Language-based software security, fourth lecture

Tempus fugit:
timing attacks and cache attacks

Xavier Leroy
2022-03-31

Collège de France, chair of software sciences
xavier.leroy@college-de-france.fr
Execution time as an information channel
User experience with an old Unix system

login: dmr
password: *****
(two seconds later…)

Login incorrect
User experience with an old Unix system

login: dmr
password: ******
(two seconds later...)
Login incorrect

Second attempt:

login: foo
password: ******
(immediately)
Login incorrect
Why this difference in response time?

int check_login(char * username, char * password)
{
    struct passwd * userinfo = getpwnam(username);
    if (userinfo == NULL) return 0; // no user with this name
    char * hash = crypt(password); // takes 2 seconds
    return (strcmp(hash, userinfo->pw_password) == 0);
}

The function terminates faster if there is no account named username than if there is one.

⇒ Enables an attacker to easily check if a given account exists.
for (int i = 0; i < N; i++) {
    if (input[i] != pin[i]) return false;
}
return true;

This loop takes time proportional to the number of correct digits at the beginning of input.

⇒ An attacker can find a $N$-digit PIN in time $10^N$ instead of $10^N$. 
Checking a PIN

An alternate implementation where the loop always runs for \( N \) iterations:

```c
valid = true;
for (int i = 0; i < N; i++) {
    if (input[i] != pin[i]) valid = false;
}
return valid;
```

Now, the execution time is \( a + bn \), where \( n \) is the number of wrong digits (= number of assignments \( valid = false \)).

\( \implies \) An efficient attack remains possible.
Let’s make the code more symmetrical by counting the number of correct digits and the number of wrong digits.

```c
valid = 0; invalid = 0;
for (int i = 0; i < N; i++) {
    if (input[i] != pin[i]) ++invalid; else ++valid;
}
return (invalid == 0);
```

Branch prediction in the processor can still cause variations in execution time, but it’s getting hard to exploit them.
Checking a PIN in constant time

The correct way to write this PIN-checking code is to use constant-time operations only, that is, operations that run in time independent of the values of their arguments.

```c
int d = 0;
for (int i = 0; i < N; i++) {
    d = d | (input[i] ^ pin[i]);
}
return (d == 0);
```

(Variable `d` accumulates the bits that differ between `input` and `password`; it remains 0 if and only if there are no differences.)
RSA encryption and signature

Based on **modular exponentiation**:

\[ M \overset{\text{encryption}}{\rightarrow} C = M^e \mod N \overset{\text{decryption}}{\rightarrow} C^d \mod N \]

\[ M \overset{\text{signature}}{\rightarrow} S = M^d \mod N \overset{\text{verification}}{\rightarrow} S^e \mod N \]

A modulus \( N = pq \) product of two prime numbers \( p, q \).

A secret exponent \( d \) and a public exponent \( e \) such that \( de \mod (p - 1)(q - 1) = 1 \).

The public key is \((N, e)\).

The secret key is \( d \) or sometimes \((p, q, d)\).
Computing modular exponentiation

The “Russian peasant” algorithm for fast exponentiation:

Decompose $d$ in bits $d_n, \ldots, d_0$ \hspace{1cm} ($d = \sum_{i=0}^{n} d_i 2^i$)

\begin{align*}
C &:= 1; \quad z := M; \\
\text{for} \ i = 0 \ \text{to} \ n \ \text{do} \\
\quad \text{if} \ d_i \ \text{then} \ C &:= C \cdot z \ \text{mod} \ N \\
\quad z &:= z^2 \ \text{mod} \ N \\
\text{done}
\end{align*}

$z$ takes successive values $M, M^2, M^4, M^8, \ldots, M^{2^n}$ (mod $N$).

At the end we have $C = \prod \{M^{2^i} \mid d_i = 1\} = M^\sum\{2^i \mid d_i = 1\} = M^d$
for $i = 0$ to $n$ do
    if $d_i$ then $C := C \cdot z \mod N$
    $z := z^2 \mod N$
done

The running time of the loop depends on the $d_i$, obviously: we perform $w + n + 1$ modular multiplications, where $w$ is the Hamming weight (number of 1 bits) of the secret $d$.

However, knowing $w$ doesn’t help to guess $d$.

