CSE 127: Introduction to Security

Lecture 16: Side Channels and Constant-Time Code

Nadia Heninger and Deian Stefan

UCSD

Fall 2019

Some material from Dan Boneh, Stefan Savage
Reminder: Side-channel attacks

You saw before how timing information (from caches or implementation choices) could leak secret information from a running program.

This lecture:

- A variety of different side-channel attacks
- How side-channel attacks can be used against cryptography
- How to mitigate timing side channels in code
Different types of side channels

Computers are physical objects, so measuring them during program execution can reveal information about the program or data.

- Electromagnetic radiation
  - Voltage running through a wire produces a magnetic field
- Power consumption
  - Different paths through a circuit might consume different amounts of power
- Sound (acoustic attacks)
  - Capacitors discharging can make noises
- Timing
  - Different execution time due to program branches
  - Cache timing attacks
- Error messages
  - Error messages might reveal secret information to an attacker
- Fault attacks
TEMPEST/van Eck Phreaking

“Electromagnetic Radiation from Video Display Units: An Eavesdropping Risk?” Wim van Eck 1985

- Governments knew about emissions for decades (TEMPEST)
- Surprising that it could be done with off the shelf equipment
“Electromagnetic Eavesdropping Risks of Flat-Panel Displays” Kuhn 2004

- Image displays simultaneously along line
- Pick up radiation from screen connection cable

The quick brown fox jumps over the lazy dog.

It is well known that electronic equipment produces electromagnetic fields which may cause interference to radio and television reception. The phenomena underlying this have been thoroughly studied over the past few decades. These studies have resulted in internationally agreed methods for measuring the interference produced by equipment. These are needed because the maximum interference levels which equipment may generate have been laid down by law in most countries.

However, interference is not the only problem caused by electromagnetic radiation. It is possible in some cases to obtain information on the signals used inside the equipment when the radiation is picked up and the received signals are decoded. Especially in the case of digital equipment this possibility constitutes a problem, because remote reconstruction of signals inside the equipment may enable reconstruction of the data the equipment is processing.

This problem is not a new one; defence specialists have been aware of it for over twenty years. Information on the way in which this kind of “eavesdropping” can be prevented is not freely available. Equipment designed to protect military information will probably be three or four times more expensive than the equipment likely to be used for processing of non-military information.
US/NATO define TEMPEST shielding standards
“Timing Analysis of Keystrokes and Timing Attacks on SSH” Song Wagner Tian 2001

- In interactive SSH, keystrokes sent in individual packets
- Build model of inter-keystroke delays by finger, key pair
- Measure packet timing off network, do Viterbi decoding
Side-Channel Attacks and Cryptography

Traditional security models for cryptography focus on indistinguishability of ciphertexts, adversaries who can request encryption or decryption oracles.

Cryptographic program execution can leak information about secrets.

Outside of traditional security models for cryptography.
Timing Attacks on Modular Exponentiation
Kocher 96

RSA performs modular exponentiation: \( m = c^d \mod N \)

Pseudocode for “square and multiply” modular exponentiation algorithm:

\[
\begin{align*}
m &= 1 \\
\text{for } i &= 0 \ldots \text{ len}(d): \\
\quad &\text{if } d[i] = 1: \\
\qquad &m = c \times m \mod N \\
\qquad &m = \text{square}(m) \mod N \\
\text{return } m
\end{align*}
\]

- Number of multiplications performed leaks Hamming weight of private key
- Secret-dependent program execution time
- Turn into full attack by cleverly choosing ciphertexts
Power Analysis Attacks on Modular Exponentiation
Kocher Jaffe Jun 1998

Simple power analysis attacks plot power consumption over time.

The textbook square and multiply implementation clearly leaks secret key bits in a power trace.

Fig. 11 SPA leaks from an RSA implementation
**Reminder: Memory caches and cache attacks**

Caches hold local (fast) copy of recently-accessed 64-byte chunks of memory.

### CPU
- Sends address, Receives data

### MEMORY CACHE
- hash(addr) to map to cache set

#### MAIN MEMORY
- Big, slow
  - e.g. 16GB SDRAM

<table>
<thead>
<tr>
<th>Set</th>
<th>Addr</th>
<th>Cached Data ~64B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>F0016280 31C6F4C0 339DD740 614F8480</td>
<td>B5 F5 80 21 E3 2C..</td>
</tr>
<tr>
<td>1</td>
<td>71685100 132A4880 2A1C0700 C017E9C0</td>
<td>7D BD 5D 2E 84 29..</td>
</tr>
<tr>
<td>2</td>
<td>311956C0 002D47C0 91507E80 55194040</td>
<td>0A 55 47 82 86 4E..</td>
</tr>
<tr>
<td>3</td>
<td>9B27F8C0 BE771100 A001FB40 317178C0</td>
<td>84 A0 7F C7 4E BC..</td>
</tr>
<tr>
<td>4</td>
<td>6618E980 BA0CDB40 89E92C00 090F9C40</td>
<td>35 11 4A E0 2E F1..</td>
</tr>
</tbody>
</table>

In a cache attack, an attack program runs on the same processor as a victim program.

The attack program measures memory access times to determine which data the victim loaded into cache.
Remainder: Memory caches and cache attacks

Caches hold local (fast) copy of recently-accessed 64-byte chunks of memory

### CPU
- Sends address, Receives data

### Addr: 2A1C0700

### MEMORY CACHE
- hash(addr) to map to cache set

<table>
<thead>
<tr>
<th>Set</th>
<th>Addr</th>
<th>Cached Data ~64B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>F0016280</td>
<td>B5 F5 80 21 E3 2C 9A DA 59 11 48 F2 C7 D7 A0 86 67 18 17 4C 59 B8 58 A7</td>
</tr>
<tr>
<td>1</td>
<td>71685100</td>
<td>27 BD 5D 2E 84 29 30 B2 8F 27 05 9C 9E C3 DA EE B7 D9 D1 76 16 54 51 5B</td>
</tr>
<tr>
<td>2</td>
<td>311956C0</td>
<td>0A 55 47 82 86 4E C4 15 4D 78 B5 C4 60 D0 2C DD 78 14 DF 66 E9 D0 11 43</td>
</tr>
<tr>
<td>3</td>
<td>9B27F8C0</td>
<td>84 A0 7F C7 4E BC 3B OB 20 0C DB 58 29 D9 F5 6A 72 50 35 82 CB 91 78 8B</td>
</tr>
<tr>
<td>4</td>
<td>6618E980</td>
<td>35 11 4A E0 2E F1 B0 FC 5A 20 D0 7F 1C 50 A4 F8 EB 6F BB 71 ED 16 07 1F</td>
</tr>
</tbody>
</table>

