With great speed comes great leakage

How processor performance is tied to side-channel leakage

Moritz Lipp & Daniel Gruss, Graz University of Technology

May 18, 2017 — Qualcomm Mobile Security Summit

Who are we

- **Moritz Lipp**
- PhD Student @ Graz University Of Technology
- @mlqxyz
- moritz.lipp@iaik.tugraz.at

Who are we

- **Daniel Gruss**
- PhD student @ Graz University Of Technology
- @lavados
- daniel.gruss@iaik.tugraz.at

Goals of this talk

- Gain a comprehensive overview on microarchitectural attacks on ARM
- Understand the connection between performance and side-channel leakage

Outline

- Background
- Practical attacks
- More attack vectors
- Rowhammer attacks
- Defenses
- Conclusion

Let's improve memory latency. It's faster, what's wrong with that?

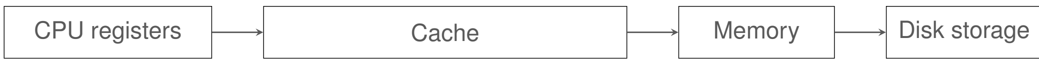
Memory Hierarchy

- Data can reside in CPU registers, the main memory or on disk

Memory Hierarchy

- Data can reside in CPU registers, the main memory or on disk
- We want data as fast as possible

Memory Hierarchy



- Data can reside in CPU registers, the main memory or on disk
- We want data as fast as possible
- CPU Cache: Small but fast memory

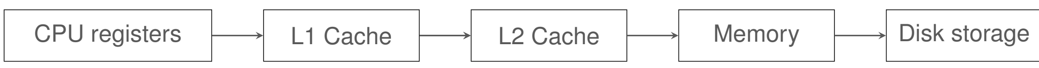
Memory Hierarchy

- Data can reside in CPU registers, the main memory or on disk
- We want data as fast as possible
- CPU Cache: Small but fast memory

Memory Hierarchy

- Data can reside in CPU registers, the main memory or on disk
- We want data as fast as possible
- CPU Cache: Small but fast memory
    - Different levels of the cache

Memory Hierarchy

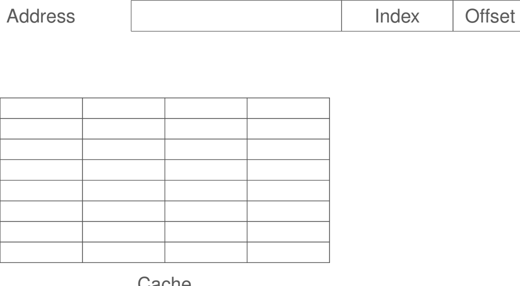


- Data can reside in CPU registers, the main memory or on disk
- We want data as fast as possible
- CPU Cache: Small but fast memory
    - Different levels of the cache

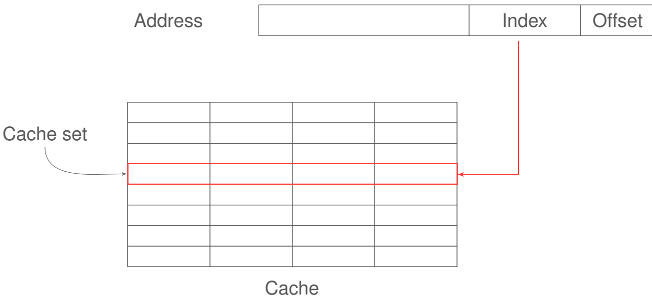
Performance

CPU-Cache for fast memory look-ups

Set-Associative Caches

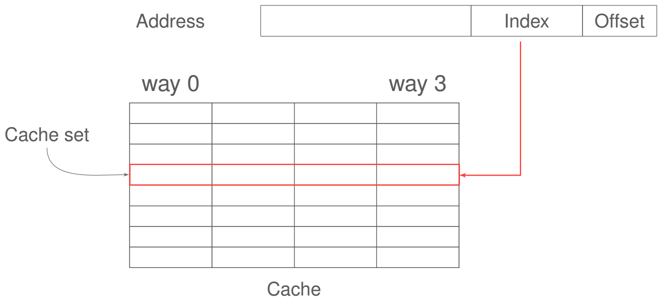


Set-Associative Caches



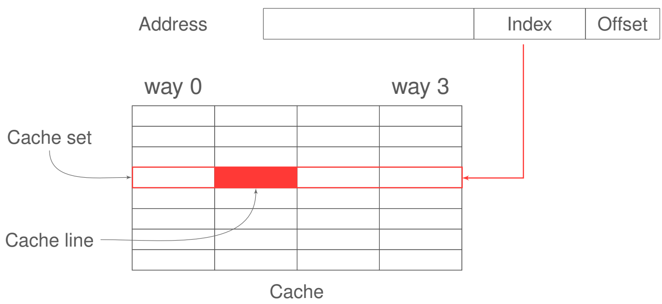
Data loaded in a specific set depending on its address

Set-Associative Caches



- Data loaded in a specific set depending on its address
- Several ways per set

Set-Associative Caches



Data loaded in a specific set depending on its address

Several ways per set

Cache line loaded in a specific way depending on the replacement policy

Cache Attacks

- Exploit timing differences of memory accesses:

Cache Attacks

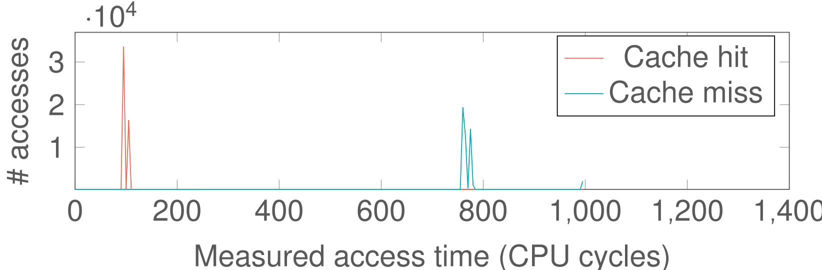
- Exploit timing differences of memory accesses:
  - cache $\rightarrow$ fast (cache hit)

Cache Attacks

- Exploit timing differences of memory accesses:
  - cache $\rightarrow$ fast (cache hit)
  - main memory $\rightarrow$ slow (cache miss)

Cache Attacks

- Exploit timing differences of memory accesses:
  - cache → fast (cache hit)
  - main memory → slow (cache miss)



Cache Attacks Requirements

- Capability to:

Cache Attacks Requirements

- Capability to:
  - Accurate timing measurements to distinguish cache hit and cache misses

Cache Attacks Requirements

- Capability to:
  - Accurate timing measurements to distinguish cache hit and cache misses
  - Remove from and load data into the cache

Timing measurements

- Need fine-grained timing measurements

Timing measurements

- Need fine-grained timing measurements
- Intel x86: Unprivileged `rdtsc` instruction for cycle count

Timing measurements

- Need fine-grained timing measurements
- Intel x86: Unprivileged `rdtsc` instruction for cycle count
- ARM: Cycle counter only in privileged mode

Timing measurements

- Need fine-grained timing measurements
- Intel x86: Unprivileged `rdtsc` instruction for cycle count
- ARM: Cycle counter only in privileged mode

$\rightarrow$ Previous attacks required root access

Unprivileged timing on ARM

1. Performance counter: Privileged

Unprivileged timing on ARM

1. Performance counter: Privileged
2. `perf_event_open`: Unprivileged syscall

Unprivileged timing on ARM

1. Performance counter: Privileged
2. `perf_event_open`: Unprivileged syscall
    - Unavailable on new devices: Privileged or kernel compiled with `CONFIG_PERF=n`

Unprivileged timing on ARM

1. Performance counter: Privileged
2. `perf_event_open`: Unprivileged syscall
   - Unavailable on new devices: Privileged or kernel compiled with `CONFIG_PERF=n`
3. `clock_gettime`: Unprivileged `POSIX` function

Unprivileged timing on ARM

1. Performance counter: Privileged
2. `perf_event_open`: Unprivileged syscall
   - Unavailable on new devices: Privileged or kernel compiled with `CONFIG_PERF=n`
3. `clock_gettime`: Unprivileged `POSIX` function
4. Thread counter: Unprivileged, multithreaded

Unprivileged timing on ARM

1. Performance counter: Privileged
2. `perf_event_open`: Unprivileged syscall
   - Unavailable on new devices: Privileged or kernel compiled with `CONFIG_PERF=n`
3. `clock_gettime`: Unprivileged `POSIX` function
4. Thread counter: Unprivileged, multithreaded

- Unprivileged timing sources

Unprivileged timing on ARM

1. Performance counter: Privileged
2. `perf_event_open`: Unprivileged syscall
    - Unavailable on new devices: Privileged or kernel compiled with `CONFIG_PERF=n`
3. `clock_gettime`: Unprivileged `POSIX` function
4. Thread counter: Unprivileged, multithreaded

- Unprivileged timing sources
- Nanosecond resolution for all sources

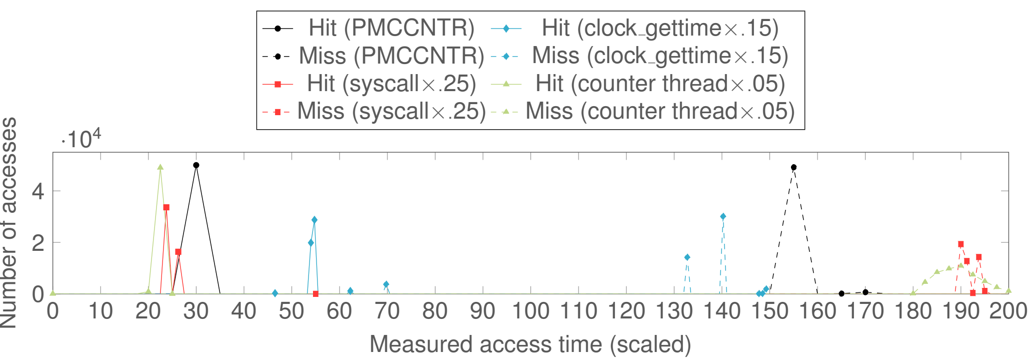
Unprivileged timing on ARM

1. Performance counter: Privileged
2. `perf_event_open`: Unprivileged syscall
   - Unavailable on new devices: Privileged or kernel compiled with `CONFIG_PERF=n`
3. `clock_gettime`: Unprivileged `POSIX` function
4. Thread counter: Unprivileged, multithreaded