Moreover, we can easily remove this dependence on $w$:

if $d_i$ then $C := C \cdot z \mod N$ else $tmp := C \cdot z \mod N$
Running time of modular exponentiation

for $i = 0$ to $n$ do
    if $d_i$ then $C := C \cdot z \mod N$ else $tmp := C \cdot z \mod N$
    $z := z^2 \mod N$
done

The time it takes to compute $C \cdot z \mod N$ depends significantly on the value of $C$, even more so if clever algorithms are used. This suffices to mount attacks on RSA by observing execution times.
P. Kocher’s timing attack (1996)

Take $k$ random messages $M_1, \ldots, M_k$.

Have them signed: $S_i = M_i^d \mod N$ and measure the time $T_i$.

Guess the bits of $d$ one after the other:

- $d_0 = 1$ always.
- Assume $d_1 = 1$. Then, the computation of $S_i$ would start by computing $M_i \cdot M_i^2 \mod N$.
  - Measure the times $t_i$ to compute $M_i \cdot M_i^2 \mod N$.
  - If the $t_i$ are correlated with the $T_i$, we do have $d_1 = 1$.
  - If there’s no correlation, we have $d_1 = 0$.
- Iterate for the following bits.
OpenSSL has a more efficient implementation of RSA:

- Uses the **Chinese remainder theorem**: compute $M^d \mod p$ and $M^d \mod q$, then combine the results to obtain $M^d \mod N$ (with $N = pq$).

- Uses **Montgomery’s representation** to speed up modular multiplications $C \cdot z \mod q$.

- Several multiplication algorithms, selected based on the sizes of the arguments.

Each of these features contributes to leak more information through execution times...
Montgomery’s algorithm performs additional reduction steps when the product $g$ gets close to the modulus $p$ or $q$.

(Butley & Boneh, 2003)
D. Brumley and D. Boneh’s timing attack

A binary search that identifies the most significant bits of the $q$ factor. Once half the bits are known, Coppersmith’s algorithm recovers the whole secret key.

The attack can be conducted across a network connection!
Cache memory as an information channel
Cache memory

Speed up accesses to a memory location that has been accessed recently (temporal locality), or that is near a recently-accessed location (spatial locality).
Cache attacks

The time taken by a memory read varies greatly whether a data at a nearby location has been accessed recently.

Overall structure of a cache attach:

1. Flush the cache (L1 or more) (clflush instruction, etc)
2. Execute privileged code that manipulates secret data.
3. Measure access times for several memory locations.
4. Infer which locations were accessed by the privileged code.
5. Deduce information on the secret data.

(In step 1-, instead of emptying the cache, we can also pre-fill it with locations that conflict with the locations we want to observe.)

(2- and 3- can take place concurrently.)
Note: it is not necessary to have read and write permissions on the memory area we want to observe. We can use any memory area that shares the same cache entries.
Example: normalizing a character string

Put letters in uppercase and normalize non-printable characters.

```c
void normalize(unsigned char * s, size_t len)
{
    static const unsigned char tbl[256] = "\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0";
    for (size_t i = 0; i < len; i++)
        s[i] = tbl[s[i]];
}
```
An example of a cache attack

get_hashed_password: a protected function that reads a password from the keyboard, normalizes it with normalize, and hashes it.

1. Flush the cache.
2. Call get_hashed_password.
3. Measure the time taken by normalize on inputs "a", "b", "c", etc.
4. Infer the elements of the tbl array accessed by normalize when called from get_hashed_password.
   (Assuming cache lines of size 1 byte.)
5. Deduce which letters a, b, c, etc, appear in the password.
The AES-128 symmetric cipher

(A. G. Wadday et al, 2018)
The AES-128 symmetric cipher

Software implementations of AES usually tabulate the subst/shift/mix steps.

\( T_0, T_1, T_2, T_3 \): tables of 256 32-bit constants.

