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.

acks

ois at Chicago
950
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 ta

1. How to advert
   an AES candi
2. How to leak k
   timings: basic
3. How to break
   by forcing cac
4. How to skew
5. How to leak k
   timings: adva
6. How to break
   without cache
7. How to misde
   a cryptograph

`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

`to/papers.html`
2005:

rts successful
omplete AES key
erver
uter.
ver used its key
data using the
nplementation
.”

l in paper.
le.

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 ar

1997: US NIST a
cipher competitio
replacing DES as
approved block c

1999: NIST anno
RC6, Rijndael, Se
as AES finalists.

2001: NIST publ
the development
Encryption Stand
explaining selecti
AES.

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

1997: US NIST announces block-cipher competition. Goal: AES, replacing DES as US government-approved block cipher.

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

2001: NIST publishes "Report on the development of the Advanced Encryption Standard (AES)," explaining selection of Rijndael as AES.

alk:
tise
date
keys through
c techniques
AES remotely
che misses
a benchmark
keys through
nced techniques
AES remotely
e misses
sign
ic architecture

1. Advertising an AES candidate

1997: US NIST announces block-cipher competition. Goal: AES, replacing DES as US government-approved block cipher.

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

2001: NIST publishes "Report on the development of the Advanced Encryption Standard (AES)," explaining selection of Rijndael as AES.

1996: Kocher ex
from *timings* of a

Clear threat to b
too. As stated in

"In some environ
timing attacks ca
against operation
in different amou
depending on the

1. Advertising an AES candidate

1997: US NIST announces block-cipher competition. Goal: AES, replacing DES as US government-approved block cipher.

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

2001: NIST publishes "Report on the development of the Advanced Encryption Standard (AES)," explaining selection of Rijndael as AES.

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.

n AES candidate

announces block-
on. Goal: AES,
s US government-
cipher.

ounces MARS,
erpent, Twofish

lishes "Report on
of the Advanced
dard (AES),"
on of Rijndael as

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 defen
timing attacks is
each encryption a
operation runs in
amount of time.

"Table lookup: n
timing attacks . .

"Multiplication/d
or variable shift/
most difficult to

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 . . .

tracts RSA key
a server.

lock-cipher keys
n NIST's report:

ıments,
an be effected
ns that execute
ınts of time,
eir 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 Se
only Boolean ope
table lookups, an
shifts/rotations.
are the easiest to
attacks. . . .

"Finalist profiles.
operations used b
among the easies
against power an
attacks. . . . Rijnd
gain a major spe
over its competit
protections are c

“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. . . .

se against
to ensure that
and decryption
the same
. . .

not vulnerable to
.

division/squaring
rotation:
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 Ri
best overall algor
AES. Rijndael ap
consistently good
Its key setup tim
and its key agility
Rijndael's operat
the easiest to def
power and timing
Finally, Rijndael's
round structure a
good potential to
instruction-level
(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.)

rpent use
erations,
nd fixed
These operations
o defend against

. . . The
by Rijndael are
st to defend
d timing
lael appears to
ed advantage
tors when such
onsidered. . . .

“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 desig
Rijmen) publish
against implemen
a comparative st
proposals”:

“Table lookups:
is not susceptible
attack. . . . Favor
that use only log
table-lookups and
and that are ther
easy to secure. T
of this group are
Magenta, Rijnda

"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."

jndael to be the
rithm for the
pears to be a
l performer . . .
e is excellent,
y is good. . . .
ions are among
fend against
g attacks. . . .
s internal
appears to have
o benefit from
parallelism.”
l.)

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 w
reports “should t
the measures to
thwart these atta

2005, after AES
vulnerable, amaz
position: Timing
“irrelevant for cr
design.” Schneie
“The problem is
attacks are pract
pretty much anyt
really enter into

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."

ners (Daemen,
“Resistance
ntation attacks:
udy of the AES

This instruction
e to a timing
rable: Algorithms
ical operations,
d fixed shifts,
refore relatively
The algorithms
Crypton, DEAL,
el 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.”

2. Leaking keys t

Most obvious tim
skipping an opera
than doing it.

1970s: TENEX c
compares user-su
against secret pa
character a a tim
first difference. A
comparison time,
of difference. A f
reveal secret pass

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.

rite: Speed
ake into account
be taken to
acks.”

is shown to be
ing change of
attacks are
yptographic
r, 2005:
that side-channel
ical against
thing, so it didn’t
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 co
password compar

Old:

```
   for (i = 0
     if (x[i]
       return
   return 1;
```

New:

```
   diff = 0;
   for (i = 0
     diff |= x
   return !di
```

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;
```

through timings

ning variability:
ation is faster

operating system
upplied string
ssword one
ne, stopping at
Attackers monitor
, deduce position
few hundred tries
sword.

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 po
attacks on crypto
Example: key-de
in modular reduc
large-integer sub
inputs and not o

My reaction at th
Eliminate variabl
from cryptograph
Beware microSPA
data-dependent
use Fermat inste
inversion in ECC
avoid S-boxes in

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.

nstant-time
rison.

```
;i < n;++i)
 != y[i])
 0;
```

```
;i < n;++i)
x[i] ^ y[i];
ff;
```

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 Q
fast timing attac
implementation”
used input-depen

AES has functio
bytes to bytes. A
S' computed as f

```
byte Sprim
  byte c =
  if (c<12
  return (c
}
```

Timing leaks bit
$c < 128$.

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:

```
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 $c < 128$.

ints out timing
ographic key bits.
pendent branch
tion, performing
traction for some
thers, leaking key.

he time: Yikes!
e-time operations
nic software!
ARC-IIep
FPU timings;
ad of Euclid for
;
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:

```
   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 solutio
replace branch by

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

CPUs handle this
in constant time.

Koeune Quisquat
"The result prese
not an attack ag
but against
bad implementat

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."

uisquater publish
k on a "careless
of AES that
ndent branches.

ns S, S' mapping
ttack is against
follows:

```
e(byte b) {
 S(b);
3) return c+c;
c+c)^283;
```

of c: faster if

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 obv
variability: L2 ca
than DRAM. Sim
is faster than L2

Reading from cac
takes less time th
reading from unc

Variability mentio
Kocher, 2000 Ke
Wagner Hall ("W
based on cache h
S-box ciphers like
and Khufu are po
Ferguson Schneie

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.

n:

y arithmetic.

```
) ;
) ;
<1)^X;
```

s arithmetic

ter:

ented here is

ainst Rijndael,

ions 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 publ

algorithm to find

from high-bandw

information. DP

plaintexts, each s

empty cache. Al

for each plaintex

lookups that mis

Avoid empty cac

some S-box entri

guarantee this as

countermeasure

the cache with th

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."

vious timing
che is faster
nilarly, L1 cache
cache.

ched line
han
cached line.

oned by 1996
lsey Schneier
Ve believe attacks
hit ratio in large
e Blowfish, CAST
ossible"), 2003
er.

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, S
Shigeri, Miyauch
algorithm to find
low-bandwidth ti
Many plaintexts,
with empty cach
input: for each p
encryption time.

"If a total-data l
before processing
between the freq
misses will not b
making it imposs
the relationships
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.”

ishes fast
DES key
idth timing
A-style. Many
starting with
gorithm input:
t, list of S-box
sed the cache.

he by preloading
es? "To
s an effective
we need to warm
he entirety of all

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 se
and 16-byte sequ
AES produces
16-byte sequence

Uses table looku
e0 = tab[k[13]
e1 =
tab[k[0]⊕n[0]
etc.
$\text{AES}_k(n) = ($e78

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 $\mathrm{AES}_k(n)$.

Uses table lookup and ⊕ (xor):

`e0 = tab[k[13]]`⊕`1`

`e1 =`

`tab[k[0]`⊕`n[0]]`⊕`k[0]`⊕`e0`

etc.

$\mathrm{AES}_k(n) = (\texttt{e784}, \ldots, \texttt{e799})$.

aito, Suzaki,
i publish fast
DES key from
ming information.
each starting
e. Algorithm
laintext,

oad is executed
g, differences
uencies of cache
e observed,
sible to determine
between sets of

3. Breaking AES

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

Uses table lookup and ⊕ (xor):
e0 = tab[k[13]]⊕1
e1 =
tab[k[0]⊕n[0]]⊕k[0]⊕e0
etc.
$\mathrm{AES}_k(n) = (\mathtt{e784}, \ldots, \mathtt{e799})$.

High-speed AES
registers, several
Operations: byte
bytes to 1 byte),
byte to 4 byte),

Attacker can forc
table entries out
observe encryptio
Each cache miss
signal, clearly vis
from other AES
other software, e
Repeat for many
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 ⊕ (xor):

