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?

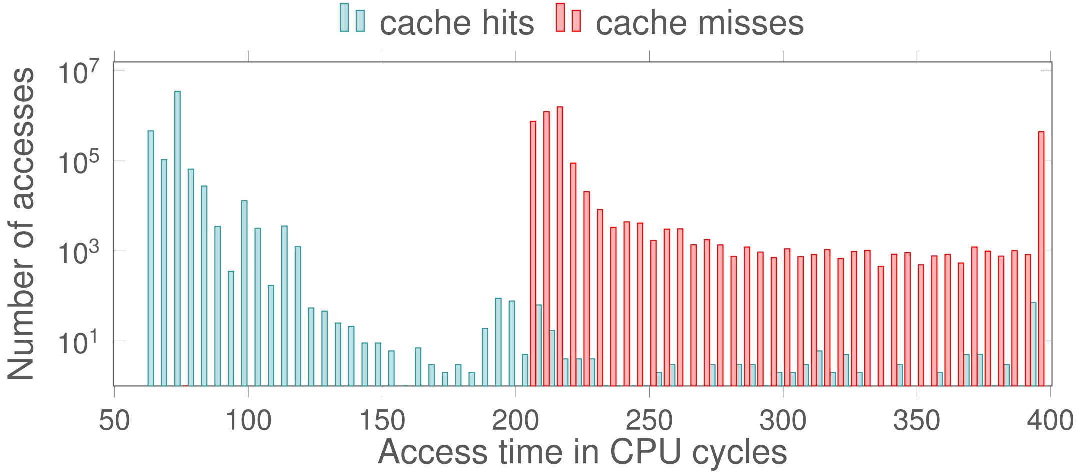
TLDR

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

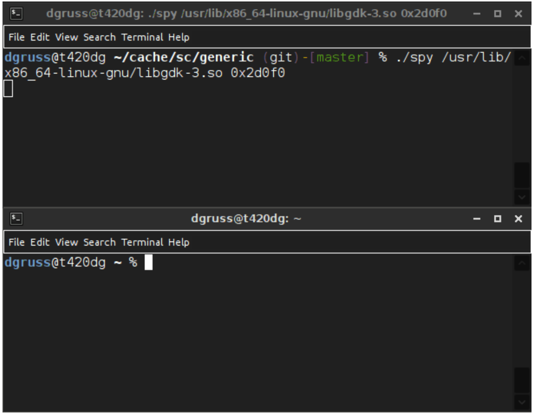
We identified and solved challenges systematically to:

- 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)



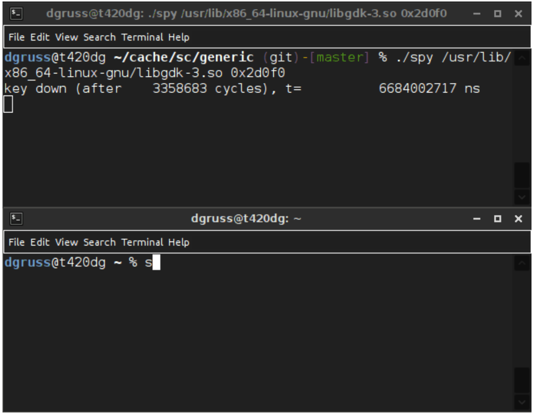
What is a cache attack? (2)



```
dgruss@t420dg ~/cache/sc/generic (git)-[master] % ./spy /usr/lib/
x86_64-linux-gnu/libgdk-3.so 0x2d0f0
```

```
dgruss@t420dg ~ %
```

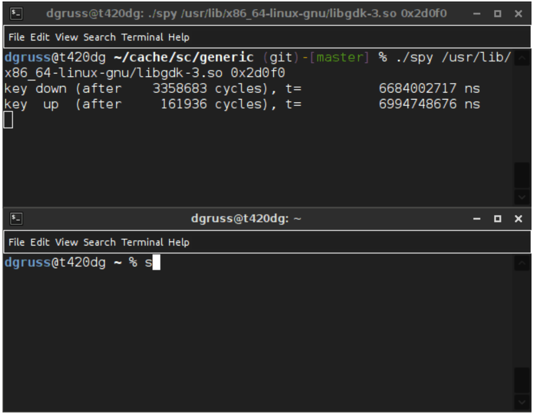
What is a cache attack? (2)



```
dgruss@t420dg ~/cache/sc/generic (git)-[master] % ./spy /usr/lib/
x86_64-linux-gnu/libgdk-3.so 0x2d0f0
key down (after    3358683 cycles), t=         6684002717 ns
```

```
dgruss@t420dg ~ % s
```

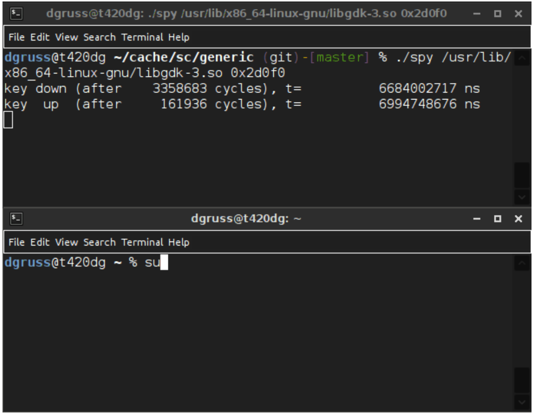
What is a cache attack? (2)



```
dgruss@t420dg ~/cache/sc/generic (git)-[master] % ./spy /usr/lib/
x86_64-linux-gnu/libgdk-3.so 0x2d0f0
key down (after    3358683 cycles), t=         6684002717 ns
key  up  (after     161936 cycles), t=         6994748676 ns
```

```
dgruss@t420dg ~ % s
```