### MAIN MEMORY
- Big, slow e.g. 16GB SDRAM

- In a cache attack, an attack program runs on the same processor as a victim program.
- The attack program measures memory access times to determine which data the victim loaded into cache.
Reminder: Memory caches and cache attacks

Caches hold local (fast) copy of recently-accessed 64-byte chunks of memory

- In a cache attack, an attack program runs on the same processor as a victim program.
- The attack program measures memory access times to determine which data the victim loaded into cache.

### CPU
- Sends address, receives data

### Addr: 2A1C0700

#### MEMORY CACHE

<table>
<thead>
<tr>
<th>Set</th>
<th>Addr</th>
<th>Cached Data ~64B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00016280</td>
<td>B5 F5 80 21 E3 2C 9A DA 59 11 48 F2</td>
</tr>
<tr>
<td></td>
<td>31C6F40</td>
<td>C7 D7 A0 86 67 18 17 4C 59 B8 58 A7</td>
</tr>
<tr>
<td></td>
<td>339DF740</td>
<td></td>
</tr>
<tr>
<td></td>
<td>614F8480</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>71685100</td>
<td>27 BD 5D 2E 84 29 30 B2 8F 27 05 9C</td>
</tr>
<tr>
<td></td>
<td>132A4880</td>
<td>9E C3 DA EE B7 D9</td>
</tr>
<tr>
<td></td>
<td>2A1C0700</td>
<td></td>
</tr>
<tr>
<td></td>
<td>C017E9C0</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>311956C0</td>
<td>0A 55 47 82 86 4E</td>
</tr>
<tr>
<td></td>
<td>002D47C0</td>
<td>C4 15 4D 78 B5 C4</td>
</tr>
<tr>
<td></td>
<td>91507E80</td>
<td>60 D0 2C DD 78 14</td>
</tr>
<tr>
<td></td>
<td>55194040</td>
<td>DF 66 E9 D0 11 43</td>
</tr>
<tr>
<td>3</td>
<td>9B27F8C0</td>
<td>84 A0 7F C7 4E BC</td>
</tr>
<tr>
<td></td>
<td>BE7711C0</td>
<td>3B 0B 20 0C DB 58</td>
</tr>
<tr>
<td></td>
<td>A001FB40</td>
<td>29 D9 F5 6A 72 50</td>
</tr>
<tr>
<td></td>
<td>317178C0</td>
<td>35 82 CB 91 78 8B</td>
</tr>
<tr>
<td>4</td>
<td>6618E980</td>
<td>35 11 4A E0 2E F1</td>
</tr>
<tr>
<td></td>
<td>B0A0CDB40</td>
<td>B0 FC 5A 20 D0 7F</td>
</tr>
<tr>
<td></td>
<td>89E92C00</td>
<td>1C 50 A4 F8 EB 6F</td>
</tr>
<tr>
<td></td>
<td>090F9C40</td>
<td>BB 71 ED 16 07 1F</td>
</tr>
</tbody>
</table>
**Reminder: Memory caches and cache attacks**

Caches hold local (fast) copy of recently-accessed 64-byte chunks of memory.

<table>
<thead>
<tr>
<th>Set</th>
<th>Addr</th>
<th>Cached Data ~64B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0016280 31C6F4C0 339DD740 614F8480</td>
<td>B5 F5 80 21 E3 2C 9A DA 59 11 48 F2 C7 D7 A0 86 67 18 17 4C 59 B8 58 A7 . . .</td>
</tr>
<tr>
<td>1</td>
<td>71685100 122A4880 2A1C0700 2017E9C0</td>
<td>27 BD 5D 2E 84 29 30 B2 8F 27 05 9C 9E C3 DA EE B7 D9 D1 76 16 54 51 5B . . .</td>
</tr>
<tr>
<td>2</td>
<td>311956C0 002D47C0 91507E80 55194040</td>
<td>0A 55 47 82 86 4E 0A 15 4D 78 B5 C4 60 D0 2C 7B 78 14 DF 66 E9 D0 11 43 . . .</td>
</tr>
<tr>
<td>3</td>
<td>9B27F8C0 BE771100 A001FB40 317178C0</td>
<td>84 A0 7F C7 4E BC 3B 0B 20 0C DB 58 02 D9 F5 6A 72 50 35 92 CB 91 78 8B . . .</td>
</tr>
<tr>
<td>4</td>
<td>6618E980 BA0CDB40 89E92C00 090F9C40</td>
<td>35 11 4A E0 2E F1 B0 FC 5A 20 D0 7F 1C 50 A4 F8 EB 6F BB 71 ED 16 07 1F . . .</td>
</tr>
</tbody>
</table>

**CPU**

Sends address, Receives data

**Addr:** 2A1C0700

**MEMORY CACHE**

hash(addr) to map to cache set

- In a cache attack, an attack program runs on the same processor as a victim program.
- The attack program measures memory access times to determine which data the victim loaded into cache.
Reminder: Memory caches and cache attacks

Caches hold local (fast) copy of recently-accessed 64-byte chunks of memory.