- Unprivileged timing sources
- Nanosecond resolution for all sources

→ Allows distinguishing cache hits from cache misses

Unprivileged timing on ARM



Cache maintenance

- Basic operation for cache attacks: invalidate cache lines

Cache maintenance

- Basic operation for cache attacks: invalidate cache lines
- Cache maintenance instructions

Cache maintenance

- Basic operation for cache attacks: invalidate cache lines
- Cache maintenance instructions
  - Intel x86: Unprivileged `clflush` instruction

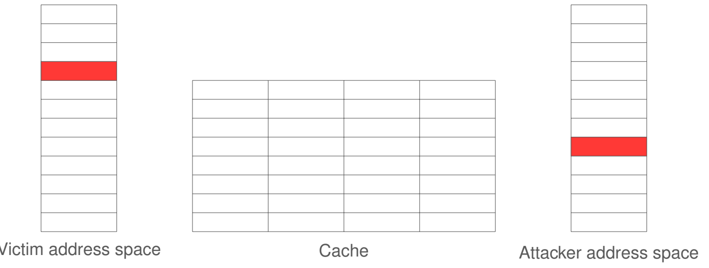
Cache maintenance

- Basic operation for cache attacks: invalidate cache lines
- Cache maintenance instructions
  - Intel x86: Unprivileged `clflush` instruction
  - ARMv7-A: Only privileged cache maintenance instructions

Cache maintenance

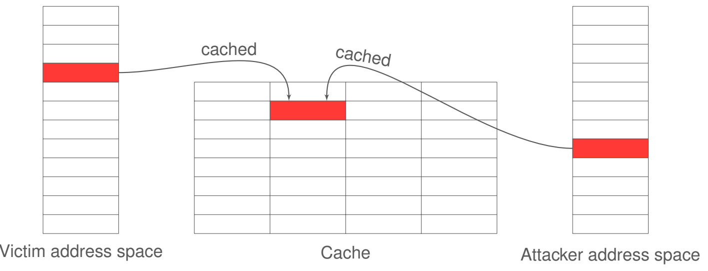
- Basic operation for cache attacks: invalidate cache lines
- Cache maintenance instructions
  - Intel x86: Unprivileged `clflush` instruction
  - ARMv7-A: Only privileged cache maintenance instructions
  - ARMv8-A: Privileged instructions can be unlocked for userspace

Cache Attacks: Flush+Reload



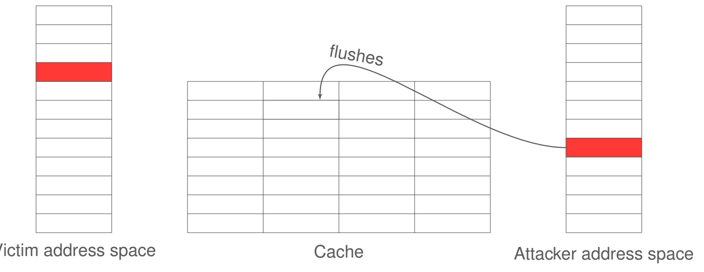
Step 1: Attacker maps shared library (shared memory, in cache)

Cache Attacks: Flush+Reload



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

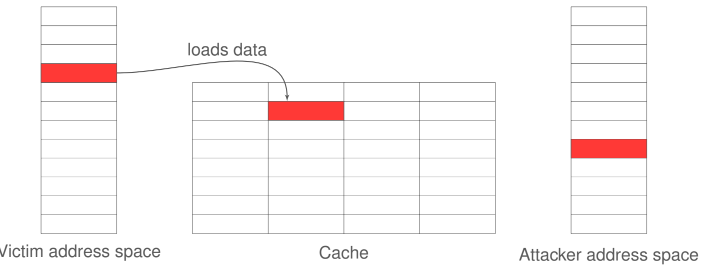
Cache Attacks: Flush+Reload



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

Step 2: Attacker flushes the shared cache line

Cache Attacks: Flush+Reload

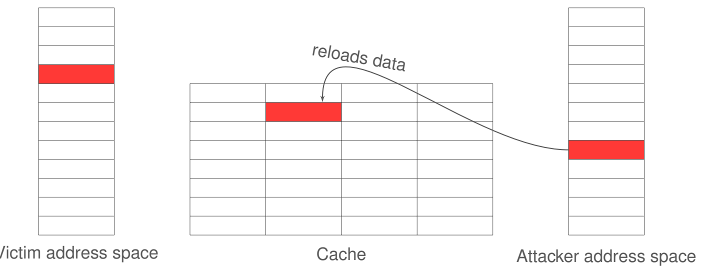


Step 1: Attacker maps shared library (shared memory, in cache)

Step 2: Attacker flushes the shared cache line

Step 3: Victim loads the data

Cache Attacks: 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

So we disable flush instructions and “problem solved”, right?

No unprivileged flush instruction on ARMv7

- Replace the missing flush instruction with cache eviction

No unprivileged flush instruction on ARMv7

- Replace the missing flush instruction with cache eviction
- Works on Intel x86

No unprivileged flush instruction on ARMv7

- Replace the missing flush instruction with cache eviction
- Works on Intel x86
  - Flush+Reload $\rightarrow$ Evict+Reload
  - Prime+Probe

Cache eviction

- Fill the whole cache

Moritz Lipp & Daniel Gruss, Graz University of Technology

Cache eviction

- Fill the whole cache $\rightarrow$ too slow

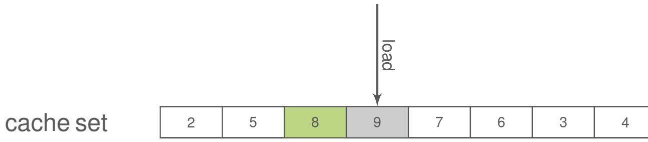
Cache eviction

- Fill the whole cache $\rightarrow$ too slow
- Fill a specific cache set

cache set	2	5	8	1	7	6	3	4

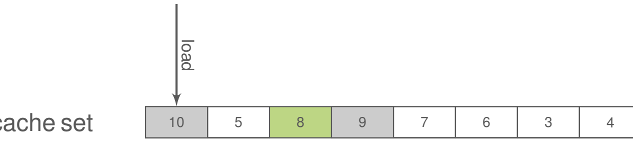
Cache eviction

- Fill the whole cache → too slow
- Fill a specific cache set



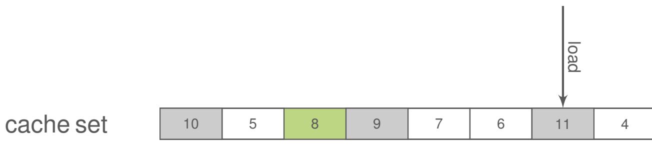
Cache eviction

- Fill the whole cache → too slow
- Fill a specific cache set



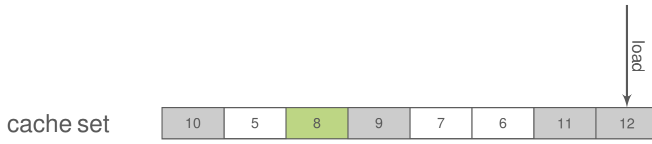
Cache eviction

- Fill the whole cache → too slow
- Fill a specific cache set



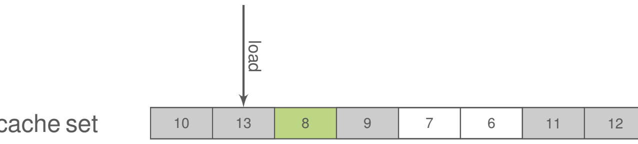
Cache eviction

- Fill the whole cache → too slow
- Fill a specific cache set



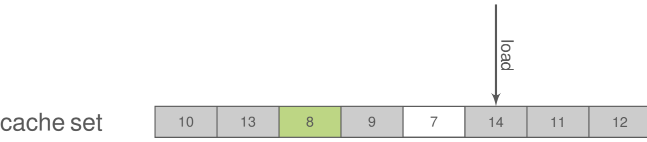
Cache eviction

- Fill the whole cache → too slow
- Fill a specific cache set



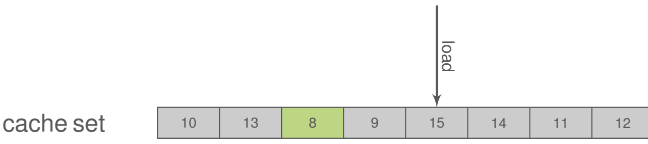
Cache eviction

- Fill the whole cache → too slow
- Fill a specific cache set



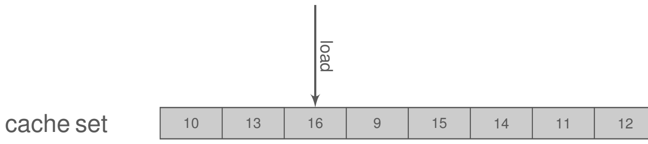
Cache eviction

- Fill the whole cache → too slow
- Fill a specific cache set



Cache eviction

- Fill the whole cache $\rightarrow$ too slow
- Fill a specific cache set
- Until the target address is evicted from the cache



Cache eviction

- Fill the whole cache → too slow
- Fill a specific cache set
- Until the target address is evicted from the cache

cache set

10	13	16	9	15	14	11	12

→ Ideal case with LRU replacement policy

Cache eviction (what actually happens)

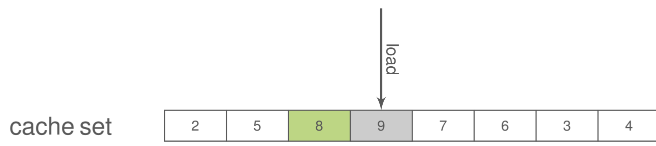
- Pseudo-random cache replacement policy

cache set

2	5	8	1	7	6	3	4

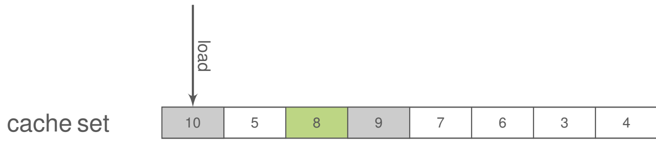
Cache eviction (what actually happens)

- Pseudo-random cache replacement policy



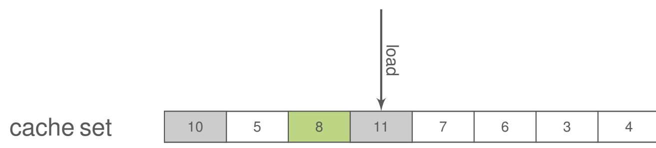
Cache eviction (what actually happens)

