ARMageddon: Cache Attacks on Mobile Devices

Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, Stefan Mangard
Graz University of Technology

August 11, 2016 — Usenix Security 2016
TLDR

- powerful cache attacks (like Flush+Reload) on x86
- why not on ARM?
TLDR

- powerful cache attacks (like Flush+Reload) on x86
- why not on ARM?

We identified and solved challenges systematically to:

- make all cache attack techniques applicable to ARM
- monitor user activity
- attack weak Android crypto
- show that ARM TrustZone leaks through the cache
What is a cache attack? (1)
What is a cache attack? (2)
What is a cache attack? (2)
What is a cache attack? (2)
What is a cache attack? (2)
What is a cache attack? (2)
What is a cache attack? (2)
What is a cache attack? (2)
What is a cache attack? (2)
Cache attack techniques

Most important techniques:

- Flush+Reload
- Prime+Probe

Both work on the last-level cache → across cores
Flush+Reload

step 0: attacker maps shared library $\rightarrow$ shared memory, shared in cache
Flush+Reload

**Step 0:** attacker maps shared library $\rightarrow$ shared memory, shared in cache
Flush+Reload

step 0: attacker maps shared library → shared memory, shared in cache
step 1: attacker flushes the shared line
Flush+Reload

**step 0**: attacker maps shared library → shared memory, shared in cache

**step 1**: attacker flushes the shared line

**step 2**: victim loads data while performing encryption
Flush+Reload

step 0: attacker maps shared library $\rightarrow$ shared memory, shared in cache
step 1: attacker flushes the shared line
step 2: victim loads data while performing encryption
step 3: attacker reloads data $\rightarrow$ fast access if the victim loaded the line
Prime+Probe

step 0: attacker fills the cache (prime)
Prime+Probe

**Step 0**: attacker fills the cache (prime)
Prime+Probe

**step 0**: attacker fills the cache (prime)
Prime+Probe

step 0: attacker fills the cache (prime)
step 1: victim evicts cache lines while performing encryption
**Prime+Probe**

**Step 0:** Attacker fills the cache (prime)

**Step 1:** Victim evicts cache lines while performing encryption
Prime+Probe

**step 0**: attacker fills the cache (prime)

**step 1**: victim evicts cache lines while performing encryption
Prime+Probe

**step 0**: attacker fills the cache (prime)

**step 1**: victim evicts cache lines while performing encryption
**Prime+Probe**

**Attacker address space**

**Cache**

**Victim address space**

**step 0**: attacker fills the cache (prime)

**step 1**: victim evicts cache lines while performing encryption
Prime+Probe

**Step 0:** Attacker fills the cache (prime)

**Step 1:** Victim evicts cache lines while performing encryption

**Step 2:** Attacker probes data to determine if the set was accessed
Prime+Probe

**step 0**: attacker fills the cache (prime)
**step 1**: victim evicts cache lines while performing encryption
**step 2**: attacker probes data to determine if the set was accessed
Prime+Probe

**step 0**: attacker fills the cache (prime)
**step 1**: victim evicts cache lines while performing encryption
**step 2**: attacker probes data to determine if the set was accessed
Caches on Intel CPUs

last-level cache (L3):
  - shared
  - inclusive

= shared memory is shared in cache, across cores!
Caches on ARM Cortex-A CPUs

last-level cache (L2):

- shared
- but not inclusive

= shared memory not in L2 is not shared in cache
Caches on ARM Cortex-A CPUs

last-level cache (L2):

- shared
- but not inclusive
  - shared memory not in L2 is not shared in cache

Challenge #1: non-inclusive caches
Modern ARM SoCs

- big.LITTLE architecture (A53 + A57)
  - multiple CPUs with no shared cache
Modern ARM SoCs