- In a cache attack, an attack program runs on the same processor as a victim program.
- The attack program measures memory access times to determine which data the victim loaded into cache.
Reminder: Memory caches and cache attacks

Caches hold local (fast) copy of recently-accessed 64-byte chunks of memory

<table>
<thead>
<tr>
<th>Set</th>
<th>Addr</th>
<th>Cached Data ~64B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>F0016280 31C6F4C0 339DD740 614F8480</td>
<td>B5 F5 80 21 E3 2C 9A DA 59 11 48 F2 C7 D7 A0 86 67 18 17 4C 59 B8 58 A7...</td>
</tr>
<tr>
<td>1</td>
<td>71685100 132A4880 2A1C0700 C017E9C0</td>
<td>27 BD 5D 2E 84 29 30 B2 8F 27 05 9C 9E C3 DA EE B7 D9 D1 76 16 54 51 5B...</td>
</tr>
<tr>
<td>2</td>
<td>311956C0 002D47C0 91507E80 55190400</td>
<td>0A 55 47 82 86 4E C4 15 4D 78 B5 C4 60 D0 2C DD 78 14 DF 66 E9 D0 11 43...</td>
</tr>
<tr>
<td>3</td>
<td>9B27F8C0 BE771100 A001FB40 317178C0</td>
<td>84 A0 7F C7 4E BC 3B 0B 20 0C DB 58 29 D9 F5 6A 72 50 35 82 CB 91 78 8B...</td>
</tr>
<tr>
<td>4</td>
<td>6618E980 BA0CDB40 89E92C00 090F9C40</td>
<td>35 11 4A E0 2E F1 B0 FC 5A 20 D0 7F 1C 50 A4 F8 EB 6F BB 71 ED 16 07 1F...</td>
</tr>
</tbody>
</table>

- In a cache attack, an attack program runs on the same processor as a victim program.
- The attack program measures memory access times to determine which data the victim loaded into cache.
Reminder: Memory caches and cache attacks

Caches hold local (fast) copy of recently-accessed 64-byte chunks of memory

<table>
<thead>
<tr>
<th>Set</th>
<th>Addr</th>
<th>Cached Data ~64B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>F0016280</td>
<td>B5 F5 80 21 E3 2C..</td>
</tr>
<tr>
<td></td>
<td>31C6F4C0</td>
<td>9A DA 59 11 48 F2..</td>
</tr>
<tr>
<td></td>
<td>339D740</td>
<td>C7 D7 A0 86 67 18..</td>
</tr>
<tr>
<td></td>
<td>614F8480</td>
<td>17 4C 59 B8 58 A7..</td>
</tr>
<tr>
<td>1</td>
<td>71685100</td>
<td>27 BD 5D 2E 84 29..</td>
</tr>
<tr>
<td></td>
<td>132A4880</td>
<td>30 B3 8F 27 05 9C..</td>
</tr>
<tr>
<td></td>
<td>2A1C0700</td>
<td>9E C3 DA EE B7 D9..</td>
</tr>
<tr>
<td></td>
<td>C017E9C0</td>
<td>D1 76 16 54 51 5B..</td>
</tr>
<tr>
<td>2</td>
<td>311956C0</td>
<td>0A 55 47 82 86 4E..</td>
</tr>
<tr>
<td></td>
<td>002D47C0</td>
<td>C4 15 4D 78 B5 C4..</td>
</tr>
<tr>
<td></td>
<td>91507E80</td>
<td>60 D0 2C DD 78 14..</td>
</tr>
<tr>
<td></td>
<td>55194040</td>
<td>DF 66 E9 D0 11 43..</td>
</tr>
<tr>
<td>3</td>
<td>9B27F8C0</td>
<td>84 A0 7F C7 4E BC..</td>
</tr>
<tr>
<td></td>
<td>BE771100</td>
<td>3B 0B 20 0C DB 58..</td>
</tr>
<tr>
<td></td>
<td>A001FB40</td>
<td>29 D9 F5 6A 72 50..</td>
</tr>
<tr>
<td></td>
<td>317178C0</td>
<td>35 82 CB 91 78 8B..</td>
</tr>
<tr>
<td>4</td>
<td>6618E980</td>
<td>35 11 4A E0 2E F1..</td>
</tr>
<tr>
<td></td>
<td>BA0CDB40</td>
<td>B0 FC 5A 20 D0 7F..</td>
</tr>
<tr>
<td></td>
<td>89E92C00</td>
<td>1C 50 A4 F8 EB 6F..</td>
</tr>
<tr>
<td></td>
<td>090F9C40</td>
<td>BB 71 ED 16 07 1F..</td>
</tr>
</tbody>
</table>

- In a cache attack, an attack program runs on the same processor as a victim program.
- The attack program measures memory access times to determine which data the victim loaded into cache.
Reminder: Memory caches and cache attacks

Caches hold local (fast) copy of recently-accessed 64-byte chunks of memory.

In a cache attack, an attack program runs on the same processor as a victim program.

The attack program measures memory access times to determine which data the victim loaded into cache.
Reminder: Memory caches and cache attacks

Caches hold local (fast) copy of recently-accessed 64-byte chunks of memory.

<table>
<thead>
<tr>
<th>Set</th>
<th>Addr</th>
<th>Cached Data ~64B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>F0016280 31C6F4C0 339DD740 614F8480</td>
<td>B5 F5 80 21 E3 2C 9A 69 59 11 48 F2 C7 D7 A0 86 67 18 17 4C 59 B8 58 A7 10 4B</td>
</tr>
<tr>
<td>1</td>
<td>71685100 132A4880 2A1C0700 C017E9C0</td>
<td>27 BD 5D 2E 84 29 30 B2 8F 27 05 9C 9E C3 DA EE B7 D9 D1 76 16 54 51 5B</td>
</tr>
<tr>
<td>2</td>
<td>311956C0 002D47C0 91507E80 55194040</td>
<td>0A 55 47 82 86 4E C4 15 4D 78 B5 C4 60 D0 2C DD 78 14 DF 66 E9 D0 11 43</td>
</tr>
<tr>
<td>3</td>
<td>9B27F8C0 BE771100 A001FB40 317178C0</td>
<td>84 A0 7F C7 4E BC 3B 0B 20 0C DB 58 29 D9 F5 6A 72 50 Evict to make room</td>
</tr>
<tr>
<td>4</td>
<td>6618E980 BA0CDB40 89E92C00 090F9C40</td>
<td>35 11 4A E0 2E F1 B0 FC 5A 20 D0 7F 1C 50 A4 F8 EB 6F BB 71 ED 16 07 1F</td>
</tr>
</tbody>
</table>

- In a cache attack, an attack program runs on the same processor as a victim program.
- The attack program measures memory access times to determine which data the victim loaded into cache.
Caches hold local (fast) copy of recently-accessed 64-byte chunks of memory.