```
e0 = tab[k[13]]⊕1
e1 =
tab[k[0]⊕n[0]]⊕k[0]⊕e0
```

etc.

$\text{AES}_k(n) = (\texttt{e784}, \ldots, \texttt{e799})$.

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.

2

quence n
uence k,

e AES$_k(n)$.

p and ⊕ (xor):
]⊕1

]⊕k[0]⊕e0

4, . . . , e799).

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: tab[k[
hundreds of extra
tab entry is not

Knock tab[13] o
signal when k[0]
Deduce k[0] as n
(Complication: c
need more work
bottom bits of k

More efficient: K
tab entries out o
Then first n[0] li
to half of its pos

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: `tab[k[0]` $\oplus$ `n[0]]` costs hundreds of extra cycles if this `tab` entry is not in L2 cache.

Knock `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.

uses 4-byte
1024-byte tables.
e extraction (4
table lookup (1
$\oplus$.

ce selected
of L2 cache,
on time.
creates timing
ible despite noise
cache misses,
tc.
plaintexts,
y.

Example: `tab[k[0]` $\oplus$ `n[0]]` costs hundreds of extra cycles if this `tab` entry is not in L2 cache.

Knock `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
L1 cache is 2-wa
three 64-byte line
address modulo 3
the first line is fo
L1 cache.

Athlon's 524288-
is 16-way associa
with the same ad
8192 are read, th
forced out of the

Force `tab[13]` ou
by accessing sele
locations.

Example: `tab[k[0] ⊕ n[0]]` costs hundreds of extra cycles if this `tab` entry is not in L2 cache.

Knock `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 `tab`[13] out of cache by accessing selected memory locations.

0] ⊕ n[0]] costs
a cycles if this
in L2 cache.

ut of cache. See
$\oplus\ n[0] = 13$.
$n[0] \oplus 13$.
cache lines;
to find
[0].)

Knock half of the
of cache.
mits $k[0]$
sibilities.

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 tab[13] out of cache by accessing selected memory locations.