\( x_0, \ldots, x_{15} \): the 16 bytes of the current state.

\[
(x_0', x_1', x_2', x_3') = T_0[x_0] \oplus T_1[x_5] \oplus T_2[x_{10}] \oplus T_3[x_{15}] \oplus K_0
\]

\[
(x_4', x_5', x_6', x_7') = T_0[x_4] \oplus T_1[x_9] \oplus T_2[x_{14}] \oplus T_3[x_3] \oplus K_1
\]

\[
(x_8', x_9', x_{10}', x_{11}') = T_0[x_8] \oplus T_1[x_{13}] \oplus T_2[x_2] \oplus T_3[x_7] \oplus K_2
\]

\[
(x_{12}', x_{13}', x_{14}', x_{15}') = T_0[x_{12}] \oplus T_1[x_1] \oplus T_2[x_6] \oplus T_3[x_{11}] \oplus K_3
\]
Cache attack on AES

(Osvik, Shamir, Tromer, Cache attacks and countermeasures: the case of AES, 2005. Ashokummar, Giri, Menezes, Highly efficient algorithms for AES key retrieval in cache access attacks, 2016.)

\[(x_0', x_1', x_2', x_3') = T_0[x_0] \oplus T_1[x_5] \oplus T_2[x_{10}] \oplus T_3[x_{15}] \oplus K_0\]
\[(x_4', x_5', x_6', x_7') = T_0[x_4] \oplus T_1[x_9] \oplus T_2[x_{14}] \oplus T_3[x_3] \oplus K_1\]
\[(x_8', x_9', x_{10}', x_{11}') = T_0[x_8] \oplus T_1[x_{13}] \oplus T_2[x_2] \oplus T_3[x_7] \oplus K_2\]
\[(x_{12}', x_{13}', x_{14}', x_{15}') = T_0[x_{12}] \oplus T_1[x_1] \oplus T_2[x_6] \oplus T_3[x_{11}] \oplus K_3\]

Assuming cache lines are 4 32-bit words, each access \(T_i[x_j]\) “leaks” the 4 most significant bits of \(x_j\).

First round: \(x = \text{chosen text } \oplus \text{key}\), hence we can recover the 4 most significant bits of each byte of the key.
Cache attack on AES


\[
\begin{align*}
(x_0', x_1', x_2', x_3') &= T_0[x_0] \oplus T_1[x_5] \oplus T_2[x_{10}] \oplus T_3[x_{15}] \oplus K_0 \\
(x_4', x_5', x_6', x_7') &= T_0[x_4] \oplus T_1[x_9] \oplus T_2[x_{14}] \oplus T_3[x_3] \oplus K_1 \\
(x_8', x_9', x_{10}', x_{11}') &= T_0[x_8] \oplus T_1[x_{13}] \oplus T_2[x_2] \oplus T_3[x_7] \oplus K_2 \\
(x_{12}', x_{13}', x_{14}', x_{15}') &= T_0[x_{12}] \oplus T_1[x_1] \oplus T_2[x_6] \oplus T_3[x_{11}] \oplus K_3
\end{align*}
\]

Assuming cache lines are 4 32-bit words, each access \(T_i[x_j]\) “leaks” the 4 most significant bits of \(x_j\).

Finer analysis of the second round lets us recover the whole key using a small number of encryptions (10 to 1000).
Protections against timing attacks
How can we avoid leaking information through execution time?

Various approaches:

- “Constant-time” programming.
- Prevent precise time measurements.
- Quantize execution or communication times.
- Blinding secrets with random noise.
Precise time measurements

Many timing attacks cannot be conducted remotely: the attacker must run on the same machine as the attacked code.

To measure elapsed time precisely, the attacker needs

- access to a high-resolution hardware clock (e.g. the Time Stamp Counter register on x86 processors);
- or parallel execution of two threads.

```plaintext
while true do
  time := time + 1
  computation to be timed;
done

T_0 := time;
T_1 := time
```
Prevent precise time measurements

The operating system or the execution environment can:

• Prohibit access to high-resolution clocks (e.g. block the `rdtsc` x86 instruction).

• Force the threads of the attacker to run on the same processor core as the attacked code, by interleaving.

• Schedule threads independently from execution time.
Scheduling based on instruction counts

(Stefan et al, *Eliminating cache-based timing attacks with instruction-based scheduling*, 2013.)

```
fillArray(L);
if secret then
  fillArray(H)
else
  skip
for i = 1 to n
  do skip done;
for i = 1 to n + m
  do skip done;
readArray(L);
x := 0
x := 1
```

With a schedule based on time slices (preemption after a fixed time $T$), we terminate with $x = 0$ or $x = 1$ depending on the running time of $\text{readArray}(L)$, which depends on the state of the cache.
Scheduling based on instruction counts

(Stefan et al, *Eliminating cache-based timing attacks with instruction-based scheduling*, 2013.)