### CPU
- Sends address,
- Receives data

### MAIN MEMORY
- Big, slow
e.g. 16GB SDRAM

### MEMORY CACHE

<table>
<thead>
<tr>
<th>Set</th>
<th>Addr</th>
<th>Cached Data ~64B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>F001E6280</td>
<td>B5 F5 80 21 E3 2C..</td>
</tr>
<tr>
<td></td>
<td>31C6F4C0</td>
<td>9A DA 59 11 48 F2..</td>
</tr>
<tr>
<td></td>
<td>339D740</td>
<td>C7 D7 A0 86 67 18..</td>
</tr>
<tr>
<td></td>
<td>614F8480</td>
<td>17 4C 59 B8 58 A7..</td>
</tr>
<tr>
<td>1</td>
<td>71685100</td>
<td>27 BD 5D 2E 84 29..</td>
</tr>
<tr>
<td></td>
<td>132A4880</td>
<td>30 B2 8F 27 05 9C..</td>
</tr>
<tr>
<td></td>
<td>2A1C0700</td>
<td>9E C3 DA EE B7 D9..</td>
</tr>
<tr>
<td></td>
<td>C017E9C0</td>
<td>D1 76 16 54 51 5B..</td>
</tr>
<tr>
<td>2</td>
<td>311596C0</td>
<td>0A 55 47 82 86 4E..</td>
</tr>
<tr>
<td></td>
<td>002D47C0</td>
<td>C4 15 4D 7B 55 C4..</td>
</tr>
<tr>
<td></td>
<td>91507E80</td>
<td>60 D0 2C DD 78 14..</td>
</tr>
<tr>
<td></td>
<td>55194040</td>
<td>DF 66 E9 D0 11 43..</td>
</tr>
<tr>
<td>3</td>
<td>9B27F8C0</td>
<td>84 A0 7F C7 4E BC..</td>
</tr>
<tr>
<td></td>
<td>BE771100</td>
<td>3B 0B 20 0C DB 58..</td>
</tr>
<tr>
<td></td>
<td>A001FB40</td>
<td>29 D9 F5 6A 72 50..</td>
</tr>
<tr>
<td></td>
<td>132E1340</td>
<td>AC 99 17 8F 44 09..</td>
</tr>
<tr>
<td>4</td>
<td>6618E980</td>
<td>35 11 4A E0 2E F1..</td>
</tr>
<tr>
<td></td>
<td>BA0CDB40</td>
<td>B0 FC 5A 20 D0 7F..</td>
</tr>
<tr>
<td></td>
<td>89E92C00</td>
<td>1C 50 A4 F8 EB 6F..</td>
</tr>
<tr>
<td></td>
<td>090F9C40</td>
<td>BB 71 ED 16 07 1F..</td>
</tr>
</tbody>
</table>

- **Address:** 132E1340

- **Data:** AC 99 17 8F 44 09..

---

• In a cache attack, an attack program runs on the same processor as a victim program.

• The attack program measures memory access times to determine which data the victim loaded into cache.
### Reminder: Memory caches and cache attacks

Caches hold local (fast) copy of recently-accessed 64-byte chunks of memory.

<table>
<thead>
<tr>
<th>Set</th>
<th>Addr</th>
<th>Cached Data ~64B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>F0016280</td>
<td>31C6F4C0 C39DD740 614F8480 B5 F5 80 21 E3 2C..</td>
</tr>
<tr>
<td>1</td>
<td>71685100</td>
<td>32A4880 2A1C0700 C017E9C0 27 BD 5D 2E 84 29..</td>
</tr>
<tr>
<td>2</td>
<td>311956C0</td>
<td>002D47C0 91507E80 S5194040 0A 55 47 82 86 4E..</td>
</tr>
<tr>
<td>3</td>
<td>9B27F8C0</td>
<td>BE771100 A001FB40 132E1340 84 A0 7F C7 4E BC..</td>
</tr>
<tr>
<td>4</td>
<td>6618E980</td>
<td>BA0CDB40 89E92C00 090F9C40 35 11 4A E0 2E F1..</td>
</tr>
</tbody>
</table>

**CPU**
- Sends address,
- Receives data

**Addr:** 2A1C0700
- Data: 9E C3 DA EE B7 D3..

**Addr:** 132E1340
- Data: AC 99 17 8F 44 09..

**MEMORY CACHE**
- hash(addr) to map to cache set
- Fast
- Slow

**Address:** 132E1340
- Data: AC 99 17 8F 44 09..

**MAIN MEMORY**
- Big, slow e.g. 16GB SDRAM

- In a cache attack, an attack program runs on the same processor as a victim program.
- The attack program measures memory access times to determine which data the victim loaded into cache.
In a cache attack, an attack program runs on the same processor as a victim program.

The attack program measures memory access times to determine which data the victim loaded into cache.
Reminder: Memory caches and cache attacks

Caches hold local (fast) copy of recently-accessed 64-byte chunks of memory

- In a cache attack, an attack program runs on the same processor as a victim program.
- The attack program measures memory access times to determine which data the victim loaded into cache.
Reminder: Memory caches and cache attacks

Caches hold local (fast) copy of recently-accessed 64-byte chunks of memory

| CPU | Sends address, Receives data |
| Addr: 2A1C0700 |
| Data: 9E C3 DA EE B7 D3.. |

