Cache-timing attacks
D. J. Bernstein

Thanks to:
University of Illinois at Chicago
NSF CCR–9983950
Alfred P. Sloan Foundation

http://cr.yp.to/papers.html#cachetiming, 2005:
“This paper reports successful extraction of a complete AES key from a network server on another computer. The targeted server used its key solely to encrypt data using the OpenSSL AES implementation on a Pentium III.”

All code included in paper. Easily reproducible.
Cache-timing attacks

D. J. Bernstein

Thanks to:
University of Illinois at Chicago
NSF CCR–9983950
Alfred P. Sloan Foundation

http://cr.yp.to/papers.html
#cachetiming

2005:
“This paper reports successful extraction of a complete AES key from a network server on another computer. The targeted server used its key solely to encrypt data using the OpenSSL AES implementation on a Pentium III.”

All code included in paper. Easily reproducible.

Outline of this talk:
1. How to advertise an AES candidate
2. How to leak keys through timings: basic techniques
3. How to break AES remotely by forcing cache misses
4. How to skew a benchmark
5. How to leak keys through timings: advanced techniques
6. How to break AES remotely without cache misses
7. How to misdesign a cryptographic architecture
This paper reports successful extraction of a complete AES key from a network server on another computer. The targeted server used its key solely to encrypt data using the OpenSSL AES implementation on a Pentium III.

All code included in paper. Easily reproducible.
Outline of this talk:
1. How to advertise an AES candidate
2. How to leak keys through timings: basic techniques
3. How to break AES remotely by forcing cache misses
4. How to skew a benchmark
5. How to leak keys through timings: advanced techniques
6. How to break AES remotely without cache misses
7. How to misdesign a cryptographic architecture

1. Advertising an AES candidate
1999: NIST announces MARS, RC6, Rijndael, Serpent, Twofish as AES finalists.
Outline of this talk:
1. How to advertise an AES candidate
2. How to leak keys through timings: basic techniques
3. How to break AES remotely by forcing cache misses
4. How to skew a benchmark
5. How to leak keys through timings: advanced techniques
6. How to break AES remotely without cache misses
7. How to misdesign a cryptographic architecture

1. Advertising an AES candidate

1999: NIST announces MARS, RC6, Rijndael, Serpent, Twofish as AES finalists.

1. Advertising an AES candidate


1999: NIST announces MARS, RC6, Rijndael, Serpent, Twofish as AES finalists.


1996: Kocher extracts RSA key from timings of a server. Clear threat to block-cipher keys too. As stated in NIST's report: “In some environments, timing attacks can be effected against operations that execute in different amounts of time, depending on their arguments.”
1. Advertising an AES candidate


1999: NIST announces MARS, RC6, Rijndael, Serpent, Twofish as AES finalists.


1996: Kocher extracts RSA key from *timings* of a server.

Clear threat to block-cipher keys too. As stated in NIST’s report:

“In some environments, timing attacks can be effected against operations that execute in different amounts of time, depending on their arguments.
Advertising an AES candidate

1999: NIST announces MARS, RC6, Rijndael, Serpent, Twofish as AES finalists.


1996: Kocher extracts RSA key from timings of a server.

Clear threat to block-cipher keys too. As stated in NIST’s report:

“In some environments, timing attacks can be effected against operations that execute in different amounts of time, depending on their arguments.

“Table lookup: not vulnerable to timing attacks ...

“Multiplication/division/squaring or variable shift/rotation: most difficult to defend ...

“A general defense against timing attacks is to ensure that each encryption and decryption operation runs in the same amount of time.

“Table lookup: not vulnerable to timing attacks . . .

“Multiplication/division/squaring or variable shift/rotation: most difficult to defend . . .

Table lookup: not vulnerable to timing attacks . . .
1996: Kocher extracts RSA key from *timings* of a server.

Clear threat to block-cipher keys too. As stated in NIST’s report:

“In some environments, timing attacks can be effected against operations that execute in different amounts of time, depending on their arguments.

“A general defense against timing attacks is to ensure that each encryption and decryption operation runs in the same amount of time. . . .

“Table lookup: not vulnerable to timing attacks . . .

“Multiplication/division/squaring or variable shift/rotation: most difficult to defend . . .
Kocher extracts RSA key from a server. Clear threat to block-cipher keys too. As stated in NIST’s report: “In some environments, timing attacks can be effected against operations that execute in different amounts of time, depending on their arguments.

“A general defense against timing attacks is to ensure that each encryption and decryption operation runs in the same amount of time. . . .

“Table lookup: not vulnerable to timing attacks . . .

“Multiplication/division/squaring or variable shift/rotation: most difficult to defend . . .

“Finalist profiles. The operations used by Rijndael are among the easiest to defend against power and timing attacks. . . . Rijndael appears to gain a major speed advantage over its competitors when such protections are considered.

“Rijndael and Serpent use only Boolean operations, table lookups, and fixed shifts/rotations. These operations are the easiest to defend against attacks. . . .
“A general defense against timing attacks is to ensure that each encryption and decryption operation runs in the same amount of time. . .

“Table lookup: not vulnerable to timing attacks . . .

“Multiplication/division/squaring or variable shift/rotation: most difficult to defend . . .

“Rijndael and Serpent use only Boolean operations, table lookups, and fixed shifts/rotations. These operations are the easiest to defend against attacks. . .

“Finalist profiles. . . The operations used by Rijndael are among the easiest to defend against power and timing attacks. . . Rijndael appears to gain a major speed advantage over its competitors when such protections are considered. . .
"A general defense against timing attacks is to ensure that each encryption and decryption operation runs in the same amount of time. . . .

Table lookup: not vulnerable to timing attacks.

Multiplication/division/squaring or variable shift/rotation: most difficult to defend.

"Rijndael and Serpent use only Boolean operations, table lookups, and fixed shifts/rotations. These operations are the easiest to defend against attacks. . . .

"Finalist profiles. . . . The operations used by Rijndael are among the easiest to defend against power and timing attacks. . . . Rijndael appears to gain a major speed advantage over its competitors when such protections are considered. . . .

"NIST judged Rijndael to be the best overall algorithm for the AES. Rijndael appears to be a consistently good performer. Its key setup time is excellent, and its key agility is good. Rijndael’s operations are among the easiest to defend against power and timing attacks. Finally, Rijndael’s internal round structure appears to have good potential to benefit from instruction-level parallelism."

(Emphasis added.)
“Rijndael and Serpent use only Boolean operations, table lookups, and fixed shifts/rotations. These operations are the easiest to defend against attacks. . . .

“Finalist profiles. . . . The operations used by Rijndael are among the easiest to defend against power and timing attacks. . . . Rijndael appears to gain a major speed advantage over its competitors when such protections are considered. . . .

“NIST judged Rijndael to be the best overall algorithm for the AES. Rijndael appears to be a consistently good performer . . . Its key setup time is excellent, and its key agility is good. . . . Rijndael’s operations are among the easiest to defend against power and timing attacks. . . . Finally, Rijndael’s internal round structure appears to have good potential to benefit from instruction-level parallelism.”

(Emphasis added.)
Rijndael and Serpent use only Boolean operations, table lookups, and fixed shifts/rotations. These operations are the easiest to defend against attacks.

---

Rijndael’s operations are among the easiest to defend against power and timing attacks. Rijndael appears to gain a major speed advantage over its competitors when such protections are considered.

NIST judged Rijndael to be the best overall algorithm for the AES. Rijndael appears to be a consistently good performer. Its key setup time is excellent, and its key agility is good. … Rijndael’s operations are among the easiest to defend against power and timing attacks. … Finally, Rijndael’s internal round structure appears to have good potential to benefit from instruction-level parallelism.” (Emphasis added.)

