

Drammer: Deterministic Rowhammer Attacks on Mobile Platforms

Victor van der Veen[§] Yanick Fratantonio[†] Martina Lindorfer[†]
Daniel Gruss[‡] Clémentine Maurice[‡] Giovanni Vigna[†]
Herbert Bos[§] Kaveh Razavi[§] Cristiano Giuffrida[§]

[§] **Vrije Universiteit Amsterdam**[†] **UC Santa Barbara**
[‡] **Graz University of Technology**

Presented at CCS'16, October 24–28, 2016, Vienna, Austria

Presented by Kevin Thommen
28/05/2020

Outline

- Background, Motivation & Goal
- The Attack - DRAMMER
 - Preparation
 - Implementing the Primitives on Mobile Devices
- Evaluation of Attack
- Mitigation Techniques
- Summary
- Critique
- Discussion

Executive Summary

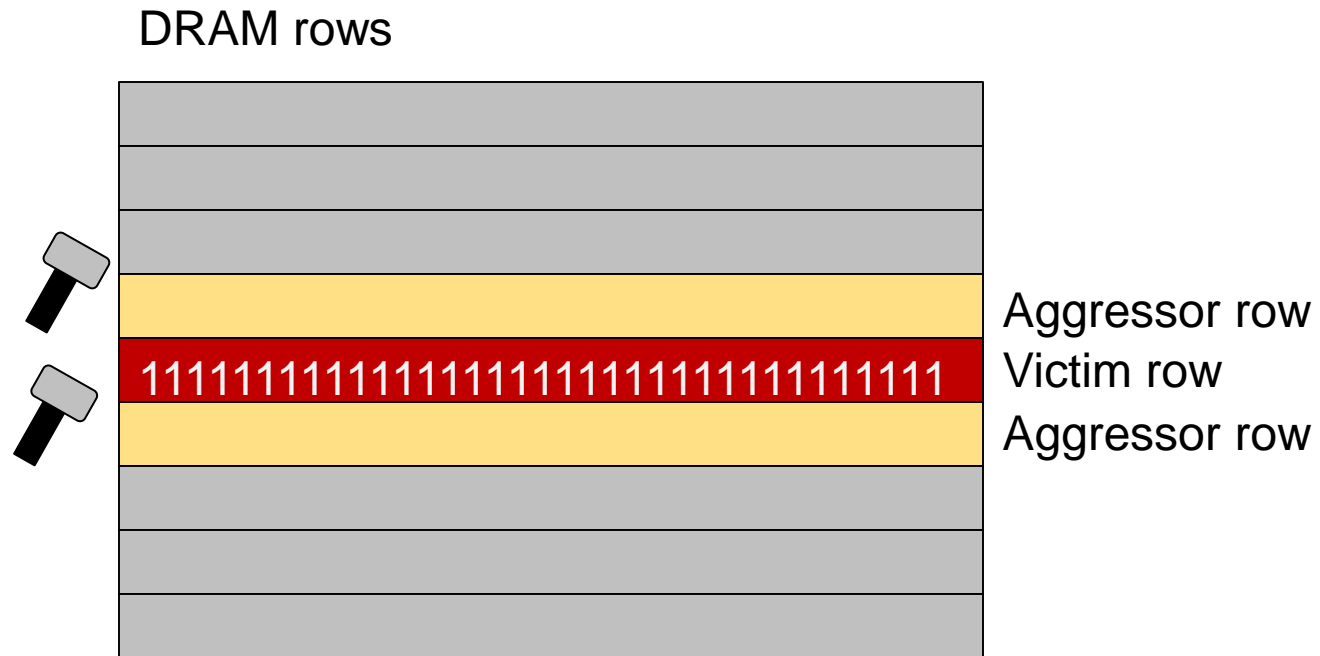
- **Motivation:** Current Rowhammer exploits are mostly **probabilistic** and most studies only focus on **x86**.
- **Goal:** Implement **deterministic** Rowhammer attack on **ARM** devices.
- **Challenges:**
 - ❑ Fast uncached memory access
 - ❑ Put page table into Rowhammer exploitable memory
 - ❑ Find aggressor rows
- **Key Ideas:**
 - ❑ Use Android's DMA buffers to get uncached contiguous memory
 - ❑ Use predictability of memory allocator to put a page table into exploitable memory
- **Result:** Many **ARMv7** devices and one **ARMv8** device have been **successfully exploited** using DRAMMER.

Background, Motivation & Goals

The Rowhammer Hardware Vulnerability

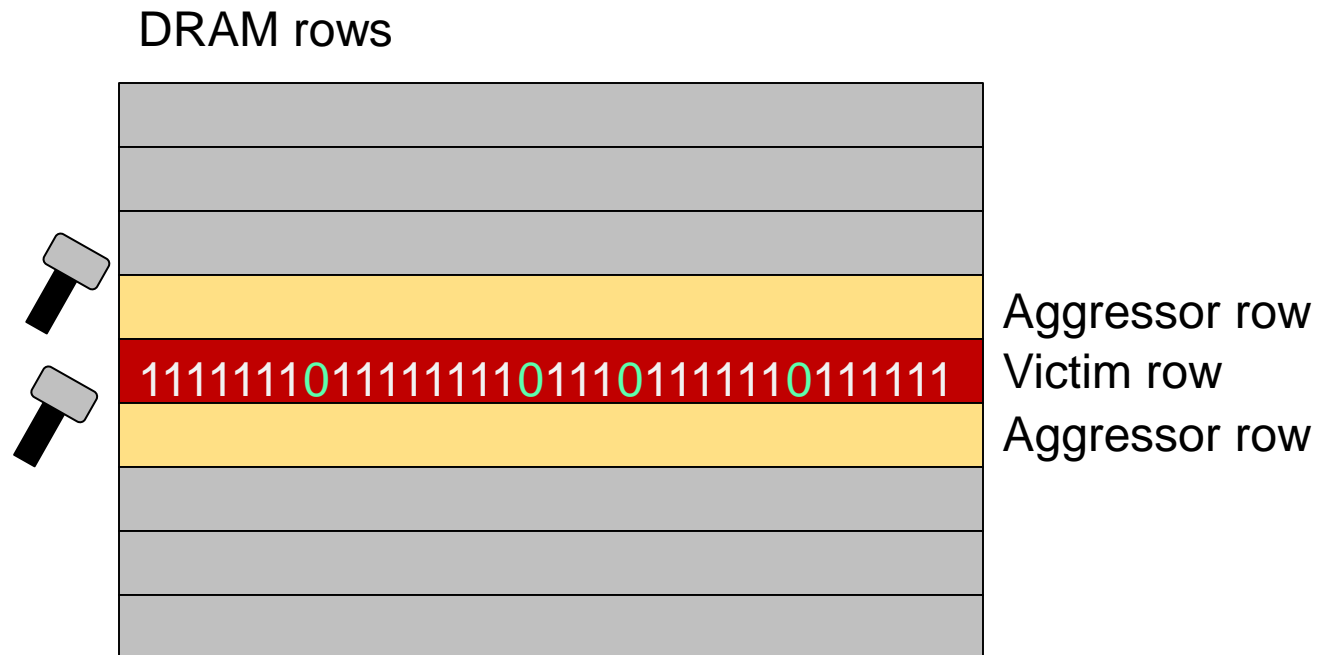


A Software-Induced Hardware Fault



- Repeatedly accessing ("hammering") aggressor rows can cause bit flips in neighboring rows. The example above demonstrates a double-sided Rowhammer.