<p>| MEMORY CACHE |</p>
<table>
<thead>
<tr>
<th>Set</th>
<th>Addr</th>
<th>Cached Data ~64B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>F0016280 31C6F4C0 339DD740 614F8480</td>
<td>B5 F5 80 21 E3 2C.. 9A DA 59 11 48 F2.. C7 D7 A0 86 67 18.. 17 4C 59 B8 58 A7..</td>
</tr>
<tr>
<td>1</td>
<td>71685100 132A4880 2A1C0700 C017E9C0</td>
<td>27 BD 5D 2E 84 29.. 30 B2 8F 27 05 9C.. 9E C3 DA EE B7 D9.. D1 76 16 54 51 5B..</td>
</tr>
<tr>
<td>2</td>
<td>311956C0 002D47C0 91507E80 55194040</td>
<td>0A 55 47 82 86 4E.. C4 15 4D 78 B5 C4.. 60 D0 2C DD 78 14.. DF 66 E9 D0 11 43..</td>
</tr>
<tr>
<td>3</td>
<td>9B27F8C0 BE771100 A001FB40 132E1340</td>
<td>84 A0 7F C7 4E BC.. 3B 0B 20 0C DB 58.. 29 D9 F5 6A 72 50..</td>
</tr>
<tr>
<td>4</td>
<td>6618E980 BA0CDB40 89E92C00 090F9C40</td>
<td>35 11 4A E0 2E F1.. B0 FC 5A 20 D0 7F.. 1C 50 A4 F8 EB 6F.. BB 71 ED 16 07 1F..</td>
</tr>
</tbody>
</table>

Address: 132E1340
Data: AC 99 17 8F 44 09.. 

Fast
Addr: 2A1C0700
Data: 9E C3 DA EE B7 D3...
Set
Addr: 132E1340
Data: AC 99 17 8F 44 09...

Evict
Addr: 132E1340
Data: AC 99 17 8F 44 09...

Reads change system state:
- Read to newly-cached location is fast
- Read to evicted location is slow

• In a cache attack, an attack program runs on the same processor as a victim program.
• The attack program measures memory access times to determine which data the victim loaded into cache.

MAIN MEMORY
Big, slow e.g. 16GB SDRAM
Cache Attacks against AES
Bernstein 2005

- AES algorithm consists of xors, shifts, substitutions
- For speed, operations precomputed as a lookup table
- Table queries dependent on key values
- A cache attack can reveal the lookup locations and thus the secret key
Speculative Execution

• CPUs can guess likely program path and do speculative execution
• Example

```python
if (uncached_value == 1) // load from memory
    a = compute(b)
```

• Branch predictor guesses if() is true based on prior history
• Starts executing compute(b) speculatively
• When value arrives from memory, check if guess was correct:
  • Correct: Save speculative work $\rightarrow$ performance gain
  • Incorrect: Discard speculative work $\rightarrow$ no harm?
Spectre and Meltdown
Lipp et al., Kocher et al. 2017

Misspeculation

- Exceptions and incorrect branch prediction can cause “rollback” of transient instructions
- Old register states are preserved, can be restored
- Memory writes are buffered, can be discarded
- Cache modifications are not restored

- Spectre and Meltdown carry out cache attacks against speculatively loaded data so that an unprivileged attacker process can read kernel memory, break ASLR, etc.
Spectre variant 1 attack

Before attack:

- Train branch predictor to expect if() is true (e.g. call with \( x < \text{array1\_size} \))
- Evict \( \text{array1\_size} \) and \( \text{array2[]} \) from cache

```java
if (x < array1_size)
    y = array2[array1[x]*4096];
```
Spectre variant 1 attack

if (x < array1_size)
    y = array2[array1[x]*4096];

Attacker calls victim with x=1000
Spectre variant 1 attack

```c
if (x < array1_size)
y = array2[array1[x]*4096];
```

Attacker calls victim with \( x=1000 \)
if (x < array1_size)
    y = array2[array1[x]*4096];

Attacker calls victim with x=1000
Speculative exec while waiting for array1_size:

Memory & Cache Status
array1_size = 00000008

Memory at array1 base:
  8 bytes of data (value doesn’t matter)
Memory at array1 base+1000:
  09 F1 98 CC 90... (something secret)

array2[ 0*4096]
array2[ 1*4096]
array2[ 2*4096]
array2[ 3*4096]
array2[ 4*4096]
array2[ 5*4096]
array2[ 6*4096]
array2[ 7*4096]
array2[ 8*4096]
array2[ 9*4096]
array2[10*4096]
array2[11*4096]
...

Contents don’t matter
only care about cache status
Uncached  Cached
Spectre variant 1 attack

if (x < array1_size)
  y = array2[array1[x] * 4096];

Attacker calls victim with x=1000
Speculative exec while waiting for array1_size:
  › Predict that if() is true
Spectre variant 1 attack

```c
if (x < array1_size)
    y = array2[array1[x]*4096];
```

Attacker calls victim with \( x = 1000 \)

Speculative exec while waiting for `array1_size`:
- Predict that `if()` is true
- Read address `array1 base + x` (using out-of-bounds \( x = 1000 \))

### Memory & Cache Status

- `array1_size = 00000008`
- Memory at `array1 base`:
  - 8 bytes of data (value doesn’t matter)
- Memory at `array1 base+1000`:
  - `09 F1 98 CC 90...` (something secret)

- `array2[ 0*4096]`
- `array2[ 1*4096]`
- `array2[ 2*4096]`
- `array2[ 3*4096]`
- `array2[ 4*4096]`
- `array2[ 5*4096]`
- `array2[ 6*4096]`
- `array2[ 7*4096]`
- `array2[ 8*4096]`
- `array2[ 9*4096]`
- `array2[10*4096]`
- `array2[11*4096]`
- ...

Contents don’t matter only care about cache status

- Uncached
- Cached
Spectre variant 1 attack

```c
if (x < array1_size)
    y = array2[array1[x] * 4096];
```

Attacker calls victim with $x=1000$

Speculative exec while waiting for `array1_size`:

- Predict that if() is true
- Read address (`array1` base + $x$) (using out-of-bounds $x=1000$)
- Read returns secret byte = 09 (in cache ⇒ fast)

Memory & Cache Status

```
allocation
Memory & Cache Status

array1_size = 00000008

Memory at array1 base:
  8 bytes of data (value doesn't matter)
Memory at array1 base+1000:

  09
F1 98 CC 90... (something secret)
```

Contents don't matter only care about cache status

- Uncached
- Cached
Spectre variant 1 attack

Attacker calls victim with \(x=1000\)

Next:
- Request mem at \((\text{array2 base} + 09 \times 4096)\)
- Brings \(\text{array2}[09 \times 4096]\) into the cache
- Realize if() is false: discard speculative work

Finish operation & return to caller
Spectre variant 1 attack

```c
if (x < array1_size)
    y = array2[array1[x]*4096];
```

Attacker calls victim with \( x=1000 \)

Attacker:
- measures read time for \( \text{array2}[i*4096] \)
- Read for \( i=09 \) is fast (cached), reveals secret byte !
- Repeat with many \( x \) (10KB/s)
Fault Attacks

“Using Memory Errors to Attack a Virtual Machine” Govindavajhala Appel 2003

Java heap overflow via glitched address of function pointer.