How does attack
necessary accesse
multiuser compu
account. Almost
an account: e.g.,
Java applet to us

What if compute
no buffer overflo
still possible to c
attack from anot
by figuring out p
sent to (e.g.) Lin
accesses of appro
locations. Nobo
Would make a ni

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 `tab`[13] out of cache by accessing selected memory locations.

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!

: 65536-byte
y associative. If
es with the same
32768 are read,
orced out of the

-byte L2 cache
tive. If 17 lines
ddress modulo
e first line is
 L2 cache.

t of cache
cted memory

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" co
reading all AES t
starting AES con

Even if this were
eliminate cache r
entries can drop
during the compu

Typical AES soft
different arrays:
output, stack, S-
sometimes kicks
lines out of L1 ca
(e.g.) the key an

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.

er do the
es? Trivial on
ter if attacker has
as easy without
, attacker sends
ser's browser.

er has no browser,
ws, etc.? Clearly
arry out the
her computer
ackets that, when
ux kernel, cause
opriate memory
dy has done this!
ice 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
implementation,
implementation,
variables into a li
of arrays. But th
eliminate cache r

Computers run m
simultaneous pro
AES software car
by another proce
lines out of L1 ca
even L2 cache. E
partial-AES cach
the timing of the

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.

untermeasure,
tables before
nputation?

free, it wouldn't
misses. Table
out of cache
utation.

ware uses several
input, key,
boxes. Software
its own S-box
ache by accessing
d 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
accident.

Can force much
frequent interrup
"hyperthreading"
Shamir Tromer,
2005 Percival—g