A Software-Induced Hardware Fault



- Repeatedly accessing (“hammering”) aggressor rows can cause bit flips in neighboring rows. The example above demonstrates a **double-sided** Rowhammer.

Motivation

- No previous study has been done on the impact of Rowhammer on ARM devices.
- All known techniques only target x86 architectures and cannot be readily translated to ARM.
- Prior to this paper it was unclear if the Rowhammer vulnerability even occurs on ARM devices.
- Exploitation techniques are either probabilistic or rely on special memory management features.
- These probabilistic Rowhammer attacks offer weak reliability guarantees:
 - No guarantee that victim object is actually placed in vulnerable physical memory location
 - No reliable prediction of outcome of corruption the victim object.

Goals

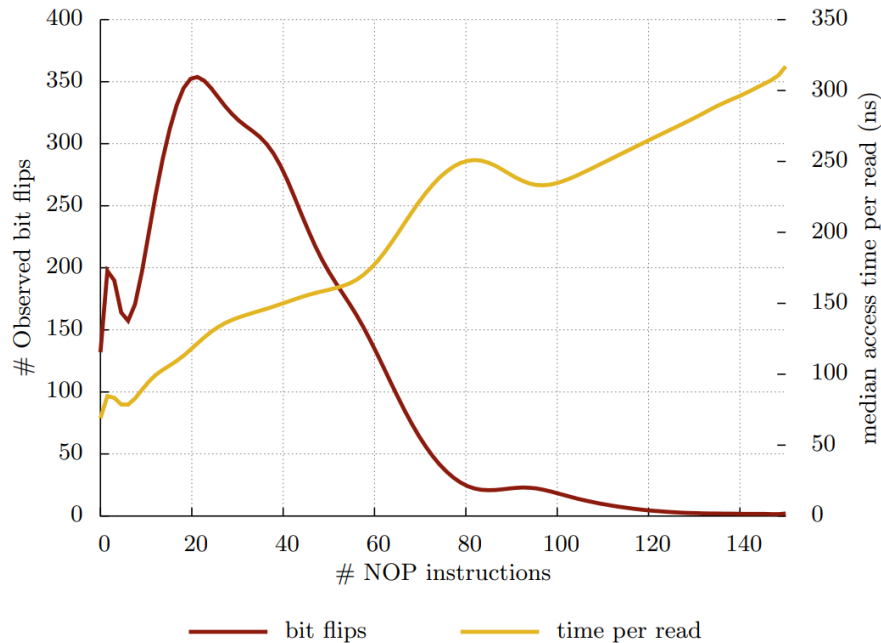
- Show that deterministic Rowhammer exploitation using only commodity features on Android/ARM is possible.
- Evaluate effectiveness of DRAMMER.

DRAMMER - Preparation

Rowhammer possible on ARM?

- ARM memory controllers are slower than the ones used in x86.
- Is the memory controller too slow to trigger the Rowhammer vulnerability?

Rowhammer possible on ARM?



- Double-sided Rowhammer on the same 5MB of physical memory with artificially increasing time in between two read operations. Performed on LG Nexus 5 device running Android 6.0.1.
- The attack was performed with full privileges.
- This shows that the Rowhammer vulnerability can be induced.

Threat Model

Following assumptions are made:

- We run an **ARM**-based device running Android 6.0.1 with:
 - all **updates** installed
 - all **security measures** activated
 - and **no special features** enabled
- The attacker has control over an **unprivileged Android App** without any permissions.
- The attacker aims to perform a privilege **escalation attack** to acquire **root privilege**.

Rowhammer - The Three Exploitation Primitives

- **P1. Fast uncached memory access.** The ability to activate rows **fast enough** to trigger the Rowhammer bug while **bypassing cache**.
- **P2. Physical memory massaging.** **Trick the victim component** to use a memory cell that is subject to the Rowhammer bug.
- **P3. Physical memory addressing.** Understand how **physical memory addresses** are used in the **virtual address space** of an unprivileged process.

How are these primitives handled by **currently known Rowhammer exploitation techniques?**

Rowhammer - The Three Exploitation Primitives

- **P1. Fast uncached memory access.** The ability to activate rows **fast enough** to trigger the Rowhammer bug while **bypassing cache**.
- **P2. Physical memory massaging.** **Trick the victim component** to use a memory cell that is subject to the Rowhammer bug.
- **P3. Physical memory addressing.** Understand how **physical memory addresses** are used in the **virtual address space** of an unprivileged process.

How are these primitives handled by **currently known Rowhammer exploitation techniques?**

The Three Exploitation Primitives – P1

- **P1. Fast uncached memory access.** The ability to activate rows **fast enough** to trigger the Rowhammer bug while **bypassing cache**.
 - CPU **memory might not be fast enough** and read instructions are masked out by multiple **layers of cache**.
- **x86 techniques:**
 - **Explicit cache flush** using the **clflush** instruction.
 - Repeatedly access addresses belonging to the same **cache eviction set**.
 - Issue memory reads using **non-temporal access instructions**.

- The cache flush on Android is **privileged**.
- Using cache eviction sets is **too slow** to trigger vulnerability.
- ARM non-temporal instructions serve only as **hints** for CPU.

The Three Exploitation Primitives

- **P1. Fast uncached memory access.** The ability to activate rows **fast enough** to trigger the Rowhammer bug while **bypassing cache**.
- **P2. Physical memory massaging.** **Trick the victim component** to use a memory cell that is subject to the Rowhammer bug.
- **P3. Physical memory addressing.** Understand how **physical memory addresses** are used in the **virtual address space** of an unprivileged process.

The Three Exploitation Primitives – P2

- **P2. Physical memory massaging.** Trick the victim component to use a memory cell that is subject to the Rowhammer bug.
 - ❑ “Massaging” memory precisely enough to push the victim page to use the vulnerable cell to store security-sensitive data.
 - x86 techniques:
 - ❑ Spray memory with page tables, hoping for one of them to land in a vulnerable physical memory page.
 - ❑ Memory deduplication
 - ❑ MMU paravirtualization.
- Spraying memory is probabilistic and the other features are not enabled by default or don't exist on stock Android.

The Three Exploitation Primitives

- **P1. Fast uncached memory access.** The ability to activate rows **fast enough** to trigger the Rowhammer bug while **bypassing cache**.
- **P2. Physical memory massaging.** **Trick the victim component** to use a memory cell that is subject to the Rowhammer bug.
- **P3. Physical memory addressing.** Understand how **physical memory addresses** are used in the **virtual address space** of an unprivileged process.

The Three Exploitation Primitives – P3

- **P3. Physical memory addressing.** Understand how **physical memory addresses** are used in the **virtual address space** of an unprivileged process.
 - Find virtual addresses that map to aggressor rows.
- x86 techniques:
 - Access the **pagemap interface** file which contains complete information about the mapping of virtual to physical addresses
 - Use **huge virtual pages** that are backed by physically contiguous physical pages.

- The pagemap is not available in **userland**.
 - Huge virtual pages are **not enabled** by default.