fillArray(L);

if secret then
  fillArray(H)
else
  skip

for i = 1 to n
do skip done;

for i = 1 to n + m
do skip done;

readArray(L);
x := 0

x := 1

With a schedule based on instruction counts (preemption after $N$ instructions were executed), the final value of $x$ is independent from the cache state.
Add a delay at the end of computation to guarantee that it runs in constant time $D_{\text{max}}$.

\[
T_0 := \text{now};
\]
\[
\text{for } i = 0 \text{ to } n \text{ do }
\]
\[
\text{if } d_i \text{ then } C := C \cdot z \mod N \text{ else } tmp := C \cdot z \mod N
\]
\[
z := z^2 \mod N
\]
\]
\[
done
\]
\[
D = \text{now} - T_0;
\]
\[
\text{sleep}(D_{\text{max}} - D)
\]

No more temporal leaks!

… at the cost of slowing down all computations.

$D_{\text{max}}$ can be hard to determine \textit{a priori}.
Variation: adjust running time to an integer multiple of $\Delta$.
(E.g. $\Delta = 10^7$ cycles for the Brumley-Boney attack.)

\[
T_0 := \text{now};
\]
\[
\ldots
\]
\[
D = \text{now} - T_0;
\]
\[
\text{sleep}(\text{ceil}(D/\Delta) \times \Delta - D)
\]
Time Quantization

(Askarov, Zhang, Myers, *Predictive black-box mitigation of timing channels*, 2010.)

Variation: adjust $D_{\text{max}}$ on the fly, following an exponential law.

\[
T_0 := \text{now};
\]
\[
\ldots
\]
\[
D := \text{now} - T_0;
\]
\[
\text{if } D > D_{\text{max}} \text{ then } D_{\text{max}} := D_{\text{max}} \times (1 + \varepsilon)
\]
\[
\text{else sleep}(D_{\text{max}} - D)
\]

The attacker gains one bit of information each time $D > D_{\text{max}}$. This happens at most one bit by time slice of duration $(1 + \varepsilon)^k$. Hence, information leakage is $O(\log^2 t)$.

More subtle laws can be used, see Askarov *et al.*
Blinding

Inject randomness in the computation so that running time is no longer correlated with the value of the secret.

Artificial example:
checking a PIN code using a random permutation.

```c
// draw a random permutation S of \{0, \ldots, n - 1\}
for (int i = 0; i < N; i++) {
    if (input[S[i]] != pin[S[i]]) return false;
}
return true;
```

The running time for one execution only gives a lower bound on the number of correct digits.
RSA with message blinding

If $M$ is the message to be signed, we can blind it using a random number $R$ before modular exponentiation.

$$
C \overset{\text{def}}{=} (R^e \cdot M)^d = (R^e)^d \cdot M^d = R^{ed} \cdot M^d = R \cdot M^d \pmod{N}
$$

since $ed \mod \varphi(N) = 1$ and $R^{\varphi(N)} = 1 \pmod{N}$ (Euler’s theorem).

Then, we can un-blind, obtaining the correct result:

$$
S \overset{\text{def}}{=} R^{-1} \cdot C \pmod{N}
$$

The time it takes to compute $(R^e \cdot M)^d$ gives no information to the attackers, since they choose $M$ but not $R$. 
Constant-time programming
Constant-time programming

A programming discipline to write programs that run in time independent from secret data.

Relies on a classification of the base operations of the programming language / of the instructions of the processor:

- **Constant-time operations**: same execution time regardless of the values of the arguments of the operation and of the state of the processor.

- **Variable-time operations**: timing is sensitive to the values of the arguments or to the processor state (caches, branch predictors, etc).
## A standard classification

<table>
<thead>
<tr>
<th></th>
<th>Constant time</th>
<th>Variable time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer arithmetic (¹)</td>
<td>+ − * &amp;</td>
<td>^</td>
</tr>
<tr>
<td>Memory reads and writes (²)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Conditional branches</td>
<td>if while &amp; &amp;</td>
<td></td>
</tr>
</tbody>
</table>

(1) Some processors have variable-time integer multiplication.

(2) For writes \( x[i] = v \) and \( *p = v \), execution time depends on \( x, i, p \) (the accessed address) but not on \( v \) (the stored value).
The “constant-time” criterion

An information flow property:

A value at level $H$ (secret) must not be used as argument to a non-constant-time operation.