---

1999: AES designers (Daemen, Rijmen) publish “Resistance against implementation attacks: a comparative study of the AES proposals”: “Table lookups: This instruction is not susceptible to a timing attack. … Favorable: Algorithms that use only logical, table-lookups and shifts, and that are therefore relatively easy to secure. The algorithms of this group are Crypton, DEAL, Magenta, Rijndael, and Serpent.”
“NIST judged Rijndael to be the best overall algorithm for the AES. Rijndael appears to be a consistently good performer . . . Its key setup time is excellent, and its key agility is good. . . . Rijndael’s operations are among the easiest to defend against power and timing attacks. . . . Finally, Rijndael’s internal round structure appears to have good potential to benefit from instruction-level parallelism.”

(Emphasis added.)

1999: AES designers (Daemen, Rijmen) publish “Resistance against implementation attacks: a comparative study of the AES proposals”:

“Table lookups: This instruction is not susceptible to a timing attack. . . . Favorable: Algorithms that use only logical operations, table-lookups and fixed shifts, and that are therefore relatively easy to secure. The algorithms of this group are Crypton, DEAL, Magenta, Rijndael and Serpent.”
1999: AES designers (Daemen, Rijmen) publish “Resistance against implementation attacks: a comparative study of the AES proposals”:

“Table lookups: This instruction is not susceptible to a timing attack. . . . Favorable: Algorithms that use only logical operations, table-lookups and fixed shifts, and that are therefore relatively easy to secure. The algorithms of this group are Crypton, DEAL, Magenta, Rijndael and Serpent.”

2005, after AES is shown to be vulnerable, amazing change of position: Timing attacks are “irrelevant for cryptographic design.” Schneier says: “The problem is that side-channel attacks are practical against pretty much anything, so it didn’t really enter into consideration.”

AES judged Rijndael to be the best overall algorithm for the AES. Rijndael appears to be a consistently good performer . . .

Its key setup time is excellent, and its key agility is good. . . .

Rijndael’s operations are among the easiest to defend against timing attacks. . . .

Its internal round structure appears to have good potential to benefit from instruction-level parallelism.”
1999: AES designers (Daemen, Rijmen) publish “Resistance against implementation attacks: a comparative study of the AES proposals”:

“Table lookups: This instruction is not susceptible to a timing attack. . . . Favorable: Algorithms that use only logical operations, table-lookups and fixed shifts, and that are therefore relatively easy to secure. The algorithms of this group are Crypton, DEAL, Magenta, Rijndael and Serpent.”

AES designers write: Speed reports “should take into account the measures to be taken to thwart these attacks.”

2005, after AES is shown to be vulnerable, amazing change of position: Timing attacks are “irrelevant for cryptographic design.” Schneier, 2005: “The problem is that side-channel attacks are practical against pretty much anything, so it didn’t really enter into consideration.”
1999: AES designers (Daemen, Rijmen) publish “Resistance against implementation attacks: a comparative study of the AES proposals”:
- Table lookups: This instruction is not susceptible to a timing attack.

Favorable: Algorithms that use only logical operations, table-lookups and fixed shifts, and that are therefore relatively easy to secure. The algorithms of this group are Crypton, DEAL, Magenta, Rijndael and Serpent.

2005, after AES is shown to be vulnerable, amazing change of position: Timing attacks are irrelevant for cryptographic design. Schneier, 2005: “The problem is that side-channel attacks are practical against pretty much anything, so it didn’t really enter into consideration.”

2. Leaking keys through timings

Most obvious timing variability: skipping an operation is faster than doing it.

1970s: TENEX operating system compares user-supplied strings against secret password one character at a time, stopping at first difference. Attackers monitor comparison time, deduce position of difference. A few hundred tries reveal secret password.
AES designers write: Speed reports “should take into account the measures to be taken to thwart these attacks.”

2005, after AES is shown to be vulnerable, amazing change of position: Timing attacks are “irrelevant for cryptographic design.” Schneier, 2005: “The problem is that side-channel attacks are practical against pretty much anything, so it didn’t really enter into consideration.”

2. Leaking keys through timings
Most obvious timing variability: skipping an operation is faster than doing it.

1970s: TENEX operating system compares user-supplied string against secret password one character a a time, stopping at first difference. Attackers monitor comparison time, deduce position of difference. A few hundred tries reveal secret password.
AES designers write: Speed reports “should take into account the measures to be taken to thwart these attacks.”

2005, after AES is shown to be vulnerable, amazing change of position: Timing attacks are “irrelevant for cryptographic design.” Schneier, 2005: “The problem is that side-channel attacks are practical against pretty much anything, so it didn’t really enter into consideration.”

2. Leaking keys through timings

Most obvious timing variability: skipping an operation is faster than doing it.

1970s: TENEX operating system compares user-supplied string against secret password one character a a time, stopping at first difference. Attackers monitor comparison time, deduce position of difference. A few hundred tries reveal secret password.

Solution: Use constant-time password comparison.

Old:
```
for (i = 0; i < n; ++i)
    if (x[i] != y[i])
        return 0;
return 1;
```

New:
```
diff = 0;
for (i = 0; i < n; ++i)
    diff |= x[i] ^ y[i];
return !diff;
```
2. Leaking keys through timings

Most obvious timing variability: skipping an operation is faster than doing it.

1970s: TENEX operating system compares user-supplied string against secret password one character at a time, stopping at first difference. Attackers monitor comparison time, deduce position of difference. A few hundred tries reveal secret password.

Solution: Use constant-time password comparison.

Old:
```
for (i = 0; i < n; ++i)
    if (x[i] != y[i])
        return 0;
return 1;
```

New:
```
diff = 0;
for (i = 0; i < n; ++i)
    diff |= x[i] ^ y[i];
return !diff;
```
Leaking keys through timings

Most obvious timing variability: skipping an operation is faster than doing it.

1970s: TENEX operating system compares user-supplied string against secret password one character at a time, stopping at the first difference. Attackers monitor comparison time, deduce position of difference. A few hundred tries reveal secret password.

Solution: Use constant-time password comparison.

Old:
```c
for (i = 0; i < n; ++i)
    if (x[i] != y[i])
        return 0;
return 1;
```

New:
```c
diff = 0;
for (i = 0; i < n; ++i)
    diff |= x[i] ^ y[i];
return !diff;
```

1996: Kocher points out timing attacks on cryptographic key bits. Example: key-dependent branch in modular reduction, performing large-integer subtraction for some inputs and not others, leaking key.

My reaction at the time: Yikes! Eliminate variable-time operations from cryptographic code! Beware microSPARC-IIe data-dependent FPU timings; use Fermat instead for inversion in ECC; avoid S-boxes in ciphers.
Solution: Use constant-time password comparison.

Old:
for (i = 0; i < n; ++i)
    if (x[i] != y[i])
        return 0;
return 1;

New:
diff = 0;
for (i = 0; i < n; ++i)
diff |= x[i] ^ y[i];
return !diff;

1996: Kocher points out timing attacks on cryptographic key bits. Example: key-dependent branch in modular reduction, performing large-integer subtraction for some inputs and not others, leaking key.

My reaction at the time: Yikes! Eliminate variable-time operations from cryptographic software!

Beware microSPARC-IIeP data-dependent FPU timings; use Fermat instead of Euclid for inversion in ECC; avoid S-boxes in ciphers; etc.
1996: Kocher points out timing attacks on cryptographic key bits. Example: key-dependent branch in modular reduction, performing large-integer subtraction for some inputs and not others, leaking key.

My reaction at the time: Yikes! Eliminate variable-time operations from cryptographic software!

Beware microSPARC-IIep data-dependent FPU timings; use Fermat instead of Euclid for inversion in ECC; avoid S-boxes in ciphers; etc.

1999: Koeune Quisquater publish fast timing attack on a “careless implementation” of AES that used input-dependent branches. AES has functions \( S \); \( S' \) computed as follows:

```c
byte Sprime(byte b) {
    byte c = S(b);
    if (c<128) return c+c;
    return (c+c)^283;
}
```

Timing leaks bit of \( c \): faster if \( c < 128 \).
1996: Kocher points out timing attacks on cryptographic key bits. Example: key-dependent branch in modular reduction, performing large-integer subtraction for some inputs and not others, leaking key.