DRAMMER – Implementing the Primitives on Mobile Devices

Attacks on Android

P1. Fast uncached memory access.

P2. Physical memory massaging.

P3. Physical memory addressing.

- How do we implement these three primitives on Android?

Support for P1 and P3

- To support efficient memory sharing between different hardware components, the OS provides **direct memory access (DMA)** memory management mechanisms.
- Android provides DMA Buffer Management APIs through its main memory manager called **ION**, giving userland apps access to **uncached, physically contiguous memory**.

Support for P1 and P3

- **P1. Fast uncached memory access:**

- Processing pipelines involving DMA buffers **bypass the CPU** and its caches.

- **P3. Physical memory addressing:**

- Most devices perform DMA operations to **physically contiguous memory pages** only, so the OS provides allocators that support this kind of memory.

P2. Physical Memory Massaging

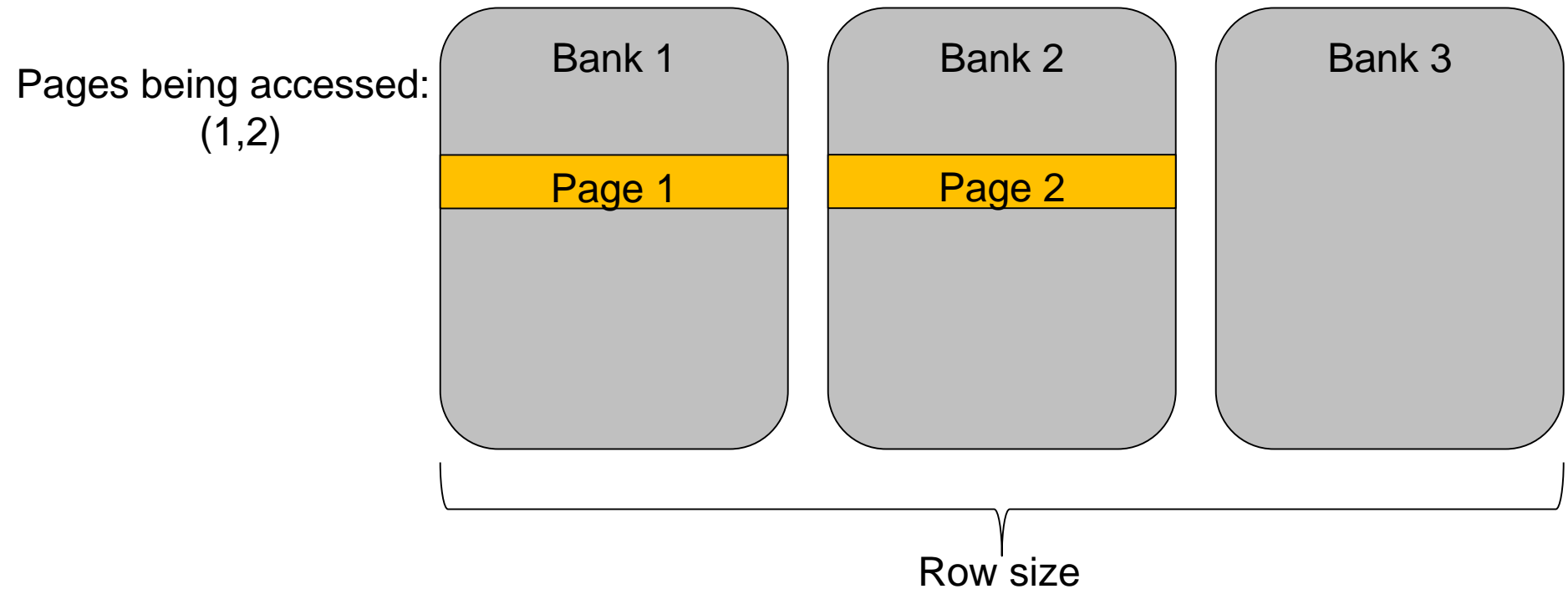
- To **deterministically land security-sensitive data** in a vulnerable physical memory page following steps are taken:
 - ❑ **Determine DRAM chip's row size** to understand memory model for later **templating**
 - ❑ **Memory Templating** to find memory locations **susceptible** to Rowhammer
 - ❑ **Land sensitive data** in **susceptible location**
 - ❑ **Reproduce the bit flip** with **security-sensitive data** now in susceptible location

P2. Physical Memory Massaging

- ❑ **Determine DRAM chip's row size** to understand memory model for later templating
- ❑ **Memory Templating** to find memory locations susceptible to Rowhammer
- ❑ **Land sensitive data** in susceptible location
- ❑ **Reproduce the bit flip** with security-sensitive data now in susceptible location

Determine DRAM chip's row size

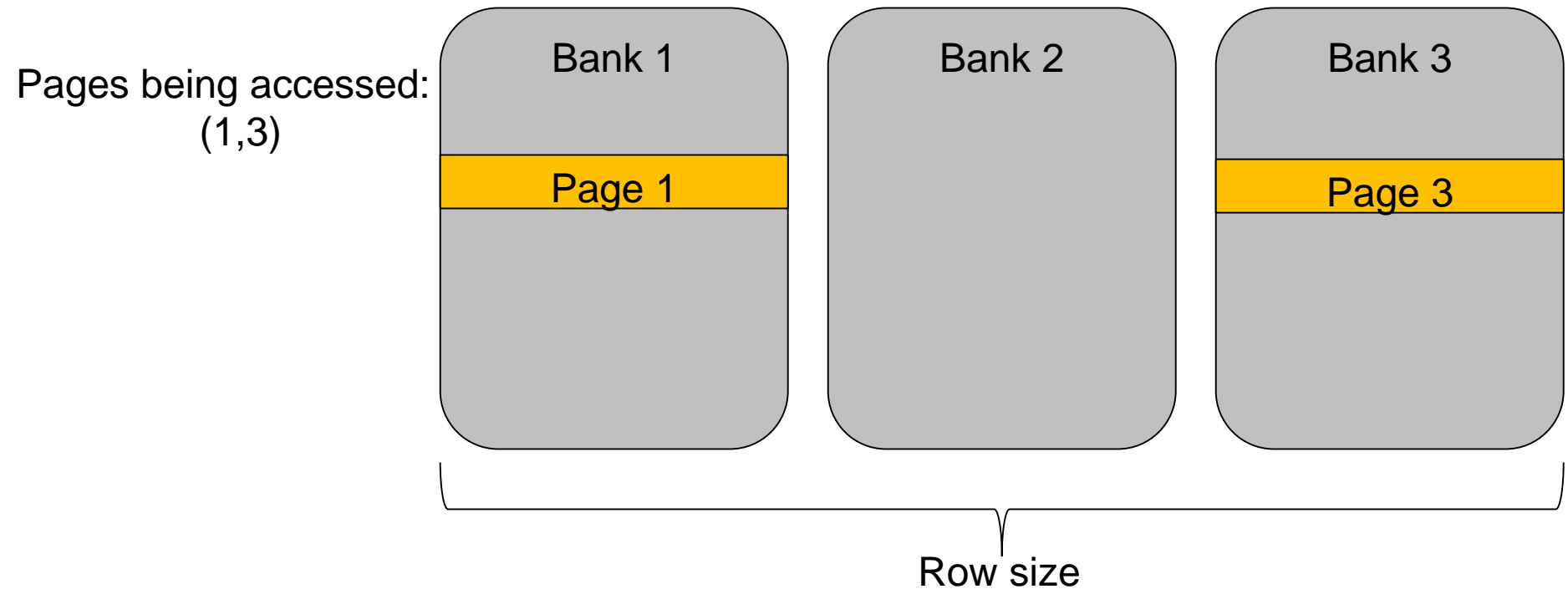
- ARM does not document row size, nor does it provide a instructions for fingerprinting DRAM modules.
- Use a timing-based side channel:



- Reading from same bank takes longer than reading from two different banks.