Examples:

✔ $z^H := x^H + y^H$

✘ $z^H := x^H / y^H$

✘ if $x^H < y^H$ then $z^H := 1$

✘ $x^H := t^L[i^H]$
Typing rules for constant time

In the style of the type systems for information flow of lecture #2.

\[ \vdash a_1 : \ell \quad a_2 : \ell \quad \vdash a_1 + a_2 : \ell \]

\[ \vdash a_1 : L \quad a_2 : L \quad \vdash a_1 / a_2 : \ell \]

\[ \vdash b : L \quad \vdash c_1 : * \quad \vdash c_2 : * \quad \vdash \text{if } b \text{ then } c_1 \text{ else } c_2 : * \]

\[ \vdash b : L \quad \vdash c : * \]

\[ \vdash \text{while } b \text{ do } c \]

Note: indirect flows (if \( b^H \) then \( x^L := 1 \) else \( x^L := 0 \)) are automatically excluded; no need to trace the \( pc \) level any longer.
A semantics for timing leaks

We can materialize information leaks caused by non-constant-time operations by a program transformation:

\[
\begin{align*}
[z := x + y] &= z := x + y \\
[z := x/y] &= \text{out}(x); \text{out}(y); z := x/y \\
\text{if } x < y \text{ then } c_1 \text{ else } c_2 \ &= \text{out}(x); \text{out}(y); \\
\text{if } x < y \text{ then } [c_1] \text{ else } [c_2] \\
\text{while } x < y \text{ do } c \text{ done} &= \text{out}(x); \text{out}(y); \\
\text{while } x < y \text{ do } \\
\text{[c]; out}(x); \text{out}(y) \text{ done}
\end{align*}
\]
Non-interference in the presence of timing leaks

initial state 1 \quad \text{same values}\quad \text{for the } x^L \quad \text{initial state 2}

program execution (trace $T_1$)

final state 1 \quad \text{same values for the } x^L \quad \text{and same traces } T_1 = T_2 \quad \text{final state 2}

program execution (trace $T_2$)
Programming in “constant-time” style

On secret data: no conditionals, no array indexing, just arithmetic and bitwise operations $\approx$ combinatorial circuits.

Example (reminder):

```c
d = 0;
for (int i = 0; i < N; i++) {
    if (input[i] != pin[i]) d = 1;  
}  
return (d == 0);
```
Programming in “constant-time” style

On secret data: no conditionals, no array indexing, just arithmetic and bitwise operations $\approx$ combinatorial circuits.

Example (reminder):

```c
    d = 0;
    for (int i = 0; i < N; i++) {
        d = d | (input[i] ^ pin[i]);
    }
    return (d == 0);
```
Avoiding arrays and indexing

\(N\)-bit integers can replace arrays of \(N\) Booleans.

Example: a DES S-box = a function 6 bits \(\rightarrow\) 4 bits.

The usual tabulated implementation:

```c
int tbl[64] = { /* 64 4-bit integers */ };  
int sbox(int x) { return tbl[x]; }  
```

Tabulation using 4 64-bit integers:

```c
uint64_t tbl0 = ..., tbl1 = ..., tbl2 = ..., tbl3 = ...;  
int sbox(int x) {  
    return (tbl0 >> x & 1) << 0 | (tbl1 >> x & 1) << 1 |  
            (tbl2 >> x & 1) << 2 | (tbl3 >> x & 1) << 3;  
}  
```

(Constant-time... but much slower!)
IF-conversion: turning conditionals into selections

Base case:

\[
\begin{align*}
\text{if } b \text{ then } x &:= a_1 \text{ else } x := a_2 \implies x := \text{sel}(b, a_1, a_2) \\
\text{if } b \text{ then } x &:= a_1 \implies x := \text{sel}(b, a_1, x)
\end{align*}
\]

The \text{sel}(b, a_1, a_2) operator

- evaluates \(b, a_1\) et \(a_2\);
- returns the value of \(a_1\) if \(b\) is true;
- returns the value of \(a_2\) if \(b\) is false;

in time independent from the value of \(b\) (“constant time”).
IF-conversion: turning conditionals into selections

More generally:

- Execute both `then` and `else` branches, renaming the variables that are assigned.
- Select the final values for the variables using operator `sel`.

