Cache-timing attacks on AES

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.
Attack extracted the AES key from timings of the server.


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

Attack extracted the AES key from timings of the server.


“In some environments, timing attacks can be effected against operations that execute in different amounts of time, depending on their arguments.
Attack extracted the AES key from timings of the server.


“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
Attack extracted the AES key from *timings* of the server.


“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 . . .
Attack extracted the AES key from timings of the server.

AES selectors (US NIST), "Report on the development of the Advanced Encryption Standard (AES)," 2001:

"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 . . .

"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. . . .
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. . . . 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 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.)

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

“Table lookups: This instruction is not susceptible to a timing attack. Favorable: Algorithms that use only logical, 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.”
“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.)

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

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

“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, 1999: Speed reports “should take into account the measures to be taken to thwart these attacks.”

(Amazing change of position, 2005: Timing attacks are “irrelevant for cryptographic design.”

Bruce Schneier, 2005: “The problem is that side-channel attacks are practical against pretty much anything, so it didn’t really enter into consideration.”)
AES designers (Daemen, Rijmen), 1999: Speed reports “should take into account the measures to be taken to thwart these attacks.”

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.

The problem in a nutshell: Daemen, Rijmen, NIST were wrong. Variable-index table lookup is vulnerable to timing attacks.

AES does many lookups such as table[k[3]^n[3]], taking time that depends on k[3], leaking k[3]. It is extremely difficult to avoid this leak on Pentium, Athlon, etc. without gigantic slowdowns.
AES designers, 1999: Speed reports “should take into account the measures to be taken to thwart these attacks.”

(Amazing change of position, 2005: Timing attacks are “irrelevant for cryptographic design.”) Bruce Schneier, 2005: “The problem is that side-channel attacks are practical against pretty much anything, so it didn’t really enter into consideration.”

The problem in a nutshell: Daemen, Rijmen, NIST were wrong. Variable-index table lookup is vulnerable to timing attacks.


It is extremely difficult to avoid this leak on Pentium, Athlon, etc. without gigantic slowdowns.
The problem in a nutshell: Daemen, Rijmen, NIST were wrong. Variable-index table lookup is vulnerable to timing attacks.


It is extremely difficult to avoid this leak on Pentium, Athlon, etc. without gigantic slowdowns.

Naive reaction: “Oh, of course. A table lookup takes more time if it misses the cache.”

Mentioned by 1996 Kocher, 2000 Kelsey Schneier Wagner Hall.

The problem in a nutshell:
Daemen, Rijmen, NIST were wrong.
Variable-index table lookup is vulnerable to timing attacks.


It is extremely difficult to avoid this leak on Pentium, Athlon, etc. without gigantic slowdowns.

Naive reaction:
“Of course. A table lookup takes more time if it misses the cache.”

Mentioned by 1996 Kocher, 2000 Kelsey Schneier Wagner Hall.
The problem in a nutshell: Daemen, Rijmen, NIST were wrong.
Variable-index table lookup is vulnerable to timing attacks.

AES does many lookups such as $\text{table}[k^3]^n^3$, taking time that depends on $k^3$,
leaking $k^3$.
It is extremely difficult to avoid this leak on Pentium, Athlon, etc.
without gigantic slowdowns.

Naive reaction:
“Oh, of course. A table lookup takes more time if it misses the cache.”

Mentioned by 1996 Kocher, 2000 Kelsey Schneier Wagner Hall.

Naive reaction, continued:
“But this isn’t hard to fix. Eliminate cache misses by loading all the AES tables into cache before the AES encryption.”

Problem 1: Eliminating cache misses is actually much more difficult than this, and is not achieved by any existing high-speed AES software.

Problem 2: Even in cache, table-lookup time is variable.
Naive reaction:
“Oh, of course. A table lookup takes more time if it misses the cache.”

Mentioned by 1996 Kocher, 2000 Kelsey Schneier Wagner Hall.

Naive reaction, continued:
“But this isn’t hard to fix. Eliminate cache misses by loading all the AES tables into cache before the AES encryption.”

Problem 1: Eliminating cache misses is actually much more difficult than this, and is not achieved by any existing high-speed AES software.