Determine DRAM chip's row size

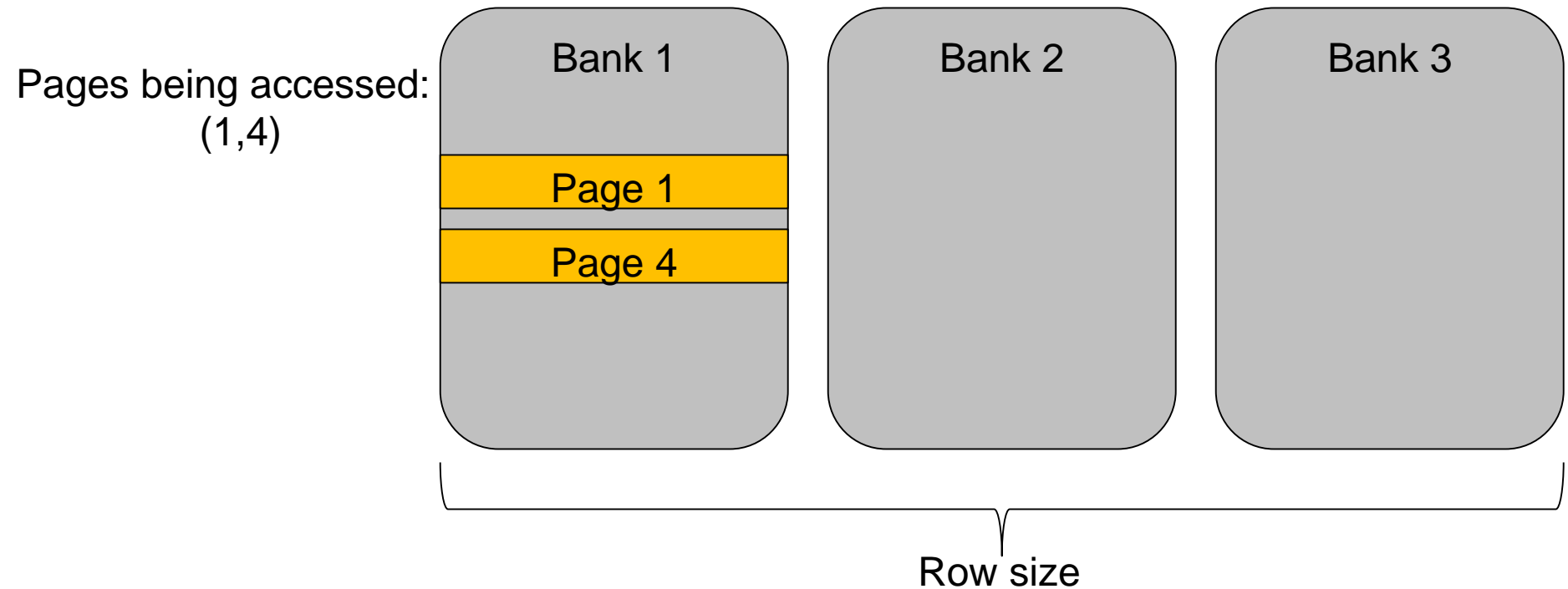
- ARM does not document row size, nor does it provide a instructions for fingerprinting DRAM modules.
- Use a timing-based side channel:



- Reading from same bank takes longer than reading from two different banks.

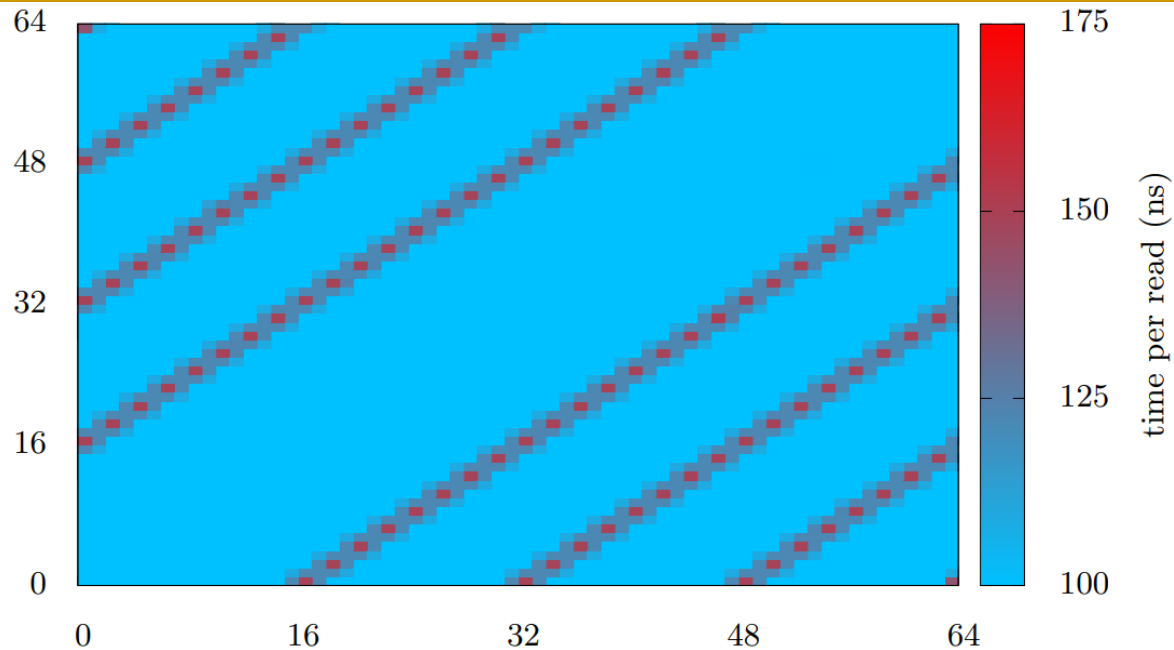
Determine DRAM chip's row size

- ARM does not document row size, nor does it provide a instructions for fingerprinting DRAM modules.
- Use a timing-based side channel:



- Reading from same bank takes longer than reading from two different banks.

Determine DRAM chip's row size



- Allows us to create heatmap representing time required to access a given pair of edges. Here the row size is 16 pages = 64KB.
- The x-axis and y-axis describe the page table that's being accessed.