Example:

\[
\text{if } b \text{ then } (x := a_1; y := a_2) \text{ else } (y := a_3; z := a_4) \\
\implies x_1 := a_1; \ y_1 := a_2[x \leftarrow x_1]; \\
y_2 := a_3; \ z_2 := a_4[y \leftarrow y_2]; \\
x := \text{sel}(b, x_1, x); \ y := \text{sel}(b, y_1, y_2); \ z := \text{sel}(b, z, z_2)
\]
Limits of IF-conversion

This transformation applies only if the then and else branches

- always terminate;
- never trigger run-time errors;
- have no effects observable by the remainder of the program.

Problematic example:

if $y \neq 0$ then $z := x/y$ else abort()

$\iff z_1 := x/y; \ abort(); \ z := \text{sel}(y \neq 0, z_1, z)$
Implementing the selection operator

Using specific processor instructions (conditional move, predicated instructions, etc).

Portably, when $b$, $a_1$ and $a_2$ have type `bool`:

$$\text{sel}(b, a_1, a_2) = b \land a_1 \lor \neg b \land a_2$$

Portably, when $a_1$ and $a_2$ are integers and $b = 0$ or 1:

$$\text{sel}(b, a_1, a_2) = b \times a_1 + (1 - b) \times a_2$$

$$\text{sel}(b, a_1, a_2) = a_2 + b \times (a_1 - a_2)$$

$$\text{sel}(b, a_1, a_2) = (-b) \land a_1 \lor (b - 1) \land a_2$$

(If $b = 0$, we have $b - 1 = 11\ldots11$ and $-b = 00\ldots00$. If $b = 1$, we have $b - 1 = 00\ldots00$ and $-b = 11\ldots11$.)
An optimizing compiler can perform “IF-conversion” itself, thus making certain conditionals constant-time:

\[
\text{if } b \text{ then } x := a_1 \text{ else } x := a_2 \rightarrow x := \text{sel}(b, a_1, a_2)
\]

But it can also introduce conditional branches to compute arithmetic or logical expressions, such as our \text{sel} implementations!

\[
x := b \times a_1 + (1 - b) \times a_2 \rightarrow \text{if } b \text{ then } x := a_1 \text{ else } x := a_2
\]
Resisting compiler optimizations


**Experiment:** 4 implementations of `sel` in portable C, compiled for x86-32 by various versions of Clang.

<table>
<thead>
<tr>
<th></th>
<th>VERSION_1</th>
<th>VERSION_2</th>
<th>VERSION_3</th>
<th>VERSION_4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>inlined</td>
<td>library</td>
<td>inlined</td>
<td>library</td>
</tr>
<tr>
<td>Clang 3.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-O0</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>-O1</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>-O2</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>-O3</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
</tr>
<tr>
<td>Clang 3.3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-O0</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>-O1</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>-O2</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>-O3</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Clang 3.9</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-O0</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>-O1</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>-O2</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>-O3</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
<td>x</td>
</tr>
</tbody>
</table>

✓ = constant-time code is generated.

✗ = a conditional branch is generated.
Attacks on speculative execution
Speculative execution in processors

Example: branch prediction.

```
load x0, [x1]
branch if x0 = 0 to L1
mul x3, x2, x3
add x4, x3, x4
branch to L2
L1:   ...
```

Memory loads take a lot of time.
Example: branch prediction.

```assembly
load x0, [x1]
branch if x0 = 0 to L1
mul x3, x2, x3
add x4, x3, x4
branch to L2
L1: ...
```

The processor *predicts* (based on previous executions) that $x0$ will not be zero and the branch will not be taken.
Example: branch prediction.

load x0, [x1]
branch if x0 = 0 to L1
mul x3, x2, x3
add x4, x3, x4
branch to L2

L1: ...

The processor executes the following instructions speculatively, in a way that can be reversed if needed.

(For example, the initial values of x3 and x4 are kept somewhere.)
Speculative execution in processors

Example: branch prediction.

```
load x0, [x1]
branch if x0 = 0 to L1
mul x3, x2, x3
add x4, x3, x4
branch to L2
L1:  ...
```

When the load terminates, the conditional branch is resolved. If x0 is 0, the prediction was wrong. The processor rolls back the speculative execution: the effects of the speculated instructions are ignored (e.g. registers x3, x4 are reset to their initial values), and execution resumes at point L1.
Speculative execution in processors

Example: branch prediction.

```plaintext
load x0, [x1]
branch if x0 = 0 to L1
mul x3, x2, x3
add x4, x3, x4
branch to L2
L1: ...
```

If x0 is not zero, the prediction is confirmed, and the processor commits the actions of the speculated instructions, then continues execution.
Speculative execution in processors

Many instructions can be executed speculatively:

- arithmetic and logical operations
- branches
- memory reads
- memory writes (as long as they stay in the write buffer).