My reaction at the time: Yikes! Eliminate variable-time operations from cryptographic software! Beware microSPARC-IIeP data-dependent FPU timings; use Fermat instead of Euclid for inversion in ECC; avoid S-boxes in ciphers; etc.

1999: Koeune Quisquater publish fast timing attack on a “careless implementation” of AES that used input-dependent branches.

AES has functions $S, S'$ mapping bytes to bytes. Attack is against $S'$ computed as follows:

```c
byte Sprime(byte b) {
    byte c = S(b);
    if (c<128) return c+c;
    return (c+c)^283;
}
```

Timing leaks bit of $c$: faster if $c < 128$. 
1999: Koeune Quisquater publish fast timing attack on a “careless implementation” of AES that used input-dependent branches.

AES has functions $S, S'$ mapping bytes to bytes. Attack is against $S'$ computed as follows:

```c
byte Sprime(byte b) {
    byte c = S(b);
    if (c<128) return c+c;
    return (c+c)^283;
}
```

Timing leaks bit of $c$: faster if $c < 128$.

Standard solution:
replace branch by arithmetic.

```c
X = c>>7;
X |= (X<<1);
X |= (X<<3);
return (c<<1)^X;
```

CPUs handle this arithmetic in constant time.

Koeune Quisquater: “The result presented here is not an attack against Rijndael, but against bad implementations.”
1999: Koeune Quisquater publish fast timing attack on a “careless implementation” of AES that used input-dependent branches.

AES has functions $S, S'$ mapping bytes to bytes. Attack is against $S'$ computed as follows:

```
byte Sprime(byte b) {
    byte c = S(b);
    if (c<128) return c+c;
    return (c+c)^283;
}
```

Timing leaks bit of $c$: faster if $c < 128$.

Standard solution:
replace branch by arithmetic.
```
X = c>>7;
X |= (X<<1);
X |= (X<<3);
return (c<<1)^X;
```
CPUs handle this arithmetic in constant time.

Koeune Quisquater:
“The result presented here is not an attack against Rijndael, but against bad implementations of it.”
Koeune Quisquater publish fast timing attack on a “careless implementation” of AES that used input-dependent branches. Functions $S, S'$ mapping bytes to bytes. Attack is against $S$ computed as follows:

```c
byte Sprime(byte b) {
    byte c = S(b);
    if (c<128) return c+c;
    return (c+c)^283;
}
```

Timing leaks bit of $c$: faster if $c<128$.

Standard solution: replace branch by arithmetic.

```c
X = c>>7;
X |= (X<<1);
X |= (X<<3);
return (c<<1)^X;
```

CPUs handle this arithmetic in constant time.

Koeune Quisquater: “The result presented here is not an attack against Rijndael, but against bad implementations of it.”

Second most obvious timing variability: L2 cache is faster than DRAM. Similarly, L1 cache is faster than L2 cache.

Reading from cached line takes less time than reading from uncached line.

Variability mentioned by 1996 Kocher, 2000 Kelsey Schneier Wagner Hall ("We believe attacks based on cache hit ratio in large S-box ciphers like Blowfish, CAST and Khufu are possible"), 2003 Ferguson Schneier.
Standard solution: replace branch by arithmetic.

\[
X = c >> 7; \\
X |= (X << 1); \\
X |= (X << 3); \\
\text{return } (c << 1) ^ X;
\]

CPUs handle this arithmetic in constant time.

Koeune Quisquater:
“The result presented here is not an attack against Rijndael, but against bad implementations of it.”

Second most obvious timing variability: L2 cache is faster than DRAM. Similarly, L1 cache is faster than L2 cache.

Reading from cached line takes less time than reading from uncached line.

Variability mentioned by 1996 Kocher, 2000 Kelsey Schneier Wagner Hall (“We believe attacks based on cache hit ratio in large S-box ciphers like Blowfish, CAST and Khufu are possible”), 2003 Ferguson Schneier.
Standard solution:
replace branch by arithmetic.

X = c>>7;
X |= (X<<1);
X |= (X<<3);
return (c<<1)^X;

CPUs handle this arithmetic in constant time.

Koeune Quisquater:
"The result presented here is not an attack against Rijndael, but against bad implementations of it."

Second most obvious timing variability: L2 cache is faster than DRAM. Similarly, L1 cache is faster than L2 cache.

Reading from cached line takes less time than reading from uncached line.

Variability mentioned by 1996 Kocher, 2000 Kelsey Schneier Wagner Hall ("We believe attacks based on cache hit ratio in large S-box ciphers like Blowfish, CAST and Khufu are possible"), 2003 Ferguson Schneier.

2002: Page publishes fast algorithm to find DES key from high-bandwidth timing information. DPA-style. Many plaintexts, each starting with empty cache. Algorithm input: for each plaintext, list of S-box lookups that missed the cache.

Avoid empty cache by preloading some S-box entries? "To guarantee this as an effective countermeasure we need to warm the cache with the entirety of all the S-boxes."
Second most obvious timing variability: L2 cache is faster than DRAM. Similarly, L1 cache is faster than L2 cache.

Reading from cached line takes less time than reading from uncached line.

Variability mentioned by 1996 Kocher, 2000 Kelsey Schneier Wagner Hall ("We believe attacks based on cache hit ratio in large S-box ciphers like Blowfish, CAST and Khufu are possible"), 2003 Ferguson Schneier.

2002: Page publishes fast algorithm to find DES key from high-bandwidth timing information. DPA-style. Many plaintexts, each starting with empty cache. Algorithm input: for each plaintext, list of S-box lookups that missed the cache.

Avoid empty cache by preloading some S-box entries? “To guarantee this as an effective countermeasure we need to warm the cache with the entirety of all the S-boxes.”
Second most obvious timing variability: L2 cache is faster than DRAM. Similarly, L1 cache is faster than L2 cache. Reading from cached line takes less time than reading from uncached line.

Motivated by 1996 Kocher, 2000 Kelsey Schneier Wagner Hall (“We believe attacks based on cache hit ratio in large S-box ciphers like Blowfish, CAST and Khufu are possible”), 2003 Ferguson Schneier.

2002: Page publishes fast algorithm to find DES key from high-bandwidth timing information. DPA-style. Many plaintexts, each starting with empty cache. Algorithm input: for each plaintext, list of S-box lookups that missed the cache.

Avoid empty cache by preloading some S-box entries? “To guarantee this as an effective countermeasure we need to warm the cache with the entirety of all the S-boxes.”

2003: Tsunoo, Saito, Suzaki, Shigeri, Miyauchi publish fast algorithm to find DES key from low-bandwidth timing information. Many plaintexts, each starting with empty cache. Algorithm input: for each plaintext, encryption time.

“If a total-data load is executed before processing, differences between the frequencies of cache misses will not be observed, making it impossible to determine the relationships between sets of S-boxes.”
2002: Page publishes fast algorithm to find DES key from high-bandwidth timing information. DPA-style. Many plaintexts, each starting with empty cache. Algorithm input: for each plaintext, list of S-box lookups that missed the cache.

Avoid empty cache by preloading some S-box entries? “To guarantee this as an effective countermeasure we need to warm the cache with the entirety of all the S-boxes.”

2003: Tsunoo, Saito, Suzaki, Shigeri, Miyauchi publish fast algorithm to find DES key from low-bandwidth timing information. Many plaintexts, each starting with empty cache. Algorithm input: for each plaintext, encryption time.

“If a total-data load is executed before processing, differences between the frequencies of cache misses will not be observed, making it impossible to determine the relationships between sets of S-boxes.”
2002: Page publishes fast algorithm to find DES key from high-bandwidth timing information. DPA-style. Many plaintexts, each starting with empty cache. Algorithm input: for each plaintext, list of S-box lookups that missed the cache.