P2. Physical Memory Massaging

- ❑ **Determine DRAM chip's row size** to understand memory model for later **templating**
- ❑ **Memory Templating** to find memory locations **susceptible** to Rowhammer
- ❑ **Land sensitive data** in **susceptible location**
- ❑ **Reproduce the bit flip** with **security-sensitive** data now in susceptible location

Memory Templating on ARM

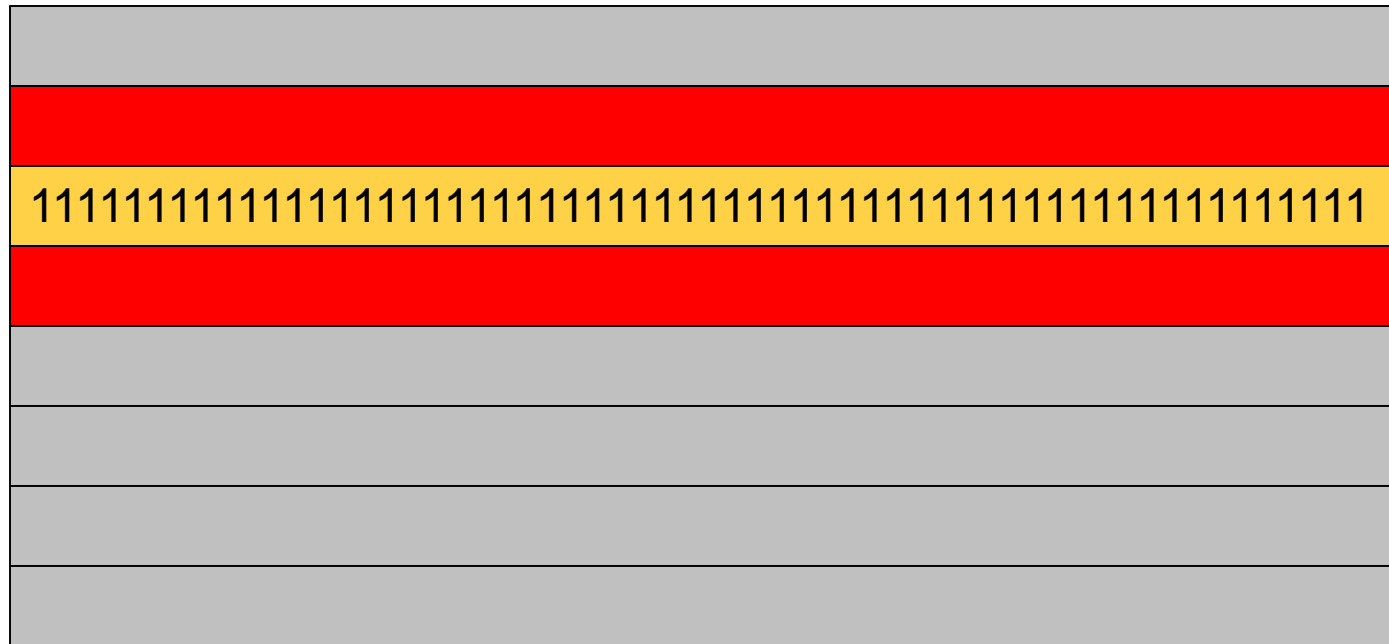
- Memory templating: Find cells **vulnerable** to bitflips.
- Remember: DMA memory allocator gives us physically **contiguous** memory.

Memory Templating on ARM

[illegible]

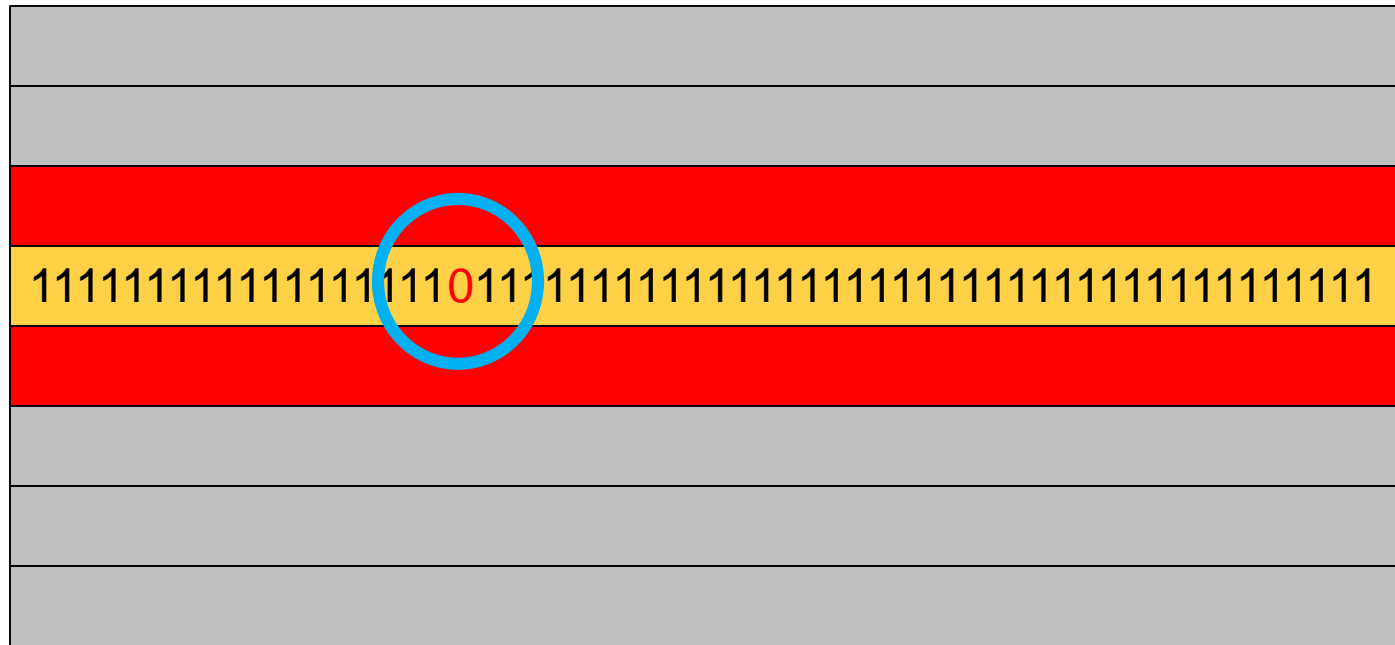
- “**Hammer**” the victim row in between and check for bitflips.

Memory Templating on ARM



- “**Hammer**” the victim row in between and check for bitflips.

Memory Templating on ARM



- “**Hammer**” the victim row in between and check for bitflips.

P2. Physical Memory Massaging

- ❑ **Determine DRAM chip's row size** to understand memory model for later **templating**
- ❑ **Memory Templating** to find memory locations **susceptible** to Rowhammer
- ❑ **Land sensitive data** in **susceptible location**
- ❑ **Reproduce the bit flip** with **security-sensitive** data now in susceptible location

What is the Sensitive Data?

- We want to store a **page table** in the susceptible vulnerable virtual page.
- Recall that page tables map **virtual addresses** to a **physical address**.

