Software side-channel attacks and fault attacks on ARM devices

Clémentine Maurice, CNRS, IRISA, EMSEC
February 18, 2020–SILM seminar, Rennes, France
Software-based side channels and fault attacks

- power consumption, electromagnetic leaks, lasers, glitching...
Software-based side channels and fault attacks

- Power consumption, electromagnetic leaks, lasers, glitching...
- Targeted attacks, physical access
  - Mostly performed on embedded devices
- Timing attacks, microarchitectural attacks, software-based fault attacks...
  - No physical access required
  - Require code co-location
Software-based side channels and fault attacks

- power consumption, electromagnetic leaks, lasers, glitching...
  - targeted attacks, physical access
  - mostly performed on embedded devices
Software-based side channels and fault attacks

• power consumption, electromagnetic leaks, lasers, glitching…
  → targeted attacks, physical access
  → mostly performed on embedded devices

• timing attacks, microarchitectural attacks, software-based fault attacks…
Software-based side channels and fault attacks

- power consumption, electromagnetic leaks, lasers, glitching...
  - targeted attacks, physical access
  - mostly performed on embedded devices

- timing attacks, microarchitectural attacks, software-based fault attacks...
  - no physical access required
  - require code co-location
Side-channel attacks
Cache attacks

• cache attacks $\rightarrow$ exploit timing differences of memory accesses
Cache attacks

- cache attacks → exploit timing differences of memory accesses
- attacker monitors which lines are accessed, not the content
Cache attacks

- cache attacks → exploit timing differences of memory accesses
- attacker monitors which lines are accessed, **not the content**
- covert channel: two processes **communicating** with each other
  - **not allowed** to do so
Cache attacks

- cache attacks \(\rightarrow\) exploit timing differences of memory accesses
- attacker monitors which lines are accessed, **not the content**
- covert channel: two processes **communicating** with each other
  - **not allowed** to do so
- side-channel attack: one malicious process **spies** on benign processes
  - e.g., steals crypto keys, spies on keystrokes
Cache attacks

- cache attacks → exploit timing differences of memory accesses
- attacker monitors which lines are accessed, not the content
- covert channel: two processes communicating with each other
  - not allowed to do so
- side-channel attack: one malicious process spies on benign processes
  - e.g., steals crypto keys, spies on keystrokes
- different techniques can be used for both covert channels and side-channel attacks
Cache attack: Flush+Reload

**Step 1:** Attacker maps shared library (shared memory, in cache)
**Cache attack: Flush+Reload**

**Step 1:** Attacker maps shared library (shared memory, in cache)
**Cache attack: Flush+Reload**

**Step 1:** Attacker maps shared library (shared memory, in cache)

**Step 2:** Attacker **flushes** the shared cache line

---

*Note: Diagram shows the victim address space, cache, and attacker address space.*
Cache attack: Flush+Reload

**Step 1:** Attacker maps shared library (shared memory, in cache)

**Step 2:** Attacker flushes the shared cache line

**Step 3:** Victim loads the data
**Cache attack: Flush+Reload**

**Step 1:** Attacker maps shared library (shared memory, in cache)

**Step 2:** Attacker *flushes* the shared cache line

**Step 3:** Victim loads the data

**Step 4:** Attacker *reloads* the data
Cache attacks: Prime+Probe

Victim address space

Cache

Attacker address space
Cache attacks: Prime+Probe

**Step 1:** Attacker primes, *i.e.*, fills, the cache (no shared memory)
Step 1: Attacker primes, i.e., fills, the cache (no shared memory)
Step 2: Victim evicts cache lines while running
Step 1: Attacker primes, *i.e.*, fills, the cache (no shared memory)

Step 2: Victim evicts cache lines while running
Cache attacks: Prime+Probe

Step 1: Attacker primes, i.e., fills, the cache (no shared memory)

Step 2: Victim evicts cache lines while running

Step 3: Attacker probes data to determine if set has been accessed
Cache attacks: Prime+Probe

**Step 1:** Attacker primes, *i.e.*, fills, the cache (no shared memory)

**Step 2:** Victim evicts cache lines while running

**Step 3:** Attacker probes data to determine if set has been accessed
A bit of history

• 2005: first cache attacks on Intel
• 2013: first cache attack on Android (ARM)
  → attack requires privileges
• 2016: first unprivileged cache attack on ARM

