Cryptographic software engineering, part 1

Daniel J. Bernstein

This is easy, right?

1. Take general principles of software engineering.
2. Apply principles to crypto.

Let's try some examples . . .

1972 Parnas "On the criteria to be used in decomposing systems into modules":

"We propose instead that one begins with a list of difficult design decisions or design decisions which are likely to change. Each module is then designed to hide such a decision from the others."

e.g. If number of cipher rounds is properly modularized as

```
#define ROUNDS 20
```

then it is easy to change.

raphic
engineering,

. Bernstein

---

easy, right?

general principles
ftware engineering.
y principles to crypto.

y some examples . . .

1972 Parnas “On the criteria to be used in decomposing systems into modules”:

“We propose instead that one begins with a list of difficult design decisions or design decisions which are likely to change. Each module is then designed to hide such a decision from the others.”

e.g. If number of cipher rounds is properly modularized as

```
#define ROUNDS 20
```

then it is easy to change.

Another
of softw
Make th
and the

ng,

n

?

rinciples
gineering.
s to crypto.

mples . . .

1972 Parnas "On the criteria to be used in decomposing systems into modules":

"We propose instead that one begins with a list of difficult design decisions or design decisions which are likely to change. Each module is then designed to hide such a decision from the others."

e.g. If number of cipher rounds is properly modularized as

```
#define ROUNDS 20
```

then it is easy to change.

Another general pr
of software engine
Make the right thi
and the wrong thi

1

1972 Parnas "On the criteria to be used in decomposing systems into modules":

"We propose instead that one begins with a list of difficult design decisions or design decisions which are likely to change. Each module is then designed to hide such a decision from the others."

e.g. If number of cipher rounds is properly modularized as

```
#define ROUNDS 20
```

then it is easy to change.

2

Another general principle of software engineering: Make the right thing simple and the wrong thing comple

1972 Parnas “On the criteria to be used in decomposing systems into modules”:

“We propose instead that one begins with a list of difficult design decisions or design decisions which are likely to change. Each module is then designed to hide such a decision from the others.”

e.g. If number of cipher rounds is properly modularized as

```
#define ROUNDS 20
```

then it is easy to change.

Another general principle of software engineering: Make the right thing simple and the wrong thing complex.

1972 Parnas "On the criteria to be used in decomposing systems into modules":

"We propose instead that one begins with a list of difficult design decisions or design decisions which are likely to change. Each module is then designed to hide such a decision from the others."

e.g. If number of cipher rounds is properly modularized as

```
#define ROUNDS 20
```

then it is easy to change.

Another general principle of software engineering: Make the right thing simple and the wrong thing complex.

e.g. Make it difficult to ignore invalid authenticators.

1972 Parnas "On the criteria to be used in decomposing systems into modules":

"We propose instead that one begins with a list of difficult design decisions or design decisions which are likely to change. Each module is then designed to hide such a decision from the others."

e.g. If number of cipher rounds is properly modularized as

```
#define ROUNDS 20
```

then it is easy to change.

Another general principle of software engineering: Make the right thing simple and the wrong thing complex.

e.g. Make it difficult to ignore invalid authenticators.

Do not design APIs like this: "The sample code used in this manual omits the checking of status values for clarity, but when using cryptlib you should check return values, particularly for critical functions . . . "

rnas “On the criteria
ed in decomposing
into modules”:

pose instead that
ins with a list of
design decisions or
ecisions which are
change. Each module
designed to hide such
on from the others.”

umber of cipher rounds
rly modularized as
`ROUNDS 20`
s easy to change.

Another general principle of software engineering: Make the right thing simple and the wrong thing complex.

e.g. Make it difficult to ignore invalid authenticators.

Do not design APIs like this: “The sample code used in this manual omits the checking of status values for clarity, but when using cryptlib you should check return values, particularly for critical functions . . . ”

Not so e

1970s: –
compare
against s
one char
stopping

- `AAAAA`
- `FAAAA`
- `FRAAA`

the criteria
omposing
ules":

ead that
list of
cisions or
hich are
Each module
o hide such
e others."

cipher rounds
rized as
0
change.

Another general principle
of software engineering:
Make the right thing simple
and the wrong thing complex.

e.g. Make it difficult to
ignore invalid authenticators.

Do not design APIs like this:
"The sample code used in
this manual omits the checking
of status values for clarity, but
when using cryptlib you should
check return values, particularly
for critical functions . . . "

Not so easy: Timi

1970s: TENEX op
compares user-sup
against secret pass
one character at a
stopping at first d

- `AAAAAA` vs. `FRIE`
- `FAAAAA` vs. `FRIE`
- `FRAAAA` vs. `FRIE`

a

ule

h

nds

Another general principle of software engineering: Make the right thing simple and the wrong thing complex.

e.g. Make it difficult to ignore invalid authenticators.

Do not design APIs like this: “The sample code used in this manual omits the checking of status values for clarity, but when using cryptlib you should check return values, particularly for critical functions . . . ”

Not so easy: Timing attacks

1970s: TENEX operating sy
compares user-supplied strin
against secret password
one character at a time,
stopping at first difference:

- `AAAAAA` vs. `FRIEND`: stop
- `FAAAAA` vs. `FRIEND`: stop
- `FRAAAA` vs. `FRIEND`: stop

Another general principle of software engineering: Make the right thing simple and the wrong thing complex.

e.g. Make it difficult to ignore invalid authenticators.

Do not design APIs like this: “The sample code used in this manual omits the checking of status values for clarity, but when using cryptlib you should check return values, particularly for critical functions . . . ”

Not so easy: Timing attacks

1970s: TENEX operating system compares user-supplied string against secret password one character at a time, stopping at first difference:

- `AAAAAA` vs. `FRIEND`: stop at 1.
- `FAAAAA` vs. `FRIEND`: stop at 2.
- `FRAAAA` vs. `FRIEND`: stop at 3.

Another general principle
of software engineering:
Make the right thing simple
and the wrong thing complex.

e.g. Make it difficult to
ignore invalid authenticators.

Do not design APIs like this:
"The sample code used in
this manual omits the checking
of status values for clarity, but
when using cryptlib you should
check return values, particularly
for critical functions . . . "

Not so easy: Timing attacks

1970s: TENEX operating system
compares user-supplied string
against secret password
one character at a time,
stopping at first difference:

- `AAAAAA` vs. `FRIEND`: stop at 1.
- `FAAAAA` vs. `FRIEND`: stop at 2.
- `FRAAAA` vs. `FRIEND`: stop at 3.

Attacker sees comparison time,
deduces position of difference.
A few hundred tries
reveal secret password.

general principle
are engineering:
e right thing simple
wrong thing complex.

ke it difficult to
nvalid authenticators.

design APIs like this:
mple code used in
nual omits the checking
s values for clarity, but
ing cryptlib you should
turn values, particularly
cal functions . . . ”

Not so easy: Timing attacks

1970s: TENEX operating system
compares user-supplied string
against secret password
one character at a time,
stopping at first difference:

- `AAAAAA` vs. `FRIEND`: stop at 1.
- `FAAAAA` vs. `FRIEND`: stop at 2.
- `FRAAAA` vs. `FRIEND`: stop at 3.

Attacker sees comparison time,
deduces position of difference.
A few hundred tries
reveal secret password.

How typ
16-byte

```
for (
  if
retur
```

rinciple
ering:
ng simple
ng complex.

ult to
henticators.

ls like this:
used in
the checking
r clarity, but
b you should
es, particularly
ns . . .”

Not so easy: Timing attacks

1970s: TENEX operating system compares user-supplied string against secret password one character at a time, stopping at first difference:

- `AAAAAA` vs. `FRIEND`: stop at 1.
- `FAAAAA` vs. `FRIEND`: stop at 2.
- `FRAAAA` vs. `FRIEND`: stop at 3.

Attacker sees comparison time, deduces position of difference. A few hundred tries reveal secret password.

How typical softwa
16-byte authentica

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

x.

s.

:

ing
ut
uld
arly

Not so easy: Timing attacks

1970s: TENEX operating system compares user-supplied string against secret password one character at a time, stopping at first difference:

- `AAAAAA` vs. `FRIEND`: stop at 1.
- `FAAAAA` vs. `FRIEND`: stop at 2.
- `FRAAAA` vs. `FRIEND`: stop at 3.

Attacker sees comparison time, deduces position of difference. A few hundred tries reveal secret password.

How typical software checks
16-byte authenticator:

```
for (i = 0;i < 16;++i)
  if (x[i] != y[i]) ret
return 1;
```

Not so easy: Timing attacks

1970s: TENEX operating system compares user-supplied string against secret password one character at a time, stopping at first difference:

- `AAAAAA` vs. `FRIEND`: stop at 1.
- `FAAAAA` vs. `FRIEND`: stop at 2.
- `FRAAAA` vs. `FRIEND`: stop at 3.

Attacker sees comparison time, deduces position of difference. A few hundred tries reveal secret password.

How typical software checks 16-byte authenticator:

```
for (i = 0;i < 16;++i)
  if (x[i] != y[i]) return 0;
return 1;
```

Not so easy: Timing attacks

1970s: TENEX operating system compares user-supplied string against secret password one character at a time, stopping at first difference:

- `AAAAAA` vs. `FRIEND`: stop at 1.
- `FAAAAA` vs. `FRIEND`: stop at 2.
- `FRAAAA` vs. `FRIEND`: stop at 3.

Attacker sees comparison time, deduces position of difference. A few hundred tries reveal secret password.

How typical software checks 16-byte authenticator:

```
for (i = 0;i < 16;++i)
  if (x[i] != y[i]) return 0;
return 1;
```

Fix, eliminating information flow from secrets to timings:

```
diff = 0;
for (i = 0;i < 16;++i)
  diff |= x[i] ^ y[i];
return 1 & ((diff-1) >> 8);
```

Notice that the language makes the wrong thing simple and the right thing complex.

easy: Timing attacks

TENEX operating system
es user-supplied string
secret password
racter at a time,
g at first difference:

A vs. FRIEND: stop at 1.
A vs. FRIEND: stop at 2.
A vs. FRIEND: stop at 3.

r sees comparison time,
position of difference.
undred tries
ecret password.

How typical software checks
16-byte authenticator:

```
for (i = 0;i < 16;++i)
  if (x[i] != y[i]) return 0;
return 1;
```

Fix, eliminating information flow
from secrets to timings:

```
diff = 0;
for (i = 0;i < 16;++i)
  diff |= x[i] ^ y[i];
return 1 & ((diff-1) >> 8);
```

Notice that the language
makes the wrong thing simple
and the right thing complex.

Languag
“right” i

So mista

ng attacks

perating system
plied string
sword
time,
ifference:

END: stop at 1.
END: stop at 2.
END: stop at 3.

parison time,
of difference.
es
word.

How typical software checks
16-byte authenticator:

```
for (i = 0;i < 16;++i)
  if (x[i] != y[i]) return 0;
return 1;
```

Fix, eliminating information flow
from secrets to timings:

```
diff = 0;
for (i = 0;i < 16;++i)
  diff |= x[i] ^ y[i];
return 1 & ((diff-1) >> 8);
```

Notice that the language
makes the wrong thing simple
and the right thing complex.

Language designer
"right" is too wea

So mistakes contir

s

ystem

g

at 1.

at 2.

at 3.

me,

ce.

How typical software checks 16-byte authenticator:

```
for (i = 0;i < 16;++i)
  if (x[i] != y[i]) return 0;
return 1;
```

Fix, eliminating information flow from secrets to timings:

```
diff = 0;
for (i = 0;i < 16;++i)
  diff |= x[i] ^ y[i];
return 1 & ((diff-1) >> 8);
```

Notice that the language makes the wrong thing simple and the right thing complex.