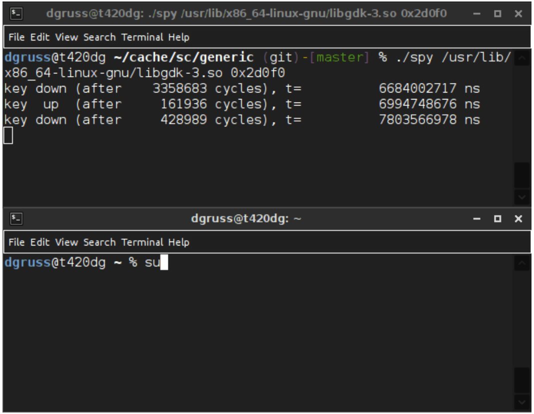
What is a cache attack? (2)



```
dgruss@t420dg ~/cache/sc/generic (git)-[master] % ./spy /usr/lib/
x86_64-linux-gnu/libgdk-3.so 0x2d0f0
key down (after    3358683 cycles), t=        6684002717 ns
key  up  (after     161936 cycles), t=        6994748676 ns
```

```
dgruss@t420dg ~ % su
```

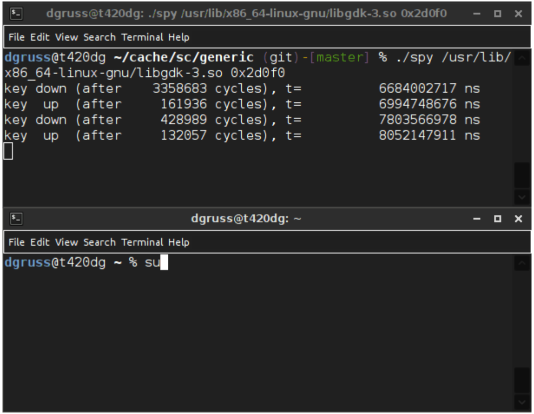
What is a cache attack? (2)



```
dgruss@t420dg ~/cache/sc/generic (git)-[master] % ./spy /usr/lib/
x86_64-linux-gnu/libgdk-3.so 0x2d0f0
key down (after    3358683 cycles), t=         6684002717 ns
key  up  (after     161936 cycles), t=         6994748676 ns
key down (after     428989 cycles), t=         7803566978 ns
```

```
dgruss@t420dg ~ % su
```

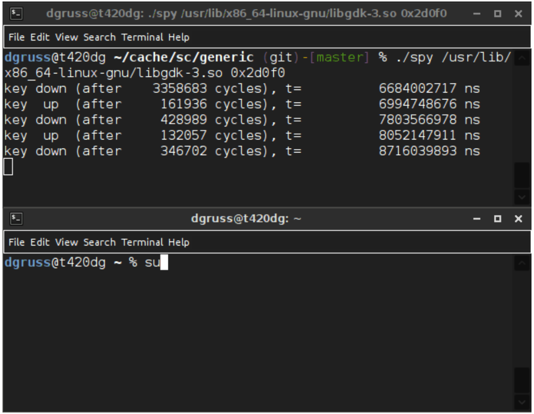
What is a cache attack? (2)



```
dgruss@t420dg ~/cache/sc/generic (git)-[master] % ./spy /usr/lib/
x86_64-linux-gnu/libgdk-3.so 0x2d0f0
key down (after    3358683 cycles), t=         6684002717 ns
key  up  (after     161936 cycles), t=         6994748676 ns
key down (after     428989 cycles), t=         7803566978 ns
key  up  (after     132057 cycles), t=         8052147911 ns
```

```
dgruss@t420dg ~ % su
```

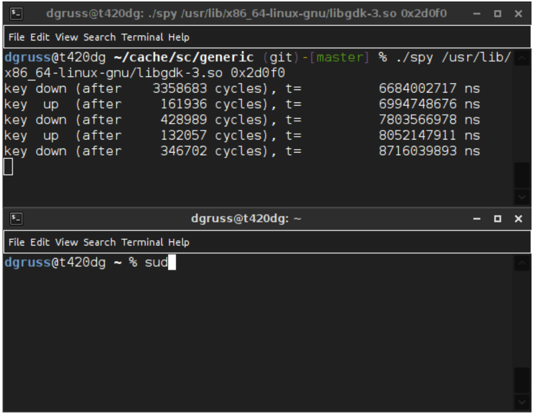
What is a cache attack? (2)



```
dgruss@t420dg ~/cache/sc/generic (git)-[master] % ./spy /usr/lib/
x86_64-linux-gnu/libgdk-3.so 0x2d0f0
key down (after   3358683 cycles), t=         6684002717 ns
key  up  (after    161936 cycles), t=         6994748676 ns
key down (after    428989 cycles), t=         7803566978 ns
key  up  (after    132057 cycles), t=         8052147911 ns
key down (after    346702 cycles), t=         8716039893 ns
```

```
dgruss@t420dg ~ % su
```

What is a cache attack? (2)



```
dgruss@t420dg ~/cache/sc/generic (git)-[master] % ./spy /usr/lib/
x86_64-linux-gnu/libgdk-3.so 0x2d0f0
key down (after    3358683 cycles), t=         6684002717 ns
key  up  (after     161936 cycles), t=         6994748676 ns
key down (after     428989 cycles), t=         7803566978 ns
key  up  (after     132057 cycles), t=         8052147911 ns
key down (after     346702 cycles), t=         8716039893 ns
```

```
dgruss@t420dg ~ % sud
```

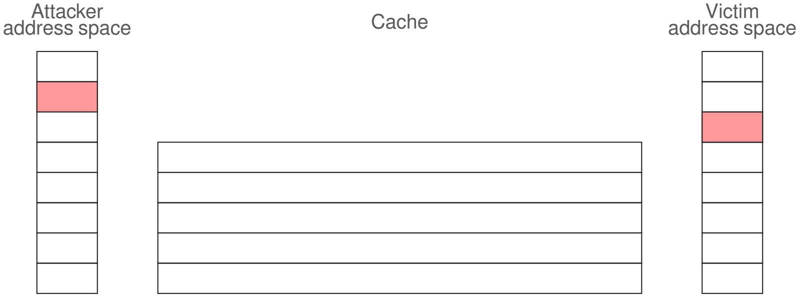
Cache attack techniques

Most important techniques:

- Flush+Reload
- Prime+Probe

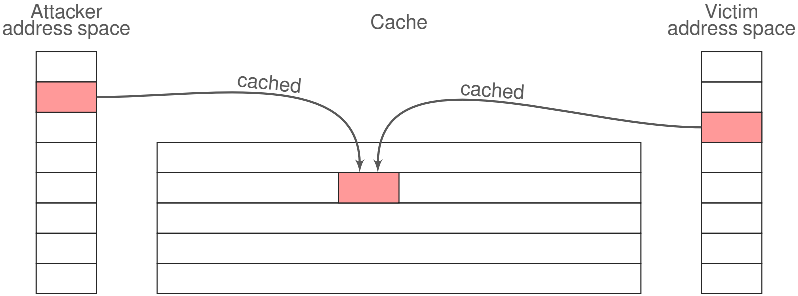
Both work on the last-level cache → across cores

Flush+Reload



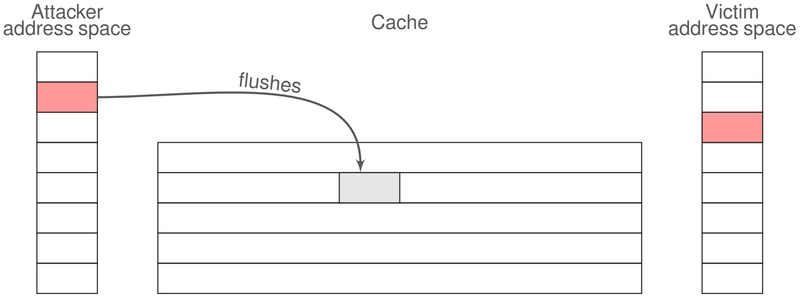
step 0: attacker maps shared library → shared memory, shared in cache

Flush+Reload



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

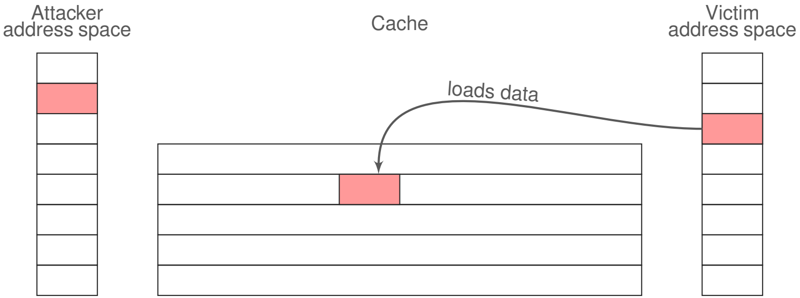
Flush+Reload



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

step 1: attacker flushes the shared line

Flush+Reload

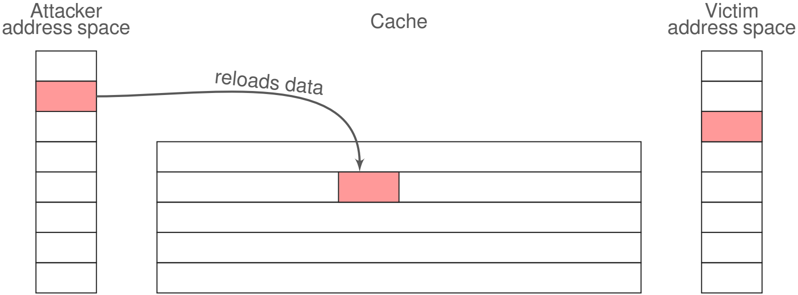


step 0: attacker maps shared library → shared memory, shared in cache

step 1: attacker flushes the shared line

step 2: victim loads data while performing encryption

Flush+Reload



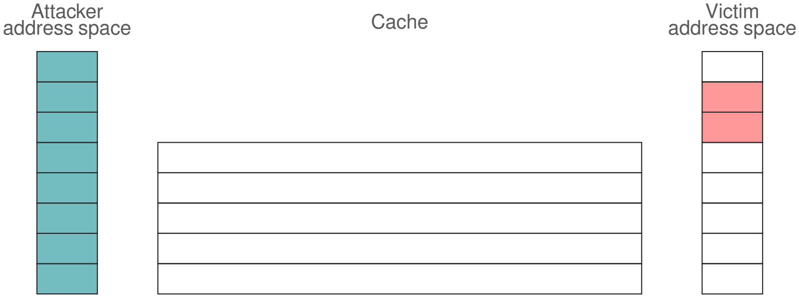
step 0: attacker maps shared library → shared memory, shared in cache

step 1: attacker flushes the shared line

step 2: victim loads data while performing encryption

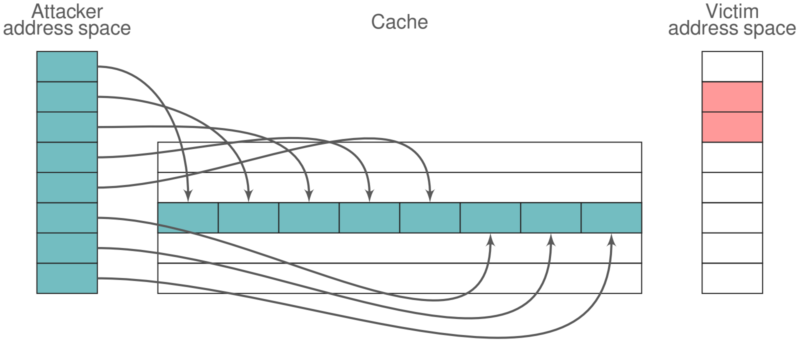
step 3: attacker reloads data → fast access if the victim loaded the line

Prime+Probe



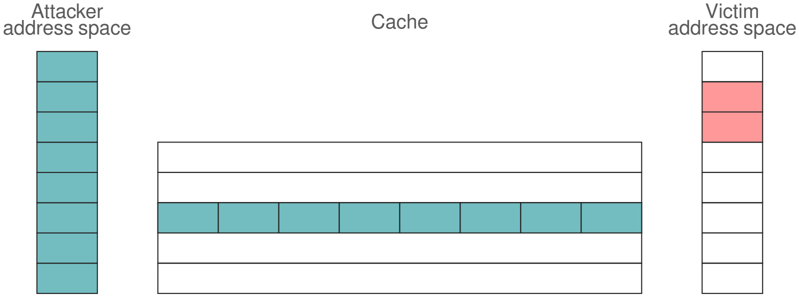
step 0: attacker fills the cache (prime)