Even today most attacks target Intel

---
Challenge #1 no flush instruction on ARMv7-A
→ use cache eviction

Challenge #1  no flush instruction on ARMv7-A
  →  use cache eviction

Challenge #2  pseudo-random replacement policy
  →  eviction strategies that use multiple accesses

**Challenge #1** no flush instruction on ARMv7-A
→ use cache eviction

**Challenge #2** pseudo-random replacement policy
→ eviction strategies that use multiple accesses

**Challenge #3** privileged cycle counter
→ thread counter (nano-second resolution)

---

Intel vs ARM

**Challenge #1** no flush instruction on ARMv7-A
   → use cache eviction

**Challenge #2** pseudo-random replacement policy
   → eviction strategies that use multiple accesses

**Challenge #3** privileged cycle counter
   → thread counter (nano-second resolution)

**Challenge #4** non-inclusive caches
   → abuse cache coherency protocol for shared memory

---

Intel vs ARM

**Challenge #1** no flush instruction on ARMv7-A
→ use cache eviction

**Challenge #2** pseudo-random replacement policy
→ eviction strategies that use multiple accesses

**Challenge #3** privileged cycle counter
→ thread counter (nano-second resolution)

**Challenge #4** non-inclusive caches
→ abuse cache coherency protocol for shared memory

**Challenge #5** no shared cache between multiple CPUs (big.LITTLE)
→ abuse cache coherency protocol for shared memory

• people tend to use their browser for everything on personal computers

https://techcrunch.com/2017/05/04/report-smartphone-owners-are-using-9-apps-per-day-30-per-month/
Threat model

- people tend to use their browser for everything on personal computers
- people tend to install a new app for everything on mobile devices

https://techcrunch.com/2017/05/04/report-smartphone-owners-are-using-9-apps-per-day-30-per-month/
Threat model

- people tend to use their browser for everything on personal computers
- people tend to install a new app for everything on mobile devices
- apps on mobile devices have specific permissions

https://techcrunch.com/2017/05/04/report-smartphone-owners-are-using-9-apps-per-day-30-per-month/
Threat model

- people tend to use their browser for everything on personal computers
- people tend to install a new app for everything on mobile devices
- apps on mobile devices have specific permissions
  → **covert channels** make a lot of sense in this context!

https://techcrunch.com/2017/05/04/report-smartphone-owners-are-using-9-apps-per-day-30-per-month/
Threat model

• people tend to use their browser for everything on personal computers
• people tend to install a new app for everything on mobile devices
• apps on mobile devices have specific permissions
 → covert channels make a lot of sense in this context!
 → e.g., one app transmitting covertly contact information to another app that does not have the permission

https://techcrunch.com/2017/05/04/report-smartphone-owners-are-using-9-apps-per-day-30-per-month/
Covert channels

- Flush+Reload, Evict+Reload and Flush+Flush, cross-core and cross-CPU
- faster than non-microarchitectural techniques

<table>
<thead>
<tr>
<th>Work</th>
<th>Type</th>
<th>Bandwidth [bps]</th>
<th>Error rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Schlegel et al.</td>
<td>Vibration settings</td>
<td>87</td>
<td>–</td>
</tr>
<tr>
<td>Schlegel et al.</td>
<td>Volume settings</td>
<td>150</td>
<td>–</td>
</tr>
<tr>
<td>Schlegel et al.</td>
<td>File locks</td>
<td>685</td>
<td>–</td>
</tr>
<tr>
<td>Marforio et al.</td>
<td>UNIX socket discovery</td>
<td>2600</td>
<td>–</td>
</tr>
<tr>
<td>Marforio et al.</td>
<td>Type of Intents</td>
<td>4300</td>
<td>–</td>
</tr>
<tr>
<td>Ours (OnePlus One)</td>
<td>Evict+Reload, cross-core</td>
<td>12537</td>
<td>5.00%</td>
</tr>
<tr>
<td>Ours (Alcatel One Touch Pop 2)</td>
<td>Evict+Reload, cross-core</td>
<td>13618</td>
<td>3.79%</td>
</tr>
<tr>
<td>Ours (Samsung Galaxy S6)</td>
<td>Flush+Flush, cross-core</td>
<td>178292</td>
<td>0.48%</td>
</tr>
<tr>
<td>Ours (Samsung Galaxy S6)</td>
<td>Flush+Reload, cross-CPU</td>
<td>257509</td>
<td>1.83%</td>
</tr>
<tr>
<td>Ours (Samsung Galaxy S6)</td>
<td>Flush+Reload, cross-core</td>
<td>1140650</td>
<td>1.10%</td>
</tr>
</tbody>
</table>
A note on Meltdown