bandwidth timing

Not clear whethe
approach can be
remotely via (e.g

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.

5 AES
Gladman's latest
etc.: squeeze
imited number
is *still* doesn't
misses!

nany
cesses. The
n be interrupted
ss that kicks
ache and maybe
Even worse, the
e state affects
e 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.

It *is* possible to s
all AES cache mi

Put AES softwar
operating-system
Disable interrupt
Disable hyperthr
Read all S-boxes
Wait for reads to
Encrypt some bl

The bad news, a
Stopping cache
enough. There a
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*.

interrupts by

more
ts with
”—2005 Osvik
independently
giving high-
g information.

er hyperthreading
carried out
.) 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 benc

Many deceptive t
the cryptographic

- Bait-and-switch
- Guesses reporte
- My-favorite-CP
- Long-message
- Timings after p
- High-variance t

Consequence: In
these functions a
much slower than

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.

stop
isses.

e into
kernel.
s.
eading etc.
into cache.
complete.
ocks of data.

s we'll see later:
misses isn't
re timing leaks

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 t
Create two versio
function, a small
and a big Fun-Sl
timings for Fun-B

Example in litera
proposes 16-byte
Says "More than
on a 200 MHz P
. . . but that's ac
breakable 4-byte

The honest alter
Focus on *one* fu

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.

hmarks

timings in
c literature:

h timings.
ed as timings.
PU timings.
timings.
precomputation.
timings.

the real world,
re often
n 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
Measure only par
computation.
Estimate the oth

Example in litera
2.2 clock cycles p
the unimplement
fast as various es

The honest alter
exactly the funct