"If a total-data load is executed before processing, differences between the frequencies of cache misses will not be observed, making it impossible to determine the relationships between sets of S-boxes."

2003: Tsunoo, Saito, Suzaki, Shigeri, Miyauchi publish fast algorithm to find DES key from low-bandwidth timing information. Many plaintexts, each starting with empty cache. Algorithm input: for each plaintext, encryption time.

"To guarantee this as an effective countermeasure we need to warm the cache with the entirety of all the S-boxes."

3. Breaking AES

Given 16-byte sequence \( n \) and 16-byte sequence \( k \), AES produces 16-byte sequence \( \text{AES}_k(n) \).

Uses table lookup and \( \overline{\text{xor}} \):

\[
\begin{align*}
e_0 &= \text{tab}[k[13]] \overline{1} \\
e_1 &= \text{tab}[k[0] \oplus n[0]] \overline{k[0]} \overline{e_0} \\
&\text{etc.}
\end{align*}
\]

\( \text{AES}_k(n) = (e_{784}, \ldots, e_{799}) \).
2003: Tsunoo, Saito, Suzaki, Shigeri, Miyauchi publish fast algorithm to find DES key from low-bandwidth timing information. Many plaintexts, each starting with empty cache. Algorithm input: for each plaintext, encryption time.

“If a total-data load is executed before processing, differences between the frequencies of cache misses will not be observed, making it impossible to determine the relationships between sets of S-boxes.”

3. Breaking AES

Given 16-byte sequence \( n \) and 16-byte sequence \( k \), AES produces 16-byte sequence \( AES_k(n) \).

Uses table lookup and \( \oplus \) (xor):
\[
e_0 = \text{tab}[k[13]] \oplus 1
\]
\[
e_1 = \text{tab}[k[0] \oplus n[0]] \oplus k[0] \oplus e_0
\]

etc.

\( AES_k(n) = (e_{784}, \ldots, e_{799}) \).
3. Breaking AES

Given 16-byte sequence \( n \) and 16-byte sequence \( k \), AES produces 16-byte sequence \( \text{AES}_k(n) \).

Uses table lookup and \( \oplus \) (xor):
\[
e_0 = \text{tab}[k[13]] \oplus 1
\]
\[
e_1 = \text{tab}[k[0] \oplus n[0]] \oplus k[0] \oplus e_0
\]

etc.
\[
\text{AES}_k(n) = (e_{784}, \ldots, e_{799}).
\]

High-speed AES uses 4-byte registers, several 1024-byte tables. Operations: byte extraction (4 bytes to 1 byte), table lookup (1 byte to 4 byte), etc.

Attacker can force selected table entries out of L2 cache, observe encryption time. Each cache miss creates timing signal, clearly visible despite noise from other AES cache misses, other software, etc.

Repeat for many plaintexts, easily deduce key.
3. Breaking AES

Given 16-byte sequence \( n \) and 16-byte sequence \( k \), AES produces 16-byte sequence \( \text{AES}_k(n) \).

Uses table lookup and \( \oplus \) (xor):
\[
e_0 = \text{tab}[k[13]] \oplus 1
\]
\[
e_1 = \text{tab}[k[0] \oplus n[0]] \oplus k[0] \oplus e_0
\]

etc.

\( \text{AES}_k(n) = (e_{784}, \ldots, e_{799}) \).

High-speed AES uses 4-byte registers, several 1024-byte tables. Operations: byte extraction (4 bytes to 1 byte), table lookup (1 byte to 4 byte), \( \oplus \).

Attacker can force selected table entries out of L2 cache, observe encryption time. Each cache miss creates timing signal, clearly visible despite noise from other AES cache misses, other software, etc.

Repeat for many plaintexts, easily deduce key.
3. Breaking AES

Given 16-byte sequence \( n \) and 16-byte sequence \( k \), AES produces 16-byte sequence \( \text{AES}_k(n) \).

Uses table lookup and \( \oplus \) (xor):
\[
e_0 = \text{tab}[k[13]] \oplus 1
\]
\[
e_1 = \text{tab}[k[0] \oplus n[0]] \oplus k[0] \oplus e_0
\]
\[
\text{etc.}
\]

\( \text{AES}_k(n) = (e_{784}; \ldots; e_{799}). \)

High-speed AES uses 4-byte registers, several 1024-byte tables. Operations: byte extraction (4 bytes to 1 byte), table lookup (1 byte to 4 byte), \( \oplus \).

Attacker can force selected table entries out of L2 cache, observe encryption time.

Each cache miss creates timing signal, clearly visible despite noise from other AES cache misses, other software, etc. Repeat for many plaintexts, easily deduce key.

Example: \( \text{tab}[k[0] \oplus n[0]] \) costs hundreds of extra cycles if this tab entry is not in L2 cache.

Knock \( \text{tab}[13] \) out of cache. See signal when \( k[0] \oplus n[0] = 13 \).

Deduce \( k[0] \) as \( n[0] \oplus 13 \).

(Complication: cache lines; need more work to find bottom bits of \( k[0] \).)

More efficient: Knock half of the tab entries out of cache.

Then first \( n[0] \) limits \( k[0] \) to half of its possibilities.

Example: \( \text{tab}[k[0] \oplus n[0]] \) costs hundreds of extra cycles if this tab entry is not in L2 cache.

Knock \( \text{tab}[13] \) out of cache. See signal when \( k[0] \oplus n[0] = 13 \).

Deduce \( k[0] \) as \( n[0] \oplus 13 \).

(Complication: cache lines; need more work to find bottom bits of \( k[0] \).)

More efficient: Knock half of the tab entries out of cache.

Then first \( n[0] \) limits \( k[0] \) to half of its possibilities.
High-speed AES uses 4-byte registers, several 1024-byte tables. Operations: byte extraction (4 bytes to 1 byte), table lookup (1 byte to 4 byte), ⊕.

Attacker can force selected table entries out of L2 cache, observe encryption time. Each cache miss creates timing signal, clearly visible despite noise from other AES cache misses, other software, etc. Repeat for many plaintexts, easily deduce key.

Example: \( \text{tab}[k[0] \oplus n[0]] \) costs hundreds of extra cycles if this tab entry is not in L2 cache.

Knock \( \text{tab}[13] \) out of cache. See signal when \( k[0] \oplus n[0] = 13 \).

Deduce \( k[0] \) as \( n[0] \oplus 13 \).

(Complication: cache lines; need more work to find bottom bits of \( k[0] \).)

More efficient: Knock half of the tab entries out of cache. Then first \( n[0] \) limits \( k[0] \) to half of its possibilities.
High-speed AES uses 4-byte registers, several 1024-byte tables.

Operations: byte extraction (4 bytes to 1 byte), table lookup (1 byte).

Attacker can force selected table entries out of L2 cache, observe encryption time.

Each cache miss creates timing signal, clearly visible despite noise from other AES cache misses, other software, etc.

Repeat for many plaintexts, easily deduce key.

Example: \( \text{tab}[k[0] \oplus n[0]] \) costs hundreds of extra cycles if this tab entry is not in L2 cache.

Knock \( \text{tab}[13] \) out of cache. See signal when \( k[0] \oplus n[0] = 13 \).

Deduce \( k[0] \) as \( n[0] \oplus 13 \).

(Complication: cache lines; need more work to find bottom bits of \( k[0] \).)

More efficient: Knock half of the tab entries out of cache.

Then first \( n[0] \) limits \( k[0] \) to half of its possibilities.

On (e.g.) Athlon: 65536-byte L1 cache is 2-way associative. If three 64-byte lines with the same address modulo 32768 are read, the first line is forced out of the L1 cache.

Athlon's 524288-byte L2 cache is 16-way associative. If 17 lines with the same address modulo 8192 are read, the first line is forced out of the L2 cache.