Language designer's notion

"right" is too weak for secu

So mistakes continue to hap

How typical software checks 16-byte authenticator:

```
for (i = 0;i < 16;++i)
  if (x[i] != y[i]) return 0;
return 1;
```

Fix, eliminating information flow from secrets to timings:

```
diff = 0;
for (i = 0;i < 16;++i)
  diff |= x[i] ^ y[i];
return 1 & ((diff-1) >> 8);
```

Notice that the language makes the wrong thing simple and the right thing complex.

Language designer's notion of "right" is too weak for security.

So mistakes continue to happen.

How typical software checks
16-byte authenticator:

```
for (i = 0;i < 16;++i)
  if (x[i] != y[i]) return 0;
return 1;
```

Fix, eliminating information flow
from secrets to timings:

```
diff = 0;
for (i = 0;i < 16;++i)
  diff |= x[i] ^ y[i];
return 1 & ((diff-1) >> 8);
```

Notice that the language
makes the wrong thing simple
and the right thing complex.

Language designer's notion of
"right" is too weak for security.

So mistakes continue to happen.

One of many examples,
part of the reference software for
one of the CAESAR candidates:

```
/* compare the tag */
int i;
for(i = 0;i < CRYPTO_ABYTES;i++)
  if(tag[i] != c[(*mlen) + i]){
    return RETURN_TAG_NO_MATCH;
  }
return RETURN_SUCCESS;
```

ical software checks
authenticator:

```
i = 0;i < 16;++i)
(x[i] != y[i]) return 0;
n 1;
```

inating information flow
rets to timings:

```
= 0;
i = 0;i < 16;++i)
f |= x[i] ^ y[i];
n 1 & ((diff-1) >> 8);
```

hat the language
ne wrong thing simple
right thing complex.

Language designer's notion of "right" is too weak for security.

So mistakes continue to happen.

One of many examples, part of the reference software for one of the CAESAR candidates:

```
/* compare the tag */
int i;
for(i = 0;i < CRYPTO_ABYTES;i++)
  if(tag[i] != c[(*mlen) + i]){
    return RETURN_TAG_NO_MATCH;
  }
return RETURN_SUCCESS;
```

Do timin

Objectio

are checks

ator:

```
 16;++i)
y[i]) return 0;
```

formation flow

nings:

```
 16;++i)
 ^ y[i];
iff-1) >> 8);
```

nguage

thing simple

g complex.

Language designer's notion of "right" is too weak for security.

So mistakes continue to happen.

One of many examples, part of the reference software for one of the CAESAR candidates:

```
/* compare the tag */
int i;
for(i = 0;i < CRYPTO_ABYTES;i++)
  if(tag[i] != c[(*mlen) + i]){
    return RETURN_TAG_NO_MATCH;
  }
return RETURN_SUCCESS;
```


Do timing attacks

Objection: "Timin

urn 0;

flow

8);

le

Language designer's notion of "right" is too weak for security.

So mistakes continue to happen.

One of many examples, part of the reference software for one of the CAESAR candidates:

```
/* compare the tag */
int i;
for(i = 0;i < CRYPTO_ABYTES;i++)
  if(tag[i] != c[(*mlen) + i]){
    return RETURN_TAG_NO_MATCH;
  }
return RETURN_SUCCESS;
```


Do timing attacks really wor

Objection: "Timings are noi

Language designer's notion of "right" is too weak for security.

So mistakes continue to happen.

One of many examples, part of the reference software for one of the CAESAR candidates:

```
/* compare the tag */
int i;
for(i = 0;i < CRYPTO_ABYTES;i++)
  if(tag[i] != c[(*mlen) + i]){
    return RETURN_TAG_NO_MATCH;
  }
return RETURN_SUCCESS;
```


Do timing attacks really work?

Objection: "Timings are noisy!"

Language designer's notion of "right" is too weak for security.

So mistakes continue to happen.

One of many examples, part of the reference software for one of the CAESAR candidates:

```
/* compare the tag */
int i;
for(i = 0;i < CRYPTO_ABYTES;i++)
  if(tag[i] != c[(*mlen) + i]){
    return RETURN_TAG_NO_MATCH;
  }
return RETURN_SUCCESS;
```


Do timing attacks really work?

Objection: "Timings are noisy!"

Answer #1:
Does noise stop *all* attacks?
To guarantee security, defender must block *all* information flow.

Language designer's notion of "right" is too weak for security.

So mistakes continue to happen.

One of many examples, part of the reference software for one of the CAESAR candidates:

```
/* compare the tag */
int i;
for(i = 0;i < CRYPTO_ABYTES;i++)
  if(tag[i] != c[(*mlen) + i]){
    return RETURN_TAG_NO_MATCH;
  }
return RETURN_SUCCESS;
```


Do timing attacks really work?

Objection: "Timings are noisy!"

Answer #1:
Does noise stop *all* attacks?
To guarantee security, defender must block *all* information flow.

Answer #2: Attacker uses statistics to eliminate noise.

Language designer's notion of "right" is too weak for security.

So mistakes continue to happen.

One of many examples, part of the reference software for one of the CAESAR candidates:

```
/* compare the tag */
int i;
for(i = 0;i < CRYPTO_ABYTES;i++)
  if(tag[i] != c[(*mlen) + i]){
    return RETURN_TAG_NO_MATCH;
  }
return RETURN_SUCCESS;
```


Do timing attacks really work?

Objection: "Timings are noisy!"

Answer #1:
Does noise stop *all* attacks?
To guarantee security, defender must block *all* information flow.

Answer #2: Attacker uses statistics to eliminate noise.

Answer #3, what the 1970s attackers actually did:
Cross page boundary,
inducing page faults,
to amplify timing signal.

ge designer's notion of
is too weak for security.

akes continue to happen.

many examples,
the reference software for
he CAESAR candidates:

```
are the tag */

 0;i < CRYPTO_ABYTES;i++)
g[i] != c[(*mlen) + i]){
urn RETURN_TAG_NO_MATCH;

RETURN_SUCCESS;
```


Do timing attacks really work?

Objection: "Timings are noisy!"

Answer #1:
Does noise stop *all* attacks?
To guarantee security, defender
must block *all* information flow.

Answer #2: Attacker uses
statistics to eliminate noise.

Answer #3, what the
1970s attackers actually did:
Cross page boundary,
inducing page faults,
to amplify timing signal.

Defende

Some of

1996 Ko
attacks

Briefly n
Kocher a
Schneier
secret ar
affect tir

2002 Pa
Suzaki–S
timing a

’s notion of
k for security.

nue to happen.

nples,
ce software for
R candidates:

```
ag */

YPTO_ABYTES;i++)
[(*mlen) + i]){
N_TAG_NO_MATCH;

CCESS;
```


Do timing attacks really work?

Objection: “Timings are noisy!”

Answer #1:
Does noise stop *all* attacks?
To guarantee security, defender must block *all* information flow.

Answer #2: Attacker uses statistics to eliminate noise.

Answer #3, what the 1970s attackers actually did:
Cross page boundary, inducing page faults, to amplify timing signal.

Defenders don’t le

Some of the litera

1996 Kocher point
attacks on cryptog

Briefly mentioned
Kocher and by 199
Schneier–Wagner–
secret array indice
affect timing via c

2002 Page, 2003 T
Suzaki–Shigeri–Mi
timing attacks on

of
rity.

pen.

e for
tes:

```
ES;i++)
 + i]){
MATCH;
```


Do timing attacks really work?

Objection: "Timings are noisy!"

Answer #1:
Does noise stop *all* attacks?
To guarantee security, defender must block *all* information flow.

Answer #2: Attacker uses statistics to eliminate noise.

Answer #3, what the 1970s attackers actually did:
Cross page boundary, inducing page faults, to amplify timing signal.

Defenders don't learn

Some of the literature:

1996 Kocher pointed out tir
attacks on cryptographic key

Briefly mentioned by
Kocher and by 1998 Kelsey–
Schneier–Wagner–Hall:
secret array indices can
affect timing via cache misse

2002 Page, 2003 Tsunoo–Sa
Suzaki–Shigeri–Miyauchi:
timing attacks on DES.

Do timing attacks really work?

Objection: "Timings are noisy!"

Answer #1:
Does noise stop *all* attacks?
To guarantee security, defender must block *all* information flow.

Answer #2: Attacker uses statistics to eliminate noise.

Answer #3, what the 1970s attackers actually did:
Cross page boundary, inducing page faults, to amplify timing signal.

Defenders don't learn

Some of the literature:

1996 Kocher pointed out timing attacks on cryptographic key bits.

Briefly mentioned by Kocher and by 1998 Kelsey–Schneier–Wagner–Hall: secret array indices can affect timing via cache misses.

2002 Page, 2003 Tsunoo–Saito–Suzaki–Shigeri–Miyauchi: timing attacks on DES.

ng attacks really work?

on: "Timings are noisy!"

#1:
ise stop *all* attacks?
antee security, defender
ock *all* information flow.

#2: Attacker uses
s to eliminate noise.

#3, what the
ttackers actually did:
ge boundary,
page faults,
fy timing signal.

Defenders don't learn

Some of the literature:

1996 Kocher pointed out timing attacks on cryptographic key bits.

Briefly mentioned by Kocher and by 1998 Kelsey–Schneier–Wagner–Hall: secret array indices can affect timing via cache misses.

2002 Page, 2003 Tsunoo–Saito–Suzaki–Shigeri–Miyauchi: timing attacks on DES.

"Guaran
load ent

really work?

ngs are noisy!"

ll attacks?
rity, defender
ormation flow.

cker uses
ate noise.

the
ctually did:
ary,
ts,
signal.

Defenders don't learn

Some of the literature:

1996 Kocher pointed out timing attacks on cryptographic key bits.

Briefly mentioned by Kocher and by 1998 Kelsey–Schneier–Wagner–Hall: secret array indices can affect timing via cache misses.

2002 Page, 2003 Tsunoo–Saito–Suzaki–Shigeri–Miyauchi: timing attacks on DES.

"Guaranteed" cou
load entire table ir

rk?

sy!”

der
low.

:

Defenders don’t learn

Some of the literature:

1996 Kocher pointed out timing attacks on cryptographic key bits.

Briefly mentioned by Kocher and by 1998 Kelsey–Schneier–Wagner–Hall: secret array indices can affect timing via cache misses.

2002 Page, 2003 Tsunoo–Saito–Suzaki–Shigeri–Miyauchi: timing attacks on DES.

“Guaranteed” countermeasu
load entire table into cache.

Defenders don't learn

Some of the literature:

1996 Kocher pointed out timing attacks on cryptographic key bits.

Briefly mentioned by Kocher and by 1998 Kelsey–Schneier–Wagner–Hall: secret array indices can affect timing via cache misses.

2002 Page, 2003 Tsunoo–Saito–Suzaki–Shigeri–Miyauchi: timing attacks on DES.

"Guaranteed" countermeasure: load entire table into cache.

Defenders don't learn

Some of the literature:

1996 Kocher pointed out timing attacks on cryptographic key bits.

Briefly mentioned by Kocher and by 1998 Kelsey–Schneier–Wagner–Hall: secret array indices can affect timing via cache misses.

2002 Page, 2003 Tsunoo–Saito–Suzaki–Shigeri–Miyauchi: timing attacks on DES.

"Guaranteed" countermeasure: load entire table into cache.

2004.11/2005.04 Bernstein: Timing attacks on AES. Countermeasure isn't safe; e.g., secret array indices can affect timing via cache-bank collisions. What *is* safe: kill all data flow from secrets to array indices.

Defenders don't learn

Some of the literature:

1996 Kocher pointed out timing attacks on cryptographic key bits.

Briefly mentioned by Kocher and by 1998 Kelsey–Schneier–Wagner–Hall: secret array indices can affect timing via cache misses.