<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel</td>
<td>●</td>
<td>●</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>ARM</td>
<td>●</td>
<td>○</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</td>
<td>○</td>
<td>○</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>

Symbols indicate whether at least one CPU model is vulnerable (filled) vs. no CPU is known to be vulnerable (empty). Glossary: reproduced (● vs. ○), first shown in this paper (⭐ vs. ⭐), not applicable (_). All tests performed without defenses enabled.
# A note on Spectre

<table>
<thead>
<tr>
<th>Method</th>
<th>Attack</th>
<th>Spectre-PHT</th>
<th>Spectre-BTB</th>
<th>Spectre-RSB</th>
<th>Spectre-STL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel</td>
<td>intra-process</td>
<td>● [48, 50]★</td>
<td>● [59]</td>
<td>● [29]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>out-of-place</td>
<td>● [13]</td>
<td>● [52, 59]</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>in-place</td>
<td>● [13, 50]</td>
<td>● [52, 59]</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>out-of-place</td>
<td>● [50]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ARM</td>
<td>intra-process</td>
<td>● [48, 50]★</td>
<td>● [6]</td>
<td>● [6]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>out-of-place</td>
<td>● [6]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>in-place</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>out-of-place</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>in-place</td>
<td>● [6, 50]★</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>out-of-place</td>
<td>● [6, 50]★</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AMD</td>
<td>intra-process</td>
<td>● [50]</td>
<td>●</td>
<td>● [29]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>out-of-place</td>
<td>●</td>
<td>●</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>in-place</td>
<td>●</td>
<td>●</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>out-of-place</td>
<td>●</td>
<td>●</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Symbols indicate whether an attack is possible and known (●), not possible and known (○), possible and previously unknown or not shown (★), or tested and did not work and previously unknown or not shown (☆). All tests performed with no defenses enabled.

---

Software-based fault attacks
DRAM organization
DRAM organization

- Channel 0
- Channel 1
DRAM organization

channel 0

channel 1

back of DIMM: rank 1

front of DIMM: rank 0
DRAM organization

channel 0

back of DIMM: rank 1

front of DIMM: rank 0

chip

channel 1
DRAM organization

- bits in cells in rows
- access: activate row, copy to row buffer
“It’s like breaking into an apartment by repeatedly slamming a neighbor’s door until the vibrations open the door you were after” – Motherboard Vice

“It’s like breaking into an apartment by repeatedly slamming a neighbor’s door until the vibrations open the door you were after” – Motherboard Vice

“It’s like breaking into an apartment by repeatedly slamming a neighbor’s door until the vibrations open the door you were after” – Motherboard Vice

“It’s like breaking into an apartment by repeatedly slamming a neighbor’s door until the vibrations open the door you were after” – Motherboard Vice

“It’s like breaking into an apartment by repeatedly slamming a neighbor’s door until the vibrations open the door you were after” – Motherboard Vice

“It’s like breaking into an apartment by repeatedly slamming a neighbor’s door until the vibrations open the door you were after” – Motherboard Vice

Impact of the CPU cache

- only **non-cached accesses** reach DRAM
- original attacks use `clflush` instruction
  → flush line from cache
  → next access will be served from DRAM
How to reach DRAM (x86 and ARM)?

1. clflush instruction → original paper (Kim et al.)
2. non-temporal accesses (Qiao et al.)
3. cache eviction → Prime+Probe (Gruss et al., Aweke et al., Frigo et al.)
4. uncached memory (van der Veen et al.)

---

P. Frigo et al. “Grand Pwning Unit: Accelerating Microarchitectural Attacks with the GPU”. In: S&P. 2018.
What works on ARM?

1. ARMv7 flush instruction is privileged \( \times \)
2. ARMv8 non-temporal stores are still cached in practice \( \times \)
3. Cache eviction seems to be too slow, except when using GPUs
   \( \rightarrow \) also works in browsers using WebGL \( \checkmark \)
4. Apps can use \( /\text{dev/ion} \) for **uncached**, physically contiguous memory, without any privilege or permission needed (since Android 4.0) \( \checkmark \)
Conclusion

• some differences in techniques used for side-channel attacks and fault attacks on x86 and ARM…
• … but nothing fundamentally different
• attacks based on optimizations
• how to get rid of the attacks while keeping the optimizations?