Force \( \text{tab}[13] \) out of cache by accessing selected memory locations.
Example: \( \text{tab}[k[0] \oplus n[0]] \) costs hundreds of extra cycles if this tab entry is not in L2 cache.

Knock \( \text{tab}[13] \) out of cache. See signal when \( k[0] \oplus n[0] = 13 \).
Deduce \( k[0] \) as \( n[0] \oplus 13 \).
(Complication: cache lines; need more work to find bottom bits of \( k[0] \).)

More efficient: Knock half of the tab entries out of cache.
Then first \( n[0] \) limits \( k[0] \) to half of its possibilities.

On (e.g.) Athlon: 65536-byte L1 cache is 2-way associative. If three 64-byte lines with the same address modulo 32768 are read, the first line is forced out of the L1 cache.

Athlon’s 524288-byte L2 cache is 16-way associative. If 17 lines with the same address modulo 8192 are read, the first line is forced out of the L2 cache.

Force \( \text{tab}[13] \) out of cache by accessing selected memory locations.
On (e.g.) Athlon: 65536-byte L1 cache is 2-way associative. If three 64-byte lines with the same address modulo 32768 are read, the first line is forced out of the L1 cache.

Athlon’s 524288-byte L2 cache is 16-way associative. If 17 lines with the same address modulo 8192 are read, the first line is forced out of the L2 cache.


How does attacker do the necessary accesses? Trivial on multiuser computer if attacker has an account. Almost as easy without an account: e.g., attacker sends Java applet to user’s browser.

What if computer has no browser, no buffer overflows, etc.? Clearly still possible to carry out the attack from another computer by figuring out packets that, when sent to (e.g.) Linux kernel, cause accesses of appropriate memory locations. Nobody has done this! Would make a nice paper!
On (e.g.) Athlon: 65536-byte L1 cache is 2-way associative. If three 64-byte lines with the same address modulo 32768 are read, the first line is forced out of the L1 cache.

Athlon’s 524288-byte L2 cache is 16-way associative. If 17 lines with the same address modulo 8192 are read, the first line is forced out of the L2 cache.


How does attacker do the necessary accesses? Trivial on multiuser computer if attacker has account. Almost as easy without an account: e.g., attacker sends Java applet to user’s browser.

What if computer has no browser, no buffer overflows, etc.? Clearly still possible to carry out the attack from another computer by figuring out packets that, when sent to (e.g.) Linux kernel, cause accesses of appropriate memory locations. Nobody has done this! Would make a nice paper!
How does attacker do the necessary accesses? Trivial on multiuser computer if attacker has account. Almost as easy without an account: e.g., attacker sends Java applet to user's browser.

What if computer has no browser, no buffer overflows, etc.? Clearly still possible to carry out the attack from another computer by figuring out packets that, when sent to (e.g.) Linux kernel, cause accesses of appropriate memory locations. Nobody has done this! Would make a nice paper!

What about the "guaranteed" countermeasure, reading all AES tables before starting AES computation?

Even if this were free, it wouldn't eliminate cache misses. Table entries can drop out of cache during the computation.

Typical AES software uses several different arrays: input, key, output, stack, S-boxes. Software sometimes kicks its own S-box lines out of L1 cache by accessing (e.g.) the key and the stack.
How does attacker do the necessary accesses? Trivial on multiuser computer if attacker has account. Almost as easy without an account: e.g., attacker sends Java applet to user’s browser.

What if computer has no browser, no buffer overflows, etc.? Clearly still possible to carry out the attack from another computer by figuring out packets that, when sent to (e.g.) Linux kernel, cause accesses of appropriate memory locations. Nobody has done this! Would make a nice paper!

What about the “guaranteed” countermeasure, reading all AES tables before starting AES computation?

Even if this were free, it wouldn’t eliminate cache misses. Table entries can drop out of cache during the computation.

Typical AES software uses several different arrays: input, key, output, stack, S-boxes. Software sometimes kicks its own S-box lines out of L1 cache by accessing (e.g.) the key and the stack.
How does attacker do the necessary accesses? Trivial on multiuser computer if attacker has account. Almost as easy without an account: e.g., attacker sends Java applet to user’s browser. If computer has no browser, Java applets, etc.? Clearly carry out the attacker computer packets that, when sent to (e.g.) Linux kernel, cause accesses of appropriate memory locations. Nobody has done this! Would make a nice paper!

What about the “guaranteed” countermeasure, reading all AES tables before starting AES computation? Even if this were free, it wouldn’t eliminate cache misses. Table entries can drop out of cache during the computation.

Typical AES software uses several different arrays: input, key, output, stack, S-boxes. Software sometimes kicks its own S-box lines out of L1 cache by accessing (e.g.) the key and the stack.

Fixed in my 2005 AES implementation, Gladman’s latest implementation, etc.: squeeze variables into a limited number of arrays. But this still doesn’t eliminate cache misses!

Computers run many simultaneous processes. The AES software can be interrupted by another process that kicks lines out of L1 cache and maybe even L2 cache. Even worse, the partial-AES cache state affects the timing of the other process.
What about the "guaranteed" countermeasure, reading all AES tables before starting AES computation?

Even if this were free, it wouldn’t eliminate cache misses. Table entries can drop out of cache during the computation.

Typical AES software uses several different arrays: input, key, output, stack, S-boxes. Software sometimes kicks its own S-box lines out of L1 cache by accessing (e.g.) the key and the stack.

Fixed in my 2005 AES implementation, Gladman’s latest implementation, etc.: squeeze variables into a limited number of arrays. But this *still* doesn’t eliminate cache misses!

Computers run many simultaneous processes. The AES software can be interrupted by another process that kicks lines out of L1 cache and maybe even L2 cache. Even worse, the partial-AES cache state affects the timing of the other process.
What about the "guaranteed" countermeasure, reading all AES tables before starting AES computation? Even if this were free, it wouldn’t eliminate cache misses. Table entries can drop out of cache during the computation.

Typical AES software uses several different arrays: input, key, output, stack, S-boxes. Software sometimes kicks its own S-box lines out of L1 cache by accessing (e.g.) the key and the stack.

Fixed in my 2005 AES implementation, Gladman’s latest implementation, etc.: squeeze variables into a limited number of arrays. But this still doesn’t eliminate cache misses!

Computers run many simultaneous processes. The AES software can be interrupted by another process that kicks lines out of L1 cache and maybe even L2 cache. Even worse, the partial-AES cache state affects the timing of the other process.

Occasional AES interrupts by accident.

Can force much more frequent interrupts with “hyperthreading”—2005 Osvik Shamir Tromer, independently 2005 Percival—giving high-bandwidth timing information.

Not clear whether hyperthreading approach can be carried out remotely via (e.g.) Linux kernel.
Fixed in my 2005 AES implementation, Gladman’s latest implementation, etc.: squeeze variables into a limited number of arrays. But this *still* doesn’t eliminate cache misses!

Computers run many simultaneous processes. The AES software can be interrupted by another process that kicks lines out of L1 cache and maybe even L2 cache. Even worse, the partial-AES cache state affects the timing of the other process.

Occasional AES interrupts by accident.

Can force much more frequent interrupts with “hyperthreading”—2005 Osvik Shamir Tromer, independently 2005 Percival—giving high-bandwidth timing information.

Not clear whether hyperthreading approach can be carried out remotely via (e.g.) Linux kernel.
Occasional AES interrupts by accident.

Can force much more frequent interrupts with “hyperthreading”—2005 Osvik Shamir Tromer, independently 2005 Percival—giving high-bandwidth timing information.

Not clear whether hyperthreading approach can be carried out remotely via (e.g.) Linux kernel.

The bad news, as we'll see later: Stopping cache misses isn't enough. There are timing leaks in cache hits.

It is possible to stop all AES cache misses.

Put AES software into operating-system kernel.

Disable interrupts.

Disable hyperthreading etc.

