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

```c
#define ROUNDS 20
```

then it is easy to change.
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.

Another general principle of software engineering:

Make the right thing simple and the wrong thing complex.
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.

Another general principle of software engineering:

Make the right thing simple and the wrong thing complex.
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.

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

We propose instead that one begins with a list of design decisions or 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 . . .”

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.
- FAAAAA 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 . . .”

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.

Not so easy: Timing attacks
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 . . .”

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.
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.
- FAAAAAA 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.
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.

Don’t 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 . . ."
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.

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.
- FAAAAAA 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:

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

How typical software checks 16-byte authenticator:

```c
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.
- FAAAAAA vs. FRIEND: stop at 2.
- FRAAAAA 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:

```c
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.
- FAAAAAA 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:

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

Fix, eliminating information flow from secrets to timings:

```c
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.
Not so easy: Timing attacks

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

- AAAA vs. FRIEND: stop at 1.
- FAAA vs. FRIEND: stop at 2.
- FRAA 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:

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

Fix, eliminating information flow from secrets to timings:

```c
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.
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:

- AAAAAB 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:

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

Fix, eliminating information flow from secrets to timings:

```c
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.
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:

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

Fix, eliminating information flow from secrets to timings:

```c
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:

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

Fix, eliminating information flow from secrets to timings:

```c
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:

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

Fix, eliminating information flow from secrets to timings:

```c
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:

```c
/* 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;
```
Typical software checks 16-byte authenticator:

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

Eliminating information flow from secrets to timings:

```c
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:

```c
/* 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!”
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;

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:

```c
/* 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;
```

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:

```c
/* 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:

```c
/* 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:

```c
/* 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:

```c
/* 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.
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:

```c
/* 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.

Defenders don’t learn

Some of the literature:
1996 Kocher pointed out timing attacks on cryptographic key bits.
Briefly mentioned by Kocher and by
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:

```c
/* 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.

Defenders don’t learn

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.

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.


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, amplify timing signal.

Defenders don’t learn

Some of the literature:


“Guaranteed” countermeasure: load entire table into cache.
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.

timing attacks on DES.

“Guaranteed” countermeasure:
load entire table into cache.
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.

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.


“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.

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:


“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.
Defenders don’t learn of the literature:

Kocher pointed out timing attacks on cryptographic key bits.

Briefly mentioned by and by 1998 Kelsey–Schneier–Wagner–Hall:

Secret array indices can affect timing via cache misses.


“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.
Defenders don’t learn some of the literature:

1996 Kocher pointed out timing attacks on cryptographic key bits.


“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.
Defenders don’t learn

Some of the literature:

1996 Kocher pointed out timing
attacks on cryptographic key bits.
Briefly mentioned by

1998 Kelsey–Schneier–Wagner–Hall:
secret array indices can
affect timing via cache misses.

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.

Intel recommends, and
OpenSSL integrates, cheaper
countermeasure: always load
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.
“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.

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.”
“Guaranteed” countermeasure: load entire table into cache.

Bernstein: AES.

 isn’t safe; indices can affect bank collisions.

all data flow through array indices.

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.

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.”
"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.

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


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.

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.
and
cheaper
always loading
lines of cache.
Rothschwabe
“Warning”:
Countermeasure isn’t safe.
Variable-time lab experiment.
Same issues described in 2004.
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.

How to write constant-time code
If possible, write code in asm
to control instruction selection.
Look for documentation
identifying variability:
“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.
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.

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.

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.
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.
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.”

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.

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.
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 fragments, but it is not believed to be exploitable, due to the large block size of existing MACs and the small size of the timing signal."

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.

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.
11
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.

12
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.
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.
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.

Suppose we know (some) const-time machine instructions.
Suppose programming language has “secret” types.
Easy for compiler to guarantee that secrets are used only by const-time instructions.
Proofs of concept: Valgrind (uninitialized data as secret), ctgrind, ct-verif, FlowTracker.
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.

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.
If possible, write code in assembly 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.

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.
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?
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?
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?

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

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.

```c
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.

```c
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.
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.

```c
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;
    }
}
```

Unacceptable: not constant-time.
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. 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.

```c
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;
  }
}
```

Unacceptable: not constant-time.
Suppose we know (some) const-time machine instructions. Suppose the programming language has "secret" types. Easy for a 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.

```c
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;
  }
}
```

Unacceptable: not constant-time.
Eliminating branches
Let's try sorting 2 integers.
Assume int32 is secret.

```c
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;
  }
}
```

Unacceptable: not constant-time.
Eliminating branches

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

```c
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;
    }
}
```

Unacceptable: not constant-time.

Safe compiler won’t allow this.
Branch timing leaks secrets.
Eliminating branches

Let's try sorting 2 integers.

Assume `int32` is secret.

```c
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.

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

```c
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];
    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];
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];
  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.

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];
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[c] = x0;
x[1 - c] = 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[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;

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;

This.

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[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;

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);
    x[0] = (c ? x1 : x0);
    x[1] = (c ? x0 : x1);
}

Since different but “?:” branch by definition:
if (x1 < x0) x[0] = x1;
else x[0] = x0;
if (x1 < x0) x[1] = x0;
else x[1] = 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);
  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);
    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);
    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);
   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.

Does safe compiler allow 
multiplication of secrets?

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

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;
}
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.