- Pseudo-random cache replacement policy



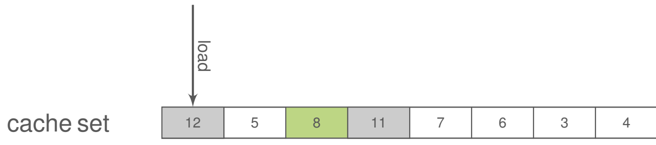
Cache eviction (what actually happens)

- Pseudo-random cache replacement policy



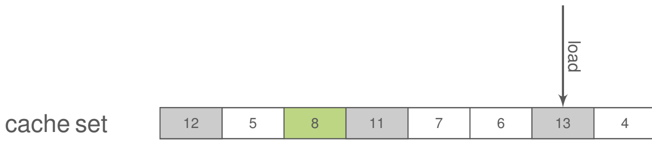
Cache eviction (what actually happens)

- Pseudo-random cache replacement policy



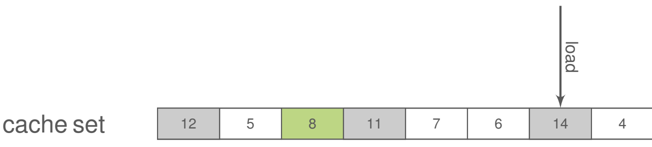
Cache eviction (what actually happens)

- Pseudo-random cache replacement policy



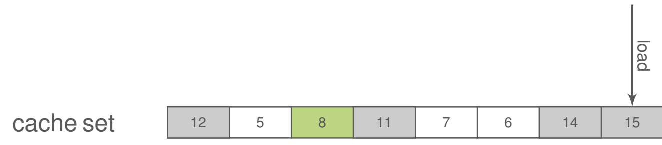
Cache eviction (what actually happens)

- Pseudo-random cache replacement policy



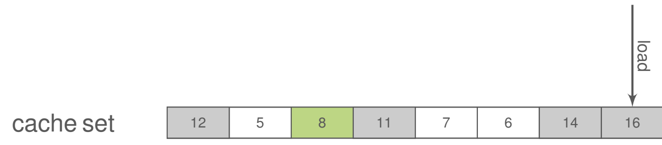
Cache eviction (what actually happens)

- Pseudo-random cache replacement policy



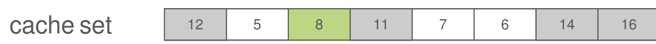
Cache eviction (what actually happens)

- Pseudo-random cache replacement policy



Cache eviction (what actually happens)

- Pseudo-random cache replacement policy



→ Simple approach highly inefficient

Pseudo-random replacement policy

- Cache line to be discarded is chosen pseudo-randomly

Pseudo-random replacement policy

- Cache line to be discarded is chosen pseudo-randomly
- Accessing once n addresses in an n-way cache set

Pseudo-random replacement policy

- Cache line to be discarded is chosen pseudo-randomly
- Accessing once *n* addresses in an *n*-way cache set

$\rightarrow$ Cache eviction slow and unreliable

Pseudo-random replacement policy

- Cache line to be discarded is chosen pseudo-randomly
- Accessing once *n* addresses in an *n*-way cache set

$\rightarrow$ Cache eviction slow and unreliable

Solution:

- Accessing unique addresses several times, with different access patterns

Pseudo-random replacement policy

Addresses	Accesses	Cycles	Eviction rate
48	48	6 517 ✓	70.78% ✗

Pseudo-random replacement policy

Addresses	Accesses	Cycles	Eviction rate
48	48	6 517 ✓	70.78% ✗
200	200	33 110 ✗	96.04% ✗

Pseudo-random replacement policy

Addresses	Accesses	Cycles	Eviction rate
48	48	6 517 ✓	70.78% ✗
200	200	33 110 ✗	96.04% ✗
800	800	142 876 ✗	99.10% ✓

Pseudo-random replacement policy

Addresses	Accesses	Cycles	Eviction rate
48	48	6 517 ✓	70.78% ✗
200	200	33 110 ✗	96.04% ✗
800	800	142 876 ✗	99.10% ✓
21	96	4 275 ✓	99.93% ✓

Pseudo-random replacement policy

Addresses	Accesses	Cycles	Eviction rate
48	48	6 517 ✓	70.78% ✗
200	200	33 110 ✗	96.04% ✗
800	800	142 876 ✗	99.10% ✓
21	96	4 275 ✓	99.93% ✓
22	102	5 101 ✓	99.99% ✓

Pseudo-random replacement policy

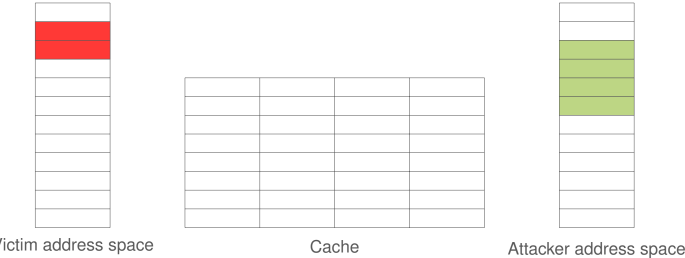
Addresses	Accesses	Cycles	Eviction rate
48	48	6 517 ✓	70.78% ✗
200	200	33 110 ✗	96.04% ✗
800	800	142 876 ✗	99.10% ✓
21	96	4 275 ✓	99.93% ✓
22	102	5 101 ✓	99.99% ✓
23	190	6 209 ✓	100.0% ✓

Pseudo-random replacement policy

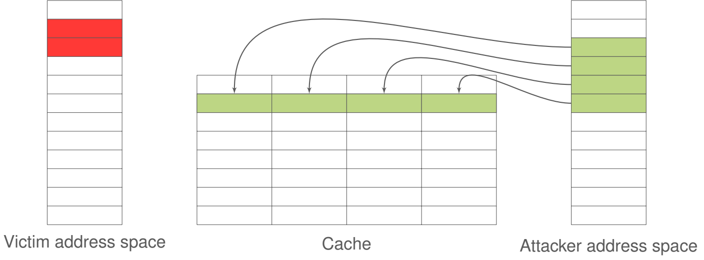
Addresses	Accesses	Cycles	Eviction rate
48	48	6 517 ✓	70.78% ✗
200	200	33 110 ✗	96.04% ✗
800	800	142 876 ✗	99.10% ✓
21	96	4 275 ✓	99.93% ✓
22	102	5 101 ✓	99.99% ✓
23	190	6 209 ✓	100.0% ✓

- We fully automated this process
- Find fast and efficient eviction strategies for any device

Cache Attacks: Prime+Probe

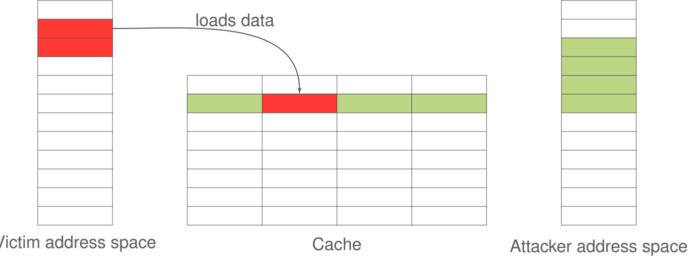


Cache Attacks: Prime+Probe



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

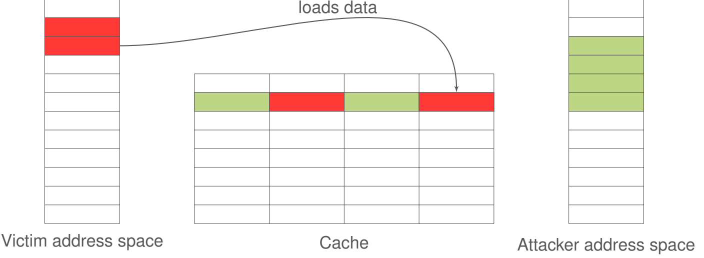
Cache Attacks: Prime+Probe



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

Step 2: Victim evicts cache lines while running

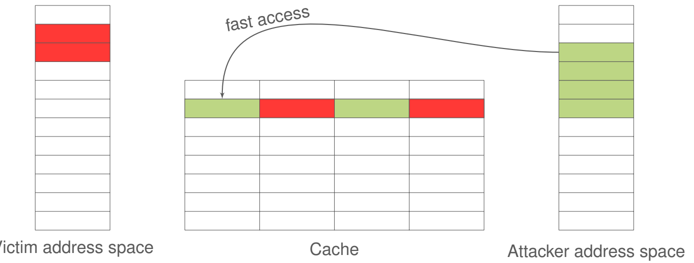
Cache Attacks: Prime+Probe



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

Step 2: Victim evicts cache lines while running

Cache Attacks: Prime+Probe

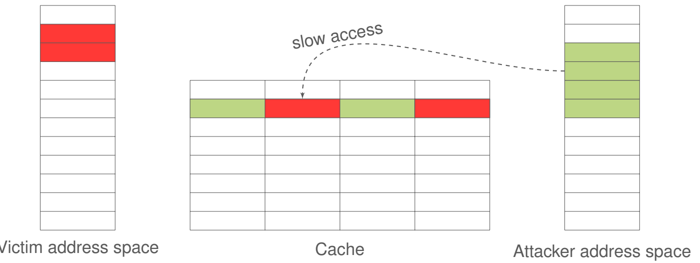


Step 1: Attacker primes, *i.e.*, fills, the cache (no shared memory)

Step 2: Victim evicts cache lines while running

Step 3: Attacker probes data to determine if set has been accessed

Cache Attacks: Prime+Probe

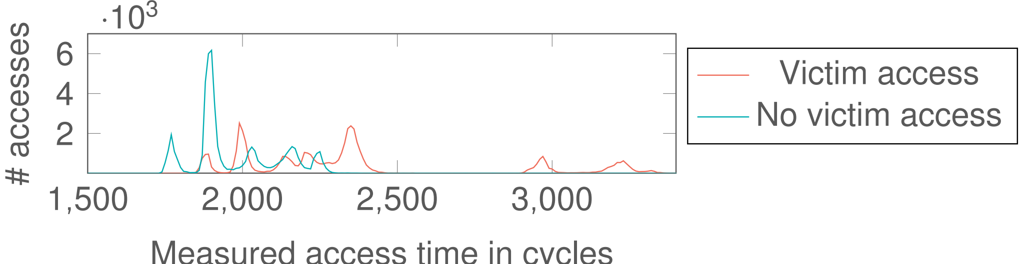


Step 1: Attacker primes, *i.e.*, fills, the cache (no shared memory)

Step 2: Victim evicts cache lines while running

Step 3: Attacker probes data to determine if set has been accessed

Prime+Probe Histogram



Those were just artificial examples and theoretical attacks, right?