that applications

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.

timings:
ons of your
Fun-Breakable
ow. Report
Breakable.

ture: Paper
authenticator.
1 Gbit/sec
entium Pro”
tually for a
authenticator.

native:
nction.

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
CPU where funct
Ignore all other C

Example in litera
were measured o
. . . because othe
many more cycle
for this particular

The honest alterr
Measure every C
If reader doesn’t
a particular chip,

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.

as timings:
rt of the

er parts.

ture: “achieves
per byte” . . . if
ed parts are as
stimates.

native: Measure
ion call
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.

Long-message tir
time only for lon
Ignore per-messa
Ignore applicatio
handle short pac

Example in litera
“2 cycles per byt
. . . plus 2000 cy

The honest alter
Report times for
for each $n \in \{0,$

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: Choose
tion is very fast.
CPUs.

ture: “All speeds
n a Pentium 4”
r chips take
s per byte
r computation.

native:
PU you can find.
care about
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 pre
Report time *afte*
a big key-depend
has been precom
and loaded into
Ignore applicatio
handle many sim

The honest alter
Measure precom
measure time to
that weren’t alre

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.

mings: Report
g messages.
ge overhead.
ns that
kets.

ture:
e”
cles per packet.

native:
n-byte packets
$1, 2, \ldots, 8192\}$.

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 tin
Measure each fur
time, on a single
Ignore possibility
in timing.

Compare functior
single timings, pr
high-variance fun

The honest alterr
Report several m
making variance

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.

ecomputation:
r
ent table