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;
}
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;
}
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;
}
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;
}

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.
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;
}

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:

```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;
}
```

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 \textit{C compiler can screw that up}. PCs.
Will want to handle this issue for fast prime-field ECC etc., but let’s dodge the issue for this sorting code:

```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;
}
```

1. Possible correctness problems (also for previous code): C standard does not define `int32` as two's-complement; says “undefined” behavior on overflow. Real CPU uses two's-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:

```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;
}
```

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:

```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;
}
```

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:

```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;
}
```

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

```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;
}
```

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**

```c
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.
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.

---

```
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

```c
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": \((b_{31}, b_{31}, \ldots, b_{31}, b_{31}, b_{31})\).
1. Possible correctness problems (also for previous code):
   C standard does not define \texttt{int32} as twos-complement; says "undefined" behavior on overflow. Real CPU uses twos-complement but C compiler can screw this up. Fix: use \texttt{gcc -fwrapv}.

2. Does safe compiler allow \texttt{"x1 < x0"} for secrets? What do we do if it doesn't? C compilers sometimes use constant-time instructions for this.

Constant-time comparisons

\begin{verbatim}
int32 isnegative(int32 x)
{ return x >> 31; }
\end{verbatim}

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": \((b_{31}, b_{31}, \ldots, b_{31}, b_{31}, b_{31})\).

\begin{verbatim}
int32 ispositive(int32 x)
{ return isnegative(-x); }
\end{verbatim}
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}).
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

```c
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}).

```c
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));
}
Constant-time comparisons

```c
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 \( b_{0} + 2b_{1} + \cdots + 2^{30}b_{30} - 2^{31}b_{31} \).

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

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

Can catch this bug by testing:

```c
define(x) {
    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)
{ return isnegative(-x); }

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

Can catch this bug by testing:

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

```c
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:

```c
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:

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

Side note illustrating `-fwrapv`:

```c
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:

```c
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:

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

Side note illustrating 

-fwrapv:

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

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

Catch this bug by testing:

```c
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
\texttt{x == -x} test, breaking this code.

\textbf{Incompetent} gcc engineering:
source of many security holes.

Incompetent language standard.
int32 ispositive(int32 x) { return isnegative(-x); }

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

Testing by testing:

```c
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.

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

```c
```
int32 ispositive(int32 x)
{ return isnegative(-x); }
This code is incorrect!
Fails for input $-2^{31}$, because 
"..."

Can catch this bug by testing:

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

Side note illustrating `-fwrapv`:

```c
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.
Side note illustrating -fwrapv:

```c
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.
Side note illustrating -fwrapv:

```c
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.
Side note illustrating `-fwrapv`:

```c
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.
```

```c
int32 isnonzero(int32 x)
{ return isnegative(x)
  || isnegative(-x); }
Not constant-time.
Second part is evaluated only if first part is zero.
```

```c
int32 isnonzero(int32 x)
{ return isnegative(x)
  | isnegative(-x); }
Constant-time logic instructions.
Safe compiler will allow this.
```
int32 ispositive(int32 x)
{ if (x == -x) return 0;
    return isnegative(-x); }

Not constant-time.

Worse: without -fwrapv, gcc can remove the 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.
Introducing -fwrapv:

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

Constant-time logic instructions.
Safe compiler will allow this.
Side note illustrating \texttt{-fwrapv}:

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

Not constant-time.

Even worse: without \texttt{-fwrapv}, current \texttt{gcc} can remove the \texttt{x == -x} test, breaking this code.

Incompetent \texttt{gcc} engineering: source of many security holes.

Incompetent language standard.

---

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

Not constant-time.

Second part is evaluated only if first part is zero.

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

Constant-time logic instructions.

Safe compiler will allow this.
int32 isnonzero(int32 x)
{ return isnegative(x)
   || isnegative(-x); }

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

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

Constant-time logic instructions.
Safe compiler will allow this.
int32 isnonzero(int32 x)
{ return isnegative(x)
  || isnegative(-x); }

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

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

This code is incorrect!
Generalization of ispositive.
Wrong for inputs (0, −2^{31}).

Constant-time logic instructions.
Safe compiler will allow this.
int32 isnonzero(int32 x)  
{ return isnegative(x)  
   || isnegative(-x); }  

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

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

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

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

This code is incorrect!
Generalization of ispositive.
Wrong for inputs \((0, -2^{31})\).
Wrong for many more inputs.
Caught quickly by random tests:

```c
for (j = 0; j < 10000000; ++j) {
  x += random(); y += random();
  c = issmaller(x, y);
  assert(c == -(x < y));
}
```
25
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:

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

26
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:

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

```c
int32 issmaller(int32 x, int32 y)
{ int32 xy = x ^ y;
  int32 c = x - y;
  c ^= xy & (c ^ x);
  return isnegative(c);
}
```
This code is incorrect! Generalization of `ispositive`. Wrong for inputs \((0, -2^{31})\).

Wrong for many more inputs. Caught quickly by random tests:

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

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

This code is incorrect!

Generalization of isnpositive.
For inputs (0, -2^{31}).

For many more inputs.
Wrong for many inputs.
Caught quickly by random tests:
for (j = 0; j < 10000000; ++j) {
    x += random(); y += random();
    c = issmaller(x, y);
    assert(c == -(x < y));
}

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.

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

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);
}

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

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

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.
```

```c
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 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); }
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);
}

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.
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); }

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.
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 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.
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.