Read all S-boxes into cache.

Wait for reads to complete.

Encrypt some blocks of data.

The bad news, as we'll see later: Stopping cache misses isn't enough. There are timing leaks in cache hits.

Occasional AES interrupts by accident.

Can force much more frequent interrupts with “hyperthreading”—2005 Osvik Shamir Tromer, independently 2005 Percival—giving high-bandwidth timing information.

Not clear whether hyperthreading approach can be carried out remotely via (e.g.) Linux kernel.

The bad news, as we'll see later: Stopping cache misses isn't enough. There are timing leaks in cache hits.

It is possible to stop all AES cache misses.

Put AES software into operating-system kernel.

Disable interrupts.

Disable hyperthreading etc.

Read all S-boxes into cache.

Wait for reads to complete.

Encrypt some blocks of data.

The bad news, as we'll see later: Stopping cache misses isn't enough. There are timing leaks in cache hits.
Occasional AES interrupts by accident.

Can force much more frequent interrupts with “hyperthreading”—2005 Osvik Shamir Tromer, independently 2005 Percival—giving high-bandwidth timing information.

Not clear whether hyperthreading approach can be carried out remotely via (e.g.) Linux kernel.

It is possible to stop all AES cache misses.

Put AES software into operating-system kernel. Disable interrupts. Disable hyperthreading etc. Read all S-boxes into cache. Wait for reads to complete. Encrypt some blocks of data.

The bad news, as we’ll see later: Stopping cache misses isn’t enough. There are timing leaks in cache hits.
Occasional AES interrupts by accident. Can force much more frequent interrupts with "hyperthreading"—2005 Osvik Shamir Tromer, independently 2005 Percival—giving high-bandwidth timing information. Not clear whether hyperthreading approach can be carried out remotely via (e.g.) Linux kernel.

It is possible to stop all AES cache misses. Put AES software into operating-system kernel. Disable interrupts. Disable hyperthreading etc. Read all S-boxes into cache. Wait for reads to complete. Encrypt some blocks of data.

The bad news, as we'll see later: Stopping cache misses isn't enough. There are timing leaks in cache *hits*.

4. Skewing benchmarks

Many deceptive timings in the cryptographic literature:

- Bait-and-switch timings.
- Guesses reported as timings.
- My-favorite-CPU timings.
- Long-message timings.
- Timings after precomputation.
- High-variance timings.

Consequence: In the real world, these functions are often much slower than advertised.
It is possible to stop all AES cache misses.

Put AES software into operating-system kernel.
Disable interrupts.
Disable hyperthreading etc.
Read all S-boxes into cache.
Wait for reads to complete.
Encrypt some blocks of data.

The bad news, as we’ll see later:
Stopping cache misses isn’t enough. There are timing leaks in cache hits.

4. Skewing benchmarks

Many deceptive timings in the cryptographic literature:
- Bait-and-switch timings.
- Guesses reported as timings.
- My-favorite-CPU timings.
- Long-message timings.
- Timings after precomputation.
- High-variance timings.

Consequence: In the real world, these functions are often much slower than advertised.
It is possible to stop all AES cache misses. Put AES software into operating-system kernel. Disable interrupts. Disable hyperthreading etc. Read all S-boxes into cache. Wait for reads to complete. Encrypt some blocks of data. The bad news, as we’ll see later: Stopping cache misses isn’t enough. There are timing leaks in cache hits.

4. Skewing benchmarks
Many deceptive timings in the cryptographic literature:
- Bait-and-switch timings.
- Guesses reported as timings.
- My-favorite-CPU timings.
- Long-message timings.
- Timings after precomputation.
- High-variance timings.

Consequence: In the real world, these functions are often much slower than advertised.

Bait-and-switch timings:
Create two versions of your function, a small Fun-Breakable and a big Fun-Slow. Report timings for Fun-Breakable. Example in literature: Paper proposes 16-byte authenticator. Says “More than 1 Gbit/sec on a 200 MHz Pentium Pro” ... but that’s actually for a breakable 4-byte authenticator.

The honest alternative:
Focus on one function.
4. Skewing benchmarks

Many deceptive timings in the cryptographic literature:

- Bait-and-switch timings.
- Guesses reported as timings.
- My-favorite-CPU timings.
- Long-message timings.
- Timings after precomputation.
- High-variance timings.

Consequence: In the real world, these functions are often much slower than advertised.

Bait-and-switch timings:
Create two versions of your function, a small Fun-Breakable and a big Fun-Slow. Report timings for Fun-Breakable.

Example in literature: Paper proposes 16-byte authenticator. Says “More than 1 Gbit/sec on a 200 MHz Pentium Pro” . . . but that’s actually for a breakable 4-byte authenticator.

The honest alternative: Focus on one function.
4. Skewing benchmarks

Many deceptive timings in the cryptographic literature:

› Bait-and-switch timings.
› Guesses reported as timings.
› My-favorite-CPU timings.
› Long-message timings.
› Timings after precomputation.
› High-variance timings.

Consequence: In the real world, these functions are often much slower than advertised.

Bait-and-switch timings:
Create two versions of your function, a small Fun-Breakable and a big Fun-Slow. Report timings for Fun-Breakable.

Example in literature: Paper proposes 16-byte authenticator. Says “More than 1 Gbit/sec on a 200 MHz Pentium Pro” … but that’s actually for a breakable 4-byte authenticator.

The honest alternative:
Focus on one function.

Guesses reported as timings:
Measure only part of the computation.
Estimate the other parts.

Example in literature: … 2.2 clock cycles per byte if the unimplemented parts are as fast as various estimates.

The honest alternative: Measure exactly the function call that applications will use.
Bait-and-switch timings:
Create two versions of your function, a small Fun-Breakable and a big Fun-Slow. Report timings for Fun-Breakable.

Example in literature: Paper proposes 16-byte authenticator. Says “More than 1 Gbit/sec on a 200 MHz Pentium Pro” … but that’s actually for a breakable 4-byte authenticator.

The honest alternative: Focus on one function.

Guesses reported as timings:
Measure only part of the computation.
Estimate the other parts.

Example in literature: “achieves 2.2 clock cycles per byte” … if the unimplemented parts are as fast as various estimates.

The honest alternative: Measure exactly the function call that applications will use.
Bait-and-switch timings: Create two versions of your function, a small Fun-Breakable and a big Fun-Slow. Report timings for Fun-Breakable.

Example in literature: Paper proposes 16-byte authenticator. Says “More than 1 Gbit/sec on a 200 MHz Pentium Pro” but that’s actually for a breakable 4-byte authenticator. The honest alternative: Focus on one function.

Guesses reported as timings: Measure only part of the computation. Estimate the other parts. Example in literature: “achieves 2.2 clock cycles per byte” ... if the unimplemented parts are as fast as various estimates. The honest alternative: Measure exactly the function call that applications will use.

My-favorite-CPU timings: Choose CPU where function is very fast. Ignore all other CPUs. Example in literature: “All speeds were measured on a Pentium 4” ... because other chips take many more cycles for this particular computation. The honest alternative: Measure every CPU you can find. If reader doesn’t care about a particular chip, he can ignore it.
Guesses reported as timings:
Measure only part of the computation.
Estimate the other parts.

Example in literature: “achieves 2.2 clock cycles per byte” ... if the unimplemented parts are as fast as various estimates.

The honest alternative: Measure exactly the function call that applications will use.

My-favorite-CPU timings: Choose CPU where function is very fast.
Ignore all other CPUs.

Example in literature: “All speeds were measured on a Pentium 4” ... because other chips take many more cycles per byte for this particular computation.

The honest alternative: Measure every CPU you can find. If reader doesn’t care about a particular chip, he can ignore it.
Guesses reported as timings:
Measure only part of the computation.
Estimate the other parts.

Example in literature: “achieves 2^2 clock cycles per byte” if the unimplemented parts are as fast as various estimates.