Problem 2: Even in cache, table-lookup time is variable.
Naive reaction, continued: “But this isn’t hard to fix. Eliminate cache misses by loading all the AES tables into cache before the AES encryption.”

Problem 1: Eliminating cache misses is actually much more difficult than this, and is not achieved by any existing high-speed AES software.

Problem 2: Even in cache, table-lookup time is variable.

Obstacles to writing code that looks up a table entry in time independent of index:

- Cache is faster than DRAM.
- L1 cache is faster than L2 cache.
- Cache associativity is limited.
- Code can be interrupted.
- Stores can interfere with loads.
- Cache-bank throughput is limited.

Perhaps there are more obstacles.

Most CPU designers fail to adequately document CPU speed!
Naive reaction, continued:
“But this isn’t hard to fix. Eliminate cache misses by loading all the AES tables into cache before the AES encryption.”

Problem 1: Eliminating cache misses is actually much more difficult than this, and is not achieved by any existing high-speed AES software.

Problem 2: Even in cache, table-lookup time is variable.

Obstacles to writing code that looks up a table entry in time independent of index:

- Cache is faster than DRAM.
- L1 cache is faster than L2 cache.
- Cache associativity is limited.
- Code can be interrupted.
- Stores can interfere with loads.
- Cache-bank throughput is limited.

Perhaps there are more obstacles. Most CPU designers fail to adequately document CPU speed!
Continued:
Naive reaction, continued:
"But this isn't hard to fix. Eliminate cache misses by loading all the AES tables into cache before the AES encryption."

Problem 1: Eliminating cache misses is actually much more difficult than this, and is not achieved by any existing high-speed AES software.

Problem 2: Even in cache, table-lookup time is variable.

Obstacles to writing code that looks up a table entry in time independent of index:

- Cache is faster than DRAM.
- L1 cache is faster than L2 cache.
- Cache associativity is limited.
- Code can be interrupted.
- Stores can interfere with loads.
- Cache-bank throughput is limited.

Perhaps there are more obstacles.

Most CPU designers fail to adequately document CPU speed!

AES software uses several arrays: tables, input, key, etc. On (e.g.) Athlon, if array positions overlap modulo 32768, accessing two arrays can knock an entry in another array out of cache.

Fix: squeeze variables into a limited number of arrays; reload tables before AES calls.
Obstacles to writing code that looks up a table entry in time independent of index:

- Cache is faster than DRAM.
- L1 cache is faster than L2 cache.
- Cache associativity is limited.
- Code can be interrupted.
- Stores can interfere with loads.
- Cache-bank throughput is limited.

Perhaps there are more obstacles. Most CPU designers fail to adequately document CPU speed!

Cache associativity is limited

AES software uses several arrays: tables, input, key, etc.

On (e.g.) Athlon, if array positions overlap modulo 32768, accessing two arrays can knock an entry in another array out of cache.

Fix: squeeze variables into a limited number of arrays; reload tables before every AES call.
Obstacles to writing code that looks up a table entry in time independent of index:

- Cache is faster than DRAM.
- L1 cache is faster than L2 cache.
- Cache associativity is limited.
- Code can be interrupted.
- Stores can interfere with loads.
- Cache-bank throughput is limited.

Perhaps there are more obstacles.
Most CPU designers fail to adequately document CPU speed!

Cache associativity is limited

AES software uses several arrays: tables, input, key, etc.

On (e.g.) Athlon, if array positions overlap modulo 32768, accessing two arrays can knock an entry in another array out of cache.

Fix: squeeze variables into a limited number of arrays; reload tables before every AES call.

Another process can knock an array entry out of cache.


Fix: put AES into operating-system kernel, disabling interrupts. Kernel call takes time; requires massive code changes; but should work.
Cache associativity is limited

AES software uses several arrays: tables, input, key, etc.

On (e.g.) Athlon, if array positions overlap modulo 32768, accessing two arrays can knock an entry in another array out of cache.

Fix: squeeze variables into a limited number of arrays; reload tables before every AES call.

Code can be interrupted

Another process can knock an array entry out of cache.


Fix: put AES into operating-system kernel, disabling interrupts. Kernel call takes time; requires massive code changes; but should work.
Cache associativity is limited

AES software uses several arrays:
tables, input, key, etc.