Prime+Probe



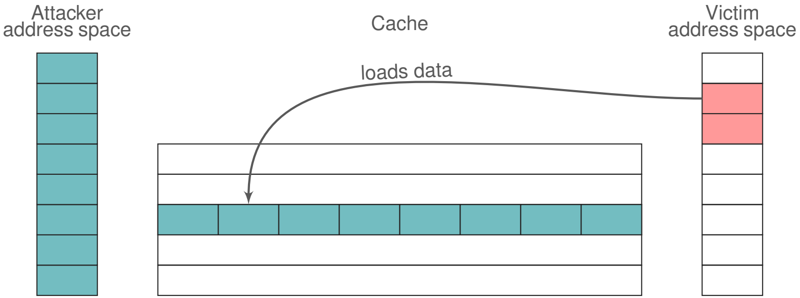
step 0: attacker fills the cache (prime)

Prime+Probe



step 0: attacker fills the cache (prime)

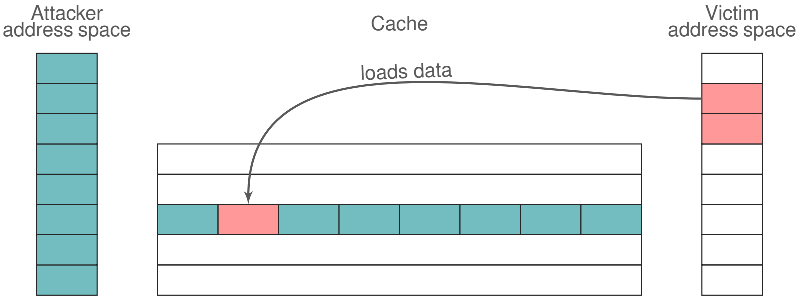
Prime+Probe



step 0: attacker fills the cache (prime)

step 1: victim evicts cache lines while performing encryption

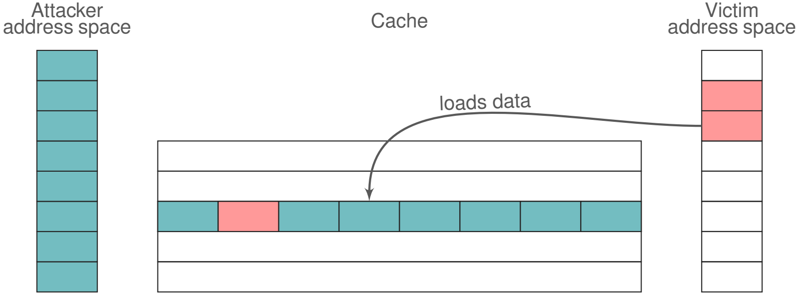
Prime+Probe



step 0: attacker fills the cache (prime)

step 1: victim evicts cache lines while performing encryption

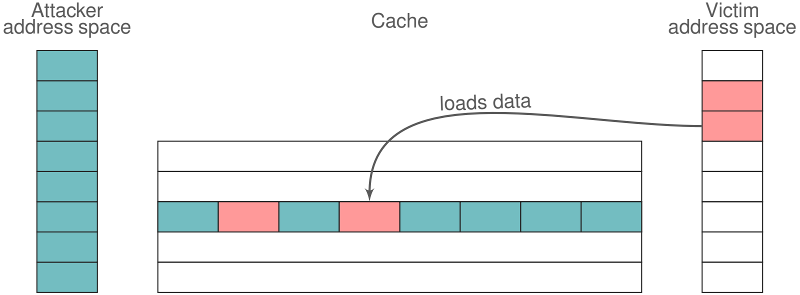
Prime+Probe



step 0: attacker fills the cache (prime)

step 1: victim evicts cache lines while performing encryption

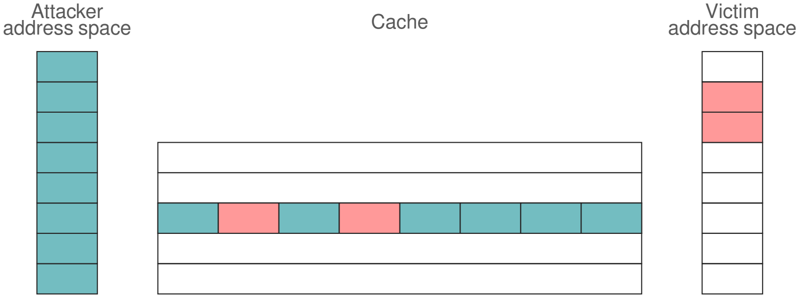
Prime+Probe



step 0: attacker fills the cache (prime)

step 1: victim evicts cache lines while performing encryption

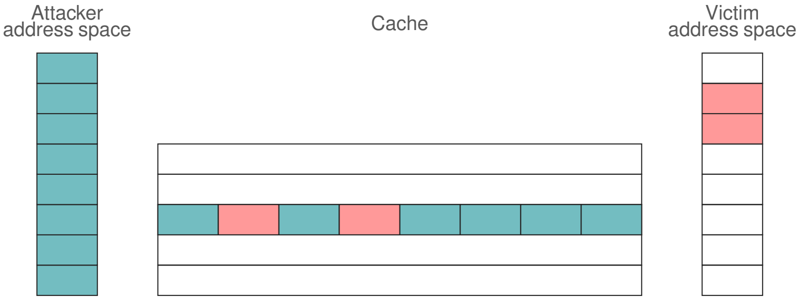
Prime+Probe



step 0: attacker fills the cache (prime)

step 1: victim evicts cache lines while performing encryption

Prime+Probe

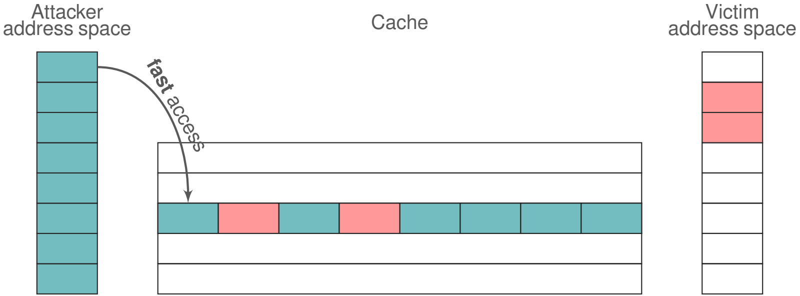


step 0: attacker fills the cache (prime)