- big.LITTLE architecture (A53 + A57)
  → multiple CPUs with no shared cache

Challenge #2: no shared cache
Cache maintenance

Instructions to enforce memory coherency

- x86: unprivileged `clflush`
- until ARMv7-A: n/a
- ARMv8-A: kernel can unlock a flush instruction for userspace
Cache maintenance

Instructions to enforce memory coherency

- x86: unprivileged clflush
- until ARMv7-A: n/a
- ARMv8-A: kernel can unlock a flush instruction for userspace

Challenge #3: no flush instruction
Cache eviction

- targeted cache eviction on ARM can be complicated:
  - existing approaches introduce much noise
  - pseudo-random replacement policy
  - unclear how randomness affects existing approaches
Cache eviction

- targeted cache eviction on ARM can be complicated:
  - existing approaches introduce much noise
  - pseudo-random replacement policy
  - unclear how randomness affects existing approaches

Challenge #4: perform fast & reliable cache eviction
Timing measurements

- x86: rdtsc provides unprivileged access to cycle count
- ARM: existing attacks require access to privileged mode cycle counter
Timing measurements

- x86: rdtsc provides unprivileged access to cycle count
- ARM: existing attacks require access to privileged mode cycle counter

Challenge #5: find unprivileged highly accurate timing sources
Challenges

#1: non-inclusive caches
#2: no shared cache
#3: no flush
#4: random eviction
#5: no unprivileged timing
Solving #1: non-inclusive caches
Solving #1: non-inclusive caches

Attacking instruction-inclusive data-non-inclusive caches
Solving #1: non-inclusive caches

Attacking instruction-inclusive data-non-inclusive caches
Solving #1: non-inclusive caches

What about entirely non-inclusive caches?

- cache coherency protocol
- fetches data from remote cores instead of DRAM
  $\Rightarrow$ remote cache hits
Solving #1: non-inclusive caches

What about entirely non-inclusive caches?
Solving #1: non-inclusive caches

Measured access time in CPU cycles (OnePlus One)

- Hit (same core)
- Hit (cross-core)
- Miss (same core)
- Miss (cross-core)

Number of accesses

-10^4

0

1

2

3

4

5

6

7

8

9

10

100

200

300

400

500

600

700

800

900

1,000

Daniel Gruss, Graz University of Technology
August 11, 2016 — Usenix Security 2016
Solving #2: no shared cache

Multiple CPUs with no shared cache
Solving #2: no shared cache

Multiple CPUs with no shared cache

- again: cache coherency protocol
- fetches data from remote CPUs instead of DRAM
- keep local L2 filled to increase probability of remote L1/L2 eviction
- timing difference between local and remote still small enough
Solving #3: no flush

- idea: replace flush instruction with cache eviction
  - Flush+Reload $\rightarrow$ Evict+Reload
Solving #3: no flush

- idea: replace flush instruction with cache eviction
  - Flush+Reload → Evict+Reload (works on x86)
Solving #3: no flush

- idea: replace flush instruction with cache eviction
  - Flush+Reload $\rightarrow$ Evict+Reload (works on x86)
- but: cache eviction is slow and can be unreliable
Solving #3: no flush

- idea: replace flush instruction with cache eviction
  - Flush+Reload $\rightarrow$ Evict+Reload (works on x86)
- but: cache eviction is slow and can be unreliable
- unless you know how to evict
Solving #3: no flush

- idea: replace flush instruction with cache eviction
  - Flush+Reload $\rightarrow$ Evict+Reload (works on x86)
- but: cache eviction is slow and can be unreliable
- unless you know how to evict
  - central idea of our Rowhammer.js paper
Solving #4: random eviction

<table>
<thead>
<tr>
<th>unique addr.</th>
<th># accesses</th>
<th>Cycles</th>
<th>Eviction rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>48</td>
<td>48</td>
<td>6517</td>
<td>70.8%</td>
</tr>
<tr>
<td>800</td>
<td>800</td>
<td>142876</td>
<td>99.1%</td>
</tr>
<tr>
<td>23</td>
<td>50</td>
<td>6209</td>
<td>100.0%</td>
</tr>
<tr>
<td>22</td>
<td>102</td>
<td>5101</td>
<td>100.0%</td>
</tr>
<tr>
<td>21</td>
<td>96</td>
<td>4275</td>
<td>99.9%</td>
</tr>
</tbody>
</table>