The processor can backtrack on these executions by rolling back the modified registers and the memory stores.
Speculative execution in processors

Many instructions can be executed speculatively:

- arithmetic and logical operations
- branches
- memory reads (incl. accessing and updating the caches)
- memory writes (as long as they stay in the write buffer).

The processor can backtrack on these executions by rolling back the modified registers and the memory stores. However, the cache state is kept, not rolled back.
Principle:

A privileged piece of code, executed speculatively, reads memory at an address that depends on a secret. The attacker measures the state of the cache and infers part of the secret.
const unsigned int len = ...;
unsigned char buf[len];

int f(unsigned int idx, int table[256 * CACHE_LINE_SIZE])
{
    int i;
    if (idx < len)
        return table[buf[idx] * CACHE_LINE_SIZE];
    else
        return -1;
}

Function $f$ runs in privileged mode, e.g. within the kernel. Parameters $idx$ and $table$ are controlled by the attacker.
const unsigned int len = ...;
unsigned char buf[len];

int f(unsigned int idx, int table[256 * CACHE_LINE_SIZE])
{
    int i;
    if (idx < len)
        return table[buf[idx] * CACHE_LINE_SIZE];
    else
        return -1;
}

The attacker calls $f$ several times with valid $idx$ values (to train branch prediction), then prepares the cache and calls $f$ with $idx$ too large.
Spectre v1: circumventing array bounds checks

```c
const unsigned int len = ...;
unsigned char buf[len];

int f(unsigned int idx, int table[256 * CACHE_LINE_SIZE])
{
    int i;
    if (idx < len)
        return table[buf[idx] * CACHE_LINE_SIZE];
    else
        return -1;
}
```

The then branch of the if is executed speculatively.
The value of the byte at buf + idx leaks via the cache.
This makes it possible to read a good chunk of the kernel memory space.
Harden array bounds checks against speculation (macros `_nospec` in the Linux kernel).

The usual access with bounds checking:

```c
T safe_read(T * tbl, unsigned len, unsigned idx)
{
    if (idx >= len) abort();
    return tbl[idx];
}
```
Harden array bounds checks against speculation (macros _nospec in the Linux kernel).

Access hardened against speculation:

```c
T safe_read_nospec(T * tbl, unsigned len, unsigned idx)
{
    if (idx >= len) abort();
    return tbl[sel(idx < len, idx, 0)];
}
```

The effect of `sel(idx < len, idx, 0)` is to clip the `idx` value so that it is never too big during speculative execution. During normal execution, `idx < len` and access takes place at index `idx`, as desired.
Variation: circumvent the BPF static code verifier

(Schlüter, Borkmann, Krysiuk, BPF and Spectre PRISC 2022.)

$r1$: valid pointer to a reachable variable.
$r2$: arbitrary integer, controlled by the attacker.

1: if $r0 \neq 0$ goto line 3  
2: $r1 = r2$  
3: if $r0 \neq 1$ goto line 5  
4: $r2 = \text{load}(r1)$  
5: // leak the value of $r2$

The verifier knows that $r0$ cannot be both 0 and 1. Therefore, $\text{load}(r1)$ at line 4 is a load from a valid address.

If both conditional branches (lines 1 and 3) are predicted as not taken, the code speculatively reads address $r2$. 

Many kinds of transient state in processors can leak data via timing channels.

→ Seminar by F. Piessens on 21/04.
Summary
Summary on timing attacks and cache attacks

Execution time is a significant source of information leaks.

These leaks are amplified by features of modern processors: caches, speculative execution, etc.

Some (mostly cryptographic) computations can be hardened against these attacks via constant-time programming, or blinding, or hardware assistance (crypto coprocessors).

Some (incomplete?) protections can be found in operating systems and in Web browsers (JavaScript execution engines).

Intel SGX enclaves are being retired, in part because they are too vulnerable to transient execution attacks.
Related attacks

By observation:

- Power consumption.
- Electromagnetic emission.

By perturbation:

- Fault injection → seminar by K. Heydemann on 07/04