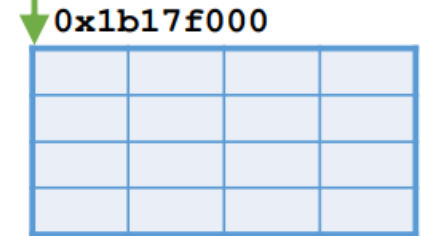
Flipping Bits in Page Tables

Entry in the (2nd level) Page Table

0 0 0 1	1 0 1 1	0 0 0 1	0 1 1 1	1 1 1 1	x x x x	x x x x	x x x x
---------	---------	---------	---------	---------	---------	---------	---------

$0x1b17f \ll 12$

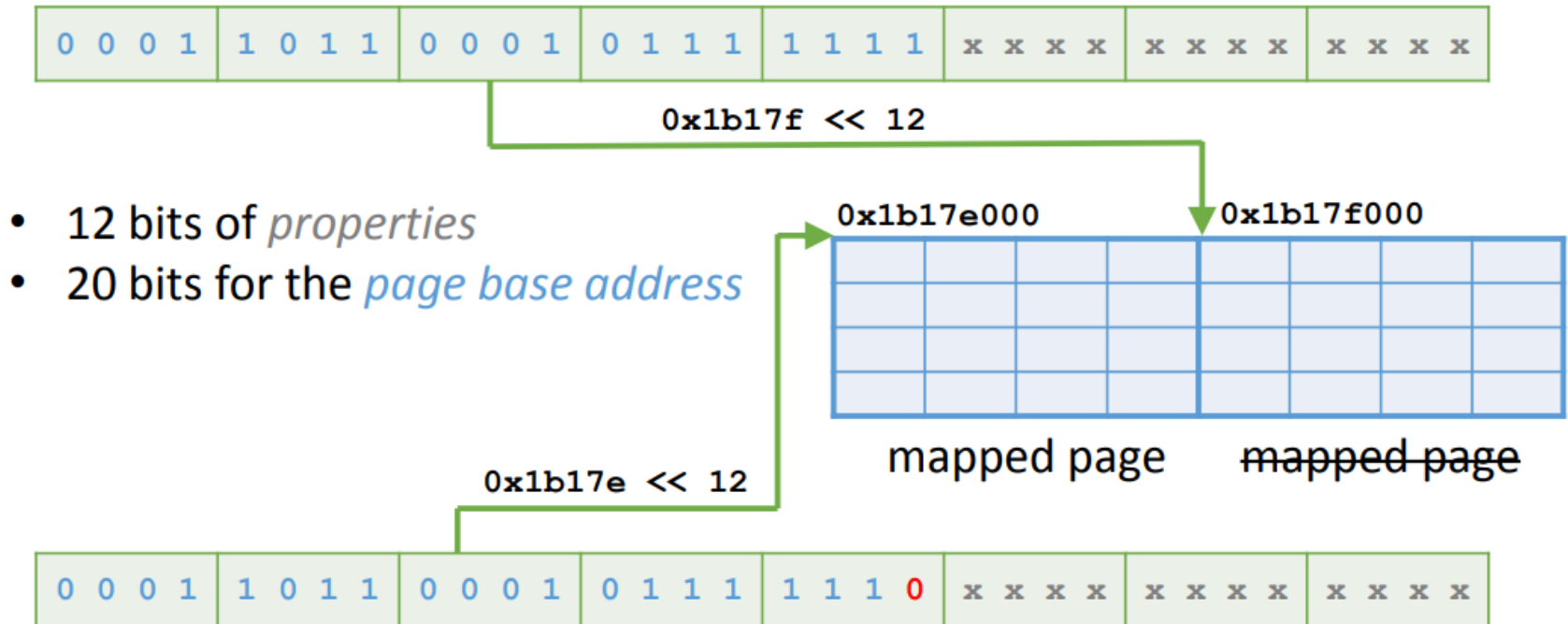
- 12 bits of *properties*
- 20 bits for the *page base address*



mapped page

Flipping Bits in Page Tables

Entry in the (2nd level) Page Table



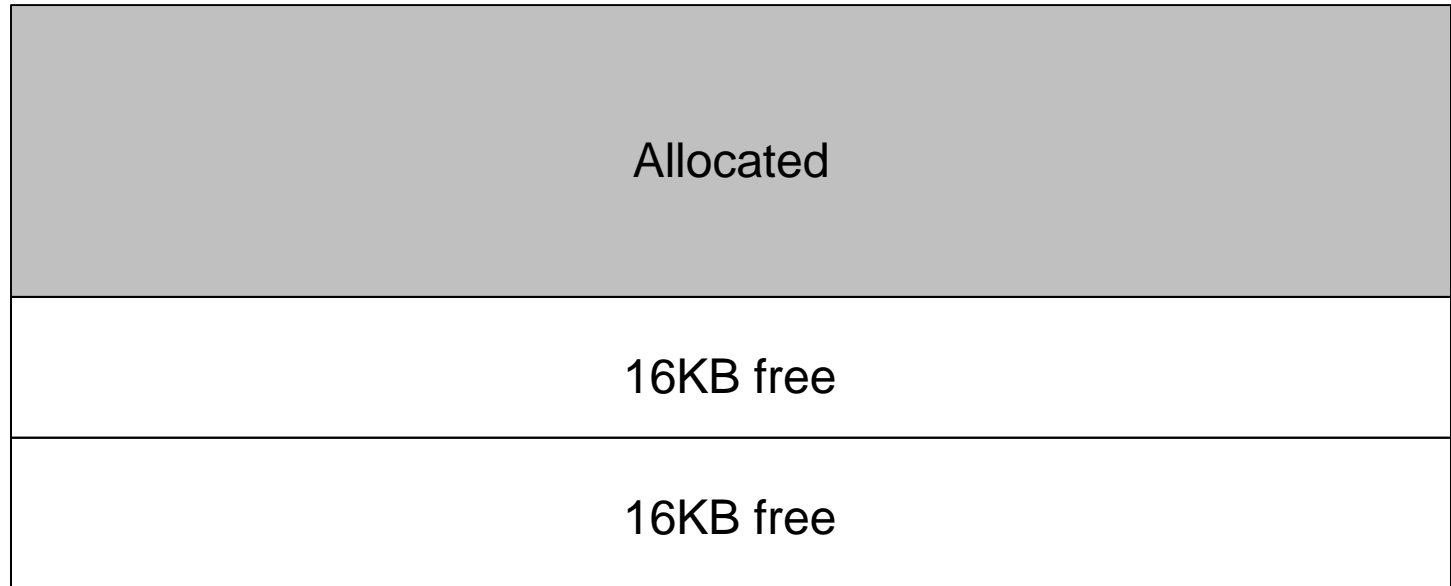
A 1-to-0 flip moves the mapping 'to the left'

- Flip offset 0: -1 page
- Flip offset 1: -2 pages
- Flip offset 2: -4 pages
- Flip offset n : -2^n pages

But how do we get a page table into a vulnerable location in physical memory?

- **Phys Feng Shui**: New technique that lures the **buddy allocator** into reusing and partitioning memory in a predictable way.

Buddy Allocator



- **Linux platforms** minimize external fragmentation by splitting and merging available memory in power-of-2 sized blocks using the buddy allocator.