2002 Page, 2003 Tsunoo–Saito–Suzaki–Shigeri–Miyauchi: timing attacks on DES.

"Guaranteed" countermeasure: load entire table into cache.

2004.11/2005.04 Bernstein: Timing attacks on AES. Countermeasure isn't safe; e.g., secret array indices can affect timing via cache-bank collisions. What *is* safe: kill all data flow from secrets to array indices.

2005 Tromer–Osvik–Shamir: 65ms to steal Linux AES key used for hard-disk encryption.

rs don't learn

the literature:

cher pointed out timing
on cryptographic key bits.

nentioned by
and by 1998 Kelsey–
–Wagner–Hall:
rray indices can
ming via cache misses.

ge, 2003 Tsunoo–Saito–
Shigeri–Miyauchi:
ttacks on DES.

"Guaranteed" countermeasure:
load entire table into cache.

2004.11/2005.04 Bernstein:
Timing attacks on AES.
Countermeasure isn't safe;
e.g., secret array indices can affect
timing via cache-bank collisions.
What *is* safe: kill all data flow
from secrets to array indices.

2005 Tromer–Osvik–Shamir:
65ms to steal Linux AES key
used for hard-disk encryption.

Intel rec
OpenSS
counterr
from kno

earn

ture:

ted out timing
graphic key bits.

by
98 Kelsey–
-Hall:
s can
ache misses.

Tsunoo–Saito–
Tyauchi:
DES.

"Guaranteed" countermeasure:
load entire table into cache.

2004.11/2005.04 Bernstein:
Timing attacks on AES.
Countermeasure isn't safe;
e.g., secret array indices can affect
timing via cache-bank collisions.
What *is* safe: kill all data flow
from secrets to array indices.

2005 Tromer–Osvik–Shamir:
65ms to steal Linux AES key
used for hard-disk encryption.

Intel recommends,
OpenSSL integrat
countermeasure: a
from known *lines*

ming
y bits.
–
es.
aito–

“Guaranteed” countermeasure: load entire table into cache.

2004.11/2005.04 Bernstein: Timing attacks on AES. Countermeasure isn’t safe; e.g., secret array indices can affect timing via cache-bank collisions. What *is* safe: kill all data flow from secrets to array indices.

2005 Tromer–Osvik–Shamir: 65ms to steal Linux AES key used for hard-disk encryption.

Intel recommends, and OpenSSL integrates, cheape
countermeasure: always loac
from known *lines* of cache.

“Guaranteed” countermeasure: load entire table into cache.

2004.11/2005.04 Bernstein: Timing attacks on AES. Countermeasure isn’t safe; e.g., secret array indices can affect timing via cache-bank collisions. What *is* safe: kill all data flow from secrets to array indices.

2005 Tromer–Osvik–Shamir: 65ms to steal Linux AES key used for hard-disk encryption.

Intel recommends, and OpenSSL integrates, cheaper countermeasure: always loading from known *lines* of cache.

“Guaranteed” countermeasure: load entire table into cache.

2004.11/2005.04 Bernstein: Timing attacks on AES. Countermeasure isn’t safe; e.g., secret array indices can affect timing via cache-bank collisions. What *is* safe: kill all data flow from secrets to array indices.

2005 Tromer–Osvik–Shamir: 65ms to steal Linux AES key used for hard-disk encryption.

Intel recommends, and OpenSSL integrates, cheaper countermeasure: always loading from known *lines* of cache.

2013 Bernstein–Schwabe “A word of warning”: This countermeasure isn’t safe. Variable-time lab experiment. Same issues described in 2004.

"Guaranteed" countermeasure: load entire table into cache.

2004.11/2005.04 Bernstein: Timing attacks on AES. Countermeasure isn't safe; e.g., secret array indices can affect timing via cache-bank collisions. What *is* safe: kill all data flow from secrets to array indices.

2005 Tromer–Osvik–Shamir: 65ms to steal Linux AES key used for hard-disk encryption.

Intel recommends, and OpenSSL integrates, cheaper countermeasure: always loading from known *lines* of cache.

2013 Bernstein–Schwabe "A word of warning": This countermeasure isn't safe. Variable-time lab experiment. Same issues described in 2004.

2016 Yarom–Genkin–Heninger "CacheBleed" steals RSA secret key via timings of OpenSSL.

teed" countermeasure:
ire table into cache.

/2005.04 Bernstein:
attacks on AES.
measure isn't safe;
ret array indices can affect
ia cache-bank collisions.
safe: kill all data flow
crets to array indices.

omer–Osvik–Shamir:
steal Linux AES key
hard-disk encryption.

Intel recommends, and OpenSSL integrates, cheaper countermeasure: always loading from known *lines* of cache.

2013 Bernstein–Schwabe "A word of warning": This countermeasure isn't safe. Variable-time lab experiment. Same issues described in 2004.

2016 Yarom–Genkin–Heninger "CacheBleed" steals RSA secret key via timings of OpenSSL.

2008 RF
Layer Se
Version
small tin
performa
extent o
fragmen
be large
due to t
existing
of the ti

ntermeasure:
nto cache.

Bernstein:
AES.
sn't safe;
ndices can affect
bank collisions.
all data flow
ray indices.

ik–Shamir:
ux AES key
encryption.

Intel recommends, and OpenSSL integrates, cheaper countermeasure: always loading from known *lines* of cache.

2013 Bernstein–Schwabe "A word of warning": This countermeasure isn't safe. Variable-time lab experiment. Same issues described in 2004.

2016 Yarom–Genkin–Heninger "CacheBleed" steals RSA secret key via timings of OpenSSL.

2008 RFC 5246 "T
Layer Security (TL
Version 1.2": "Th
small timing chann
performance depe
extent on the size
fragment, but it is
be large enough to
due to the large bl
existing MACs and
of the timing signa

re:

affect

ons.

ow

.

t

y

n.

Intel recommends, and OpenSSL integrates, cheaper countermeasure: always loading from known *lines* of cache.

2013 Bernstein–Schwabe "A word of warning": This countermeasure isn't safe. Variable-time lab experiment. Same issues described in 2004.

2016 Yarom–Genkin–Heninger "CacheBleed" steals RSA secret key via timings of OpenSSL.

2008 RFC 5246 "The Trans
Layer Security (TLS) Protoc
Version 1.2": "This leaves a
small timing channel, since
performance depends to som
extent on the size of the dat
fragment, but it is not belie
be large enough to be exploi
due to the large block size c
existing MACs and the smal
of the timing signal."

Intel recommends, and OpenSSL integrates, cheaper countermeasure: always loading from known *lines* of cache.

2013 Bernstein–Schwabe "A word of warning": This countermeasure isn't safe. Variable-time lab experiment. Same issues described in 2004.

2016 Yarom–Genkin–Heninger "CacheBleed" steals RSA secret key via timings of OpenSSL.

2008 RFC 5246 "The Transport Layer Security (TLS) Protocol, Version 1.2": "This leaves a small timing channel, since MAC performance depends to some extent on the size of the data fragment, but it is not believed to be large enough to be exploitable, due to the large block size of existing MACs and the small size of the timing signal."

Intel recommends, and OpenSSL integrates, cheaper countermeasure: always loading from known *lines* of cache.

2013 Bernstein–Schwabe "A word of warning": This countermeasure isn't safe. Variable-time lab experiment. Same issues described in 2004.

2016 Yarom–Genkin–Heninger "CacheBleed" steals RSA secret key via timings of OpenSSL.

2008 RFC 5246 "The Transport Layer Security (TLS) Protocol, Version 1.2": "This leaves a small timing channel, since MAC performance depends to some extent on the size of the data fragment, but it is not believed to be large enough to be exploitable, due to the large block size of existing MACs and the small size of the timing signal."

2013 AlFardan–Paterson "Lucky Thirteen: breaking the TLS and DTLS record protocols": exploit these timings; steal plaintext.

ommends, and
L integrates, cheaper
measure: always loading
own *lines* of cache.

rnstein–Schwabe
of warning":
untermeasure isn't safe.
-time lab experiment.
sues described in 2004.

rom–Genkin–Heninger
Bleed" steals RSA secret
timings of OpenSSL.

2008 RFC 5246 "The Transport Layer Security (TLS) Protocol, Version 1.2": "This leaves a small timing channel, since MAC performance depends to some extent on the size of the data fragment, but it is not believed to be large enough to be exploitable, due to the large block size of existing MACs and the small size of the timing signal."

2013 AlFardan–Paterson "Lucky Thirteen: breaking the TLS and DTLS record protocols": exploit these timings; steal plaintext.

How to

If possib
to contr

Look for
identifyi
"Divisio
when th
complet
cycles re
values o

Measure
trusting

and
es, cheaper
always loading
of cache.

chwabe
g”:
ure isn’t safe.
experiment.
bed in 2004.

in–Heninger
als RSA secret
OpenSSL.

2008 RFC 5246 “The Transport Layer Security (TLS) Protocol, Version 1.2”: “This leaves a small timing channel, since MAC performance depends to some extent on the size of the data fragment, but it is not believed to be large enough to be exploitable, due to the large block size of existing MACs and the small size of the timing signal.”

2013 AlFardan–Paterson “Lucky Thirteen: breaking the TLS and DTLS record protocols”: exploit these timings; steal plaintext.

How to write cons

If possible, write c
to control instruct

Look for documen
identifying variabil
“Division operatio
when the divide op
completes, with th
cycles required dep
values of the input

Measure cycles rat
trusting CPU docu

er
ding

afe.
t.
04.

ger
ecret

2008 RFC 5246 “The Transport Layer Security (TLS) Protocol, Version 1.2”: “This leaves a small timing channel, since MAC performance depends to some extent on the size of the data fragment, but it is not believed to be large enough to be exploitable, due to the large block size of existing MACs and the small size of the timing signal.”

2013 AlFardan–Paterson “Lucky Thirteen: breaking the TLS and DTLS record protocols”: exploit these timings; steal plaintext.

How to write constant-time

If possible, write code in asn
to control instruction selecti

Look for documentation identifying variability: e.g., “Division operations termina when the divide operation completes, with the number cycles required dependent o values of the input operands

Measure cycles rather than trusting CPU documentatio

2008 RFC 5246 "The Transport Layer Security (TLS) Protocol, Version 1.2": "This leaves a small timing channel, since MAC performance depends to some extent on the size of the data fragment, but it is not believed to be large enough to be exploitable, due to the large block size of existing MACs and the small size of the timing signal."

2013 AlFardan–Paterson "Lucky Thirteen: breaking the TLS and DTLS record protocols": exploit these timings; steal plaintext.

How to write constant-time code

If possible, write code in asm to control instruction selection.

Look for documentation identifying variability: e.g., "Division operations terminate when the divide operation completes, with the number of cycles required dependent on the values of the input operands."

Measure cycles rather than trusting CPU documentation.

FC 5246 "The Transport
ecurity (TLS) Protocol,
1.2": "This leaves a
ning channel, since MAC
ance depends to some
n the size of the data
t, but it is not believed to
enough to be exploitable,
he large block size of
MACs and the small size
ming signal."

Fardan–Paterson "Lucky
: breaking the TLS and
ecord protocols": exploit
mings; steal plaintext.

How to write constant-time code

If possible, write code in asm to control instruction selection.

Look for documentation identifying variability: e.g., "Division operations terminate when the divide operation completes, with the number of cycles required dependent on the values of the input operands."

Measure cycles rather than trusting CPU documentation.

Cut off a
secrets t

Cut off a
secrets t

Cut off a
secrets t

Prefer lo

Prefer ve

Watch o
variable-
Cortex-M

The Transport
LS) Protocol,
is leaves a
nel, since MAC
nds to some
of the data
not believed to
be exploitable,
lock size of
d the small size
al."

terson "Lucky
the TLS and
ocols": exploit
al plaintext.