(on the Alcatel One Touch Pop 2)
Solving #5: no unprivileged timing

Comparison of 4 different measurement techniques

- performance counter (privileged)
- \texttt{perf-event-open} (syscall, unprivileged)
- \texttt{clock-gettime} (unprivileged)
- thread counter (multithreaded, unprivileged)
Solving #5: no unprivileged timing

Hit (PMCCNTR)  Hit (clock_gettime × .15)
Miss (PMCCNTR) Miss (clock_gettime × .15)
Hit (syscall × .25)  Hit (counter thread × .05)
Miss (syscall × .25) Miss (counter thread × .05)
Flush+Flush on the Samsung Galaxy S6

![Graph showing measured execution time in CPU cycles for Flush (address cached) and Flush (address not cached).]
Prime+Probe on the Alcatel One Touch Pop 2

Execution time in CPU cycles

Cases

Victim access
No victim access
## Covert channels on Android

<table>
<thead>
<tr>
<th>Work</th>
<th>Type</th>
<th>Bandwidth [bps]</th>
<th>Error rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ours (Samsung Galaxy S6)</td>
<td>Flush+Reload, cross-core</td>
<td>1 140 650</td>
<td>1.10%</td>
</tr>
<tr>
<td>Ours (Samsung Galaxy S6)</td>
<td>Flush+Reload, cross-CPU</td>
<td>257 509</td>
<td>1.83%</td>
</tr>
<tr>
<td>Ours (Samsung Galaxy S6)</td>
<td>Flush+Flush, cross-core</td>
<td>178 292</td>
<td>0.48%</td>
</tr>
<tr>
<td>Ours (Alcatel One Touch Pop 2)</td>
<td>Evict+Reload, cross-core</td>
<td>13 618</td>
<td>3.79%</td>
</tr>
<tr>
<td>Ours (OnePlus One)</td>
<td>Evict+Reload, cross-core</td>
<td>12 537</td>
<td>5.00%</td>
</tr>
<tr>
<td>Marforio et al.</td>
<td>Type of Intents</td>
<td>4 300</td>
<td>–</td>
</tr>
<tr>
<td>Marforio et al.</td>
<td>UNIX socket discovery</td>
<td>2 600</td>
<td>–</td>
</tr>
<tr>
<td>Schlegel et al.</td>
<td>File locks</td>
<td>685</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>Vibration settings</td>
<td>87</td>
<td>–</td>
</tr>
</tbody>
</table>
Cache template attacks (CTA)

Cache template matrix for `libinput.so`
(on an Alcatel One Touch Pop 2)
Cache template attacks (CTA)

Cache template matrix for the default AOSP keyboard
(on a Samsung Galaxy S6)
CTA: taps and swipes

measured on an Alcatel One Touch Pop 2
CTA: taps and swipes

measured on a Samsung Galaxy S6
CTA: taps and swipes

measured on a OnePlus One

Access time

Time in seconds
CTA: distinguishing keys

Time in seconds

Access time

Key
Space
Bouncy Castle

- a widely used crypto library
  - WhatsApp, ...
- uses a T-table implementation
Attacking Bouncy Castle

Evict+Reload (Alcatel) vs. Flush+Reload (Samsung)
Attacking Bouncy Castle with Prime+Probe (Alcatel)
Conclusions

- all the powerful cache attacks applicable to smartphones
- monitor user activity with high accuracy
- derive crypto keys
- ARM TrustZone leaks through the cache
ARMageddon: Cache Attacks on Mobile Devices

Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, Stefan Mangard
Graz University of Technology
August 11, 2016 — Usenix Security 2016