On (e.g.) Athlon, if array positions overlap modulo 32768, accessing two arrays can knock an entry in another array out of cache.

Fix: squeeze variables into a limited number of arrays; reload tables before every AES call.

Code can be interrupted

Another process can knock an array entry out of cache.


But problem exists even without hyperthreading.

Fix: put AES into operating-system kernel, disabling interrupts. Kernel call takes time; requires massive code changes; but should work.

Stores can interfere with loads

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

Timing variation happens even if all loads are from L1 cache!

Fix: compress AES tables; control table positions modulo 4096.
Code can be interrupted
Another process can knock an array entry out of cache.
But problem exists even without hyperthreading.
Fix: put AES into operating-system kernel, disabling interrupts. Kernel call takes time; requires massive code changes; but should work.

Stores can interfere with loads
On (e.g.) Pentium III, load from L1 cache is slightly slower if it involves same cache line modulo 4096 as a recent store.
Timing variation happens even if all loads are from L1 cache!
Fix: compress AES tables; control table positions modulo 4096.
Code can be interrupted. Another process can knock an array entry out of cache. Consider hyperthreading attacks (2005 Osvik Shamir Tromer, independently 2005 Percival). But problem exists even without hyperthreading. Fix: put AES into operating-system kernel, disabling interrupts. Kernel call takes time; requires massive code changes; but should work.

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

Timing variation happens even if all loads are from L1 cache! Fix: compress AES tables; control table positions modulo 4096.

Cache-bank throughput is limited. 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.” Fix: very careful asm programming to ensure that two loads never happen in the same cycle. My AES software tries to do this.
Stores can interfere with loads
On (e.g.) Pentium III, load from L1 cache is slightly slower if it involves same cache line modulo 4096 as a recent store.

Timing variation happens even if all loads are from L1 cache!

Fix: compress AES tables; control table positions modulo 4096.

Cache-bank throughput is limited
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.”

Fix: very careful asm programming to ensure that two loads never happen in the same cycle. My AES software tries to do this.
Stores can interfere with loads
On (e.g.) Pentium III,
load from L1 cache is
slightly slower if it involves
same cache line modulo 4096
as a recent store.

Timing variation happens
even if all loads are from L1 cache!
Fix: compress AES tables; control
table positions modulo 4096.

Cache-bank throughput is limited
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.”

Fix: very careful asm programming
to ensure that two loads
never happen in the same cycle.
My AES software tries to do this.

Do we want to keep AES?
Variable-index table lookups
are dangerous.
Clearly impossible to achieve
today’s advertised AES speeds
with constant-time software.
I believe it’s possible to
achieve tolerable speeds,
but it’s extremely difficult,
and it’s fragile: new CPUs
will allow more attacks.
Cache-bank throughput is limited
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.”

Fix: very careful asm programming to ensure that two loads never happen in the same cycle. My AES software tries to do this.

Do we want to keep AES?
Variable-index table lookups are dangerous.

Clearly impossible to achieve today’s advertised AES speeds with constant-time software.

I believe it’s possible to achieve tolerable speeds, but it’s extremely difficult, and it’s fragile: new CPUs will allow more attacks.
Do we want to keep AES?

Variable-index table lookups are dangerous.

Clearly impossible to achieve today’s advertised AES speeds with constant-time software.

I believe it’s possible to achieve tolerable speeds, but it’s extremely difficult, and it’s fragile: new CPUs will allow more attacks.

Why not switch to a cipher that avoids these problems?

2005 Schneier: “Pretty much any encryption algorithm is susceptible to timing attacks, so choosing on that regard doesn’t make that much sense.”

But some fast ciphers are not susceptible to timing.

Do we want to keep AES?

Variable-index table lookups are dangerous.

Clearly impossible to achieve today’s advertised AES speeds with constant-time software.

I believe it’s possible to achieve tolerable speeds, but it’s extremely difficult, and it’s fragile: new CPUs will allow more attacks.

Why not switch to a cipher that avoids these problems?

2005 Schneier: “Pretty much any encryption algorithm is susceptible to timing attacks, so choosing on that regard doesn’t make that much sense.”

But some fast ciphers are not susceptible to timing attacks! Can build fast cipher from xor, add, constant-distance rotation. Examples: TEA, Helix, Salsa20.