Buddy Allocator



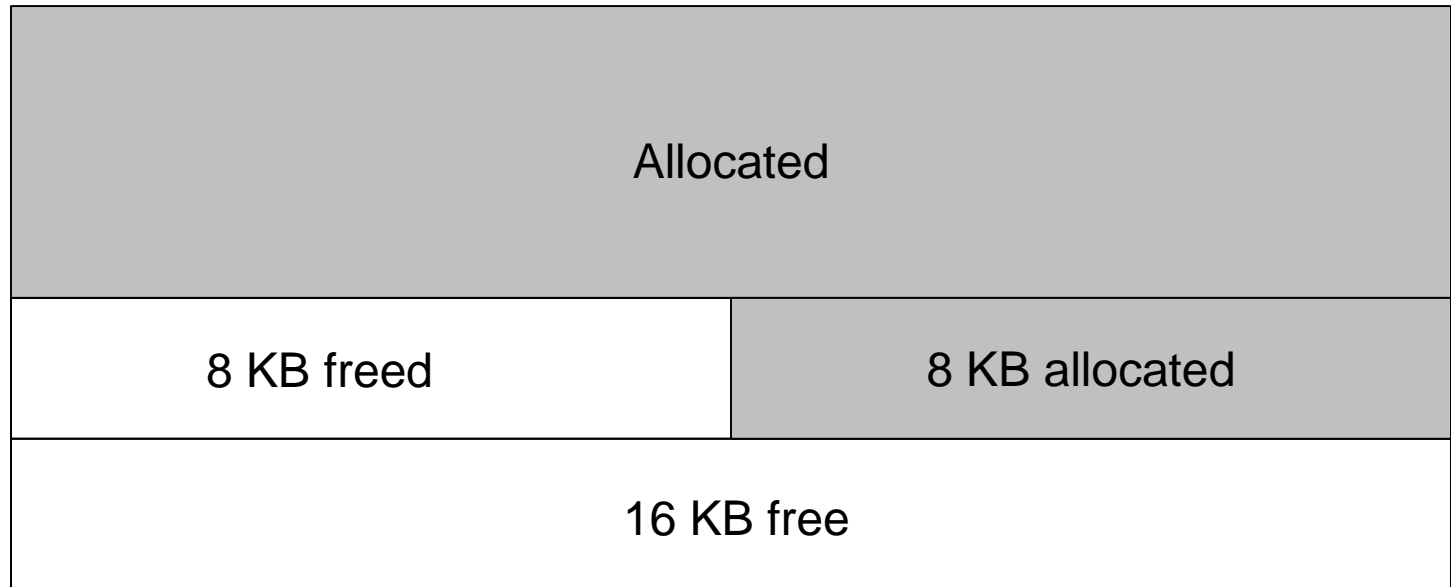
- It iteratively splits larger blocks in half as necessary until it finds matching block.

Buddy Allocator



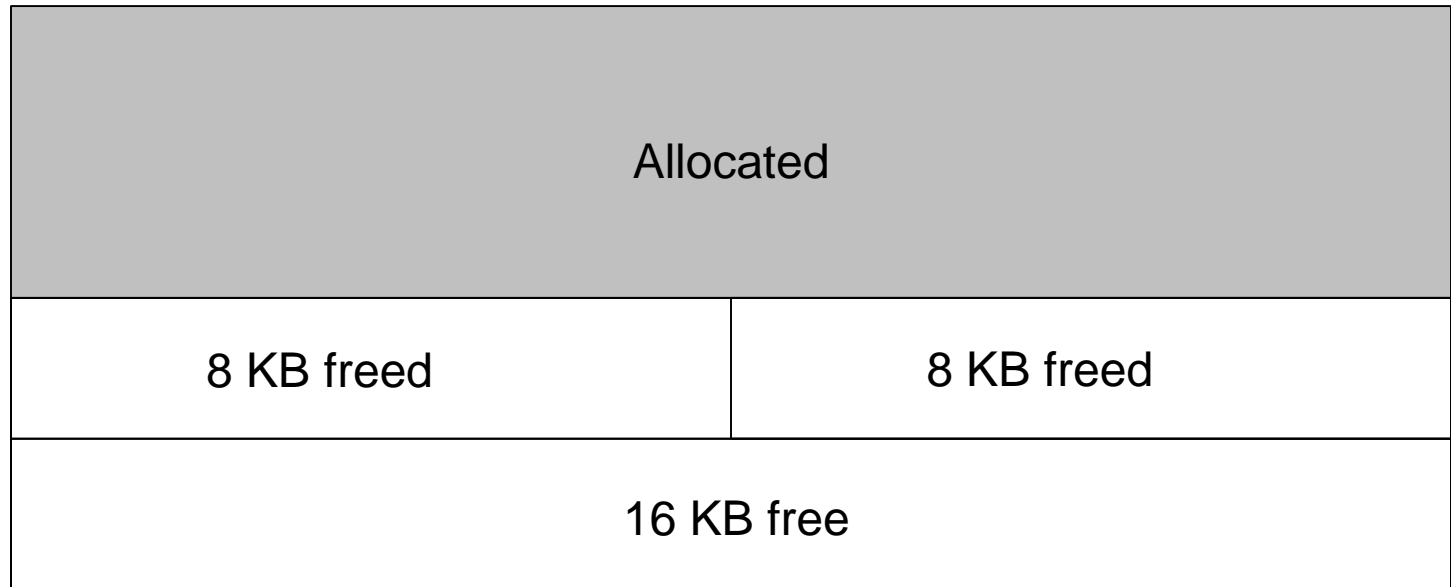
- It always prioritizes the smallest fitting block when splitting.

Buddy Allocator



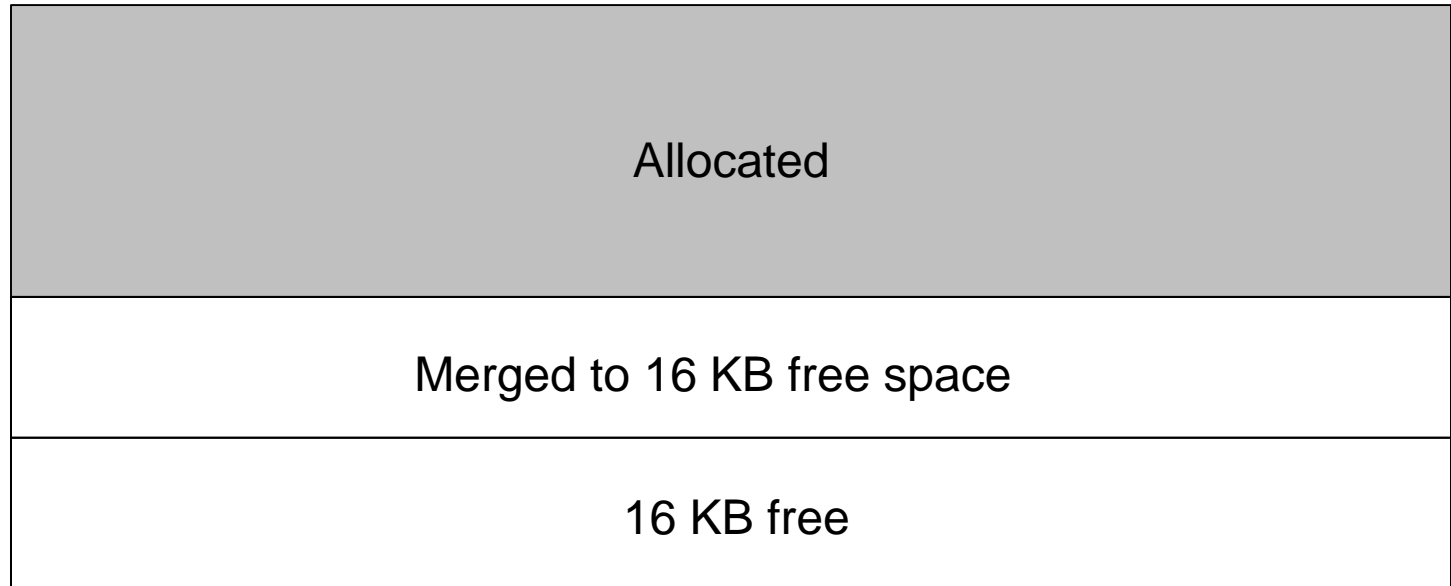
- When freeing blocks again, it tries to merge free blocks of same sizes.