How to write constant-time code

If possible, write code in asm to control instruction selection.

Look for documentation identifying variability: e.g., "Division operations terminate when the divide operation completes, with the number of cycles required dependent on the values of the input operands."

Measure cycles rather than trusting CPU documentation.

Cut off all data flo
secrets to branch c

Cut off all data flo
secrets to array ind

Cut off all data flo
secrets to shift/rot

Prefer logic instruc

Prefer vector instr

Watch out for CP
variable-time mult
Cortex-M3 and mo

port
col,
n
MAC
ne
ta
ved to
itable,
of
l size

ucky
and
ploit
t.

How to write constant-time code

If possible, write code in asm to control instruction selection.

Look for documentation identifying variability: e.g., “Division operations terminate when the divide operation completes, with the number of cycles required dependent on the values of the input operands.”

Measure cycles rather than trusting CPU documentation.

Cut off all data flow from secrets to branch conditions

Cut off all data flow from secrets to array indices.

Cut off all data flow from secrets to shift/rotate distar

Prefer logic instructions.

Prefer vector instructions.

Watch out for CPUs with variable-time multipliers: e.g Cortex-M3 and most PowerP

How to write constant-time code

If possible, write code in asm to control instruction selection.

Look for documentation identifying variability: e.g., “Division operations terminate when the divide operation completes, with the number of cycles required dependent on the values of the input operands.”

Measure cycles rather than trusting CPU documentation.

Cut off all data flow from secrets to branch conditions.

Cut off all data flow from secrets to array indices.

Cut off all data flow from secrets to shift/rotate distances.

Prefer logic instructions.

Prefer vector instructions.

Watch out for CPUs with variable-time multipliers: e.g., Cortex-M3 and most PowerPCs.

write constant-time code

le, write code in asm
ol instruction selection.

documentation
ng variability: e.g.,
n operations terminate
e divide operation
es, with the number of
equired dependent on the
f the input operands."

cycles rather than
CPU documentation.

Cut off all data flow from secrets to branch conditions.

Cut off all data flow from secrets to array indices.

Cut off all data flow from secrets to shift/rotate distances.

Prefer logic instructions.

Prefer vector instructions.

Watch out for CPUs with variable-time multipliers: e.g., Cortex-M3 and most PowerPCs.

Suppose
const-ti

Suppose
has "se

Easy for
that sec
by const

Proofs o
(uninitia
ctgrind,

stant-time code

ode in asm
ion selection.

tation
ity: e.g.,
ns terminate
peration
e number of
pendent on the
t operands."

her than
umentation.

Cut off all data flow from secrets to branch conditions.

Cut off all data flow from secrets to array indices.

Cut off all data flow from secrets to shift/rotate distances.

Prefer logic instructions.

Prefer vector instructions.

Watch out for CPUs with variable-time multipliers: e.g., Cortex-M3 and most PowerPCs.

Suppose we know
const-time machin

Suppose programm
has "`secret`" typ

Easy for compiler
that `secret` types
by const-time instr

Proofs of concept:
(uninitialized data
ctgrind, ct-verif, F

code

n

on.

ate

of

n the

s.”

n.

Cut off all data flow from secrets to branch conditions.

Cut off all data flow from secrets to array indices.

Cut off all data flow from secrets to shift/rotate distances.

Prefer logic instructions.

Prefer vector instructions.

Watch out for CPUs with variable-time multipliers: e.g., Cortex-M3 and most PowerPCs.

Suppose we know (some) const-time machine instructi

Suppose programming langu has “`secret`” types.

Easy for compiler to guarant that `secret` types are used by const-time instructions.

Proofs of concept: Valgrind (uninitialized data as `secre` ctgrind, ct-verif, FlowTracke

Cut off all data flow from secrets to branch conditions.

Cut off all data flow from secrets to array indices.

Cut off all data flow from secrets to shift/rotate distances.

Prefer logic instructions.

Prefer vector instructions.

Watch out for CPUs with variable-time multipliers: e.g., Cortex-M3 and most PowerPCs.

Suppose we know (some) const-time machine instructions.

Suppose programming language has "`secret`" types.

Easy for compiler to guarantee that `secret` types are used only by const-time instructions.

Proofs of concept: Valgrind (uninitialized data as `secret`), ctgrind, ct-verif, FlowTracker.

Cut off all data flow from secrets to branch conditions.

Cut off all data flow from secrets to array indices.

Cut off all data flow from secrets to shift/rotate distances.

Prefer logic instructions.

Prefer vector instructions.

Watch out for CPUs with variable-time multipliers: e.g., Cortex-M3 and most PowerPCs.

Suppose we know (some) const-time machine instructions.

Suppose programming language has "`secret`" types.

Easy for compiler to guarantee that `secret` types are used only by const-time instructions.

Proofs of concept: Valgrind (uninitialized data as `secret`), ctgrind, ct-verif, FlowTracker.

How can we implement, e.g., sorting of a `secret` array?

all data flow from
to branch conditions.

all data flow from
to array indices.

all data flow from
to shift/rotate distances.

gic instructions.

ector instructions.

ut for CPUs with
time multipliers: e.g.,
M3 and most PowerPCs.

Suppose we know (some) const-time machine instructions.

Suppose programming language has "`secret`" types.

Easy for compiler to guarantee that `secret` types are used only by const-time instructions.

Proofs of concept: Valgrind (uninitialized data as `secret`), ctgrind, ct-verif, FlowTracker.

How can we implement, e.g., sorting of a `secret` array?

Eliminat

Let's try

Assume

ow from
conditions.

ow from
dices.

ow from
tate distances.

ctions.

uctions.

Us with
ipliers: e.g.,
ost PowerPCs.

Suppose we know (some) const-time machine instructions.

Suppose programming language has "`secret`" types.

Easy for compiler to guarantee that `secret` types are used only by const-time instructions.

Proofs of concept: Valgrind (uninitialized data as `secret`), ctgrind, ct-verif, FlowTracker.

How can we implement, e.g., sorting of a `secret` array?

Eliminating branch

Let's try sorting 2

Assume `int32` is

.

nces.

g.,
PCs.

Suppose we know (some) const-time machine instructions.

Suppose programming language has "`secret`" types.

Easy for compiler to guarantee that `secret` types are used only by const-time instructions.

Proofs of concept: Valgrind (uninitialized data as `secret`), ctgrind, ct-verif, FlowTracker.

How can we implement, e.g., sorting of a `secret` array?

Eliminating branches

Let's try sorting 2 integers. Assume `int32` is `secret`.

Suppose we know (some) const-time machine instructions.

Suppose programming language has “`secret`” types.

Easy for compiler to guarantee that `secret` types are used only by const-time instructions.

Proofs of concept: Valgrind (uninitialized data as `secret`), ctgrind, ct-verif, FlowTracker.

How can we implement, e.g., sorting of a `secret` array?

Eliminating branches

Let’s try sorting 2 integers.
Assume `int32` is `secret`.

Suppose we know (some) const-time machine instructions.

Suppose programming language has "`secret`" types.

Easy for compiler to guarantee that `secret` types are used only by const-time instructions.

Proofs of concept: Valgrind (uninitialized data as `secret`), ctgrind, ct-verif, FlowTracker.

How can we implement, e.g., sorting of a `secret` array?

Eliminating branches

Let's try sorting 2 integers. Assume `int32` is `secret`.

```
void sort2(int32 *x)
{ int32 x0 = x[0];
  int32 x1 = x[1];
  if (x1 < x0) {
    x[0] = x1;
    x[1] = x0;
  }
}
```

Suppose we know (some) const-time machine instructions.

Suppose programming language has “secret” types.

Easy for compiler to guarantee that secret types are used only by const-time instructions.

Proofs of concept: Valgrind (uninitialized data as secret), ctgrind, ct-verif, FlowTracker.

How can we implement, e.g., sorting of a secret array?

Eliminating branches

Let’s try sorting 2 integers. Assume int32 is secret.

```
void sort2(int32 *x)
{ int32 x0 = x[0];
  int32 x1 = x[1];
  if (x1 < x0) {
    x[0] = x1;
    x[1] = x0;
  }
}
```

Unacceptable: not constant-time.

we know (some)
me machine instructions.

programming language
cret" types.

compiler to guarantee
cret types are used only
-time instructions.

of concept: Valgrind
lized data as secret),
ct-verif, FlowTracker.

n we implement, e.g.,
of a secret array?

Eliminating branches

Let's try sorting 2 integers.
Assume `int32` is `secret`.

```
void sort2(int32 *x)
{ int32 x0 = x[0];
  int32 x1 = x[1];
  if (x1 < x0) {
    x[0] = x1;
    x[1] = x0;
  }
}
```

Unacceptable: not constant-time.

```
void so
{ int32
  int32
  if (x
    x[0
    x[1
  } els
    x[0
    x[1
  }
}
```

(some)
e instructions.

ming language
es.

to guarantee
s are used only
ructions.

Valgrind
as secret),
lowTracker.

ement, e.g.,
et array?

Eliminating branches

Let's try sorting 2 integers.
Assume `int32` is `secret`.

```
void sort2(int32 *x)
{ int32 x0 = x[0];
  int32 x1 = x[1];
  if (x1 < x0) {
    x[0] = x1;
    x[1] = x0;
  }
}
```

Unacceptable: not constant-time.

```
void sort2(int32
{ int32 x0 = x[0
  int32 x1 = x[1
  if (x1 < x0) {
    x[0] = x1;
    x[1] = x0;
  } else {
    x[0] = x0;
    x[1] = x1;
  }
}
```

ions.

uage

tee

only

t),

er.

,'

Eliminating branches

Let's try sorting 2 integers.

Assume `int32` is `secret`.

```
void sort2(int32 *x)
{ int32 x0 = x[0];
  int32 x1 = x[1];
  if (x1 < x0) {
    x[0] = x1;
    x[1] = x0;
  }
}
```

Unacceptable: not constant-time.

```
void sort2(int32 *x)
{ int32 x0 = x[0];
  int32 x1 = x[1];
  if (x1 < x0) {
    x[0] = x1;
    x[1] = x0;
  } else {
    x[0] = x0;
    x[1] = x1;
  }
}
```

Eliminating branches

Let's try sorting 2 integers.

Assume `int32` is `secret`.

```
void sort2(int32 *x)
{ int32 x0 = x[0];
  int32 x1 = x[1];
  if (x1 < x0) {
    x[0] = x1;
    x[1] = x0;
  }
}
```

Unacceptable: not constant-time.


```
void sort2(int32 *x)
{ int32 x0 = x[0];
  int32 x1 = x[1];
  if (x1 < x0) {
    x[0] = x1;
    x[1] = x0;
  } else {
    x[0] = x0;
    x[1] = x1;
  }
}
```

Eliminating branches

Let's try sorting 2 integers.

Assume `int32` is `secret`.

```
void sort2(int32 *x)
{ int32 x0 = x[0];
  int32 x1 = x[1];
  if (x1 < x0) {
    x[0] = x1;
    x[1] = x0;
  }
}
```

Unacceptable: not constant-time.


```
void sort2(int32 *x)
{ int32 x0 = x[0];
  int32 x1 = x[1];
  if (x1 < x0) {
    x[0] = x1;
    x[1] = x0;
  } else {
    x[0] = x0;
    x[1] = x1;
  }
}
```

Safe compiler won't allow this.

Branch timing leaks secrets.

ing branches

sorting 2 integers.

int32 is secret.

```
rt2(int32 *x)
 x0 = x[0];
 x1 = x[1];
1 < x0) {
] = x1;
] = x0;
```

table: not constant-time.


```
void sort2(int32 *x)
{ int32 x0 = x[0];
  int32 x1 = x[1];
  if (x1 < x0) {
    x[0] = x1;
    x[1] = x0;
  } else {
    x[0] = x0;
    x[1] = x1;
  }
}
```

Safe compiler won't allow this.