The honest alternative: Measure exactly the function call that applications will use.

My-favorite-CPU timings: Choose CPU where function is very fast.
Ignore all other CPUs.

Example in literature: “All speeds were measured on a Pentium 4” because other chips take many more cycles per byte this particular computation.

The honest alternative: Measure every CPU you can find. If reader doesn’t care about a particular chip, he can ignore it.

Long-message timings: Report time only for long messages.
Ignore per-message overhead.
Ignore applications that handle short packets.

Example in literature: “2 cycles per byte” plus 2000 cycles per packet.

The honest alternative: Report times for n-byte packets for each n ∈ {0, 2^0, 2^1, 2^2, ... , 8192}.
My-favorite-CPU timings: Choose CPU where function is very fast. Ignore all other CPUs.

Example in literature: “All speeds were measured on a Pentium 4” … because other chips take many more cycles per byte for this particular computation.

The honest alternative: Measure every CPU you can find. If reader doesn’t care about a particular chip, he can ignore it.

Long-message timings: Report time only for long messages. Ignore per-message overhead. Ignore applications that handle short packets.

Example in literature: “2 cycles per byte” … plus 2000 cycles per packet.

The honest alternative: Report times for $n$-byte packets for each $n \in \{0, 1, 2, \ldots, 8192\}$.
My-favorite-CPU timings: Choose CPU where function is very fast. Ignore all other CPUs.

Example in literature: “All speeds were measured on a Pentium 4” because other chips take many more cycles per byte for this particular computation.

The honest alternative: Measure every CPU you can find. If reader doesn't care about a particular chip, he can ignore it.

Long-message timings: Report time only for long messages. Ignore per-message overhead. Ignore applications that handle short packets.

Example in literature: “2 cycles per byte” plus 2000 cycles per packet.

The honest alternative: Report times for $n$-byte packets for each $n \in \{0, 1, 2, \ldots, 8192\}$.

Timings after precomputation: Report time after a big key-dependent table has been precomputed and loaded into L1 cache. Ignore applications that handle many simultaneous keys.

The honest alternative: Measure precomputation time; measure time to load inputs that weren't already in cache.
Long-message timings: Report time only for long messages. Ignore per-message overhead. Ignore applications that handle short packets.

Example in literature: “2 cycles per byte” … plus 2000 cycles per packet.

The honest alternative: Report times for \( n \)-byte packets for each \( n \in \{0, 1, 2, \ldots, 8192\} \).

Timings after precomputation: Report time after a big key-dependent table has been precomputed and loaded into L1 cache. Ignore applications that handle many simultaneous keys.

The honest alternative: Measure precomputation time; measure time to load inputs that weren’t already in cache.
Timings after precomputation: Report time after a big key-dependent table has been precomputed and loaded into L1 cache. Ignore applications that handle many simultaneous keys.

The honest alternative: Measure precomputation time; measure time to load inputs that weren't already in cache.

High-variance timings: Measure each function a single time, on a single input. Ignore possibility of high variance in timing.

Compare functions by comparing single timings, promoting a few high-variance functions.

The honest alternative: Report several measurements, making variance clear.

---

Timings after precomputation: Report time only for long messages. Ignore per-message overhead. Ignore applications that handle short packets.

The honest alternative: Report times for $n$-byte packets for each $n \in \{0, 1, 2, \ldots, 8192\}$.
Timings after precomputation: Report time after a big key-dependent table has been precomputed and loaded into L1 cache. Ignore applications that handle many simultaneous keys.

The honest alternative: Measure precomputation time; measure time to load inputs that weren’t already in cache.

High-variance timings: Measure each function a single time, on a single input. Ignore possibility of high variance in timing.

Compare functions by comparing single timings, promoting a few high-variance functions.

The honest alternative: Report several measurements, making variance clear.
High-variance timings:
Measure each function a single
time, on a single input.
Ignore possibility of high variance
in timing.

Compare functions by comparing
single timings, promoting a few
high-variance functions.

The honest alternative:
Report several measurements,
making variance clear.

5. Advanced timing leaks

2004: I write software for
Poly1305-AES, a state-of-the-art
message authenticator. Standard
Wegman-Carter structure,
combining a provably secure
"universal" hash (Poly1305) with
a hopefully-secure stream cipher
(AES in counter mode).

Poly1305 has no precomputation.
Existing AES software slow precomputation,
making Poly1305-AES look slow. So I
write new AES software.
High-variance timings:
Measure each function a single
time, on a single input.
Ignore possibility of high variance
in timing.

Compare functions by comparing
single timings, promoting a few
high-variance functions.

The honest alternative:
Report several measurements,
making variance clear.

5. Advanced timing leaks
2004: I write software for
Poly1305-AES, a state-of-the-art
message authenticator. Standard
Wegman-Carter structure,
combining a provably secure
“universal” hash (Poly1305) with
a hopefully-secure stream cipher
(AES in counter mode).

Poly1305 has no precomputation.
Existing AES software does
slow precomputation, making
Poly1305-AES look slow. So I
write new AES software.
5. Advanced timing leaks


Poly1305 has no precomputation. Existing AES software does slow precomputation, making Poly1305-AES look slow. So I write new AES software.

I look at successive cycle counts for authenticating ten 1-byte messages: 3668 833 585 574 603 567 577 568 570...

2-byte messages: 568 572 574 575 570 563 565 569 571 574.

3-byte messages: 569 573 575 576 571 564 566 570 572 575.

Interesting. Where do these numbers come from?

Another computation, same CPU: 771 768 751 752 751 752 751 752 751 752 751 752 751 752.
5. Advanced timing leaks


Poly1305 has no precomputation. Existing AES software does slow precomputation, making Poly1305-AES look slow. So I write new AES software.

I look at successive cycle counts for authenticating ten 1-byte messages: 3668 833 585 574 603 567 577 568 570 585.

2-byte messages: 568 572 574 575 570 563 565 569 571 574.

3-byte messages: 569 573 575 576 571 564 566 570 572 575.

Interesting. Where do these numbers come from?

Another computation, same CPU: 771 768 751 752 751 752 751 752 751 752 751 752 751 752 752.
I look at successive cycle counts for authenticating ten 1-byte messages: 3668 833 585 574 603 567 577 568 570 585.

2-byte messages: 568 572 574 575 570 563 565 569 571 574.

3-byte messages: 569 573 575 576 571 564 566 570 572 575.

Interesting. Where do these numbers come from?

Another computation, same CPU: 771 768 751 752 751 752 751 752 751 752 751 752.

Load-after-store conflicts:
On (e.g.) Pentium III, load from L1 cache is slightly slower if it involves the same cache line modulo 4096 as a recent store.

This timing variation happens even if all loads are from L1 cache!
I look at successive cycle counts for authenticating ten 1-byte messages: 3668 833 585 574 603 567 577 568 570 585.

2-byte messages: 568 572 574 575 570 563 565 569 571 574.

3-byte messages: 569 573 575 576 571 564 566 570 572 575.

Interesting. Where do these numbers come from?

Another computation, same CPU: 771 768 751 752 751 752 751 752 751 752 751 752 751 752.

Load-after-store conflicts:
On (e.g.) Pentium III, load from L1 cache is slightly slower if it involves same cache line modulo 4096 as a recent store.

This timing variation happens even if all loads are from L1 cache!
I look at successive cycle counts for authenticating ten 1-byte messages: 3668 833 585 574 603 567 577 568 570 585.

2-byte messages: 568 572 574 575 570 563 565 569 571 574.

3-byte messages: 569 573 575 576 571 564 566 570 572 575.

Interesting. Where do these numbers come from?

Another computation, same CPU:

771 768 751 752 751 752 751 752 751 752 751 752 751 752.

Load-after-store conflicts:
On (e.g.) Pentium III, load from L1 cache is slightly slower if it involves same cache line modulo 4096 as a recent store.