Figure 3. Experimental setup to induce memory errors, showing a PC built from surplus components, clip-on gooseneck lamp, 50-watt spotlight bulb, and digital thermometer. Not shown is the variable AC power supply for the lamp.
Types of RAM

- **Volatile memory**: Data retained only as long as power is on.
- **Persistent memory** like flash or magnetic disks retains data without power.

**SRAM**

- SRAM retains bit value as long as power is on without any refresh.
- Faster, lower density, higher cost.
- Has a “burn-in” phenomenon where on startup it tends to flip bit to “remembered bit”.

**DRAM**

- DRAM requires periodic refresh to retain stored value.
- Capacitors charged to store data.
- Higher density, lowered cost.
- “Cold boot attacks” exploited capacitor discharge time to read sensitive data from physical memory.
Rowhammer attacks
Seaborn and Dullien 2015

- DRAM cells are grouped into rows
- All cells in a row are refreshed together

- Repeatedly opening and closing a row within a refresh interval causes disturbance errors in adjacent rows.
- Attacker running attack process on same machine as victim can cause bits to flip in victim’s memory
Mitigating timing side channels

• Eliminating all side-channels is practically impossible

• We can eliminate or mitigate some channels

➤ How do we choose?

➤ Attacker model + impact of defense + cost of defense
Sweet spot: timing channels

• Good for the attacker:
  ➤ Remote attackers can exploit timing channels
  ➤ Co-located attacker (on same physical machine) can abuse cache to amplify these attacks

• Good for defense
  ➤ Can eliminate timing channels
  ➤ Performance overhead of doing so is reasonable
To understand how to eliminate the channels we need to understand what introduces time variability.
Which runs faster?

```c
void foo(double x) {
    double z, y = 1.0;
    for (uint32_t i = 0; i < 100000000; i++) {
        z = y*x;
    }
}
```

A: foo(1.0);

B: foo(1.0e-323);

C: They take the same amount of time!

Code from D. Kohlbrenner
Which runs faster?

```c
void foo(double x) {
    double z, y = 1.0;
    for (uint32_t i = 0; i < 100000000; i++) {
        z = y*x;
    }
}
```

A: `foo(1.0);` ←

B: `foo(1.0e-323);`

C: They take the same amount of time!

Code from D. Kohlbrenner
Why? Floating-point time variability

<table>
<thead>
<tr>
<th>Processor</th>
<th>+ subnormal</th>
<th>+ special</th>
<th>× subnormal</th>
<th>× special</th>
<th>÷ subnormal</th>
<th>÷ special</th>
<th>√ subnormal</th>
<th>√ special</th>
<th>√x²</th>
<th>√x³</th>
<th>√−x</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel Core i7-7700 (Kaby Lake)</td>
<td>✔</td>
<td>✔</td>
<td>❌</td>
<td>❌</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>❌</td>
<td>❌</td>
<td>❌</td>
</tr>
<tr>
<td>Intel Core i7-6700K (Skylake)</td>
<td>✔</td>
<td>✔</td>
<td>❌</td>
<td>❌</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>❌</td>
<td>❌</td>
<td>❌</td>
</tr>
<tr>
<td>Intel Core i7-3667U (Ivy Bridge)</td>
<td>✔</td>
<td>✔</td>
<td>❌</td>
<td>❌</td>
<td>❌</td>
<td>❌</td>
<td>❌</td>
<td>❌</td>
<td>❌</td>
<td>❌</td>
<td>❌</td>
</tr>
<tr>
<td>Intel Xeon X5660 (Westmere)</td>
<td>✔</td>
<td>✔</td>
<td>❌</td>
<td>❌</td>
<td>❌</td>
<td>❌</td>
<td>❌</td>
<td>❌</td>
<td>❌</td>
<td>❌</td>
<td>❌</td>
</tr>
<tr>
<td>Intel Atom D2550 (Cedarview)</td>
<td>✔</td>
<td>✔</td>
<td>❌</td>
<td>❌</td>
<td>❌</td>
<td>❌</td>
<td>❌</td>
<td>❌</td>
<td>❌</td>
<td>❌</td>
<td>❌</td>
</tr>
<tr>
<td>AMD Phenom II X6 1100T</td>
<td>✔</td>
<td>✔</td>
<td>❌</td>
<td>❌</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>❌</td>
<td>❌</td>
<td>❌</td>
</tr>
<tr>
<td>AMD Ryzen 7 1800x</td>
<td>✔</td>
<td>✔</td>
<td>❌</td>
<td>❌</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>❌</td>
<td>❌</td>
<td>❌</td>
</tr>
</tbody>
</table>

**Single-precision operations**

**Double-precision operations**
Some instructions introduce time variability

- **Problem:** Certain instructions take different amounts of time depending on the operands
  - If input data is secret: might leak some of it!

- **Solution?**
  - In general, don’t use variable-time instructions
When ARMv8.4-DIT is implemented:

Data Independent Timing.