Branch timing leaks secrets.

```
void so
{ int32
  int32
  int32
  x[0] =
  x[1] =
}
```

nes

integers.

secret.

```
*x)
];
];
```

constant-time.


```
void sort2(int32 *x)
{ int32 x0 = x[0];
  int32 x1 = x[1];
  if (x1 < x0) {
    x[0] = x1;
    x[1] = x0;
  } else {
    x[0] = x0;
    x[1] = x1;
  }
}
```

Safe compiler won't allow this.

Branch timing leaks secrets.


```
void sort2(int32
{ int32 x0 = x[0
  int32 x1 = x[1
  int32 c = (x1
  x[0] = (c ? x1
  x[1] = (c ? x0
}
```

-time.


```
void sort2(int32 *x)
{ int32 x0 = x[0];
  int32 x1 = x[1];
  if (x1 < x0) {
    x[0] = x1;
    x[1] = x0;
  } else {
    x[0] = x0;
    x[1] = x1;
  }
}
```

Safe compiler won't allow this.
Branch timing leaks secrets.


```
void sort2(int32 *x)
{ int32 x0 = x[0];
  int32 x1 = x[1];
  int32 c = (x1 < x0);
  x[0] = (c ? x1 : x0);
  x[1] = (c ? x0 : x1);
}
```

```
void sort2(int32 *x)
{ int32 x0 = x[0];
  int32 x1 = x[1];
  if (x1 < x0) {
    x[0] = x1;
    x[1] = x0;
  } else {
    x[0] = x0;
    x[1] = x1;
  }
}
```

Safe compiler won't allow this.
Branch timing leaks secrets.


```
void sort2(int32 *x)
{ int32 x0 = x[0];
  int32 x1 = x[1];
  int32 c = (x1 < x0);
  x[0] = (c ? x1 : x0);
  x[1] = (c ? x0 : x1);
}
```

```
void sort2(int32 *x)
{ int32 x0 = x[0];
  int32 x1 = x[1];
  if (x1 < x0) {
    x[0] = x1;
    x[1] = x0;
  } else {
    x[0] = x0;
    x[1] = x1;
  }
}
```

Safe compiler won't allow this.
Branch timing leaks secrets.


```
void sort2(int32 *x)
{ int32 x0 = x[0];
  int32 x1 = x[1];
  int32 c = (x1 < x0);
  x[0] = (c ? x1 : x0);
  x[1] = (c ? x0 : x1);
}
```

Syntax is different but "`?:`" is a branch by definition:

```
  if (x1 < x0) x[0] = x1;
  else x[0] = x0;
  if (x1 < x0) x[1] = x0;
  else x[1] = x1;
```

```
rt2(int32 *x)
 x0 = x[0];
 x1 = x[1];
1 < x0) {
] = x1;
] = x0;
e {
] = x0;
] = x1;
```

mpiler won't allow this.

timing leaks secrets.


```
void sort2(int32 *x)
{ int32 x0 = x[0];
  int32 x1 = x[1];
  int32 c = (x1 < x0);
  x[0] = (c ? x1 : x0);
  x[1] = (c ? x0 : x1);
}
```

Syntax is different but "?:" is a branch by definition:

```
  if (x1 < x0) x[0] = x1;
  else x[0] = x0;
  if (x1 < x0) x[1] = x0;
  else x[1] = x1;
```

```
void so
{ int32
  int32
  int32
  x[c] =
  x[1 -
}
```

```
 *x)
];
];
```

't allow this.
ks secrets.


```
void sort2(int32 *x)
{ int32 x0 = x[0];
  int32 x1 = x[1];
  int32 c = (x1 < x0);
  x[0] = (c ? x1 : x0);
  x[1] = (c ? x0 : x1);
}
```

Syntax is different but "?:" is a branch by definition:

```
  if (x1 < x0) x[0] = x1;
  else x[0] = x0;
  if (x1 < x0) x[1] = x0;
  else x[1] = x1;
```

```
void sort2(int32
{ int32 x0 = x[0
  int32 x1 = x[1
  int32 c = (x1
  x[c] = x0;
  x[1 - c] = x1;
}
```

```
void sort2(int32 *x)
{ int32 x0 = x[0];
  int32 x1 = x[1];
  int32 c = (x1 < x0);
  x[0] = (c ? x1 : x0);
  x[1] = (c ? x0 : x1);
}
```

Syntax is different but “?:” is a branch by definition:

```
  if (x1 < x0) x[0] = x1;
  else x[0] = x0;
  if (x1 < x0) x[1] = x0;
  else x[1] = x1;
```



```
void sort2(int32 *x)
{ int32 x0 = x[0];
  int32 x1 = x[1];
  int32 c = (x1 < x0);
  x[c] = x0;
  x[1 - c] = x1;
}
```

his.

```
void sort2(int32 *x)
{ int32 x0 = x[0];
  int32 x1 = x[1];
  int32 c = (x1 < x0);
  x[0] = (c ? x1 : x0);
  x[1] = (c ? x0 : x1);
}
```

Syntax is different but “?:” is a branch by definition:

```
  if (x1 < x0) x[0] = x1;
  else x[0] = x0;
  if (x1 < x0) x[1] = x0;
  else x[1] = x1;
```



```
void sort2(int32 *x)
{ int32 x0 = x[0];
  int32 x1 = x[1];
  int32 c = (x1 < x0);
  x[c] = x0;
  x[1 - c] = x1;
}
```

```
void sort2(int32 *x)
{ int32 x0 = x[0];
  int32 x1 = x[1];
  int32 c = (x1 < x0);
  x[0] = (c ? x1 : x0);
  x[1] = (c ? x0 : x1);
}
```

Syntax is different but "?:" is a branch by definition:

```
  if (x1 < x0) x[0] = x1;
  else x[0] = x0;
  if (x1 < x0) x[1] = x0;
  else x[1] = x1;
```



```
void sort2(int32 *x)
{ int32 x0 = x[0];
  int32 x1 = x[1];
  int32 c = (x1 < x0);
  x[c] = x0;
  x[1 - c] = x1;
}
```

Safe compiler won't allow this: won't allow `secret` data to be used as an array index.

Cache timing is not constant: see earlier attack examples.

```
rt2(int32 *x)
 x0 = x[0];
 x1 = x[1];
 c = (x1 < x0);
= (c ? x1 : x0);
= (c ? x0 : x1);
```

s different but "?:"
nch by definition:

```
1 < x0) x[0] = x1;
x[0] = x0;
1 < x0) x[1] = x0;
x[1] = x1;
```



```
void sort2(int32 *x)
{ int32 x0 = x[0];
  int32 x1 = x[1];
  int32 c = (x1 < x0);
  x[c] = x0;
  x[1 - c] = x1;
}
```

Safe compiler won't allow this:
won't allow `secret` data
to be used as an array index.

Cache timing is not constant:
see earlier attack examples.

```
void so
{ int32
  int32
  int32
  c *= 
  x[0] =
  x[1] =
}
```

```
*x)
];
];
< x0);
 : x0);
 : x1);
```

but “?:”
inition:

```
[0] = x1;
;
[1] = x0;
;
```



```
void sort2(int32 *x)
{ int32 x0 = x[0];
  int32 x1 = x[1];
  int32 c = (x1 < x0);
  x[c] = x0;
  x[1 - c] = x1;
}
```

Safe compiler won’t allow this: won’t allow `secret` data to be used as an array index.

Cache timing is not constant: see earlier attack examples.

```
void sort2(int32
{ int32 x0 = x[0
  int32 x1 = x[1
  int32 c = (x1
  c *= x1 - x0;
  x[0] = x0 + c;
  x[1] = x1 - c;
}
```

```
void sort2(int32 *x)
{ int32 x0 = x[0];
  int32 x1 = x[1];
  int32 c = (x1 < x0);
  x[c] = x0;
  x[1 - c] = x1;
}
```

Safe compiler won't allow this: won't allow `secret` data to be used as an array index.

Cache timing is not constant: see earlier attack examples.


```
void sort2(int32 *x)
{ int32 x0 = x[0];
  int32 x1 = x[1];
  int32 c = (x1 < x0);
  c *= x1 - x0;
  x[0] = x0 + c;
  x[1] = x1 - c;
}
```

```
void sort2(int32 *x)
{ int32 x0 = x[0];
  int32 x1 = x[1];
  int32 c = (x1 < x0);
  x[c] = x0;
  x[1 - c] = x1;
}
```

Safe compiler won't allow this:
won't allow `secret` data
to be used as an array index.

Cache timing is not constant:
see earlier attack examples.


```
void sort2(int32 *x)
{ int32 x0 = x[0];
  int32 x1 = x[1];
  int32 c = (x1 < x0);
  c *= x1 - x0;
  x[0] = x0 + c;
  x[1] = x1 - c;
}
```

```
void sort2(int32 *x)
{ int32 x0 = x[0];
  int32 x1 = x[1];
  int32 c = (x1 < x0);
  x[c] = x0;
  x[1 - c] = x1;
}
```

Safe compiler won't allow this:
won't allow `secret` data
to be used as an array index.

Cache timing is not constant:
see earlier attack examples.


```
void sort2(int32 *x)
{ int32 x0 = x[0];
  int32 x1 = x[1];
  int32 c = (x1 < x0);
  c *= x1 - x0;
  x[0] = x0 + c;
  x[1] = x1 - c;
}
```

Does safe compiler allow
multiplication of secrets?

Recall that multiplication
takes variable time on, e.g.,
Cortex-M3 and most PowerPCs.

```
rt2(int32 *x)
 x0 = x[0];
 x1 = x[1];
 c = (x1 < x0);
= x0;
 c] = x1;
```

npiler won't allow this:
low `secret` data
ed as an array index.

ming is not constant:
er attack examples.


```
void sort2(int32 *x)
{ int32 x0 = x[0];
  int32 x1 = x[1];
  int32 c = (x1 < x0);
  c *= x1 - x0;
  x[0] = x0 + c;
  x[1] = x1 - c;
}
```

Does safe compiler allow
multiplication of secrets?

Recall that multiplication
takes variable time on, e.g.,
Cortex-M3 and most PowerPCs.

Will war
for fast
but let's
for this s

```
void so
{ int32
  int32
  int32
  c &= 
  x[0] 
  x[1] 
}
```

```
 *x)
];
];
< x0);
```

't allow this:
et data
rray index.

ot constant:
examples.


```
void sort2(int32 *x)
{ int32 x0 = x[0];
  int32 x1 = x[1];
  int32 c = (x1 < x0);
  c *= x1 - x0;
  x[0] = x0 + c;
  x[1] = x1 - c;
}
```

Does safe compiler allow multiplication of secrets?

Recall that multiplication takes variable time on, e.g., Cortex-M3 and most PowerPCs.

Will want to hand
for fast prime-field
but let's dodge the
for this sorting coc

```
void sort2(int32
{ int32 x0 = x[0
  int32 x1 = x[1
  int32 c = -(x1
  c &= x1 ^ x0;
  x[0] = x0 ^ c;
  x[1] = x1 ^ c;
}
```

his:

x.

t:


```
void sort2(int32 *x)
{ int32 x0 = x[0];
  int32 x1 = x[1];
  int32 c = (x1 < x0);
  c *= x1 - x0;
  x[0] = x0 + c;
  x[1] = x1 - c;
}
```

Does safe compiler allow multiplication of secrets?

Recall that multiplication takes variable time on, e.g., Cortex-M3 and most PowerPCs.