This timing variation happens even if all loads are from L1 cache!

Cache-bank throughput limits:
On (e.g.) Athlon, can perform two loads from L1 cache every cycle.

Exception: Second load waits for a cycle if loads are from same cache "bank."

Time for cache hit again depends on array index.

No guarantee that these are the only effects.
Load-after-store conflicts:

On (e.g.) Pentium III, load from L1 cache is slightly slower if it involves same cache line modulo 4096 as a recent store.

This timing variation happens even if all loads are from L1 cache!

Cache-bank throughput limits:

On (e.g.) Athlon, can perform two loads from L1 cache every cycle.

Exception: Second load waits for a cycle if loads are from same cache “bank.”

Time for cache hit again depends on array index.

No guarantee that these are the only effects.
Load-after-store conflicts:

On (e.g.) Pentium III, load from L1 cache is slightly slower if it involves the same cache line modulo 4096 as a recent store.

This timing variation happens even if all loads are from L1 cache!

Cache-bank throughput limits:

On (e.g.) Athlon, can perform two loads from L1 cache every cycle.

Exception: Second load waits for a cycle if loads are from same cache “bank.”

Time for cache hit again depends on array index.

No guarantee that these are the only effects.

6. Breaking AES in cache

2004: I point out cache-hit time variations in OpenSSL and other popular AES implementations.

2005: I extract complete key from OpenSSL timings, making no effort to knock table entries out of cache.

Many random known plaintexts.
Cache-bank throughput limits:
On (e.g.) Athlon, can perform two loads from L1 cache every cycle.
Exception: Second load waits for a cycle if loads are from same cache “bank.”
Time for cache hit again depends on array index.
No guarantee that these are the only effects.

6. Breaking AES in cache
2004: I point out cache-hit time variations in OpenSSL and other popular AES implementations.
2005: I extract complete key from OpenSSL timings, making no effort to knock table entries out of cache. Many random known plaintexts.
Cache-bank throughput limits:
On (e.g.) Athlon, can perform two loads from L1 cache every cycle.

Exception: Second load waits for a cycle if loads are from same cache “bank.”

Time for cache hit again depends on array index.

No guarantee that these are the only effects.

6. Breaking AES in cache

2004: I point out cache-hit time variations in OpenSSL and other popular AES implementations.

2005: I extract complete key from OpenSSL timings, making no effort to knock table entries out of cache. Many random known plaintexts.
6. Breaking AES in cache

2004: I point out cache-hit time variations in OpenSSL and other popular AES implementations.

2005: I extract complete key from OpenSSL timings, making no effort to knock table entries out of cache. Many random known plaintexts.
6. Breaking AES in cache

2004: I point out cache-hit time variations in OpenSSL and other popular AES implementations.

2005: I extract complete key from OpenSSL timings, making no effort to knock table entries out of cache.

Many random known plaintexts.

Graph has $x$-coordinates 0 through 255.


Encryption time (for this test code, this CPU, etc.) is maximized when $k[13] \oplus n[13] = 8$.

3-cycle signal.
Graph has \( x \)-coordinates 0 through 255.

\( y \)-coordinate: average cycles to encrypt random plaintext with \( k[13] \oplus n[13] = x \), minus average cycles to encrypt unrestricted random plaintext.

Encryption time (for this test code, this CPU, etc.) is maximized when \( k[13] \oplus n[13] = 8 \).

3-cycle signal.
Graph has $x$-coordinates 0 through 255.


Encryption time (for this test code, this CPU, etc.) is maximized when $k[13] \oplus n[13] = 8$. 3-cycle signal.
Graph has $x$-coordinates 0 through 255.


Encryption time (for this test code, this CPU, etc.) is maximized when $k[13] \oplus n[13] = 8$. 3-cycle signal.

Graph has $x$-coordinates 0 through 255.

$y$-coordinate: average cycles to encrypt random plaintext.

Encryption time (for this test code, this CPU, etc.) is maximized when $k \iff n = 8$.


This graph has much larger max, presumably L1 cache miss.

This graph has much larger max, presumably L1 cache miss.
2006: Mironov reports ciphertext-only attack deducing key after a few thousand ciphertexts.

Focus on last round of AES computation.

Obvious next research step:
Understand network noise!

Can we see ≈ 1-cycle signals from (e.g.) median of $10^6$ packet timings?

Would be another nice paper.

I’m not doing this; feel free to jump in.
2006: Mironov reports ciphertext-only attack deducing key after a few thousand ciphertexts.
Focus on last round of AES computation.

Obvious next research step: Understand network noise!
Can we see $\approx 1$-cycle signals from (e.g.) median of $10^6$ packet timings?

Would be another nice paper. I’m not doing this; feel free to jump in.
2006: Mironov reports ciphertext-only attack deducing key after a few thousand ciphertexts. Focus on last round of AES computation.

Obvious next research step: Understand network noise! Can we see $\approx 1$-cycle signals from (e.g.) median of $10^6$ packet timings?

Would be another nice paper. I’m not doing this; feel free to jump in.

7. Misdesigning cryptography

Primary goal of cryptography: Continued employment for cryptographers.

How to achieve this?

Example: Use 512-bit RSA.

Oops, broken? Use 768-bit RSA.

Oops, broken? Use 1024-bit RSA.
2006: Mironov reports ciphertext-only attack deducing key after a few thousand ciphertexts. Focus on last round of AES computation.

Obvious next research step: Understand network noise! Can we see $\approx 1$-cycle signals from (e.g.) median of $10^6$ packet timings?

Would be another nice paper. I’m not doing this; feel free to jump in.

7. Misdesigning cryptography

Primary goal of cryptography: Continued employment for cryptographers.

How to achieve this?

Example: Use 512-bit RSA. Oops, broken? Use 768-bit RSA. Oops, broken? Use 1024-bit RSA.
2006: Mironov reports ciphertext-only attack deducing key after a few thousand ciphertexts.

Focus on last round of AES computation.

Obvious next research step: Understand network noise!

Can we see ı cycle signals from (e.g.) median of 10⁶ packet timings?

Would be another nice paper.

I’m not doing this; feel free to jump in.

7. Misdesigning cryptography

Primary goal of cryptography: Continued employment for cryptographers.

How to achieve this?

Example: Use 512-bit RSA.
Oops, broken? Use 768-bit RSA.
Oops, broken? Use 1024-bit RSA.

Don’t believe that attacks work until they’ve been announced in the New York Times.

For timing attacks: If attack hasn’t been demonstrated, assume it doesn’t work.

Don’t use obviously-constant-time software such as Phelix.
7. Misdesigning cryptography

Primary goal of cryptography: Continued employment for cryptographers.

How to achieve this?

Example: Use 512-bit RSA.
Oops, broken? Use 768-bit RSA.
Oops, broken? Use 1024-bit RSA.

Don’t believe that attacks work until they’ve been announced in the New York Times.

For timing attacks: If attack hasn’t been demonstrated, assume it doesn’t work.

Don’t use obviously-constant-time software such as Phelix.
Don’t believe that attacks work until they’ve been announced in the New York Times.

For timing attacks: If attack hasn’t been demonstrated, assume it doesn’t work.

Don’t use obviously-constant-time software such as Phelix.

Don’t use cryptographic hardware. Build complex multi-layer cryptographic systems. Don’t communicate adequately between people designing different layers.

e.g. Most CPU designers fail to thoroughly document CPU speed.

Challenge: Market a CPU with a variable-time adder.
Don’t believe that attacks work until they’ve been announced in the New York Times.

For timing attacks: If attack hasn’t been demonstrated, assume it doesn’t work.

Don’t use obviously-constant-time software such as Phelix.

Don’t use cryptographic hardware.
Build complex multi-layer cryptographic systems.
Don’t communicate adequately between people designing different layers.
e.g. Most CPU designers fail to thoroughly document CPU speed.

Challenge: Market a CPU with a variable-time adder.