step 1: victim evicts cache lines while performing encryption

step 2: attacker probes data to determine if the set was accessed

Prime+Probe

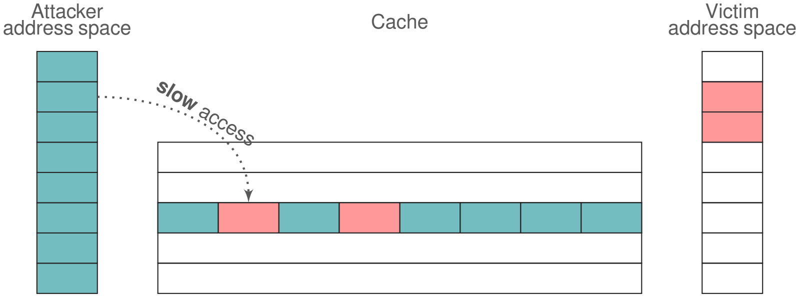


step 0: attacker fills the cache (prime)

step 1: victim evicts cache lines while performing encryption

step 2: attacker probes data to determine if the set was accessed

Prime+Probe

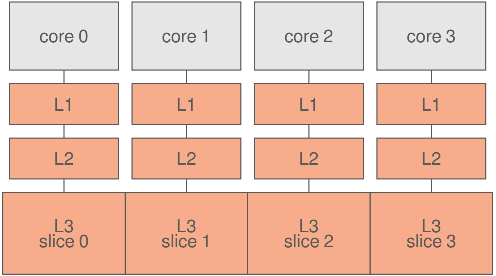


step 0: attacker fills the cache (prime)

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

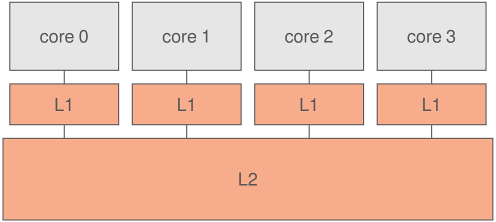


last-level cache (L3):

- shared
- inclusive

= shared memory is shared in cache, across cores!

Caches on ARM Cortex-A CPUs

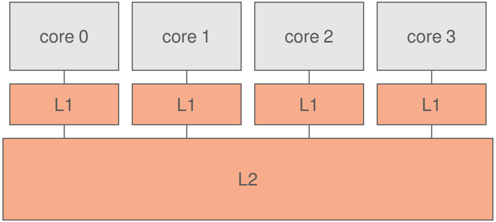


last-level cache (L2):

- 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

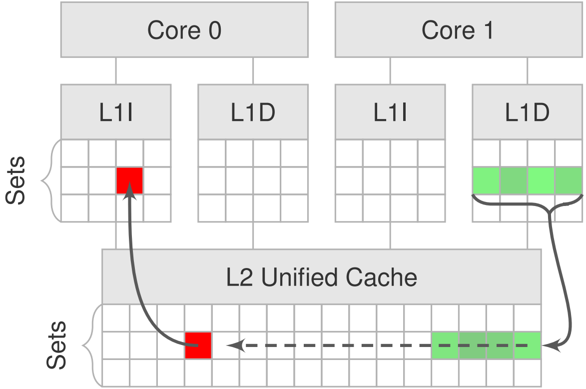
Solving #1: non-inclusive caches

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

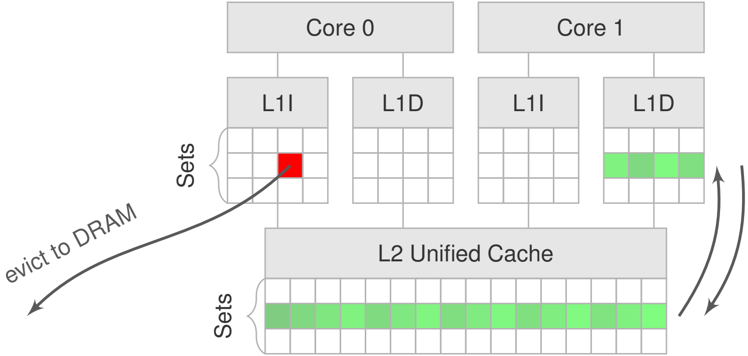
What about entirely non-inclusive caches?

- cache coherency protocol
- fetches data from remote cores instead of DRAM

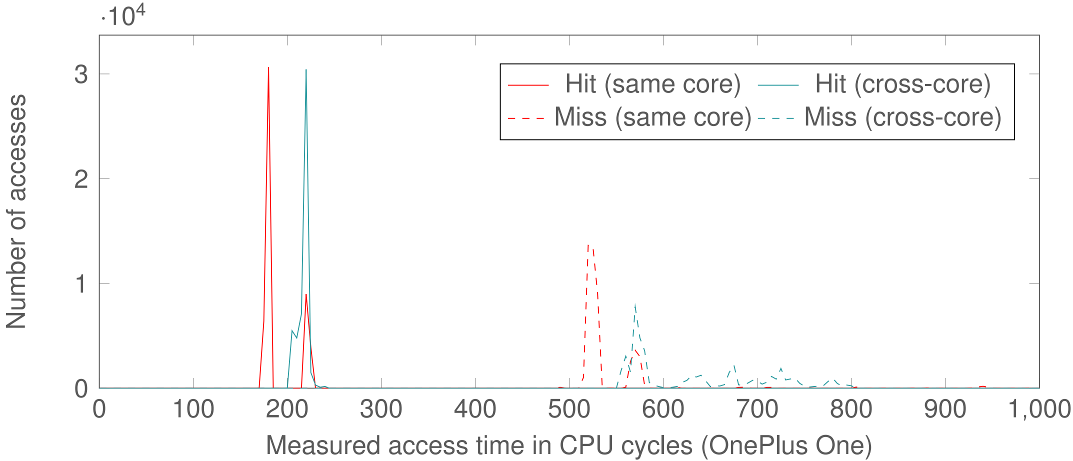
→ remote cache hits

Solving #1: non-inclusive caches

What about entirely non-inclusive caches?