Demo

Demo!

Spying on the User

- Issue: Locating event-dependent memory access

Spying on the User

- Issue: Locating event-dependent memory access

→ Cache Template Attacks

Spying on the User

- Issue: Locating event-dependent memory access

→ Cache Template Attacks

1. Shared library or executable is mapped

Spying on the User

- Issue: Locating event-dependent memory access

→ Cache Template Attacks

1. Shared library or executable is mapped
2. Trigger an event in parallel and Flush+Reload one address

Spying on the User

- Issue: Locating event-dependent memory access

→ Cache Template Attacks

1. Shared library or executable is mapped
2. Trigger an event in parallel and Flush+Reload one address

   → Cache hit: Address used by the library/executable

Spying on the User

- Issue: Locating event-dependent memory access

→ Cache Template Attacks

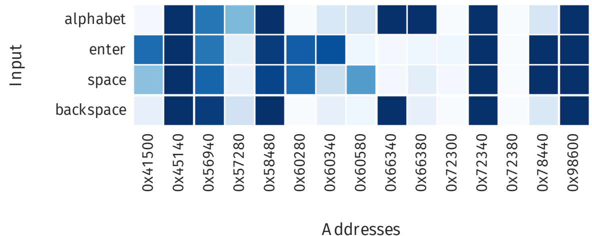
1. Shared library or executable is mapped
2. Trigger an event in parallel and Flush+Reload one address

   → Cache hit: Address used by the library/executable

3. Repeat step 2 for every address

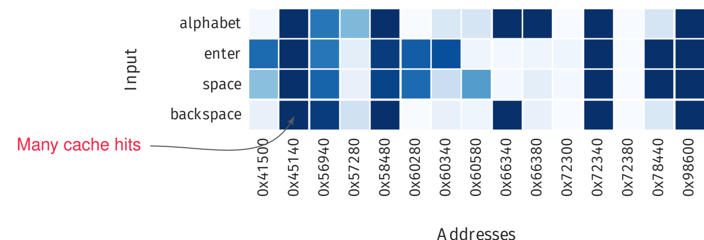
Spying on the User

- Cache template matrix
  = How many cache hits for each pair (event, address)?
- On shared library and ART binaries, e.g., AOSP keyboard



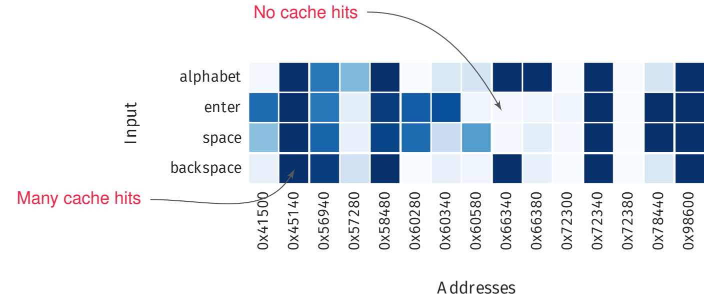
Spying on the User

- Cache template matrix
  = How many cache hits for each pair (event, address)?
- On shared library and ART binaries, e.g., AOSP keyboard



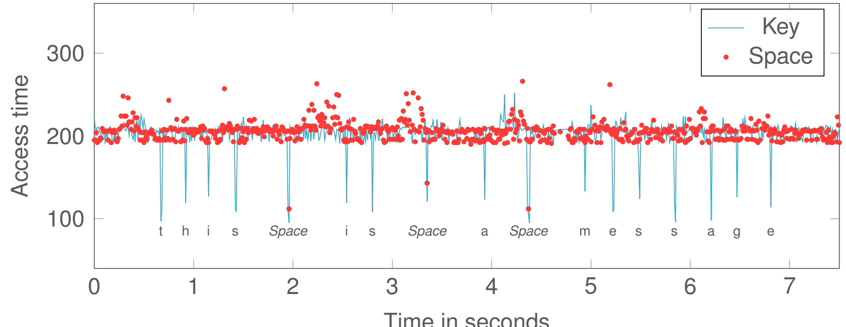
Spying on the User

- Cache template matrix
  = How many cache hits for each pair (event, address)?
- On shared library and ART binaries, e.g., AOSP keyboard



Spying on the User

Evict+Reload on two addresses on the Alcatel One Touch Pop 2 in `custpack@app@withoutlibs@LatinIME.apk@classes.dex`
→ Distinguish keys from spaces

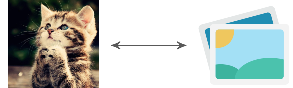


Covert Channels

- gallery app

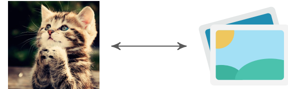
Covert Channels

- gallery app
  - No permissions except accessing your images



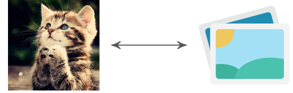
Covert Channels

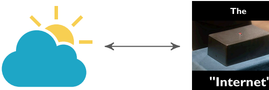
- gallery app
  - No permissions except accessing your images
- weather widget



Covert Channels

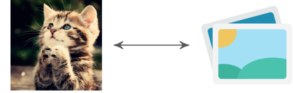
- gallery app
  - No permissions except accessing your images
- weather widget
  - No permissions except accessing the Internet

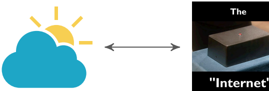




Covert Channels

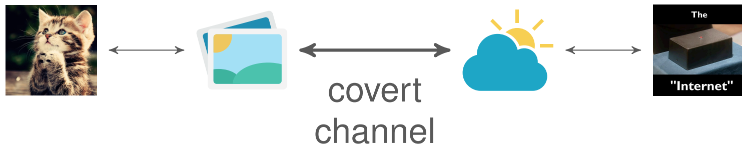
- malicious gallery app
  - No permissions except accessing your images
- malicious weather widget
  - No permissions except accessing the Internet





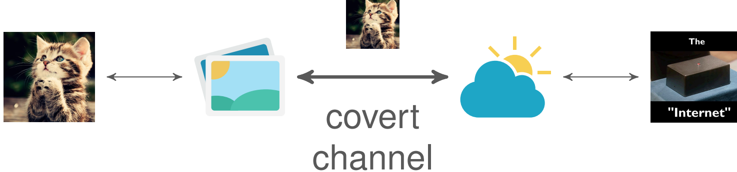
Covert Channels

- malicious gallery app
  - No permissions except accessing your images
- malicious weather widget
  - No permissions except accessing the Internet



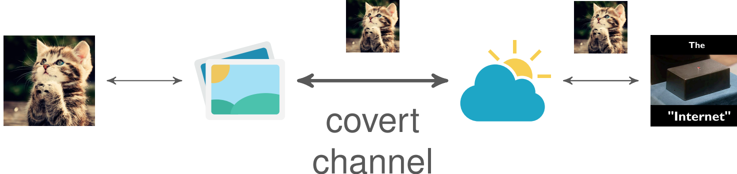
Covert Channels

- malicious gallery app
  - No permissions except accessing your images
- malicious weather widget
  - No permissions except accessing the Internet



Covert Channels

- malicious gallery app
  - No permissions except accessing your images
- malicious weather widget
  - No permissions except accessing the Internet



Covert Channels

- Two apps want to communicate with each other, but are not allowed or not able to do so

Covert Channels

- Two apps want to communicate with each other, but are not allowed or not able to do so
  - No permissions

Covert Channels

- Two apps want to communicate with each other, but are not allowed or not able to do so
  - No permissions
  - No Intents, Binders, ASHMEM, . . .

Covert Channels

- Two apps want to communicate with each other, but are not allowed or not able to do so
  - No permissions
  - No Intents, Binders, ASHMEM, . . .
- A covert channel

Covert Channels

- Two apps want to communicate with each other, but are not allowed or not able to do so
  - No permissions
  - No Intents, Binders, ASHMEM, . . .
- A covert channel
  - Enables two unprivileged apps to communicate

Covert Channels

- Two apps want to communicate with each other, but are not allowed or not able to do so
  - No permissions
  - No Intents, Binders, ASHMEM, . . .
- A covert channel
  - Enables two unprivileged apps to communicate
  - Does not use data transfer mechanisms provided by the OS

Covert Channels

- Two apps want to communicate with each other, but are not allowed or not able to do so
  - No permissions
  - No Intents, Binders, ASHMEM, . . .
- A covert channel
  - Enables two unprivileged apps to communicate
  - Does not use data transfer mechanisms provided by the OS
  - Evades the sandboxing concept and permission system

Covert Channels

- Two apps want to communicate with each other, but are not allowed or not able to do so
  - No permissions
  - No Intents, Binders, ASHMEM, ...
- A covert channel
  - Enables two unprivileged apps to communicate
  - Does not use data transfer mechanisms provided by the OS
  - Evades the sandboxing concept and permission system

→ Collusion attack

Covert Channels

- Works using Flush+Reload, Evict+Reload, and other techniques

Covert Channels

- Works using Flush+Reload, Evict+Reload, and other techniques
- Works cross-core and cross-CPU

Covert Channels

- Works using Flush+Reload, Evict+Reload, and other techniques
- Works cross-core and cross-CPU
- Faster than state of the art by several orders of magnitude

Covert Channels

- Works using Flush+Reload, Evict+Reload, and other techniques
- Works cross-core and cross-CPU
- Faster than state of the art by several orders of magnitude

Work	Type	Bandwidth [bps]	Error rate
Schlegel et al.	Vibration settings	87	–
Schlegel et al.	Volume settings	150	–
Schlegel et al.	File locks	685	–
Marforio et al.	UNIX socket discovery	2 600	–
Marforio et al.	Type of Intents	4 300	–

Covert Channels

- Works using Flush+Reload, Evict+Reload, and other techniques
- Works cross-core and cross-CPU
- Faster than state of the art by several orders of magnitude

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

But if I use crypto I'm safe, right?

Attacking AES

- Bouncy Castle $\rightarrow$ default implementation uses T-Tables

Attacking AES

- Bouncy Castle → default implementation uses T-Tables
  - key-dependent table accesses = key-dependent memory accesses

→ Let's monitor which which entry is accessed!

Attacking AES

- Bouncy Castle $\rightarrow$ default implementation uses T-Tables
    - key-dependent table accesses = key-dependent memory accesses

$\rightarrow$ Let's monitor which which entry is accessed!