Will want to handle this issu for fast prime-field ECC etc. but let's dodge the issue for this sorting code:

```
void sort2(int32 *x)
{ int32 x0 = x[0];
  int32 x1 = x[1];
  int32 c = -(x1 < x0);
  c &= x1 ^ x0;
  x[0] = x0 ^ c;
  x[1] = x1 ^ c;
}
```

```
void sort2(int32 *x)
{ int32 x0 = x[0];
  int32 x1 = x[1];
  int32 c = (x1 < x0);
  c *= x1 - x0;
  x[0] = x0 + c;
  x[1] = x1 - c;
}
```

Does safe compiler allow multiplication of secrets?

Recall that multiplication takes variable time on, e.g., Cortex-M3 and most PowerPCs.

Will want to handle this issue for fast prime-field ECC etc., but let's dodge the issue for this sorting code:

```
void sort2(int32 *x)
{ int32 x0 = x[0];
  int32 x1 = x[1];
  int32 c = -(x1 < x0);
  c &= x1 ^ x0;
  x[0] = x0 ^ c;
  x[1] = x1 ^ c;
}
```

```
rt2(int32 *x)
 x0 = x[0];
 x1 = x[1];
 c = (x1 < x0);
x1 - x0;
= x0 + c;
= x1 - c;
```

fe compiler allow
cation of secrets?

hat multiplication
riable time on, e.g.,
M3 and most PowerPCs.

Will want to handle this issue
for fast prime-field ECC etc.,
but let's dodge the issue
for this sorting code:

```
void sort2(int32 *x)
{ int32 x0 = x[0];
  int32 x1 = x[1];
  int32 c = -(x1 < x0);
  c &= x1 ^ x0;
  x[0] = x0 ^ c;
  x[1] = x1 ^ c;
}
```

1. Possi
(also for
C standa
int32 a
"undefin
Real CP
but *C co*

```
 *x)
];
];
< x0);
```

r allow
ecrets?

lication
e on, e.g.,
ost PowerPCs.

Will want to handle this issue for fast prime-field ECC etc., but let's dodge the issue for this sorting code:

```
void sort2(int32 *x)
{ int32 x0 = x[0];
  int32 x1 = x[1];
  int32 c = -(x1 < x0);
  c &= x1 ^ x0;
  x[0] = x0 ^ c;
  x[1] = x1 ^ c;
}
```

1. Possible correct
(also for previous
C standard does n
`int32` as twos-cor
"undefined" behav
Real CPU uses tw
but *C compiler ca*

"PCs.

Will want to handle this issue for fast prime-field ECC etc., but let's dodge the issue for this sorting code:

```
void sort2(int32 *x)
{ int32 x0 = x[0];
  int32 x1 = x[1];
  int32 c = -(x1 < x0);
  c &= x1 ^ x0;
  x[0] = x0 ^ c;
  x[1] = x1 ^ c;
}
```


1. Possible correctness prob (also for previous code): C standard does not define `int32` as twos-complement; "undefined" behavior on ove Real CPU uses twos-comple but *C compiler can screw th*

Will want to handle this issue for fast prime-field ECC etc., but let's dodge the issue for this sorting code:

```
void sort2(int32 *x)
{ int32 x0 = x[0];
  int32 x1 = x[1];
  int32 c = -(x1 < x0);
  c &= x1 ^ x0;
  x[0] = x0 ^ c;
  x[1] = x1 ^ c;
}
```


1. Possible correctness problems (also for previous code): C standard does not define `int32` as twos-complement; says "undefined" behavior on overflow. Real CPU uses twos-complement but *C compiler can screw this up*.

Will want to handle this issue
for fast prime-field ECC etc.,
but let's dodge the issue
for this sorting code:

```
void sort2(int32 *x)
{ int32 x0 = x[0];
  int32 x1 = x[1];
  int32 c = -(x1 < x0);
  c &= x1 ^ x0;
  x[0] = x0 ^ c;
  x[1] = x1 ^ c;
}
```


1. Possible correctness problems
(also for previous code):
C standard does not define
`int32` as twos-complement; says
"undefined" behavior on overflow.
Real CPU uses twos-complement
but *C compiler can screw this up*.

Fix: use `gcc -fwrapv`.

Will want to handle this issue
for fast prime-field ECC etc.,
but let's dodge the issue
for this sorting code:

```
void sort2(int32 *x)
{ int32 x0 = x[0];
  int32 x1 = x[1];
  int32 c = -(x1 < x0);
  c &= x1 ^ x0;
  x[0] = x0 ^ c;
  x[1] = x1 ^ c;
}
```


1. Possible correctness problems
(also for previous code):
C standard does not define
`int32` as twos-complement; says
"undefined" behavior on overflow.
Real CPU uses twos-complement
but *C compiler can screw this up*.

Fix: use `gcc -fwrapv`.

2. Does safe compiler allow
"`x1 < x0`" for secrets?
What do we do if it doesn't?

Will want to handle this issue
for fast prime-field ECC etc.,
but let's dodge the issue
for this sorting code:

```
void sort2(int32 *x)
{ int32 x0 = x[0];
  int32 x1 = x[1];
  int32 c = -(x1 < x0);
  c &= x1 ^ x0;
  x[0] = x0 ^ c;
  x[1] = x1 ^ c;
}
```


1. Possible correctness problems (also for previous code):
C standard does not define `int32` as twos-complement; says "undefined" behavior on overflow.
Real CPU uses twos-complement but *C compiler can screw this up*.

Fix: use `gcc -fwrapv`.

2. Does safe compiler allow "`x1 < x0`" for secrets?
What do we do if it doesn't?

C compilers *sometimes* use constant-time instructions for this.

nt to handle this issue
prime-field ECC etc.,
dodge the issue
sorting code:

```
rt2(int32 *x)
 x0 = x[0];
 x1 = x[1];
 c = -(x1 < x0);
x1 ^ x0;
= x0 ^ c;
= x1 ^ c;
```


1. Possible correctness problems (also for previous code):
C standard does not define `int32` as twos-complement; says "undefined" behavior on overflow.
Real CPU uses twos-complement but *C compiler can screw this up*.

Fix: use `gcc -fwrapv`.

2. Does safe compiler allow "`x1 < x0`" for secrets?
What do we do if it doesn't?

C compilers *sometimes* use constant-time instructions for this.

Constan

```
int32 i
{ retur
```

Returns

le this issue
l ECC etc.,
e issue
de:

```
*x)
];
];
< x0);
```


1. Possible correctness problems (also for previous code):
C standard does not define `int32` as twos-complement; says "undefined" behavior on overflow.
Real CPU uses twos-complement but *C compiler can screw this up*.

Fix: use `gcc -fwrapv`.

2. Does safe compiler allow "`x1 < x0`" for secrets?
What do we do if it doesn't?

C compilers *sometimes* use constant-time instructions for this.

Constant-time con

```
int32 isnegative
{ return x >> 31
```

Returns −1 if `x <`

ue

,

1. Possible correctness problems (also for previous code):
C standard does not define `int32` as twos-complement; says "undefined" behavior on overflow.
Real CPU uses twos-complement but *C compiler can screw this up*.

Fix: use `gcc -fwrapv`.

2. Does safe compiler allow "`x1 < x0`" for secrets?
What do we do if it doesn't?

C compilers *sometimes* use constant-time instructions for this.

Constant-time comparisons

```
int32 isnegative(int32 x)
{ return x >> 31; }
```

Returns −1 if `x < 0`, otherwi

1. Possible correctness problems (also for previous code): C standard does not define `int32` as twos-complement; says "undefined" behavior on overflow. Real CPU uses twos-complement but *C compiler can screw this up*.

Fix: use `gcc -fwrapv`.

2. Does safe compiler allow "`x1 < x0`" for secrets? What do we do if it doesn't?

C compilers *sometimes* use constant-time instructions for this.

Constant-time comparisons

```
int32 isnegative(int32 x)
{ return x >> 31; }
```

Returns −1 if `x` < 0, otherwise 0.

1. Possible correctness problems (also for previous code):
C standard does not define `int32` as twos-complement; says "undefined" behavior on overflow.
Real CPU uses twos-complement but *C compiler can screw this up*.

Fix: use `gcc -fwrapv`.

2. Does safe compiler allow "`x1 < x0`" for secrets?
What do we do if it doesn't?

C compilers *sometimes* use constant-time instructions for this.

Constant-time comparisons

```
int32 isnegative(int32 x)
{ return x >> 31; }
```

Returns −1 if `x < 0`, otherwise 0.

Why this works: the bits $(b_{31}, b_{30}, \ldots, b_2, b_1, b_0)$ represent the integer $b_0 + 2b_1 + 4b_2 + \cdots + 2^{30}b_{30} - 2^{31}b_{31}$.

"1-bit signed right shift": $(b_{31}, b_{31}, \ldots, b_3, b_2, b_1)$.

"31-bit signed right shift": $(b_{31}, b_{31}, \ldots, b_{31}, b_{31}, b_{31})$.

ble correctness problems
previous code):
ard does not define
s twos-complement; says
ned" behavior on overflow.
U uses twos-complement
ompiler can screw this up.

gcc -fwrapv.

safe compiler allow
0" for secrets?
o we do if it doesn't?

lers *sometimes* use
t-time instructions for this.

Constant-time comparisons

```
int32 isnegative(int32 x)
{ return x >> 31; }
```

Returns `-1` if `x` < 0, otherwise 0.

Why this works: the bits $(b_{31}, b_{30}, \ldots, b_2, b_1, b_0)$ represent the integer $b_0 + 2b_1 + 4b_2 + \cdots + 2^{30}b_{30} - 2^{31}b_{31}$.

"1-bit signed right shift": $(b_{31}, b_{31}, \ldots, b_3, b_2, b_1)$.

"31-bit signed right shift": $(b_{31}, b_{31}, \ldots, b_{31}, b_{31}, b_{31})$.

```
int32 i
{ retur
```

tness problems
code):
ot define
mplement; says
vior on overflow.
os-complement
n screw this up.

rapv.

oiler allow
rets?
it doesn't?

times use
ructions for this.

Constant-time comparisons

```
int32 isnegative(int32 x)
{ return x >> 31; }
```

Returns −1 if `x` < 0, otherwise 0.

Why this works: the bits $(b_{31}, b_{30}, \ldots, b_2, b_1, b_0)$ represent the integer $b_0 + 2b_1 + 4b_2 + \cdots + 2^{30}b_{30} - 2^{31}b_{31}$.

"1-bit signed right shift": $(b_{31}, b_{31}, \ldots, b_3, b_2, b_1)$.

"31-bit signed right shift": $(b_{31}, b_{31}, \ldots, b_{31}, b_{31}, b_{31})$.

```
int32 ispositive
{ return isnegat
```

lems

says
erflow.
ment
is up.

?

or this.

Constant-time comparisons

```
int32 isnegative(int32 x)
{ return x >> 31; }
```

Returns −1 if `x` < 0, otherwise 0.

Why this works: the bits $(b_{31}, b_{30}, \ldots, b_2, b_1, b_0)$ represent the integer $b_0 + 2b_1 + 4b_2 + \cdots + 2^{30}b_{30} - 2^{31}b_{31}$.

"1-bit signed right shift": $(b_{31}, b_{31}, \ldots, b_3, b_2, b_1)$.

"31-bit signed right shift": $(b_{31}, b_{31}, \ldots, b_{31}, b_{31}, b_{31})$.

```
int32 ispositive(int32 x)
{ return isnegative(-x);
```

Constant-time comparisons

```
int32 isnegative(int32 x)
{ return x >> 31; }
```

Returns −1 if `x` < 0, otherwise 0.

Why this works: the bits $(b_{31}, b_{30}, \ldots, b_2, b_1, b_0)$ represent the integer $b_0 + 2b_1 + 4b_2 + \cdots + 2^{30}b_{30} - 2^{31}b_{31}$.

