplus K_0]$ is set
Partition traces by some predicted intermediate bit

Guessing $K_0 = 02$, traces where high bit of $S[P_0 \oplus K_0]$ is set
Partition traces by some predicted intermediate bit

Guessing $K_0 = 02$, traces where high bit of $S[P_0 \oplus K_0]$ is set

$\approx 10,000$ encryptions
Perfect countermeasures are very difficult
Even further down the rabbit hole...
CPU Noise

Collects sounds which emanates from CPU

Capacitor emits sound from its foils due to fast pulse charging and discharging

Adi Purwono, 2008
Photon emissions

Sergei Skorobogatov, 2009
Lessons
Foot-Shooting Prevention Agreement

I, ________, promise that once I see how simple AES really is, I will not implement it in production code even though it would be really fun.

This agreement shall be in effect until the undersigned creates a meaningful interpretive dance that compares and contrasts cache-based, timing, and other side channel attacks and their countermeasures.

[Signature]

[Date]

Jeff Moser
Do break whatever you can

**Algorithms:**
- Elliptic curve DSS/DH
- Pairing-based algorithms
- AES-GCM authentication
- SHA-3 candidates

**Side-channels:**
- Motherboard sensors
- CPU debug registers

**Killer target:**
- Cross-VM key compromise
Thank you
Complication: cache lines
Complication: cache lines
Complication: cache lines

diagram showing cache lines and table lookup
Complication: Table families

\[
\begin{align*}
t_0 &= \text{Te0}[(s_0 \gg 24)] \uparrow \\
    &= \text{Te1}[(s_1 \gg 16) \& 0xff] \uparrow \\
    &= \text{Te2}[(s_2 \gg 8) \& 0xff] \uparrow \\
    &= \text{Te3}[(s_3) \& 0xff] \uparrow \\
    &= \text{rk}[0];
\end{align*}
\]

\[
\begin{align*}
t_1 &= \text{Te0}[(s_1 \gg 24)] \uparrow \\
    &= \text{Te1}[(s_2 \gg 16) \& 0xff] \uparrow \\
    &= \text{Te2}[(s_3 \gg 8) \& 0xff] \uparrow \\
    &= \text{Te3}[(s_0) \& 0xff] \uparrow \\
    &= \text{rk}[1];
\end{align*}
\]

\[
\begin{align*}
t_2 &= \text{Te0}[(s_2 \gg 24)] \uparrow \\
    &= \text{Te1}[(s_3 \gg 16) \& 0xff] \uparrow \\
    &= \text{Te2}[(s_0 \gg 8) \& 0xff] \uparrow \\
    &= \text{Te3}[(s_1) \& 0xff] \uparrow \\
    &= \text{rk}[2];
\end{align*}
\]

\[
\begin{align*}
t_3 &= \text{Te0}[(s_3 \gg 24)] \uparrow \\
    &= \text{Te1}[(s_0 \gg 16) \& 0xff] \uparrow \\
    &= \text{Te2}[(s_1 \gg 8) \& 0xff] \uparrow \\
    &= \text{Te3}[(s_2) \& 0xff] \uparrow \\
    &= \text{rk}[3];
\end{align*}
\]
Final round is much better to attack
Final round is much better to attack

\[ C_i \oplus K_i = S[X] \]
\[ C_j \oplus K_j = S[Y] \]
Final round is much better to attack

\[ C_i \oplus K_i = S[X] \]
\[ C_j \oplus K_j = S[Y] \]

\[ X = Y \Rightarrow \]
\[ C_i \oplus K_i = C_j \oplus K_j \]
\[ C_i \oplus C_j = K_i \oplus K_j \]