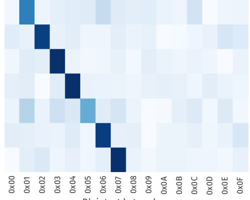
- Java VM creates a copy of the T-tables when the app starts

Attacking AES

- Bouncy Castle → default implementation uses T-Tables
  - key-dependent table accesses = key-dependent memory accesses

→ Let's monitor which which entry is accessed!

- Java VM creates a copy of the T-tables when the app starts
- No shared memory → only Prime+Probe possible



Monitoring ARM TrustZone

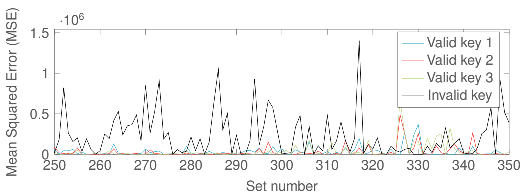
- Timing difference for different RSA signature keys

Monitoring ARM TrustZone

- Timing difference for different RSA signature keys
- No knowledge of the TrustZone/trustlet implementation

Monitoring ARM TrustZone

- Timing difference for different RSA signature keys
- No knowledge of the TrustZone/trustlet implementation
- Valid keys and invalid keys are distinguishable



- full attack (on AES) was performed by Zhang et al. 2016

It’s not fast enough! Let’s add more optimizations!

`PRFM` / `prefetch`: Unusual instructions (1)

- tells the CPU “I might need that later”

`PRFM` / `prefetch`: Unusual instructions (1)

- tells the CPU “I might need that later”
- hint—may be ignored by the CPU

`PRFM` / `prefetch`: Unusual instructions (1)

- tells the CPU “I might need that later”
- hint—may be ignored by the CPU
- generates no faults

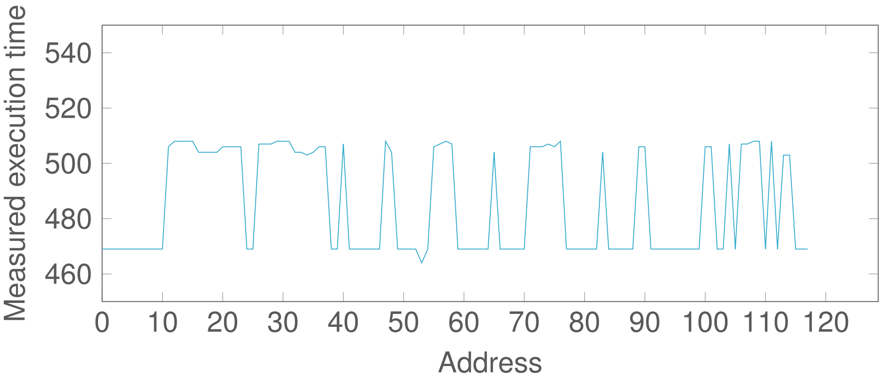
`PRFM` / `prefetch`: Unusual instructions (2)

- operand is a virtual address

`PRFM` / `prefetch`: Unusual instructions (2)

- operand is a virtual address
- but it needs to translate the virtual address to a physical address

Address space scan on OnePlus One



Flush+Flush

- Execution time of the flush instruction varies
  - if the address is cached
  - or not cached

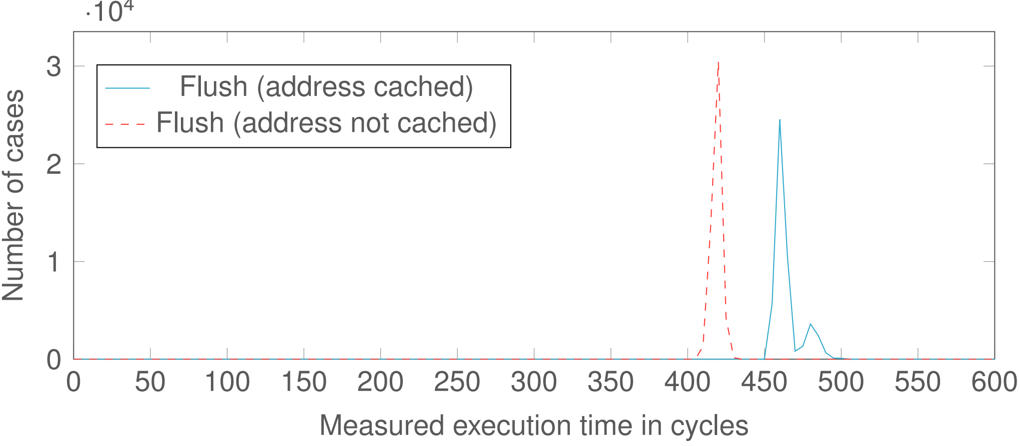
Flush+Flush

- Execution time of the flush instruction varies
  - if the address is cached
  - or not cached
- No additional memory accesses (stealthy)

Flush+Flush

- Execution time of the flush instruction varies
  - if the address is cached
  - or not cached
- No additional memory accesses (stealthy)
- Higher frequency

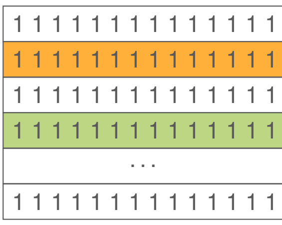
Flush+Flush on the Samsung Galaxy S6



At least the hardware still functionally works as it is supposed to...

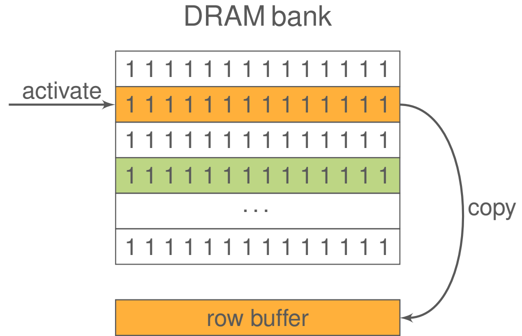
Rowhammer

“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



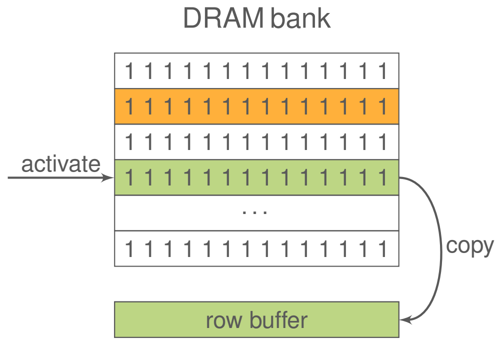
Rowhammer

"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



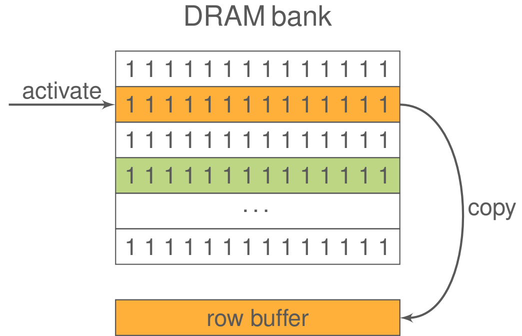
Rowhammer

"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



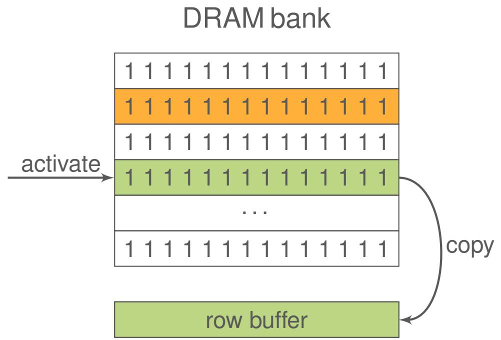
Rowhammer

“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



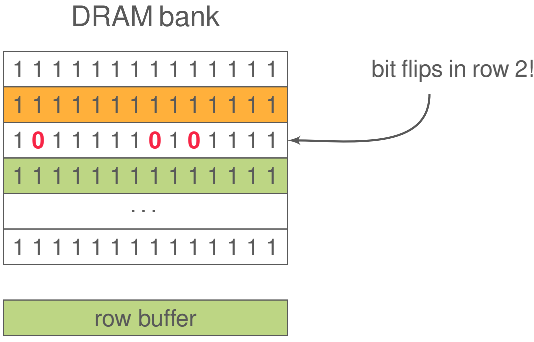
Rowhammer

"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



Rowhammer

“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



Requirements for Rowhammer

Memory accesses must be

- uncached to reach DRAM
- fast enough to win the race against the next row refresh
- targeted and hit specific DRAM rows

Access techniques

1. `clflush` instruction $\rightarrow$ original paper (Kim et al. 2014)
2. cache eviction (Gruss, Maurice, and Mangard 2016; Aweke et al. 2016)
3. non-temporal accesses (Qiao et al. 2016)
4. uncached memory (Veen et al. 2016)

Physical addresses and DRAM

- fixed map: physical addresses $\rightarrow$ DRAM cells
- undocumented for most processors

Physical addresses and DRAM

- fixed map: physical addresses $\rightarrow$ DRAM cells
- undocumented for most processors
- can be automatically reverse-egineered based on timing differences (Pessl et al. 2016)

How to exploit random bit flips?

- They are not random $\rightarrow$ highly reproducible flip pattern!

1. choose a data structure that you can place at arbitrary memory locations
2. scan for “good” flips
3. place data structure there
4. trigger bit flip again

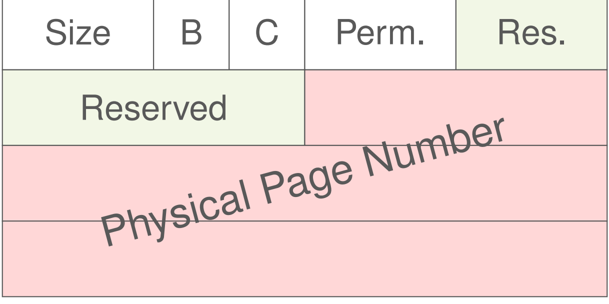
Page Table Entries

Size	B	C	Perm.	