"1-bit signed right shift": $(b_{31}, b_{31}, \ldots, b_3, b_2, b_1)$.

"31-bit signed right shift": $(b_{31}, b_{31}, \ldots, b_{31}, b_{31}, b_{31})$.


```
int32 ispositive(int32 x)
{ return isnegative(-x); }
```

Constant-time comparisons

```
int32 isnegative(int32 x)
{ return x >> 31; }
```

Returns `-1` if `x < 0`, otherwise `0`.

Why this works: the bits $(b_{31}, b_{30}, \ldots, b_2, b_1, b_0)$ represent the integer $b_0 + 2b_1 + 4b_2 + \cdots + 2^{30}b_{30} - 2^{31}b_{31}$.

"1-bit signed right shift": $(b_{31}, b_{31}, \ldots, b_3, b_2, b_1)$.

"31-bit signed right shift": $(b_{31}, b_{31}, \ldots, b_{31}, b_{31}, b_{31})$.


```
int32 ispositive(int32 x)
{ return isnegative(-x); }
```

This code is incorrect!

Fails for input -2^{31}, because "`-x`" produces -2^{31}.

Constant-time comparisons

```
int32 isnegative(int32 x)
{ return x >> 31; }
```

Returns −1 if `x < 0`, otherwise 0.

Why this works: the bits $(b_{31}, b_{30}, \ldots, b_2, b_1, b_0)$ represent the integer $b_0 + 2b_1 + 4b_2 + \cdots + 2^{30}b_{30} - 2^{31}b_{31}$.

"1-bit signed right shift": $(b_{31}, b_{31}, \ldots, b_3, b_2, b_1)$.

"31-bit signed right shift": $(b_{31}, b_{31}, \ldots, b_{31}, b_{31}, b_{31})$.


```
int32 ispositive(int32 x)
{ return isnegative(-x); }
```

This code is incorrect!
Fails for input -2^{31},
because "`-x`" produces -2^{31}.

Can catch this bug by testing:

```
int64 x; int32 c;
for (x = INT32_MIN;
     x <= INT32_MAX;++x) {
  c = ispositive(x);
  assert(c == -(x > 0));
}
```

t-time comparisons

```
snegative(int32 x)
n x >> 31; }
```

−1 if x < 0, otherwise 0.

s works: the bits

$), \ldots, b_2, b_1, b_0)$

t the integer $b_0 + 2b_1 +$

$\cdot + 2^{30}b_{30} - 2^{31}b_{31}$.

gned right shift":

$, \ldots, b_3, b_2, b_1)$.

signed right shift":

$, \ldots, b_{31}, b_{31}, b_{31})$.


```
int32 ispositive(int32 x)
{ return isnegative(-x); }
```

This code is incorrect!

Fails for input -2^{31},

because "-x" produces -2^{31}.

Can catch this bug by testing:

```
int64 x; int32 c;
for (x = INT32_MIN;
     x <= INT32_MAX;++x) {
  c = ispositive(x);
  assert(c == -(x > 0));
}
```

Side not

```
int32 is
{ if (x
  retur
```

mparisons

(int32 x)

; }

0, otherwise 0.

he bits

$_1, b_0)$

ger $b_0 + 2b_1 +$

$_0 - 2^{31} b_{31}$.

t shift”:

$_2, b_1)$.

nt shift”:

$b_{31}, b_{31})$.


```
int32 ispositive(int32 x)
{ return isnegative(-x); }
```

This code is incorrect!
Fails for input -2^{31},
because “-x” produces -2^{31}.

Can catch this bug by testing:

```
int64 x; int32 c;
for (x = INT32_MIN;
     x <= INT32_MAX;++x) {
  c = ispositive(x);
  assert(c == -(x > 0));
}
```

Side note illustrati

```
int32 ispositive
{ if (x == -x) r
  return isnegat
```

se 0.

$b_1 +$

.


```
int32 ispositive(int32 x)
{ return isnegative(-x); }
```

This code is incorrect!

Fails for input -2^{31},

because "`-x`" produces -2^{31}.

Can catch this bug by testing:

```
int64 x; int32 c;
for (x = INT32_MIN;
     x <= INT32_MAX;++x) {
  c = ispositive(x);
  assert(c == -(x > 0));
}
```


Side note illustrating `-fwra`

```
int32 ispositive(int32 x)
{ if (x == -x) return 0;
  return isnegative(-x);
```

```
int32 ispositive(int32 x)
{ return isnegative(-x); }
```

This code is incorrect!

Fails for input -2^{31},

because "`-x`" produces -2^{31}.

Can catch this bug by testing:

```
int64 x; int32 c;
for (x = INT32_MIN;
     x <= INT32_MAX;++x) {
  c = ispositive(x);
  assert(c == -(x > 0));
}
```


Side note illustrating `-fwrapv`:

```
int32 ispositive(int32 x)
{ if (x == -x) return 0;
  return isnegative(-x); }
```

```
int32 ispositive(int32 x)
{ return isnegative(-x); }
```

This code is incorrect!

Fails for input -2^{31},

because "`-x`" produces -2^{31}.

Can catch this bug by testing:

```
int64 x; int32 c;
for (x = INT32_MIN;
     x <= INT32_MAX;++x) {
  c = ispositive(x);
  assert(c == -(x > 0));
}
```


Side note illustrating `-fwrapv`:

```
int32 ispositive(int32 x)
{ if (x == -x) return 0;
  return isnegative(-x); }
```

Not constant-time.

```
int32 ispositive(int32 x)
{ return isnegative(-x); }
```

This code is incorrect!

Fails for input -2^{31},

because "`-x`" produces -2^{31}.

Can catch this bug by testing:

```
int64 x; int32 c;
for (x = INT32_MIN;
     x <= INT32_MAX;++x) {
  c = ispositive(x);
  assert(c == -(x > 0));
}
```


Side note illustrating `-fwrapv`:

```
int32 ispositive(int32 x)
{ if (x == -x) return 0;
  return isnegative(-x); }
```

Not constant-time.

Even worse: without `-fwrapv`, current gcc can remove the `x == -x` test, breaking this code.

```
int32 ispositive(int32 x)
{ return isnegative(-x); }
```

This code is incorrect!
Fails for input -2^{31},
because "-x" produces -2^{31}.

Can catch this bug by testing:

```
int64 x; int32 c;
for (x = INT32_MIN;
     x <= INT32_MAX;++x) {
  c = ispositive(x);
  assert(c == -(x > 0));
}
```


Side note illustrating `-fwrapv`:

```
int32 ispositive(int32 x)
{ if (x == -x) return 0;
  return isnegative(-x); }
```

Not constant-time.

Even worse: without `-fwrapv`, current gcc can remove the `x == -x` test, breaking this code.

Incompetent gcc engineering: source of many security holes. Incompetent language standard.

```
spositive(int32 x)
n isnegative(-x); }
```

de is incorrect!

input -2^{31},

“-x” produces -2^{31}.

ch this bug by testing:

```
; int32 c;
= INT32_MIN;
<= INT32_MAX;++x) {
spositive(x);
t(c == -(x > 0));
```


Side note illustrating `-fwrapv`:

```
int32 ispositive(int32 x)
{ if (x == -x) return 0;
  return isnegative(-x); }
```

Not constant-time.

Even worse: without `-fwrapv`, current gcc can remove the `x == -x` test, breaking this code.

Incompetent gcc engineering: source of many security holes. Incompetent language standard.

```
int32 i
{ retur
  || is
```

```
(int32 x)
ive(-x); }
```

rect!

^{31},

duces -2^{31}.

g by testing:

```
;
IN;
MAX;++x) {
(x);
x > 0));
```


Side note illustrating `-fwrapv`:

```
int32 ispositive(int32 x)
{ if (x == -x) return 0;
  return isnegative(-x); }
```

Not constant-time.

Even worse: without `-fwrapv`, current gcc can remove the `x == -x` test, breaking this code.

Incompetent gcc engineering: source of many security holes. Incompetent language standard.

```
int32 isnonzero(
{ return isnegat
  || isnegative(
```

Side note illustrating `-fwrapv`:

```
int32 ispositive(int32 x)
{ if (x == -x) return 0;
  return isnegative(-x); }
```

Not constant-time.

Even worse: without `-fwrapv`, current gcc can remove the `x == -x` test, breaking this code.

Incompetent gcc engineering: source of many security holes. Incompetent language standard.


```
int32 isnonzero(int32 x)
{ return isnegative(x)
    || isnegative(-x); }
```

Side note illustrating `-fwrapv`:

```
int32 ispositive(int32 x)
{ if (x == -x) return 0;
  return isnegative(-x); }
```

Not constant-time.

Even worse: without `-fwrapv`, current gcc can remove the `x == -x` test, breaking this code.

Incompetent gcc engineering: source of many security holes. Incompetent language standard.


```
int32 isnonzero(int32 x)
{ return isnegative(x)
  || isnegative(-x); }
```

Side note illustrating `-fwrapv`:

```
int32 ispositive(int32 x)
{ if (x == -x) return 0;
  return isnegative(-x); }
```

Not constant-time.

Even worse: without `-fwrapv`, current gcc can remove the `x == -x` test, breaking this code.

Incompetent gcc engineering: source of many security holes. Incompetent language standard.


```
int32 isnonzero(int32 x)
{ return isnegative(x)
  || isnegative(-x); }
```

Not constant-time.
Second part is evaluated only if first part is zero.

Side note illustrating `-fwrapv`:

```
int32 ispositive(int32 x)
{ if (x == -x) return 0;
  return isnegative(-x); }
```

Not constant-time.

Even worse: without `-fwrapv`, current gcc can remove the `x == -x` test, breaking this code.

Incompetent gcc engineering: source of many security holes. Incompetent language standard.


```
int32 isnonzero(int32 x)
{ return isnegative(x)
  || isnegative(-x); }
```

Not constant-time.
Second part is evaluated only if first part is zero.

```
int32 isnonzero(int32 x)
{ return isnegative(x)
  | isnegative(-x); }
```

Constant-time logic instructions.
Safe compiler will allow this.

e illustrating -fwrapv:

```
spositive(int32 x)
 == -x) return 0;
n isnegative(-x); }
```

stant-time.

rse: without -fwrapv,
gcc can remove the
 test, breaking this code.

tent gcc engineering:
f many security holes.
tent language standard.


```
int32 isnonzero(int32 x)
{ return isnegative(x)
  || isnegative(-x); }
```

Not constant-time.
Second part is evaluated
only if first part is zero.

```
int32 isnonzero(int32 x)
{ return isnegative(x)
  | isnegative(-x); }
```

Constant-time logic instructions.
Safe compiler will allow this.

```
int32 i
{ retur
```

ng -fwrapv:

```
(int32 x)
eturn 0;
ive(-x); }
```

e.

out -fwrapv,
emove the
king this code.

engineering:
curity holes.
uage standard.


```
int32 isnonzero(int32 x)
{ return isnegative(x)
  || isnegative(-x); }
```

Not constant-time.
Second part is evaluated
only if first part is zero.

```
int32 isnonzero(int32 x)
{ return isnegative(x)
  | isnegative(-x); }
```

Constant-time logic instructions.
Safe compiler will allow this.

```
int32 issmaller(
{ return isnegat
```

pv:

}

pv,

code.

g:

es.

ard.


```
int32 isnonzero(int32 x)
{ return isnegative(x)
    || isnegative(-x); }
```

Not constant-time.
Second part is evaluated
only if first part is zero.

```
int32 isnonzero(int32 x)
{ return isnegative(x)
    | isnegative(-x); }
```

Constant-time logic instructions.
Safe compiler will allow this.

```
int32 issmaller(int32 x,i
{ return isnegative(x - y
```

```
int32 isnonzero(int32 x)
{ return isnegative(x)
  || isnegative(-x); }
```

Not constant-time.
Second part is evaluated
only if first part is zero.

```
int32 isnonzero(int32 x)
{ return isnegative(x)
  | isnegative(-x); }
```

Constant-time logic instructions.
Safe compiler will allow this.


```
int32 issmaller(int32 x,int32 y)
{ return isnegative(x - y); }
```

```
int32 isnonzero(int32 x)
{ return isnegative(x)
  || isnegative(-x); }
```

Not constant-time.