puted
_1 cache.
ns that
ultaneous keys.

native:
putation time;
load inputs
ady 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.

5. Advanced tim

2004: I write sof
Poly1305-AES, a
message authent
Wegman-Carter s
combining a prov
"universal" hash
a hopefully-secur
(AES in counter

Poly1305 has no
Existing AES sof
slow precomputa
Poly1305-AES lo
write new AES s

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.

nings:
nction a single
input.
of high variance

ns by comparing
romoting a few
nctions.

native:
easurements,
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.

I look at successi
for authenticating
messages: 3668 8
567 577 568 570

2-byte messages:
575 570 563 565

3-byte messages:
576 571 564 566

Interesting. Whe
numbers come fr

Another computa
771 768 751 752
751 752 751 752

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.

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.

ing leaks

tware for
state-of-the-art
icator. Standard
structure,
vably secure
(Poly1305) with
e stream cipher
mode).

precomputation.
tware does
tion, making
ok slow. So I
oftware.

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

On (e.g.) Pentiu
load from L1 cac
slightly slower if
same cache line
as a recent store.

This timing varia
even if all loads a
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!

ive cycle counts
g ten 1-byte
833 585 574 603
585.

568 572 574
569 571 574.

569 573 575
570 572 575.

ere do these
om?

ation, same CPU:
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 thro

On (e.g.) Athlon
can perform two
from L1 cache ev

Exception: Secon
waits for a cycle
are from same ca

Time for cache *h*
again depends on

No guarantee tha
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.

conflicts:

m III,
che is
it involves
modulo 4096

tion happens
are from L1