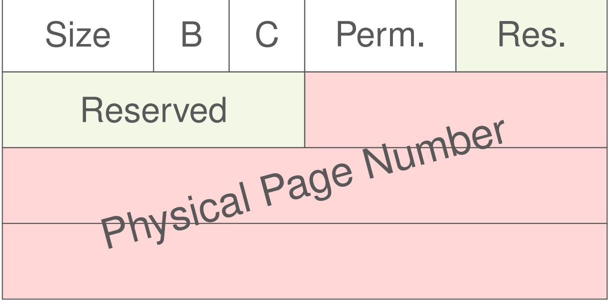
Page Table Entries

<table>
<tr><td>Size</td><td>B</td><td>C</td><td>Perm.</td><td>Res.</td></tr>
<tr><td colspan="3">Reserved</td><td colspan="2"></td></tr>
<tr><td colspan="5"></td></tr>
<tr><td colspan="5"></td></tr>
</table>

Page Table Entries

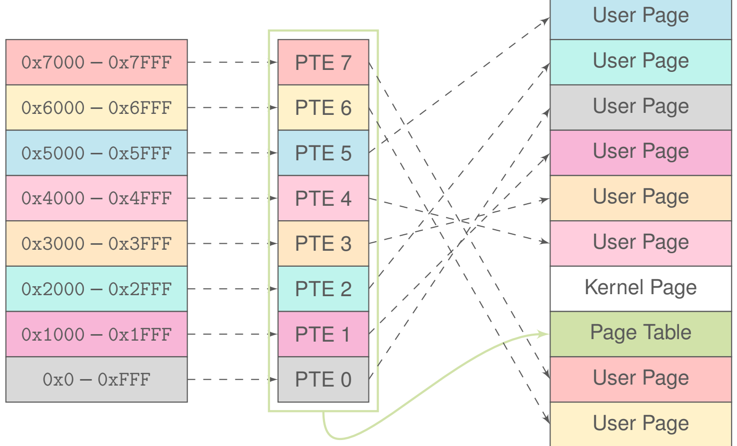


Page Table Entries

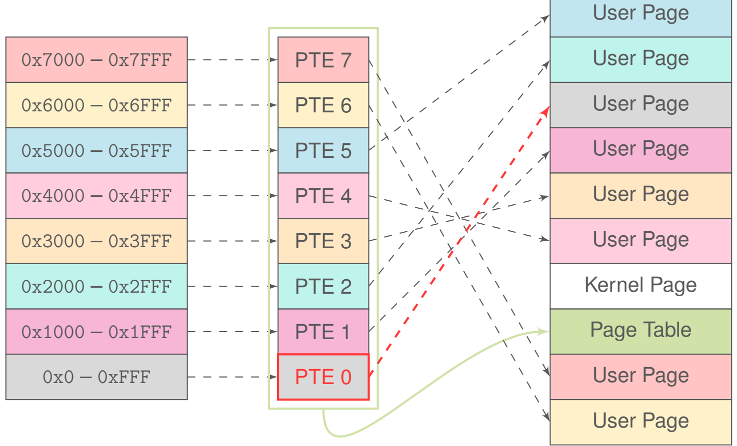


Most bits are physical page number bits

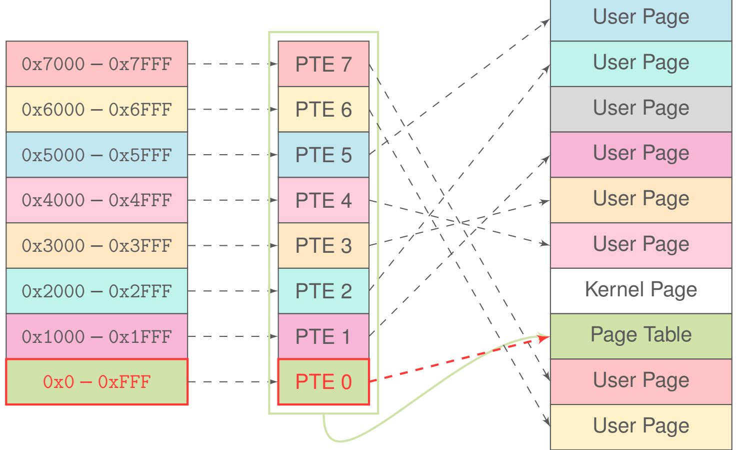
Page Table Manipulation



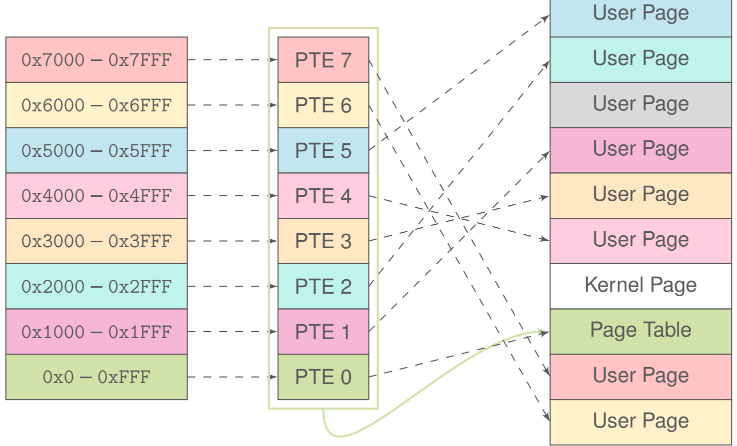
Page Table Manipulation



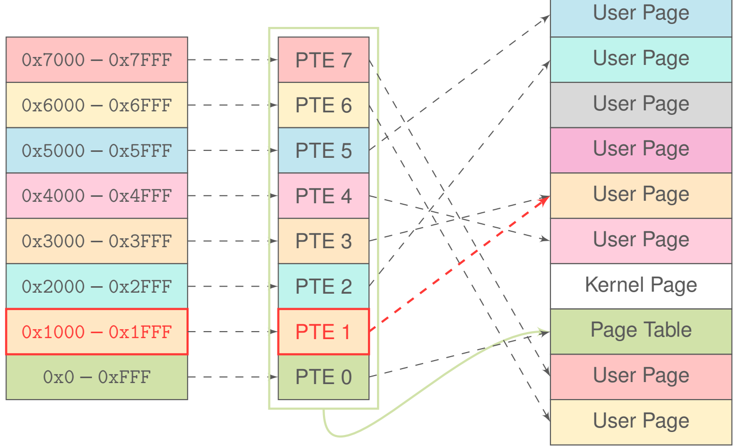
Page Table Manipulation



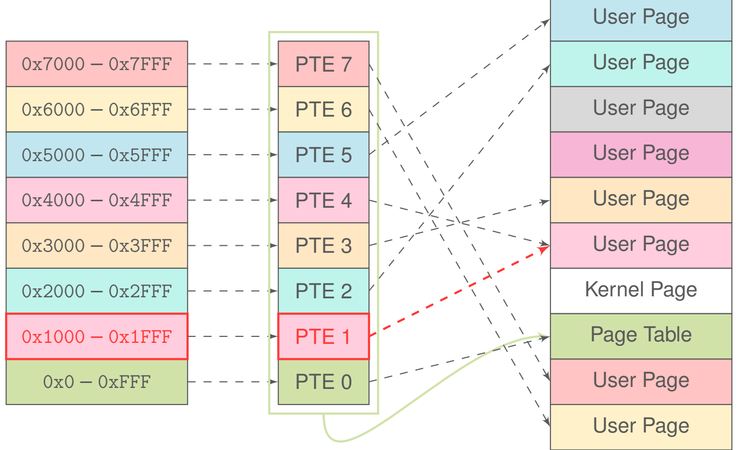
Page Table Manipulation



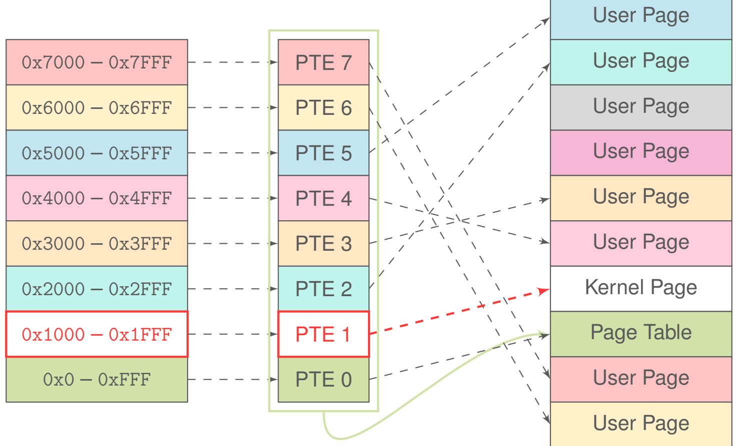
Page Table Manipulation



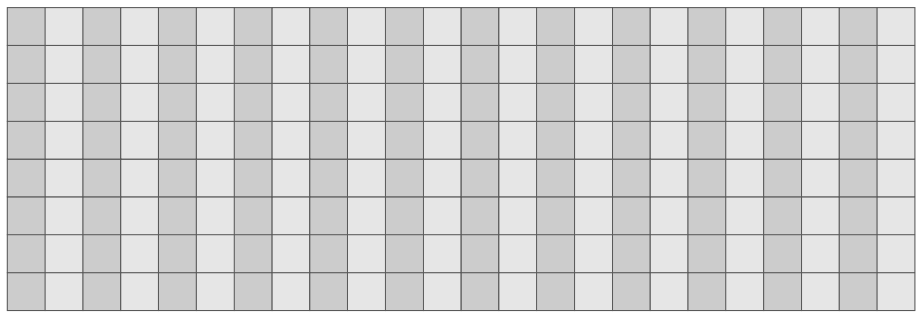
Page Table Manipulation



Page Table Manipulation

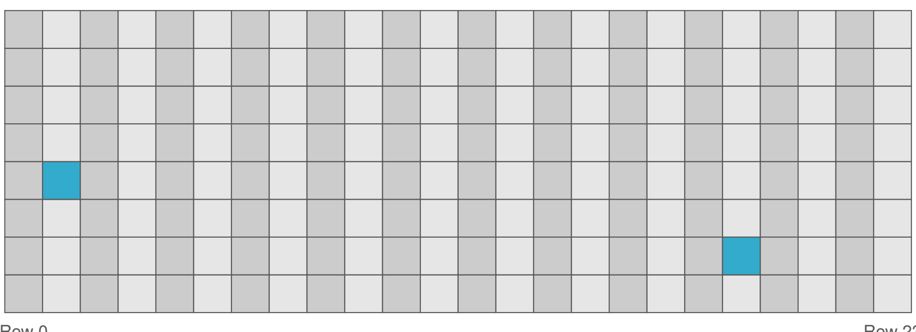


Search for page with flip



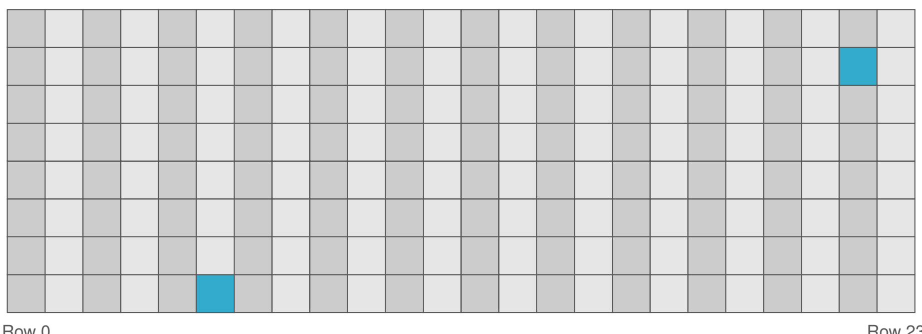
Hammering memory locations in different rows