Second part is evaluated only if first part is zero.

```
int32 isnonzero(int32 x)
{ return isnegative(x)
  | isnegative(-x); }
```

Constant-time logic instructions.

Safe compiler will allow this.


```
int32 issmaller(int32 x,int32 y)
{ return isnegative(x - y); }
```

This code is incorrect!

Generalization of `ispositive`.

Wrong for inputs $(0, -2^{31})$.

```
int32 isnonzero(int32 x)
{ return isnegative(x)
  || isnegative(-x); }
```

Not constant-time.

Second part is evaluated only if first part is zero.

```
int32 isnonzero(int32 x)
{ return isnegative(x)
  | isnegative(-x); }
```

Constant-time logic instructions.

Safe compiler will allow this.


```
int32 issmaller(int32 x,int32 y)
{ return isnegative(x - y); }
```

This code is incorrect!

Generalization of `ispositive`.

Wrong for inputs $(0, -2^{31})$.

Wrong for many more inputs.

Caught quickly by random tests:

```
for (j = 0;j < 10000000;++j) {
  x += random(); y += random();
  c = issmaller(x,y);
  assert(c == -(x < y));
}
```

```
snonzero(int32 x)
n isnegative(x)
negative(-x); }
```

stant-time.
part is evaluated
rst part is zero.

```
snonzero(int32 x)
n isnegative(x)
egative(-x); }
```

t-time logic instructions.
npiler will allow this.


```
int32 issmaller(int32 x,int32 y)
{ return isnegative(x - y); }
```

This code is incorrect!

Generalization of `ispositive`.

Wrong for inputs $(0, -2^{31})$.

Wrong for many more inputs.

Caught quickly by random tests:

```
for (j = 0;j < 10000000;++j) {
  x += random(); y += random();
  c = issmaller(x,y);
  assert(c == -(x < y));
}
```

```
int32 i
{ int32
  int32
  c ^= i
  retur
}
```

```
int32 x)
ive(x)
-x); }
```

.

luated

zero.

```
int32 x)
ive(x)
x); }
```

ic instructions.

allow this.


```
int32 issmaller(int32 x,int32 y)
{ return isnegative(x - y); }
```

This code is incorrect!

Generalization of `ispositive`.

Wrong for inputs $(0, -2^{31})$.

Wrong for many more inputs.

Caught quickly by random tests:

```
for (j = 0;j < 10000000;++j) {
  x += random(); y += random();
  c = issmaller(x,y);
  assert(c == -(x < y));
}
```

```
int32 issmaller(
{ int32 xy = x ^
  int32 c = x -
  c ^= xy & (c ^
  return isnegat
}
```

ions.

.


```
int32 issmaller(int32 x,int32 y)
{ return isnegative(x - y); }
```

This code is incorrect!

Generalization of `ispositive`.

Wrong for inputs $(0, -2^{31})$.

Wrong for many more inputs.

Caught quickly by random tests:

```
for (j = 0;j < 10000000;++j) {
  x += random(); y += random();
  c = issmaller(x,y);
  assert(c == -(x < y));
}
```

```
int32 issmaller(int32 x,i
{ int32 xy = x ^ y;
  int32 c = x - y;
  c ^= xy & (c ^ x);
  return isnegative(c);
}
```

```
int32 issmaller(int32 x,int32 y)
{ return isnegative(x - y); }
```

This code is incorrect!

Generalization of `ispositive`.

Wrong for inputs $(0, -2^{31})$.

Wrong for many more inputs.

Caught quickly by random tests:

```
for (j = 0;j < 10000000;++j) {
  x += random(); y += random();
  c = issmaller(x,y);
  assert(c == -(x < y));
}
```



```
int32 issmaller(int32 x,int32 y)
{ int32 xy = x ^ y;
  int32 c = x - y;
  c ^= xy & (c ^ x);
  return isnegative(c);
}
```

```
int32 issmaller(int32 x,int32 y)
{ return isnegative(x - y); }
```

This code is incorrect!

Generalization of `ispositive`.

Wrong for inputs $(0, -2^{31})$.

Wrong for many more inputs.

Caught quickly by random tests:

```
for (j = 0;j < 10000000;++j) {
  x += random(); y += random();
  c = issmaller(x,y);
  assert(c == -(x < y));
}
```



```
int32 issmaller(int32 x,int32 y)
{ int32 xy = x ^ y;
  int32 c = x - y;
  c ^= xy & (c ^ x);
  return isnegative(c);
}
```

Some verification strategies:

- Think this through.
- Write a proof.
- Formally verify proof.
- Automate proof construction.
- Test many random inputs.
- A bit painful: test all inputs.
- Faster: test `int16` version.

```
ssmaller(int32 x,int32 y)
n isnegative(x - y); }
```

de is incorrect!

zation of `ispositive`.

or inputs $(0, -2^{31})$.

or many more inputs.

quickly by random tests:

```
= 0;j < 10000000;++j) {
random(); y += random();
ssmaller(x,y);
t(c == -(x < y));
```



```
int32 issmaller(int32 x,int32 y)
{ int32 xy = x ^ y;
  int32 c = x - y;
  c ^= xy & (c ^ x);
  return isnegative(c);
}
```

Some verification strategies:

- Think this through.
- Write a proof.
- Formally verify proof.
- Automate proof construction.
- Test many random inputs.
- A bit painful: test all inputs.
- Faster: test `int16` version.

```
void mi
{ int32
  int32
  int32
  int32
  c ^= a
  c >>=
  c &= a
  *x = a
  *y = b
}

void so
{ minma
```

```
int32 x,int32 y)
ive(x - y); }
```

rect!

ispositive.

$(0, -2^{31})$.

nore inputs.

random tests:

```
0000000;++j) {
 y += random();
x,y);
x < y));
```



```
int32 issmaller(int32 x,int32 y)
{ int32 xy = x ^ y;
  int32 c = x - y;
  c ^= xy & (c ^ x);
  return isnegative(c);
}
```

Some verification strategies:

- Think this through.
- Write a proof.
- Formally verify proof.
- Automate proof construction.
- Test many random inputs.
- A bit painful: test all inputs.
- Faster: test `int16` version.

```
void minmax(int3
{ int32 a = *x;
  int32 b = *y;
  int32 ab = b ^
  int32 c = b -
  c ^= ab & (c ^
  c >>= 31;
  c &= ab;
  *x = a ^ c;
  *y = b ^ c;
}

void sort2(int32
{ minmax(x,x + 1
```

```
nt32 y)
); }
```

ve.

s.

ests:

```
+j) {
dom();
```



```
int32 issmaller(int32 x,int32 y)
{ int32 xy = x ^ y;
  int32 c = x - y;
  c ^= xy & (c ^ x);
  return isnegative(c);
}
```

Some verification strategies:

- Think this through.
- Write a proof.
- Formally verify proof.
- Automate proof construction.
- Test many random inputs.
- A bit painful: test all inputs.
- Faster: test `int16` version.

```
void minmax(int32 *x,int3
{ int32 a = *x;
  int32 b = *y;
  int32 ab = b ^ a;
  int32 c = b - a;
  c ^= ab & (c ^ b);
  c >>= 31;
  c &= ab;
  *x = a ^ c;
  *y = b ^ c;
}

void sort2(int32 *x)
{ minmax(x,x + 1); }
```

```
int32 issmaller(int32 x,int32 y)
{ int32 xy = x ^ y;
  int32 c = x - y;
  c ^= xy & (c ^ x);
  return isnegative(c);
}
```

Some verification strategies:

- Think this through.
- Write a proof.
- Formally verify proof.
- Automate proof construction.
- Test many random inputs.
- A bit painful: test all inputs.
- Faster: test `int16` version.


```
void minmax(int32 *x,int32 *y)
{ int32 a = *x;
  int32 b = *y;
  int32 ab = b ^ a;
  int32 c = b - a;
  c ^= ab & (c ^ b);
  c >>= 31;
  c &= ab;
  *x = a ^ c;
  *y = b ^ c;
}

void sort2(int32 *x)
{ minmax(x,x + 1); }
```

```
ssmaller(int32 x,int32 y)
 xy = x ^ y;
 c = x - y;
xy & (c ^ x);
n isnegative(c);
```

erification strategies:
this through.
a proof.
lly verify proof.
nate proof construction.
nany random inputs.
painful: test all inputs.
: test `int16` version.


```
void minmax(int32 *x,int32 *y)
{ int32 a = *x;
  int32 b = *y;
  int32 ab = b ^ a;
  int32 c = b - a;
  c ^= ab & (c ^ b);
  c >>= 31;
  c &= ab;
  *x = a ^ c;
  *y = b ^ c;
}

void sort2(int32 *x)
{ minmax(x,x + 1); }
```

```
int32 i
{ int32
  c ^= 
  retur
}

void so
{ long 
  for (
    for
      mi
}
```

Safe co
if array

```
int32 x,int32 y)
 y;
y;
 x);
ive(c);
```

strategies:
gh.

proof.
construction.
om inputs.
est all inputs.
16 version.


```
void minmax(int32 *x,int32 *y)
{ int32 a = *x;
  int32 b = *y;
  int32 ab = b ^ a;
  int32 c = b - a;
  c ^= ab & (c ^ b);
  c >>= 31;
  c &= ab;
  *x = a ^ c;
  *y = b ^ c;
}

void sort2(int32 *x)
{ minmax(x,x + 1); }
```

```
int32 ispositive
{ int32 c = -x;
  c ^= x & c;
  return isnegat
}

void sort(int32
{ long long i,j;
  for (j = 0;j <
    for (i = j -
      minmax(x +
}
```

Safe compiler will
if array length n is

```
nt32 y)
```

ion.

ts.

.


```
void minmax(int32 *x,int32 *y)
{ int32 a = *x;
  int32 b = *y;
  int32 ab = b ^ a;
  int32 c = b - a;
  c ^= ab & (c ^ b);
  c >>= 31;
  c &= ab;
  *x = a ^ c;
  *y = b ^ c;
}

void sort2(int32 *x)
{ minmax(x,x + 1); }
```

```
int32 ispositive(int32 x)
{ int32 c = -x;
  c ^= x & c;
  return isnegative(c);
}

void sort(int32 *x,long l
{ long long i,j;
  for (j = 0;j < n;++j)
    for (i = j - 1;i >= 0
      minmax(x + i,x + i
}
```

Safe compiler will allow this

if array length n is not secr

```
void minmax(int32 *x,int32 *y)
{ int32 a = *x;
  int32 b = *y;
  int32 ab = b ^ a;
  int32 c = b - a;
  c ^= ab & (c ^ b);
  c >>= 31;
  c &= ab;
  *x = a ^ c;
  *y = b ^ c;
}

void sort2(int32 *x)
{ minmax(x,x + 1); }
```



```
int32 ispositive(int32 x)
{ int32 c = -x;
  c ^= x & c;
  return isnegative(c);
}

void sort(int32 *x,long long n)
{ long long i,j;
  for (j = 0;j < n;++j)
    for (i = j - 1;i >= 0;--i)
      minmax(x + i,x + i + 1);
}
```

Safe compiler will allow this
if array length `n` is not secret.