Solving #1: non-inclusive caches



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 → 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 → 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 → 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 → 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

unique addr.	# accesses	Cycles	Eviction rate
48	48	6 517	70.8%
800	800	142 876	99.1%
23	50	6 209	100.0%
22	102	5 101	100.0%
21	96	4 275	99.9%

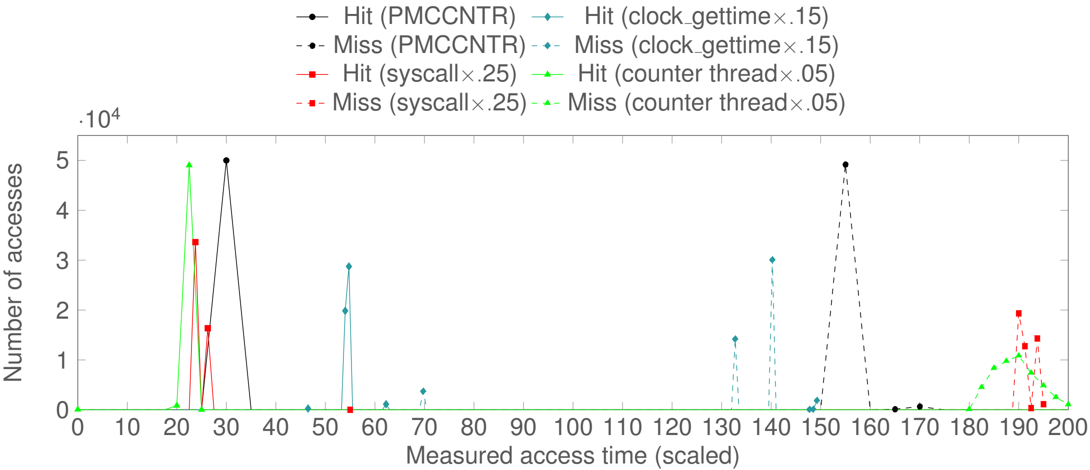
(on the Alcatel One Touch Pop 2)

Solving #5: no unprivileged timing

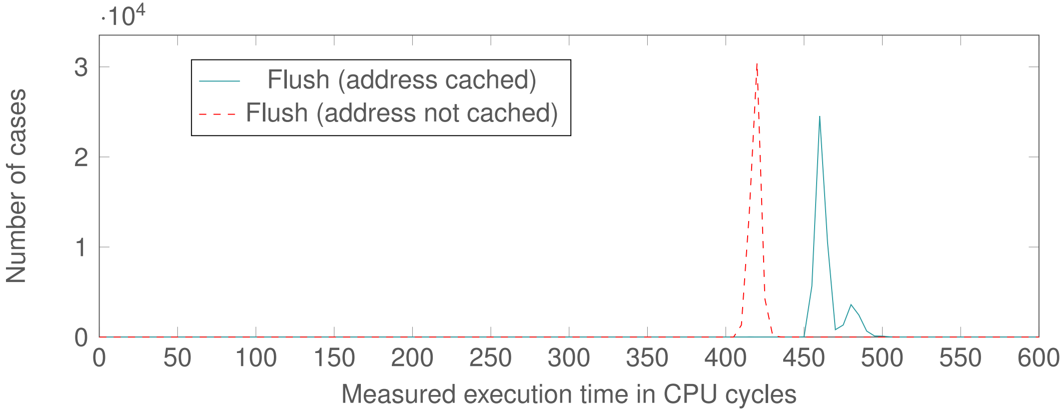
Comparison of 4 different measurement techniques

- performance counter (privileged)
- `perf_event_open` (syscall, unprivileged)
- `clock_gettime` (unprivileged)
- thread counter (multithreaded, unprivileged)

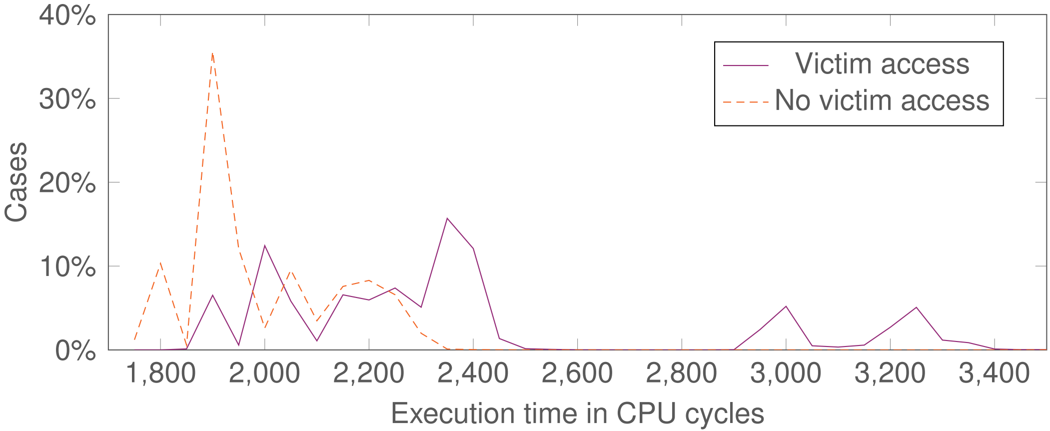
Solving #5: no unprivileged timing



Flush+Flush on the Samsung Galaxy S6



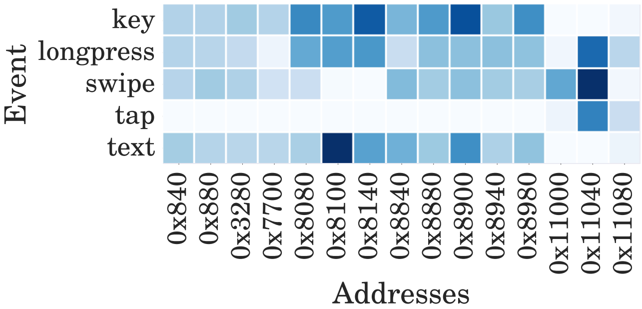
Prime+Probe on the Alcatel One Touch Pop 2