Search for page with flip



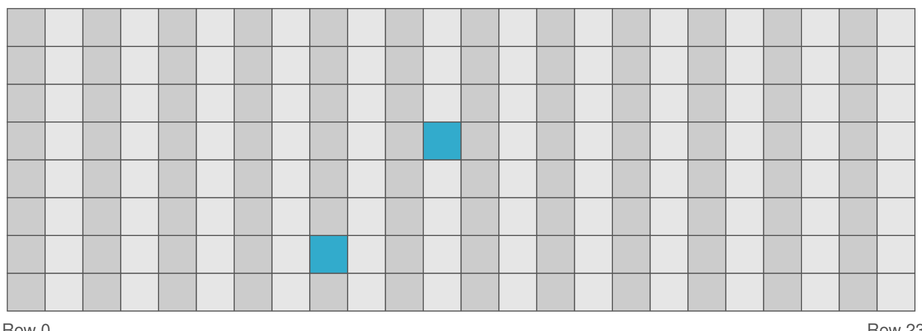
Hammering memory locations in different rows

Search for page with flip



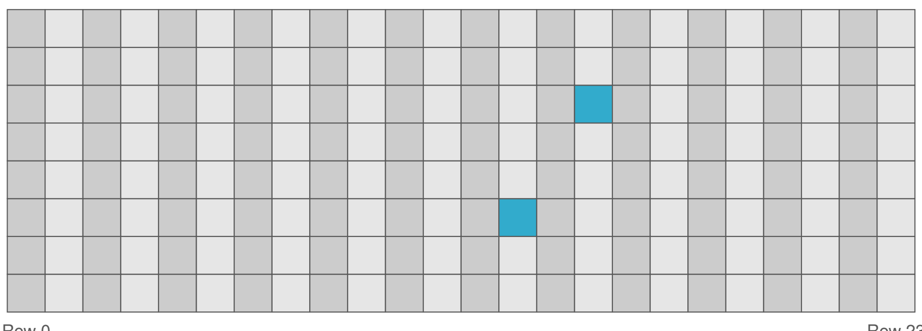
Hammering memory locations in different rows

Search for page with flip



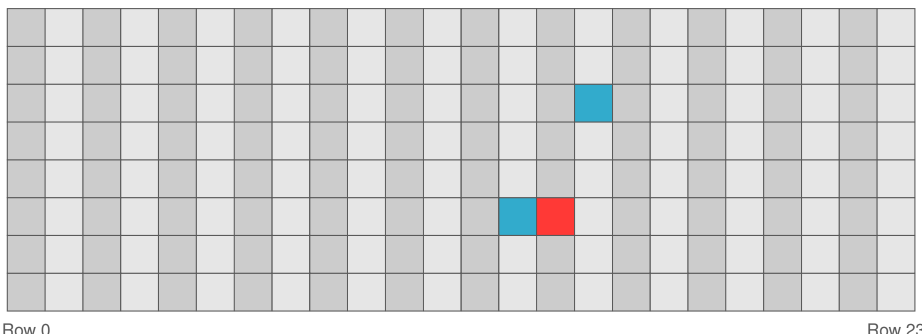
Hammering memory locations in different rows

Search for page with flip



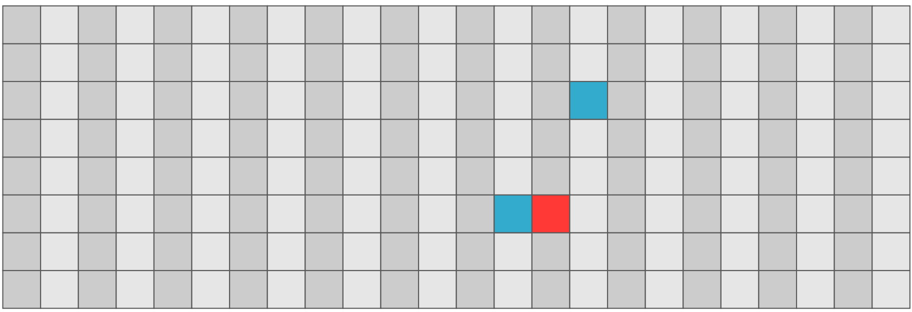
Hammering memory locations in different rows

Search for page with flip

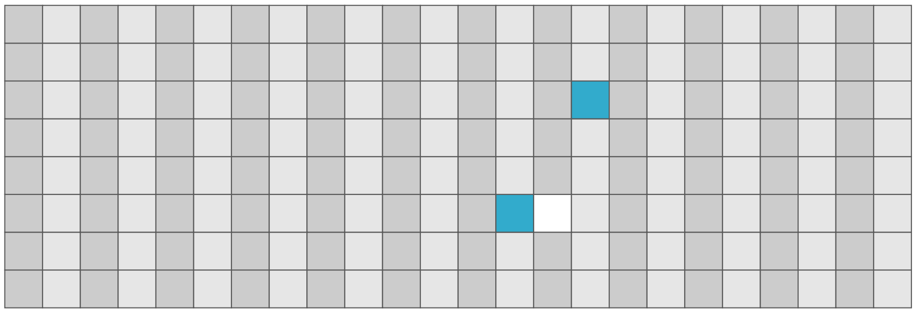


Hammering memory locations in different rows

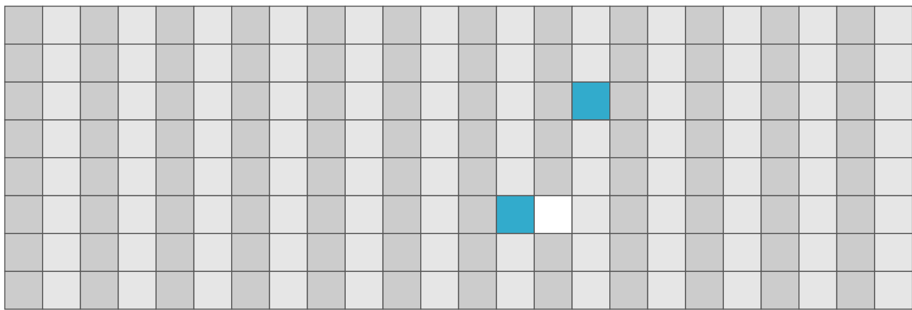
Release page with flip



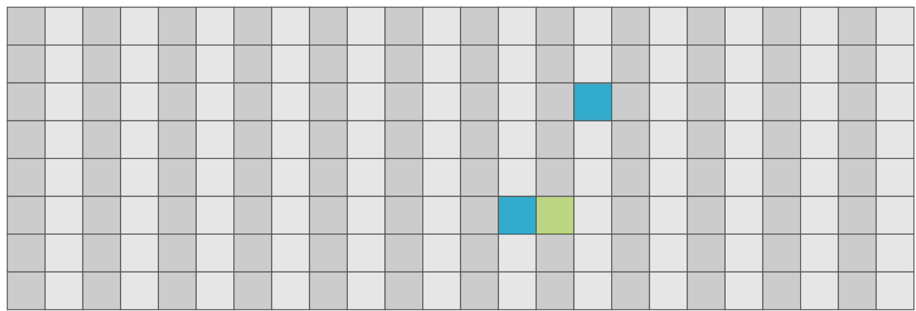
Release page with flip



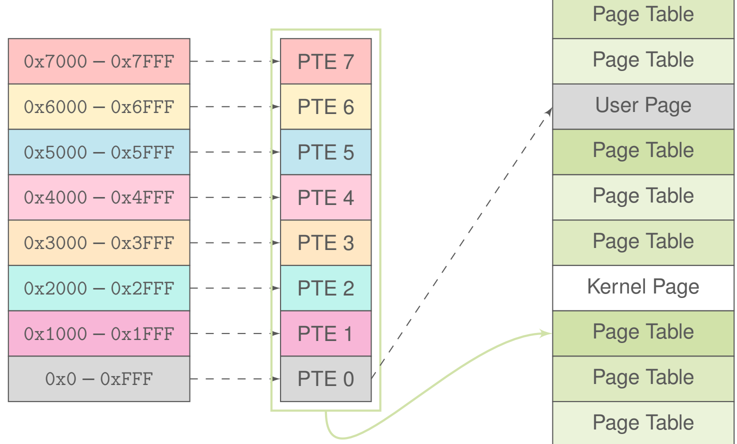
Fill all remaining memory with page tables



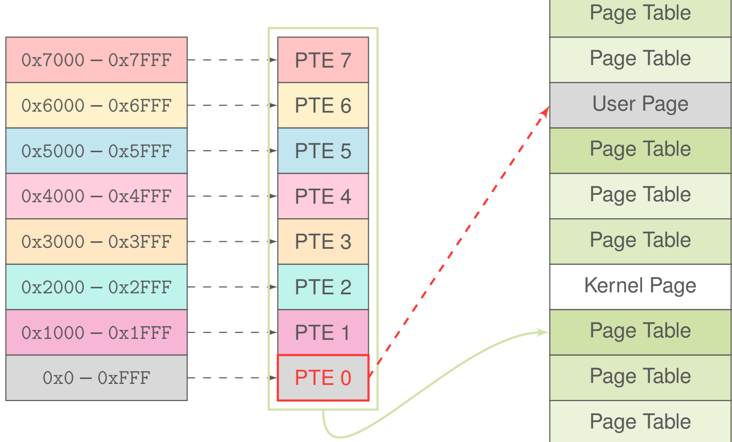
Fill all remaining memory with page tables



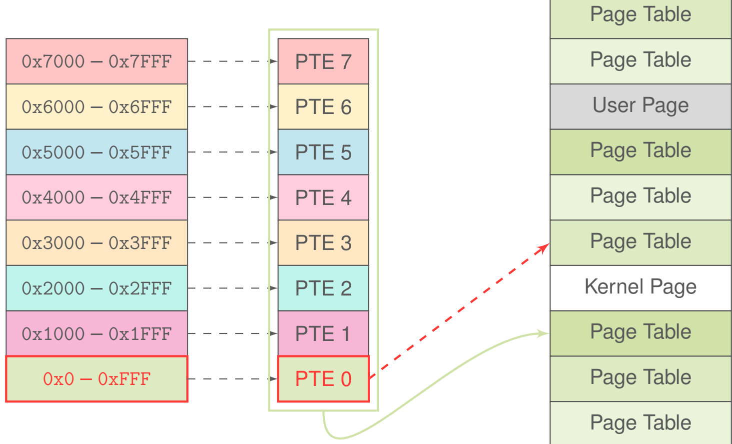
Page Table Manipulation



Page Table Manipulation



Page Table Manipulation



Strategy: Flipping Page Table PPN bits

1. scan for flips
2. exhaust or massage memory to place a page table at target location
3. gain access to your own page table $\rightarrow$ kernel privileges

Flipping Page Table PPN bits

- idea from Seaborn et al. 2015
- same idea applied in several other works:
  - Rowhammer.js (Gruss, Maurice, and Mangard 2016)
  - One bit flips, one cloud flops (Xiao et al. 2016)
  - Drammer (Veen et al. 2016)

(Post-)Rowhammer Exploitation

- modify binary pages executed in root privileges (Xiao et al. 2016)
- modify credential structs (Veen et al. 2016)
- read keys (Xiao et al. 2016)
- corrupt RSA signatures (Bhattacharya et al. 2016)
- modify certificates / configurations (Razavi et al. 2016)

Is there a happy end to this story?