<table>
<thead>
<tr>
<th>DIT</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b0</td>
<td>The architecture makes no statement about the timing properties of any instructions.</td>
</tr>
</tbody>
</table>
| 0b1 | The architecture requires that:  
  - The timing of every load and store instruction is insensitive to the value of the data being loaded or stored.  
  - For certain data processing instructions, the instruction takes a time which is independent of:  
    - The values of the data supplied in any of its registers.  
    - The values of the NZCV flags.  
  - For certain data processing instructions, the response of the instruction to asynchronous exceptions does not vary based on:  
    - The values of the data supplied in any of its registers.  
    - The values of the NZCV flags. |

The data processing instructions affected by this bit are:

- All cryptographic instructions. These instructions are:
  - AESD, AESE, AESIMC, AESMC, SHA1C, SHA1H, SHA1M, SHA1P, SHA1SU0, SHA1SU1, SHA256H, SHA256H2, SHA256SU0, and SHA256SU1.
- A subset of those instructions which use the general-purpose register file. For these instructions, the effects of CPSR.DIT apply only if they do not use R15 as either their source or destination and pass their condition execution check. The instructions are:
  - BFI, BFC, CLZ, CMN, CMP, MLA, MLAS, MLS, MOVT, MUL, MULS, NOP, PKHBT, PKHTB, RBIT, REV, REV16, REVSH, RRX, SADD16, SADD8, SASX, SBFX, SHADD16, SHADD8, SHASX, SHSAX, SHSUB16, SHSUB8, SMLAL1*, SMLAW1*, SMLSD1*, SMMLA1*, SMMLS1*, SMMLU1*, SMMLU1*, SMUAD1*, SMUL1*, SSAX, SSHU8, SSHU16, SSHU8, SXTAB1*, SXTAH, SXTB*, SXTH, TEO, TST, UADD1*, UASX, UBFX, UHADD1*, UHASX, UHSAX, UHSUB1*, UMAAL, UMLAL, UMLALS, UMLUL, UMULL, UMULLS, USADA8, USAX, USUB1*, UXTAB1*, UXTAH, UXTH, ADC (register-shifted register), ADCS (register-shifted register), ADD (register-shifted register), ADDS (register-shifted register), AND (register-shifted register), ANDS (register-shifted register), ASR (register-shifted register), ASRS (register-shifted register), BIC (register-shifted register), BICS (register-shifted register), EOR (register-shifted register), EORS (register-shifted register), LSL (register-shifted register), LSLS (register-shifted register), LSR (register-shifted register), LSRS (register-shifted register), MOV (register-shifted register), MOVVS (register-shifted register), MVN (register-shifted register), MVNS (register-shifted register), ORR (register-shifted register), ORRS (register-shifted register), ROR (register-shifted register), RORS (register-shifted register), RSBR (register-shifted register), RSBS (register-shifted register), RSC (register-shifted register), RCS (register-shifted register), SBC (register-shifted register), SBCS (register-shifted register), SUB (register-shifted register), and SUBS (register-shifted register).
Control flow introduces time variability

m = 1
for i = 0 ... len(d):
    if d[i] = 1:
        m = c * m mod N
        m = square(m) mod N
    return m
if-statements on secrets are unsafe

```c
s0;
if (secret) {
    s1;
    s2;
}
s3;
```

<table>
<thead>
<tr>
<th>secret</th>
<th>run</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>s0;s1;s2;s3;</td>
</tr>
<tr>
<td>false</td>
<td>s0;s3;</td>
</tr>
</tbody>
</table>

4

2
Can we pad else branch?

```java
if (secret) {
    s1;
    s2;
} else {
    s1';
    s2';
}
```

where s1 and s1’ take same amount of time
Why padding branches doesn’t work

- **Problem:** Instructions are loaded from cache
  - Which instructions were loaded (or not) observable

- **Problem:** Hardware tried to predict where branch goes
  - Success (or failure) of prediction is observable

- **What can we do?**
Don’t branch on secrets!

Real code needs to branch...
Fold control flow into data flow

(assumption secret = 1 or 0)

```java
if (secret) {
    x = a;
}
```

$x = \text{secret} \times a + (1 - \text{secret}) \times x$;
Fold control flow into data flow

(assumption secret = 1 or 0)

if (secret) {
    x = a;
} else {
    x = b;
}

\[
\begin{align*}
    &x = \text{secret} \times a \\
    &+ (1-\text{secret}) \times x;
\end{align*}
\]

\[
\begin{align*}
    &x = (1-\text{secret}) \times b \\
    &+ \text{secret} \times x;
\end{align*}
\]
Fold control flow into data flow

- Multiple ways to fold control flow into data flow
  - Previous example: takes advantage of arithmetic
  - What’s another way?

```c
/* Constant-time helper macro that selects l or r depending on all-1 or all-0
 * mask m */
#define CT_SEL(m, l, r) (((m) & (l)) | (~(m) & (r)))
```
An example from mbedTLS

Goal: get the length of the padding so we can remove it
An example from mbedTLS

```c
static int get_zeros_padding(unsigned char *input, size_t input_len,
                        size_t *data_len)
{
    size_t i;

    if (NULL == input || NULL == data_len)
        return(MBEDTLS_ERR_CIPHER_BAD_INPUT_DATA);

    *data_len = 0;
    for (i = input_len; i > 0; i-- ) {
        if (input[i-1] != 0) {
            *data_len = i;
            return 0;
        }
    }

    return 0;
}
```
An example from mbedTLS

```c
static int get_zeros_padding( unsigned char *input, size_t input_len,
                              size_t *data_len )
{
    size_t i;

    if( NULL == input || NULL == data_len )
        return( MBEDTLS_ERR_CIPHER_BAD_INPUT_DATA );

    *data_len = 0;
    for( i = input_len; i > 0; i-- ) {
        if (input[i-1] != 0) {
            *data_len = i;
            return 0;
        }
    }

    return 0;
}
```

Is this safe?
An example from mbedTLS

```c
static int get_zeros_padding( unsigned char *input, size_t input_len, size_t *data_len )
{
    size_t i;

    if( NULL == input || NULL == data_len )
        return( MBEDTLS_ERR_CIPHER_BAD_INPUT_DATA );

    *data_len = 0;
    for( i = input_len; i > 0; i-- ) {
        if (input[i-1] != 0) {
            *data_len = i;
            return 0;
        }
    }

    return 0;
}
```

Is this safe?
An example from mbedTLS

```c
static int get_zeros_padding( unsigned char *input, size_t input_len, 
    size_t *data_len )
{
    size_t i
    unsigned done = 0, prev_done = 0;

    if( NULL == input || NULL == data_len )
        return( MBEDTLS_ERR_CIPHER_BAD_INPUT_DATA );

    *data_len = 0;
    for( i = input_len; i > 0; i-- ) {
        prev_done = done;
        done |= input[i-1] != 0;
        if (done & !prev_done) {
            *data_len = i;
        }
    }

    return 0;
}
```