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

2004: I point ou
cache-hit time va
in OpenSSL and
popular AES imp

2005: I extract c
from OpenSSL ti
making no effort
knock table entri
Many random kn

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.

ughput limits:

,
loads
very cycle.

nd load
if loads
ache “bank.”

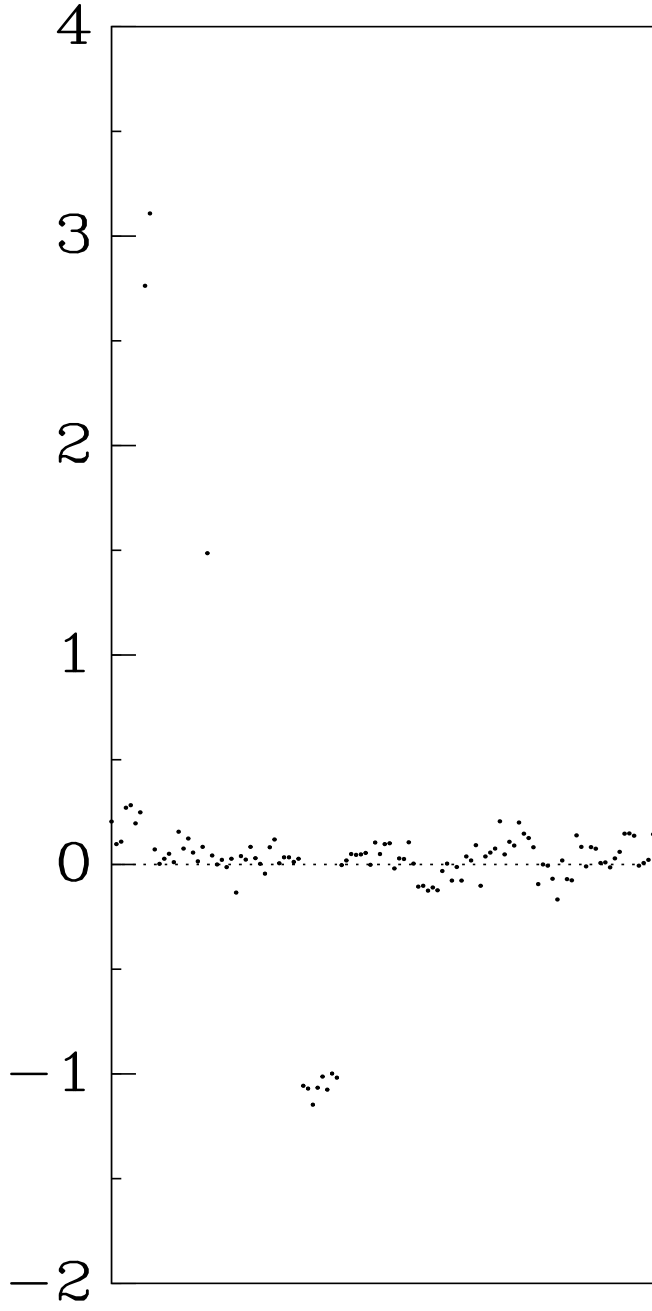
hit
n array index.

at these are the

6. Breaking AES in cache

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

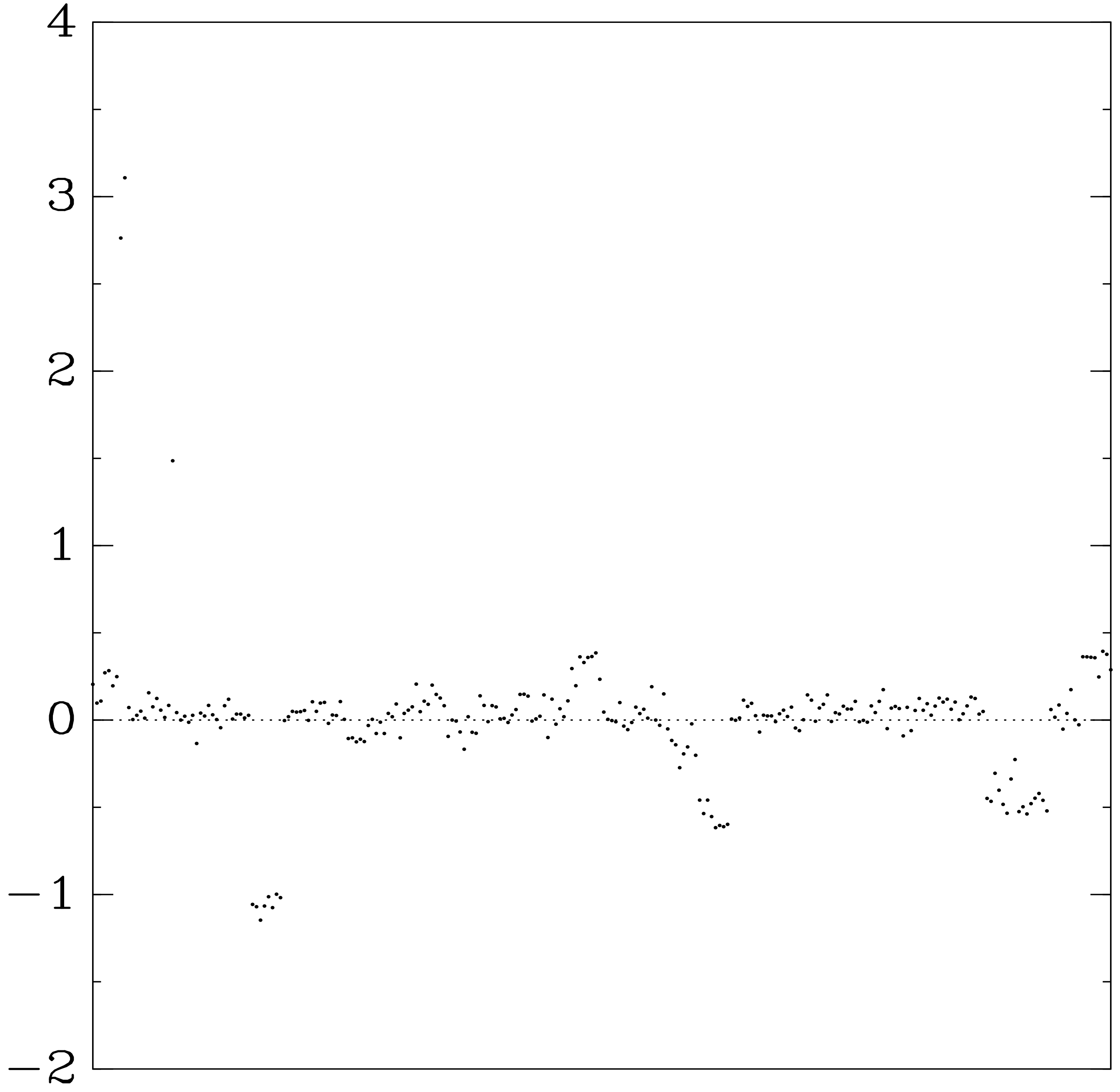
2005: I extract complete key from OpenSSL timings, making no effort to knock table entries out of cache. Many random known plaintexts.



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.



s in cache

t

ariations

other

plementations.

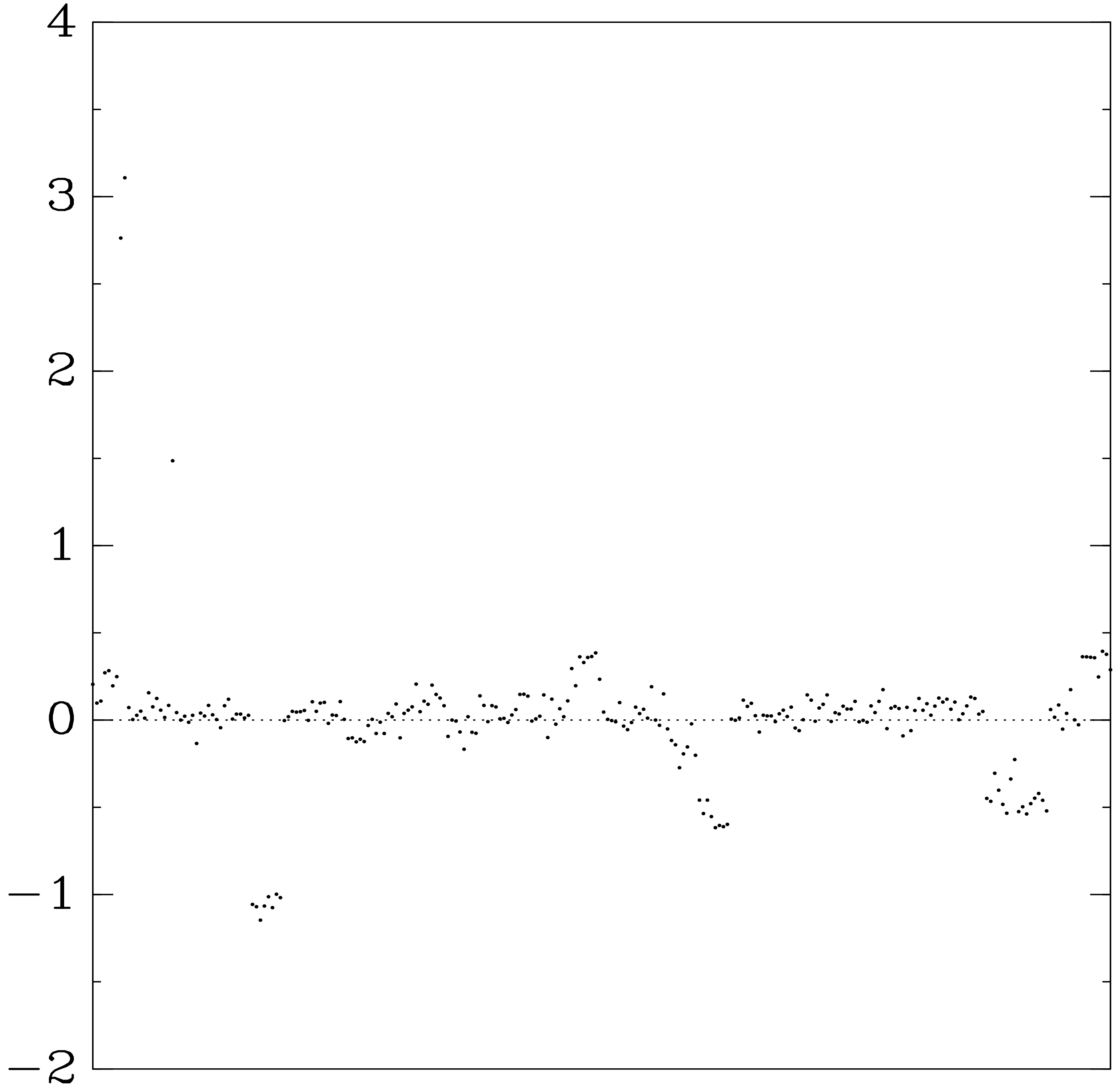
omplete key

imings,

to

es out of cache.

nown plaintexts.



Graph has x-coor

0 through 255.

y-coordinate: av

to encrypt rando

with $k[13] \oplus n[1$

minus average cy

unrestricted rand

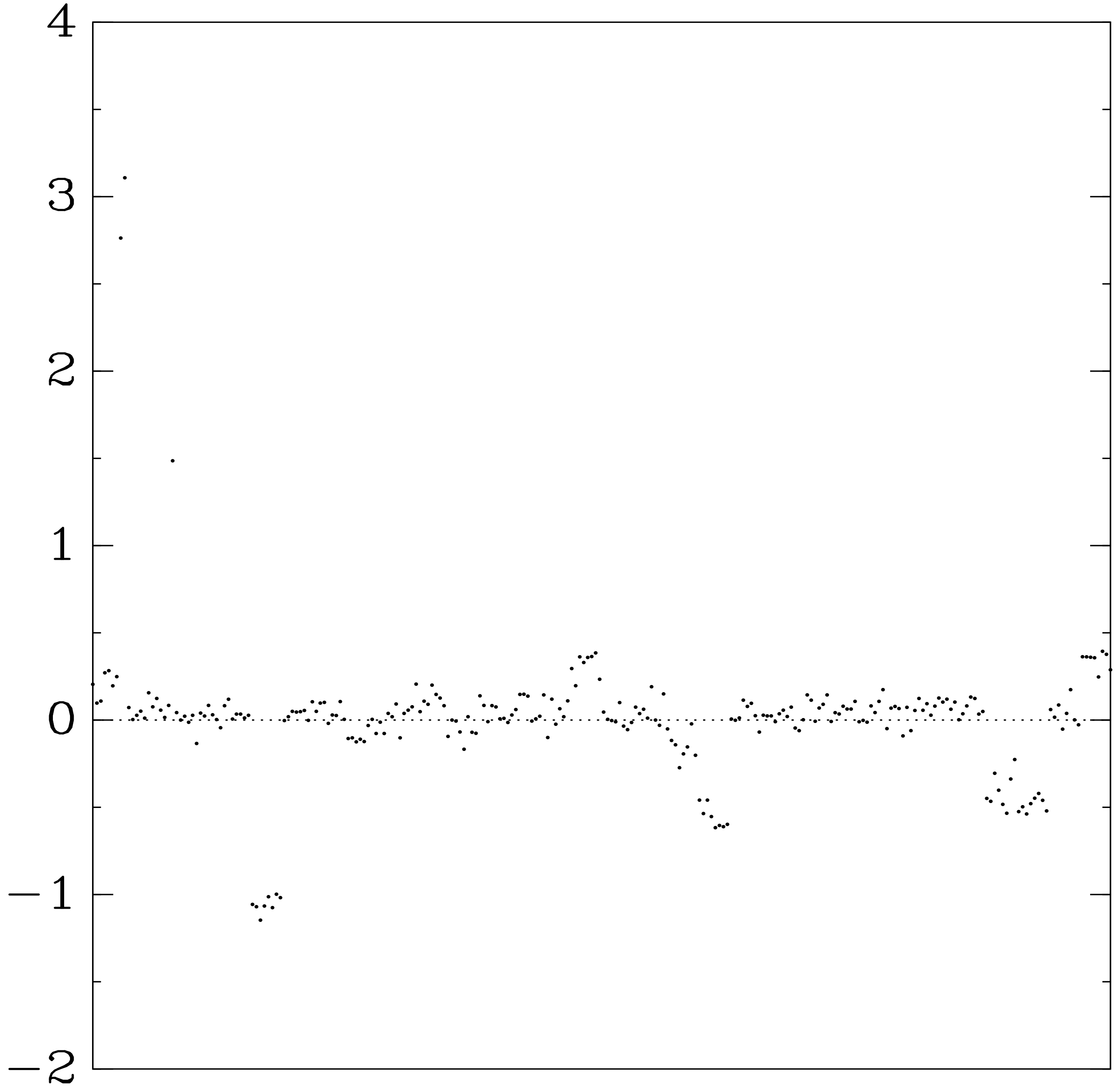
Encryption time

code, this CPU,

is maximized wh

$k[13] \oplus n[13] =$

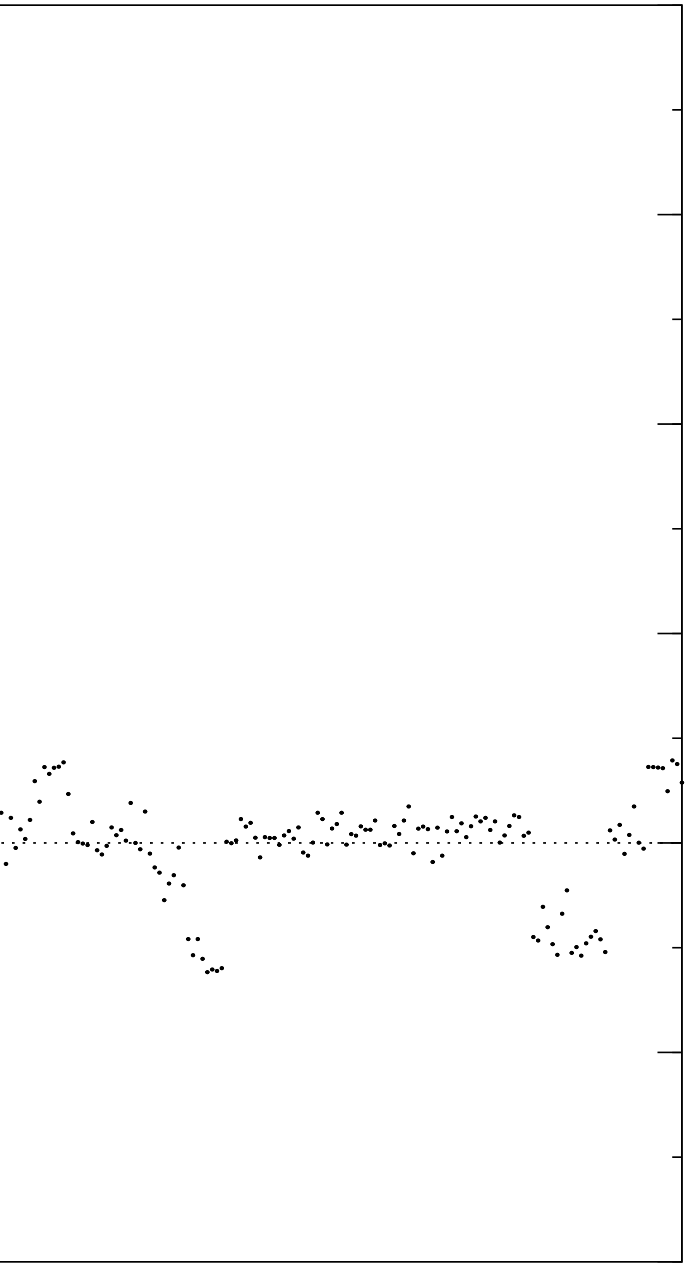
3-cycle signal.



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.

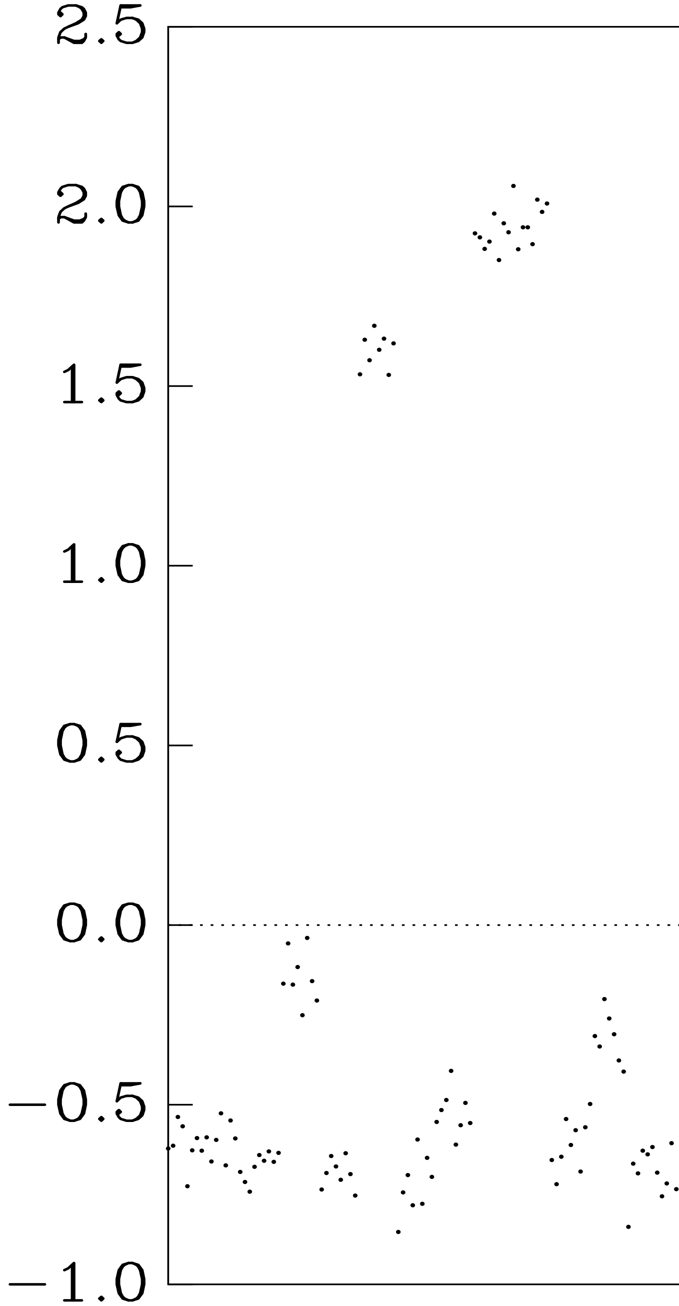
Encryption time (for this test code, this CPU, etc.) is maximized when $k[13] \oplus n[13] = 8$. 3-cycle signal.



Graph has x-coordinates 0 through 255.

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.

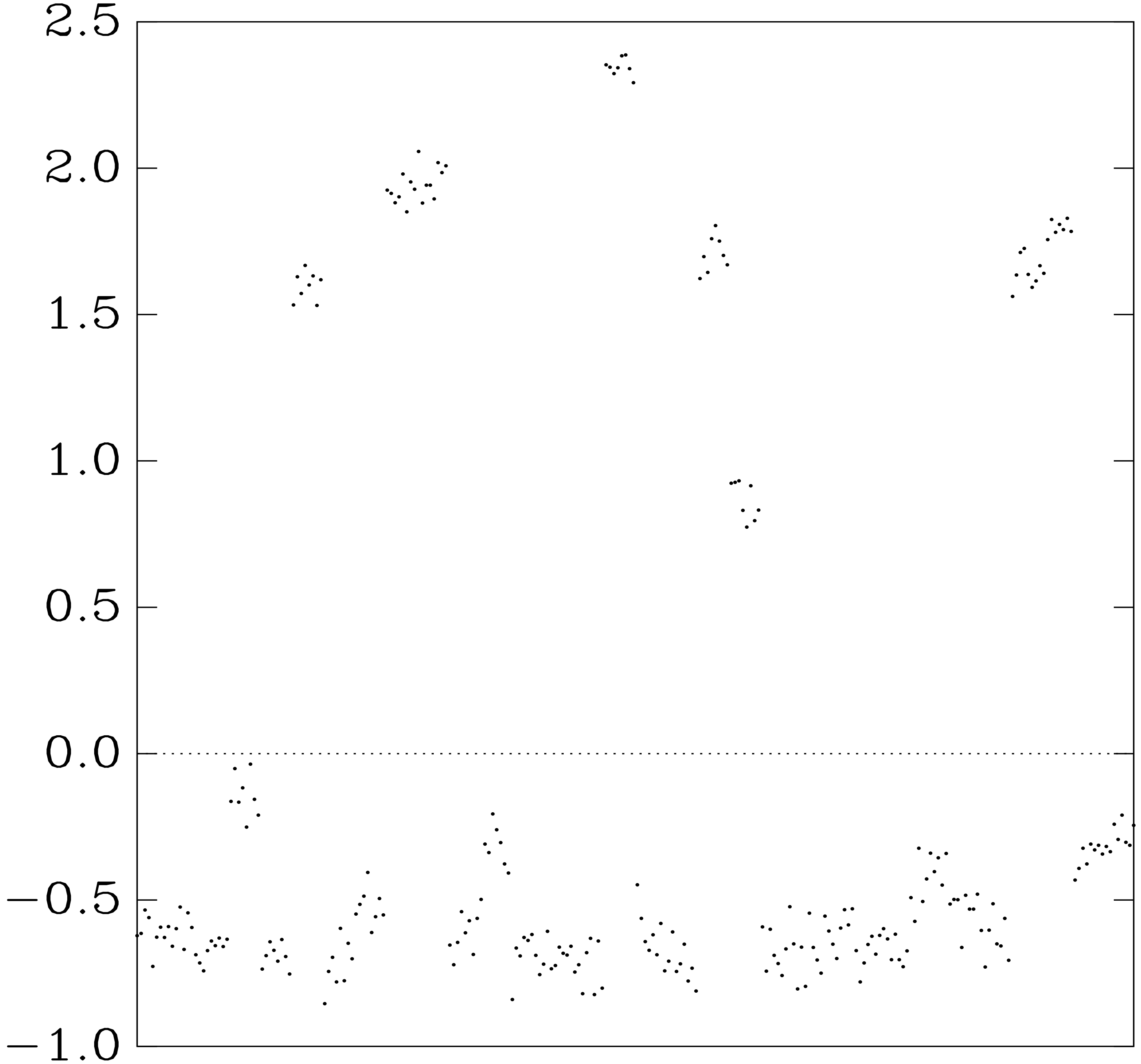


Graph for $k[5] \oplus$

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.

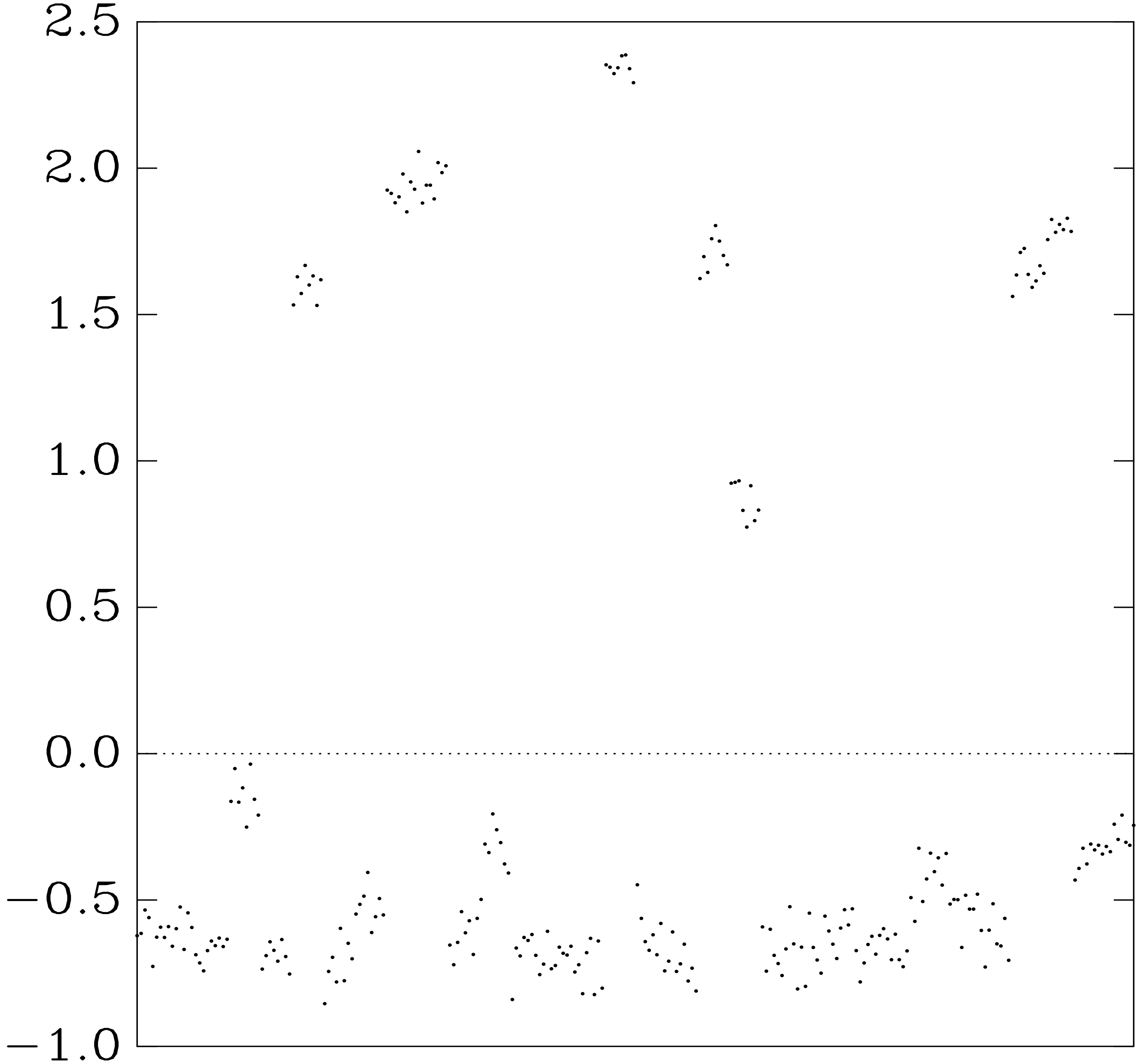


Graph for $k[5] \oplus n[5]$.

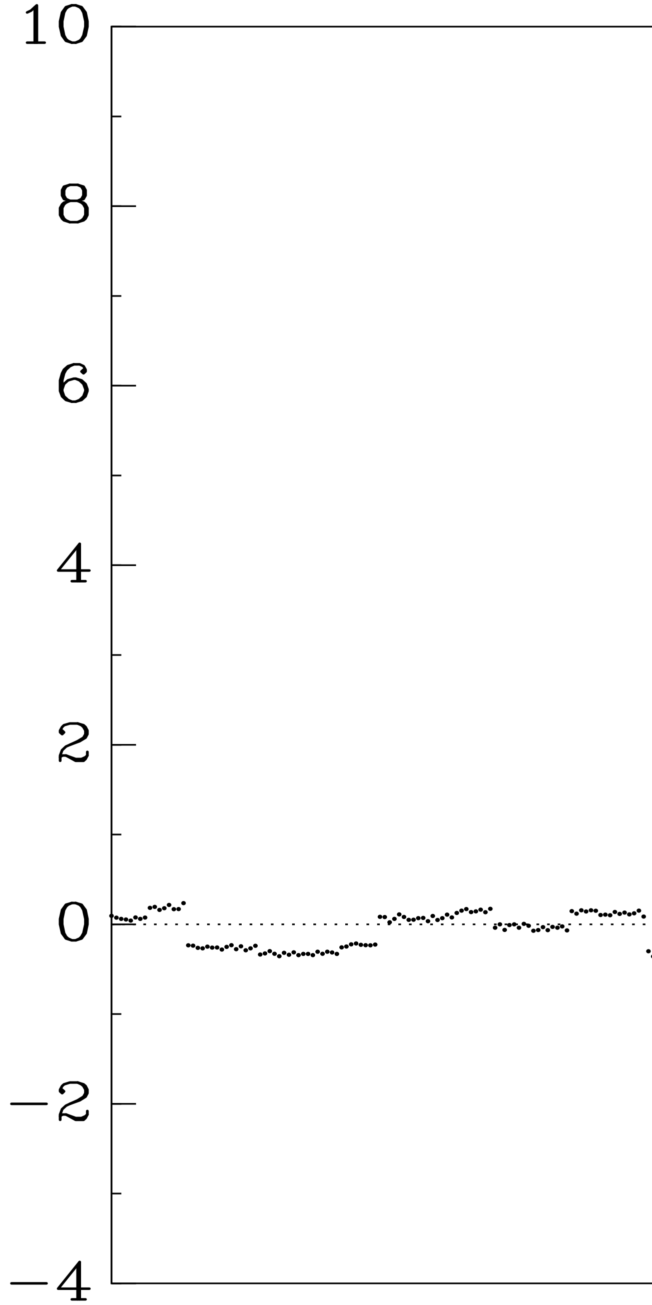
rdinates

erage cycles
m plaintext
$3] = x$,
ycles to encrypt
om plaintext.

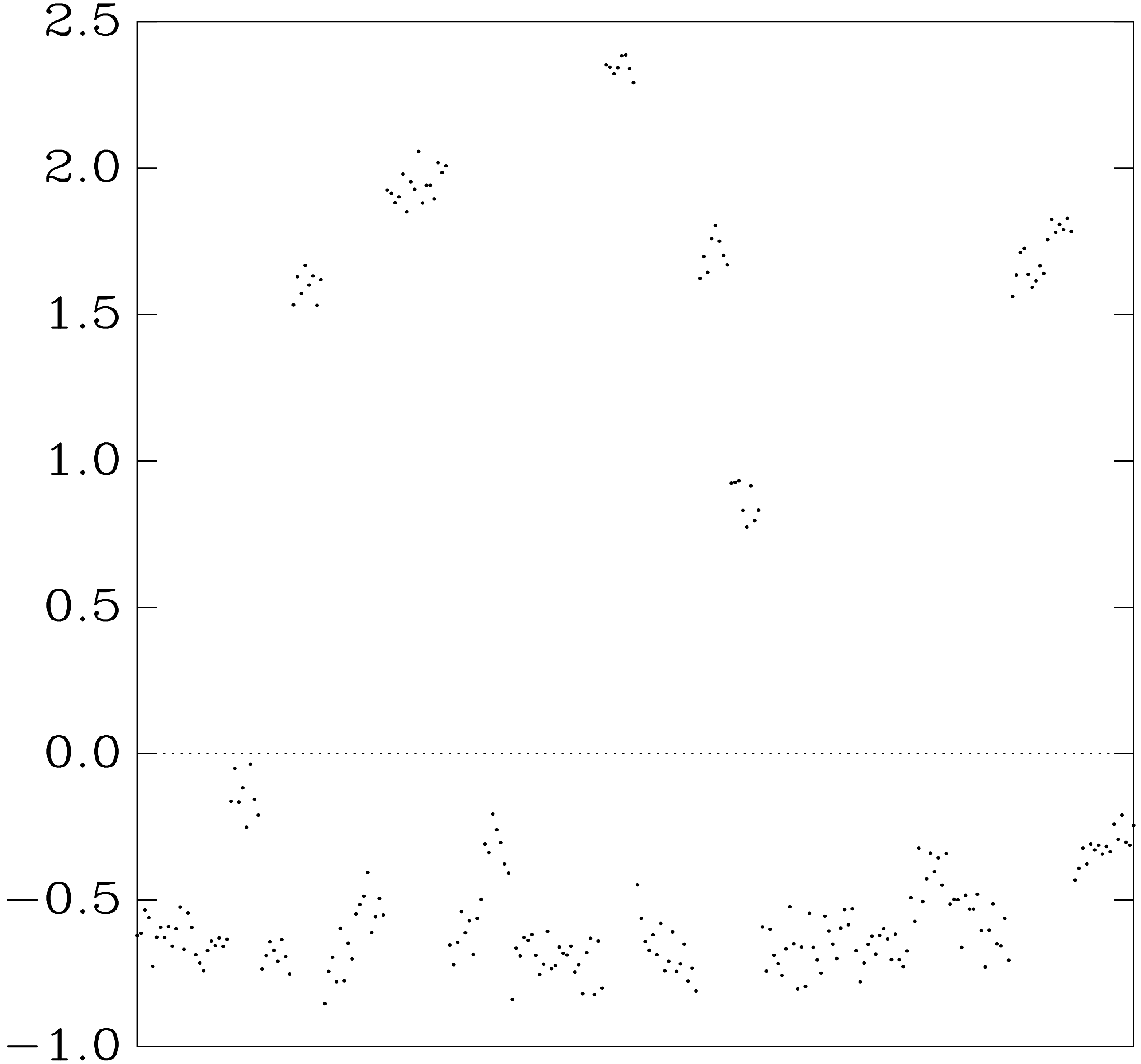