Covert channels on Android

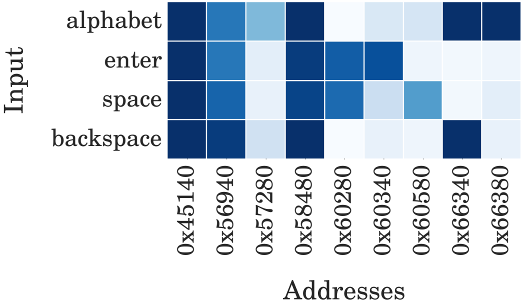
Work	Type	Bandwidth [bps]	Error rate
Ours (Samsung Galaxy S6)	Flush+Reload, cross-core	**1 140 650**	1.10%
Ours (Samsung Galaxy S6)	Flush+Reload, cross-CPU	**257 509**	1.83%
Ours (Samsung Galaxy S6)	Flush+Flush, cross-core	**178 292**	0.48%
Ours (Alcatel One Touch Pop 2)	Evict+Reload, cross-core	**13 618**	3.79%
Ours (OnePlus One)	Evict+Reload, cross-core	**12 537**	5.00%
Marforio et al.	Type of Intents	4 300	–
Marforio et al.	UNIX socket discovery	2 600	–
Schlegel et al.	File locks	685	–
Schlegel et al.	Volume settings	150	–
Schlegel et al.	Vibration settings	87	–

Cache template attacks (CTA)



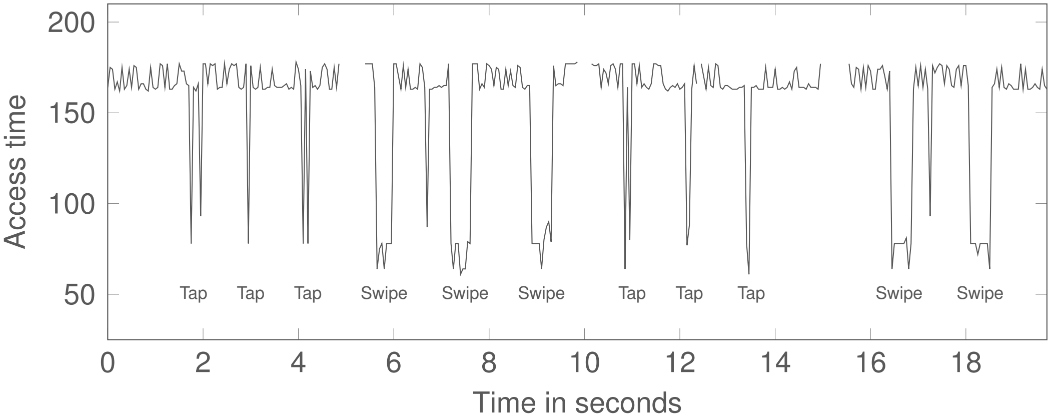
Cache template matrix for `libinput.so`
(on an Alcatel One Touch Pop 2)

Cache template attacks (CTA)



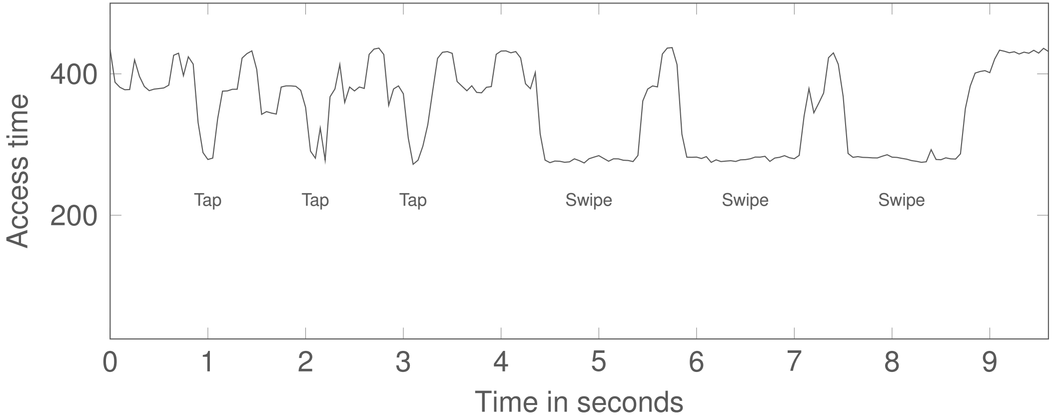
Cache template matrix for the default AOSP keyboard
(on a Samsung Galaxy S6)