Is this safe?
An example from mbedTLS

```c
static int get_zeros_padding( unsigned char *input, size_t input_len,
                               size_t *data_len )
{
    size_t i
    unsigned done = 0, prev_done = 0;

    if( NULL == input || NULL == data_len )
        return( MBEDTLS_ERR_CIPHER_BAD_INPUT_DATA );

    *data_len = 0;
    for( i = input_len; i > 0; i-- ) {
        prev_done = done;
        done |= input[i-1] != 0;
        if (done & !prev_done) {
            *data_len = i;
        }
    }

    return 0;
}
```

Is this safe?
An example from mbedTLS

static int get_zeros_padding( unsigned char *input, size_t input_len, size_t *data_len )
{
    size_t i
    unsigned done = 0, prev_done = 0;

    if( NULL == input || NULL == data_len )
        return( MBEDTLS_ERR_CIPHER_BAD_INPUT_DATA );

    *data_len = 0;
    for( i = input_len; i > 0; i-- ) {
        prev_done = done;
        done |= input[i-1] != 0;
        *data_len = CT_SEL(done & !prev_done, i, *data_len);
    }

    return 0;
}
Control flow introduces time variability

- **Problem:** Control flow that depends on secret data can lead to information leakage
  - Loops
  - If-statements (switch, etc.)
  - Early returns, goto, break, continue
  - Function calls

- **Solution:** control flow should not depend on secrets, fold secret control flow into data!
static void KeyExpansion(uint8_t* RoundKey, const uint8_t* Key) {
...
// All other round keys are found from the previous round keys.
for (i = Nk; i < Nb * (Nr + 1); ++i) {
...
k = (i - 1) * 4;
tempa[0] = RoundKey[k + 0];
tempa[1] = RoundKey[k + 1];
tempa[2] = RoundKey[k + 2];
tempa[3] = RoundKey[k + 3];
...
tempa[0] = sbox[tempa[0]];
tempa[1] = sbox[tempa[1]];
...}
How do we fix this?

- Only access memory at public index

- How do we express `arr[secret]`?

  \[
  x = arr[secret] \quad \Rightarrow \quad \text{for} (\text{size}_t \ i = 0; \ i < \text{arr}_\text{len}; \ i++) \quad x = \text{CT}_\text{SEL}(\text{EQ}(\text{secret}, \ i), \ arr[i], \ x)
  \]

/* Constant-time helper macro that selects \( l \) or \( r \) depending on all-1 or all-0
* mask \( m \) */

#define CT_SEL(m, l, r) (((m) & (l)) | (~(m) & (r)))
Summary: what introduces time variability?

• Duration of certain operations depends on data
  ➤ Do not use operators that are variable time

• Control flow
  ➤ Do not branch based on a secret

• Memory access
  ➤ Do not access memory based on a secret
Solution: constant-time programming

- Duration of certain operations depends on data
  - Transform to safe, known CT operations
- Control flow
  - Turn control flow into data flow problem: select!
- Memory access
  - Loop over public bounds of array!
Writing CT code is unholy

OpenSSL padding oracle attack

Writing CT code is unholy

OpenSSL padding oracle attack

Writing CT code is unholy

OpenSSL padding oracle attack

Lucky 13 timing attack
Writing CT code is unholy

OpenSSL padding oracle attack


Lucky 13 timing attack

Writing CT code is unholy

OpenSSL padding oracle attack

Lucky 13 timing attack
Writing CT code is unholy

OpenSSL padding oracle attack

Lucky 13 timing attack

CVE-2016-2107
Somorovsky. “Curious padding oracle in OpenSSL.”
What can we do about this?

• Design new programming languages!

➤ E.g., FaCT language lets you write code that is guaranteed to be constant time

```plaintext
export
donkey get_zeros_padding( secret uint8 input[], secret mut uint32 data_len) {

data_len = 0;
for( uint32 i = len input; i > 0; i-=1 ) {
    if (input[i-1] != 0) {
        data_len = i;
        return;
    }
}
```

Automatically transform code when possible!

```c
export void conditional_swap(secret mut uint32 x,
   secret mut uint32 y,
   secret bool cond) {

   secret mut bool __branch1 = cond;
   { // then part
   secret uint32 tmp = x;
   x = CT_SEL(__branch1, y, x);
   y = CT_SEL(__branch1, tmp, y);
   }

   __branch1 = !__branch1;
   { ... else part ...}
}
```
Raise type error otherwise!

- Some transformations not possible
  - E.g., loops bounded by secret data
- Some transformations would produce slow code
  - E.g., accessing array at secret index
What about existing code?

• Program analysis to the rescue!

➤ Symbolically execute a function and alert if you find an input that causes it to (1) branch on a secret (2) use secret as memory index or (3) use variable-time instruction

```java
if (cond₀) {
    arr[33] = 5;
} else {
    if (cond₁) {
        arr[secret] = 42;
    } else {
        arr[44] = 3;
    }
}
```
What about existing code?

- Program analysis to the rescue!

  ➤ Symbolically execute a function and alert if you find an input that causes it to (1) branch on a secret (2) use secret as memory index or (3) use variable-time instruction

```c
if (cond_0) {
    arr[33] = 5;
} else {
    if (cond_1) {
        arr[secret] = 42;
    } else {
        arr[44] = 3;
    }
}
```
What about existing code?

- Program analysis to the rescue!

  Symbolically execute a function and alert if you find an input that causes it to (1) branch on a secret (2) use secret as memory index or (3) use variable-time instruction

```c
if (cond₀) {
    arr[33] = 5;
} else {
    if (cond₁) {
        arr[secret] = 42;
    } else {
        arr[44] = 3;
    }
}
```

```
cond₀ = false
arr[33] = 55;
cond₁ = true
arr[secret] = 42;
arr[44] = 3;
```
What about existing code?

• Program analysis to the rescue!
  ➤ Symbolically execute a function and alert if you find an input that causes it to (1) branch on a secret (2) use secret as memory index or (3) use variable-time instruction

• Can even do this for branches not taken and find leaks via speculative execution!