(for this test
etc.)
en
8.



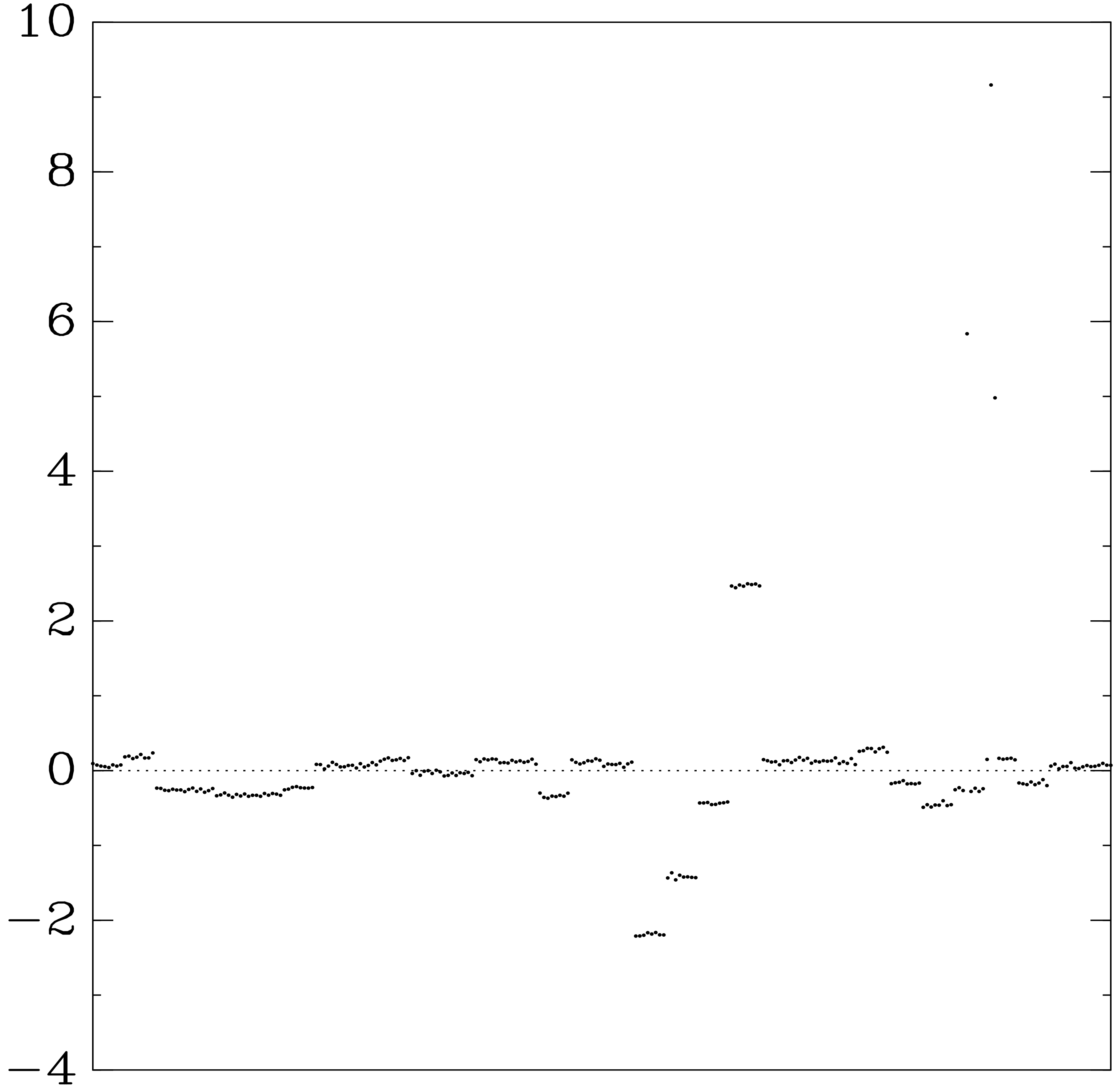
Graph for $k[5] \oplus n[5]$.



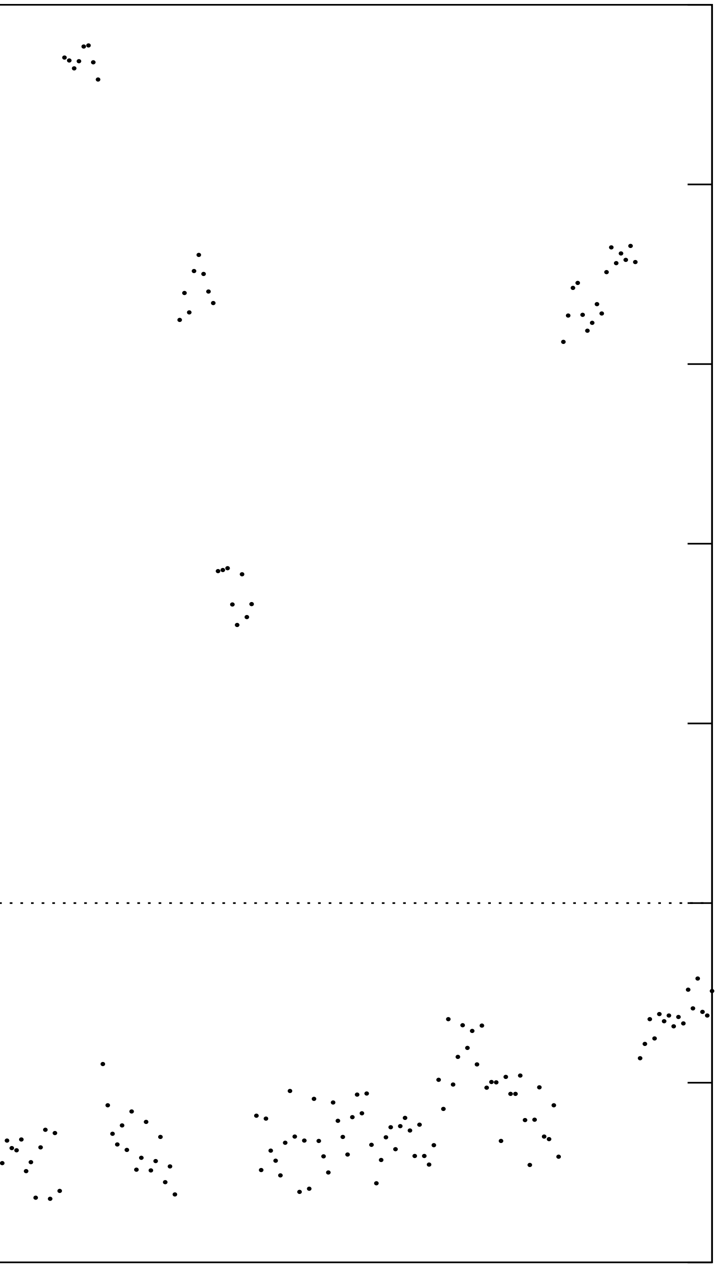
This graph has m
presumably L1 ca



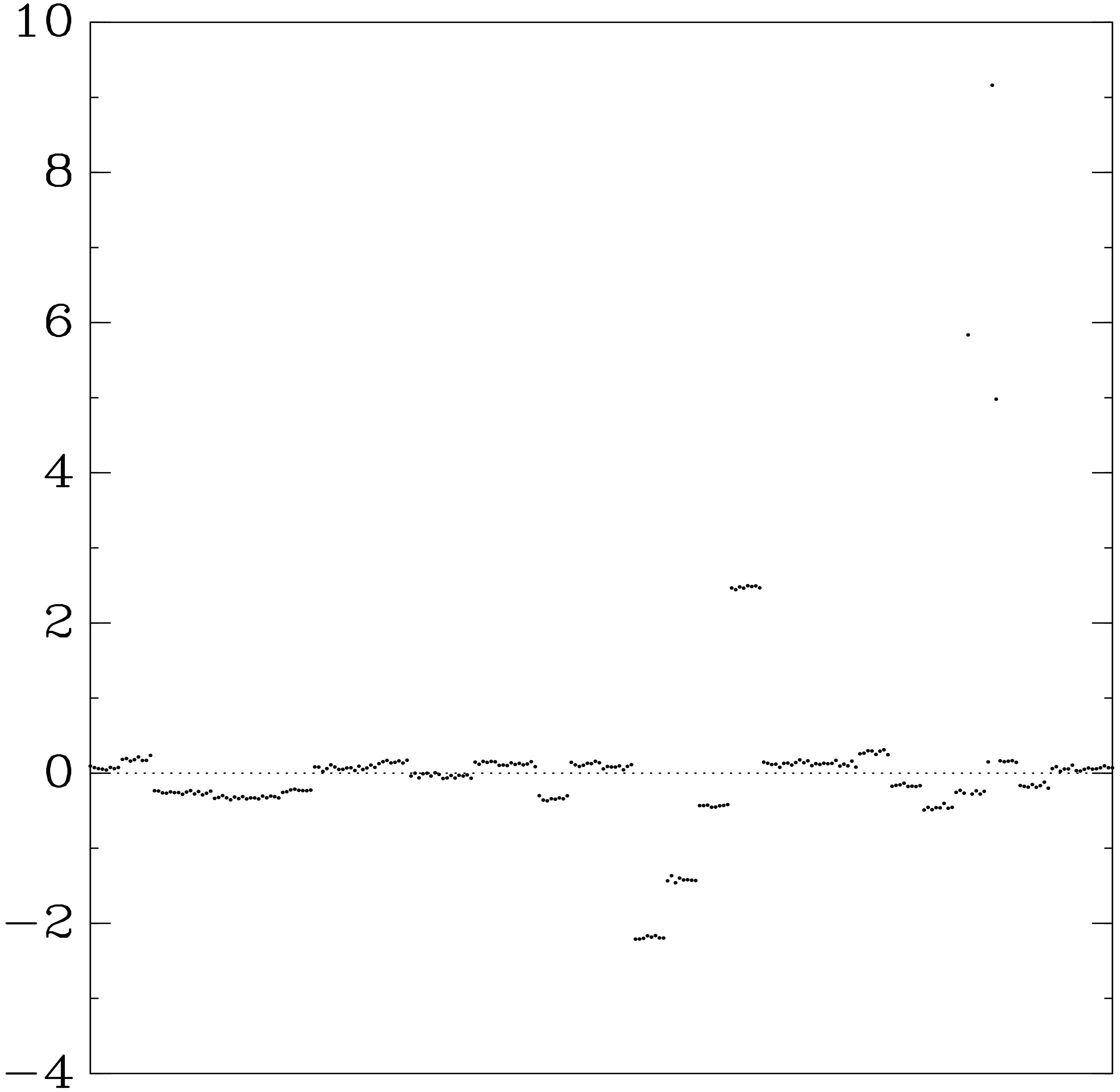
Graph for $k[5] \oplus n[5]$.



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



$n[5]$.

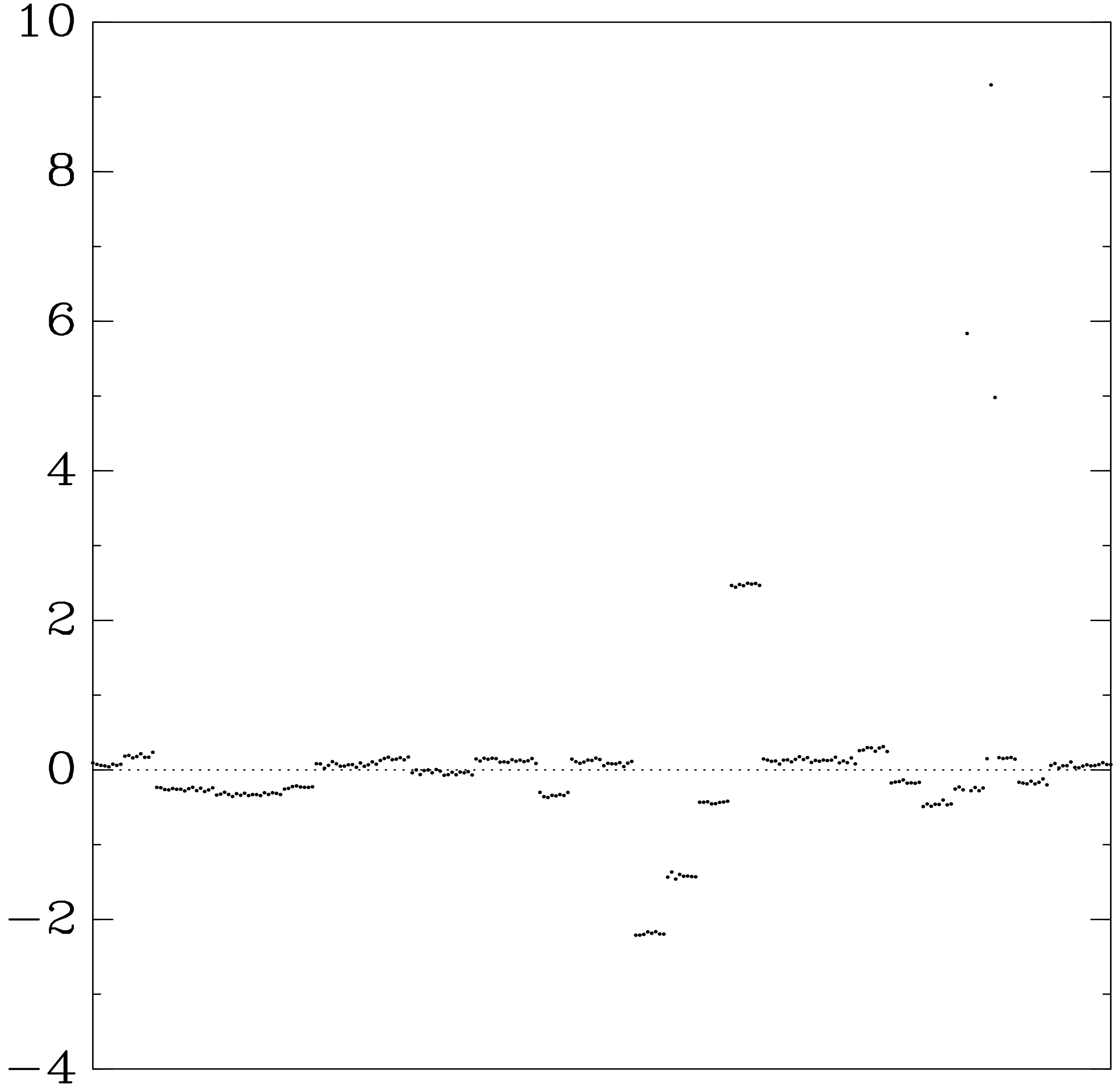


This graph has much larger max, presumably L1 cache miss.

2006: Mironov re
only attack dedu
few thousand cip
Focus on last rou
of AES computat

Obvious next res
Understand netw
Can we see ≈ 1-c
from (e.g.) media
10^6 packet timin

Would be anothe
I'm not doing thi
feel free to jump

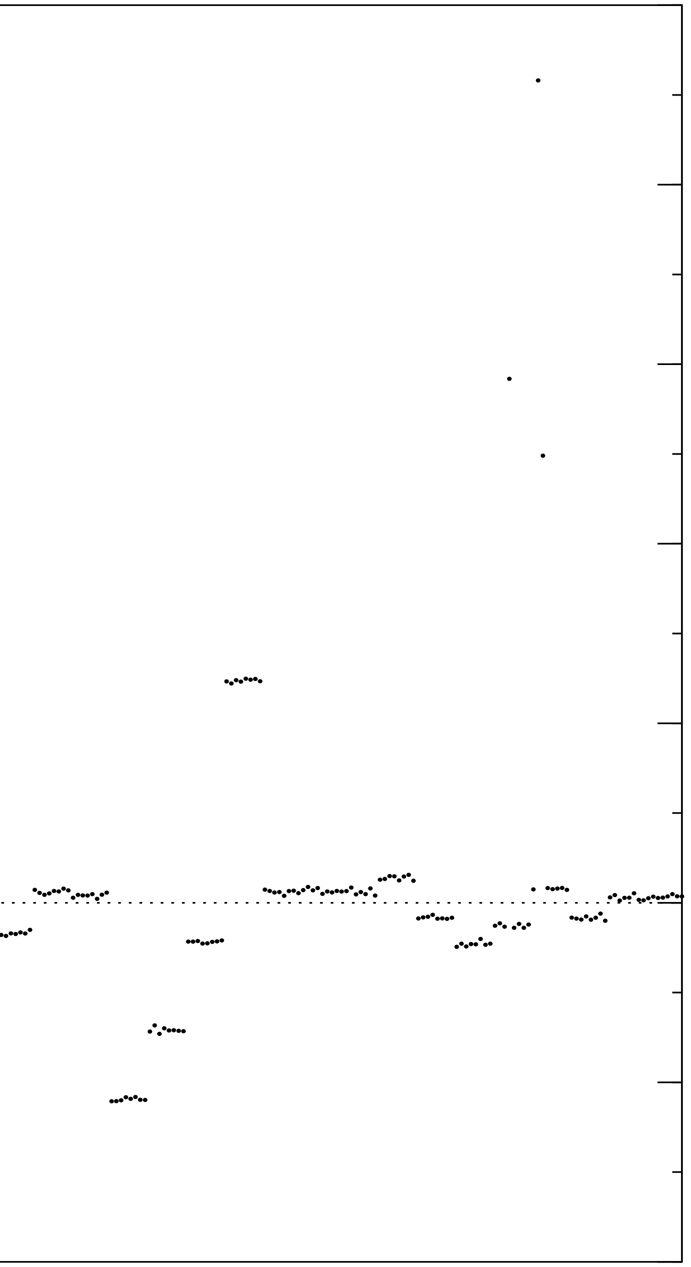


This graph has much larger max, presumably L1 cache miss.

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.



nuch larger max,
ache 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 $\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 c

Primary goal of c
Continued emplo
cryptographers.

How to achieve t

Example: Use 51
Oops, broken? U
Oops, broken? U

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.

eports ciphertext-
cing key after a
phertexts.
und
tion.

earch step:
ork noise!
cycle signals
an of
gs?

er nice paper.
is;
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 tha
until they've bee
in the New York

For timing attack
hasn't been dem
assume it doesn'

Don't use obviou
software such as

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.

cryptography

cryptography:
yment for

this?

2-bit RSA.
Use 768-bit RSA.
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 use crypto

Build complex m
cryptographic sys
Don't communic
between people
different layers.

e.g. Most CPU
thoroughly docu

Challenge: Mark
with a variable-ti

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.