CTA: taps and swipes



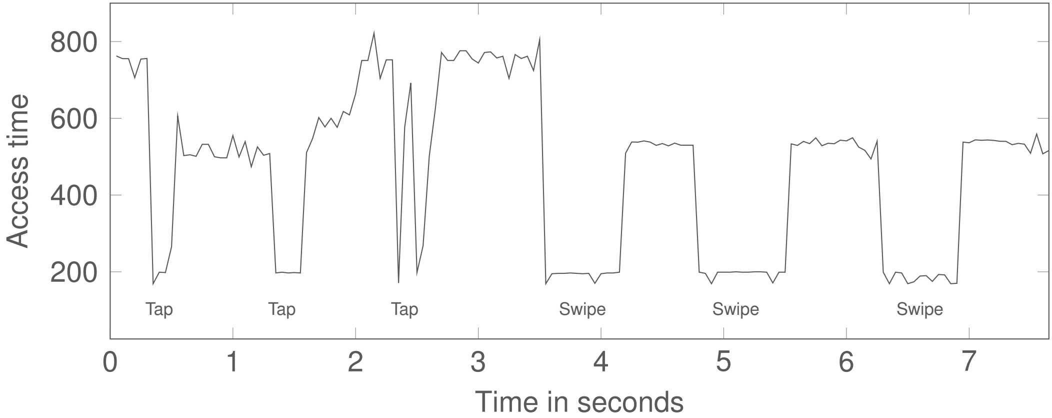
measured on an Alcatel One Touch Pop 2

CTA: taps and swipes



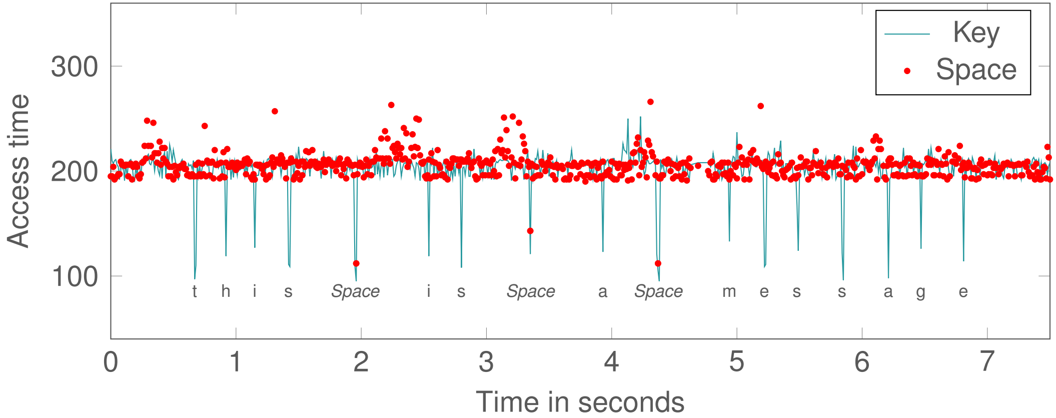
measured on a Samsung Galaxy S6

CTA: taps and swipes



measured on measured on a OnePlus One

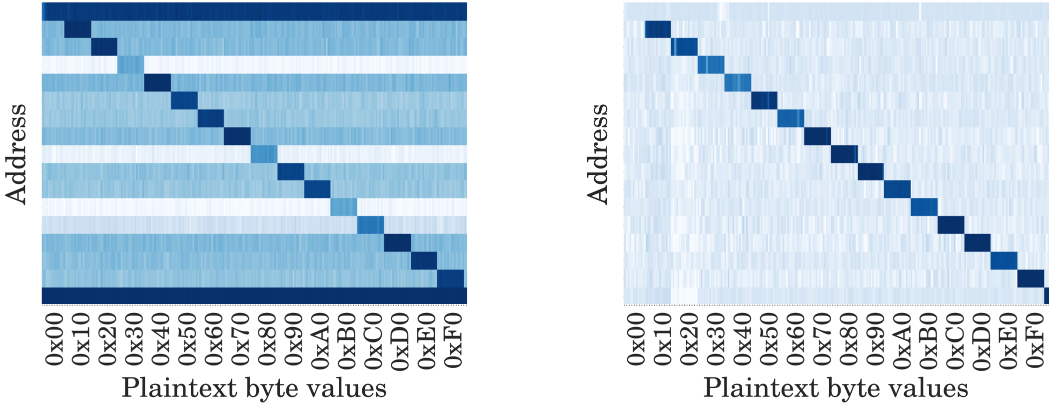
CTA: distinguishing keys



Bouncy Castle

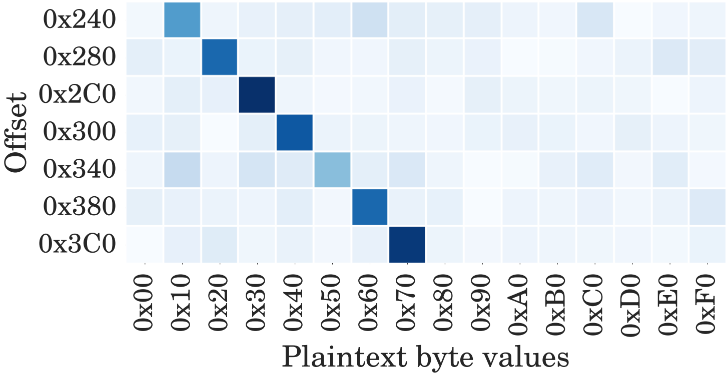
- a widely used crypto library
  - WhatsApp, ...
- uses a T-table implementation

Attacking Bouncy Castle

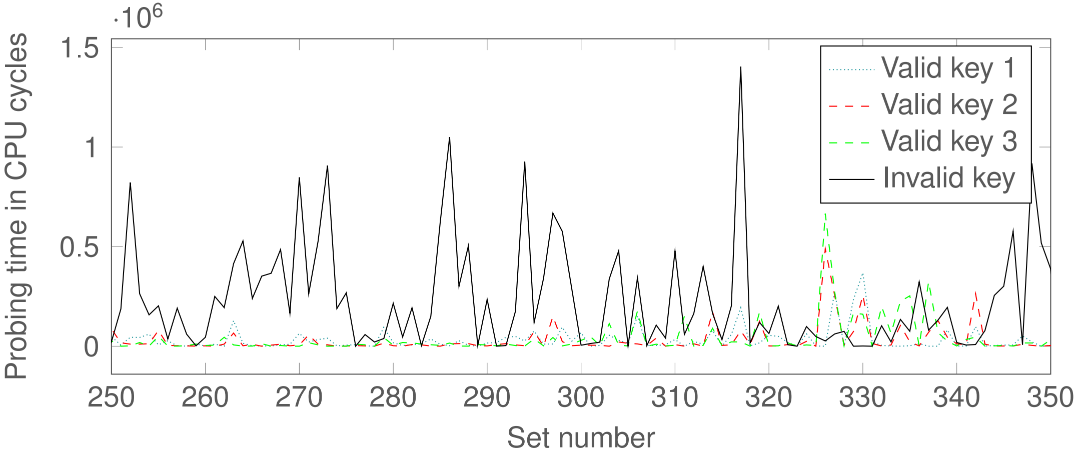


Evict+Reload (Alcatel) vs. Flush+Reload (Samsung)

Attacking Bouncy Castle with Prime+Probe (Alcatel)



Leakage from ARM TrustZone (RSA signatures)



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