Cache Attack Countermeasures

- Coarse-grained timers

Cache Attack Countermeasures

- Coarse-grained timers $\rightarrow$ thread timer is sufficient...

Cache Attack Countermeasures

- Coarse-grained timers $\rightarrow$ thread timer is sufficient...
- No shared memory

Cache Attack Countermeasures

- Coarse-grained timers → thread timer is sufficient...
- No shared memory → What about Prime+Probe?

Cache Attack Countermeasures

- Coarse-grained timers → thread timer is sufficient...
- No shared memory → What about Prime+Probe?
- Restrict system information, e.g., `pagemap`

Cache Attack Countermeasures

- Coarse-grained timers → thread timer is sufficient. . .
- No shared memory → What about Prime+Probe?
- Restrict system information, e.g., `pagemap` → Harder but still possible

Cache Attack Countermeasures

- Coarse-grained timers → thread timer is sufficient...
- No shared memory → What about Prime+Probe?
- Restrict system information, e.g., `pagemap` → Harder but still possible
- Use cryptographic instruction extensions

Cache Attack Countermeasures

- Coarse-grained timers → thread timer is sufficient...
- No shared memory → What about Prime+Probe?
- Restrict system information, e.g., `pagemap` → Harder but still possible
- Use cryptographic instruction extensions

→ Still not the default everywhere...

Cache Attack Countermeasures

- Coarse-grained timers → thread timer is sufficient...
- No shared memory → What about Prime+Probe?
- Restrict system information, e.g., `pagemap` → Harder but still possible
- Use cryptographic instruction extensions

→ Still not the default everywhere...

→ Doesn't protect from spying user behavior...

Cache Attack Countermeasures

- Coarse-grained timers → thread timer is sufficient...
- No shared memory → What about Prime+Probe?
- Restrict system information, e.g., `pagemap` → Harder but still possible
- Use cryptographic instruction extensions

  → Still not the default everywhere...

  → Doesn't protect from spying user behavior...

→ Protect crypto.

Cache Attack Countermeasures

- Coarse-grained timers → thread timer is sufficient...
- No shared memory → What about Prime+Probe?
- Restrict system information, e.g., `pagemap` → Harder but still possible
- Use cryptographic instruction extensions

  → Still not the default everywhere...
  → Doesn't protect from spying user behavior...

→ Protect crypto. For the rest...

Cache Attack Countermeasures

- Coarse-grained timers → thread timer is sufficient...
- No shared memory → What about Prime+Probe?
- Restrict system information, e.g., `pagemap` → Harder but still possible
- Use cryptographic instruction extensions

  → Still not the default everywhere...
  → Doesn't protect from spying user behavior...

→ Protect crypto. For the rest... ongoing research, stay tuned

Rowhammer Countermeasures

Most promising: Probabilistic Adjacent Row Activation (Kim et al. 2014)

- one row closed $\rightarrow$ one adjacent row opened with low probability p

Rowhammer Countermeasures

Most promising: Probabilistic Adjacent Row Activation (Kim et al. 2014)

- one row closed $\rightarrow$ one adjacent row opened with low probability p
- Rowhammer: one row opened and closed a high number of times N_{th}

Rowhammer Countermeasures

Most promising: Probabilistic Adjacent Row Activation (Kim et al. 2014)

- one row closed $\rightarrow$ one adjacent row opened with low probability p
- Rowhammer: one row opened and closed a high number of times N_{th}
- statistically, neighbor rows are refreshed $\rightarrow$ no bit flip

Rowhammer Countermeasures

Most promising: Probabilistic Adjacent Row Activation (Kim et al. 2014)

- one row closed $\rightarrow$ one adjacent row opened with low probability p
- Rowhammer: one row opened and closed a high number of times N_{th}
- statistically, neighbor rows are refreshed $\rightarrow$ no bit flip
- implementation at the memory controller level

Rowhammer Countermeasures

Most promising: Probabilistic Adjacent Row Activation (Kim et al. 2014)

- one row closed $\rightarrow$ one adjacent row opened with low probability p
- Rowhammer: one row opened and closed a high number of times N_{th}
- statistically, neighbor rows are refreshed $\rightarrow$ no bit flip
- implementation at the memory controller level
- for $p = 0.001$ and $N_{th} = 100K$, experiencing one error in one year has a probability 9.4×10^{-14}

Rowhammer Countermeasures

Most promising: Probabilistic Adjacent Row Activation (Kim et al. 2014)

- one row closed $\rightarrow$ one adjacent row opened with low probability p
- Rowhammer: one row opened and closed a high number of times N_{th}
- statistically, neighbor rows are refreshed $\rightarrow$ no bit flip
- implementation at the memory controller level
- for $p = 0.001$ and $N_{th} = 100K$, experiencing one error in one year has a probability 9.4×10^{-14}
- cheap and fast

Take aways

Conclusion

- cache attacks are a realistic threat on mobile devices
  - no permissions, no privileges
- in many cases performance comes with side-channel leaks
- fast uncached memory accesses make Rowhammer attacks possible

Conclusion

- cache attacks are a realistic threat on mobile devices
  - no permissions, no privileges
- in many cases performance comes with side-channel leaks
- fast uncached memory accesses make Rowhammer attacks possible

or: “With great speed comes great leakage”

With great speed comes great leakage

How processor performance is tied to side-channel leakage

Moritz Lipp & Daniel Gruss, Graz University of Technology

May 18, 2017 — Qualcomm Mobile Security Summit

References I

Aweke, Zelalem Birhanu, Salessawi Ferede Yitbarek, Rui Qiao, Reetuparna Das, Matthew Hicks, Yossi Oren, and Todd Austin (2016). “ANVIL: Software-based protection against next-generation rowhammer attacks”. In: ACM SIGPLAN Notices 51.4, pp. 743–755.

Bhattacharya, Sarani and Debdeep Mukhopadhyay (2016). “Curious case of Rowhammer: Flipping Secret Exponent Bits using Timing Analysis”. In: CHES’16.

Gruss, Daniel, David Bidner, and Stefan Mangard (2015). “Practical Memory Deduplication Attacks in Sandboxed JavaScript”. In: 20th European Symposium on Research in Computer Security (ESORICS’15).

References II

Gruss, Daniel, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard (2016). “Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR”. In: CCS’16.

Gruss, Daniel, Clémentine Maurice, and Stefan Mangard (2016). “Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript”. In: DIMVA’16.

Gruss, Daniel, Clémentine Maurice, Klaus Wagner, and Stefan Mangard (2016). “Flush+Flush: A Fast and Stealthy Cache Attack”. In: DIMVA’16.

Gruss, Daniel, Raphael Spreitzer, and Stefan Mangard (2015). “Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches”. In: USENIX Security Symposium.

References III

Kim, Yoongu, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu (2014). "Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors". In: ISCA'14.

Lipp, Moritz, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard (2016). "ARMageddon: Cache Attacks on Mobile Devices". In: USENIX Security Symposium.

Maurice, Clémentine, Manuel Weber, Michael Schwarz, Lukas Giner, Daniel Gruss, Carlo Alberto Boano, Stefan Mangard, and Kay Römer (2017). "Hello from the Other Side: SSH over Robust Cache Covert Channels in the Cloud". In: NDSS'17. to appear.

References IV

Pessl, Peter, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard (2016). "DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks". In: USENIX Security Symposium.

Qiao, Rui and Mark Seaborn (2016). "A new approach for rowhammer attacks". In: HOST 2016.

Razavi, Kaveh, Ben Gras, Erik Bosman, Bart Preneel, Cristiano Giuffrida, and Herbert Bos (2016). "Flip Feng Shui: Hammering a Needle in the Software Stack". In: USENIX Security Symposium.

Schwarz, Michael, Clémentine Maurice, Daniel Gruss, and Stefan Mangard (2017). "Fantastic Timers and Where to Find Them: High-Resolution Microarchitectural Attacks in JavaScript". In: Proceedings of the 21th International Conference on Financial Cryptography and Data Security (FC'17).

References V

Seaborn, Mark and Thomas Dullien (2015). “Exploiting the DRAM rowhammer bug to gain kernel privileges”. In: Black Hat 2015 Briefings.

Veen, Victor van der, Yanick Fratantonio, Martina Lindorfer, Daniel Gruss, Clémentine Maurice, Giovanni Vigna, Herbert Bos, Kaveh Razavi, and Cristiano Giuffrida (2016). “Drammer: Deterministic Rowhammer Attacks on Mobile Platforms”. In: CCS’16.

Xiao, Yuan, Xiaokuan Zhang, Yinqian Zhang, and Radu Teodorescu (2016). “One Bit Flips, One Cloud Flops: Cross-VM Row Hammer Attacks and Privilege Escalation ”. In: USENIX Security Symposium.

Zhang, Ning, Kun Sun, Deborah Shands, Wenjing Lou, and Y Thomas Hou (2016). “TruSpy: Cache Side-Channel Information Leakage from the Secure World on ARM Devices”. In: Cryptology ePrint Archive: Report 2016/980.