Buddy Allocator



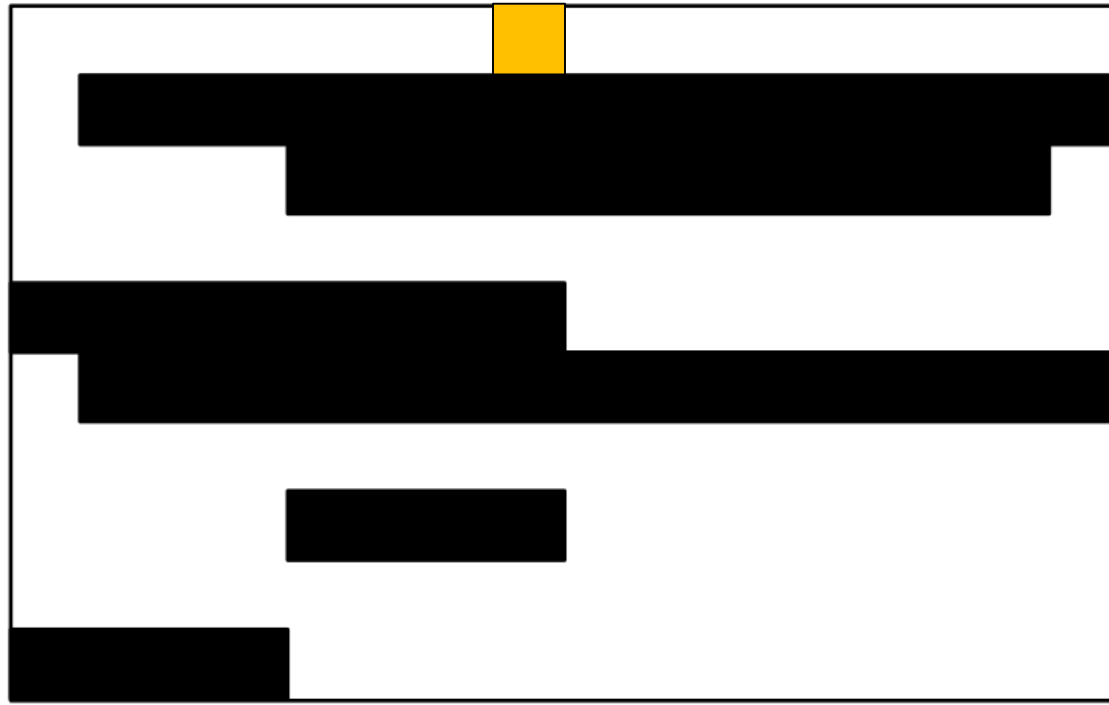
- When freeing blocks again, it tries to merge free blocks of same sizes.

Buddy Allocator



- When freeing blocks again, it tries to merge free blocks of same sizes.

Phys Feng Shui – Tricking the Allocator



UNINITIALIZED MEMORY



allocated



free



Memory chunk where we have exploitable bit flip (not allocated currently)

- The goal is to land a page table in the yellow memory chunk where we have an exploitable bit flip.
- We're going to exhaust and free memory to get page table in that yellow location.

Phys Feng Shui – Exhaust(L)+Template(L)

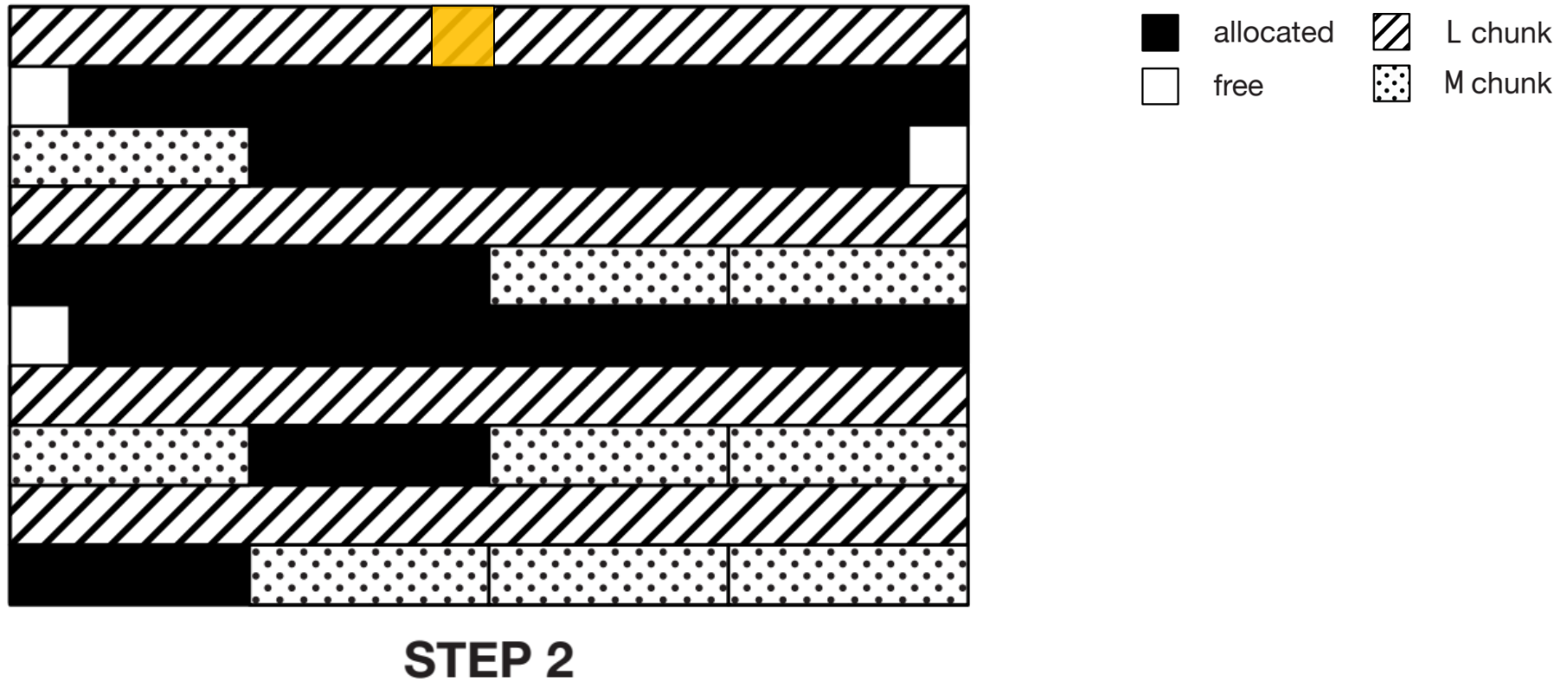



STEP 1

L chunk: Largest possible contiguous chunk of memory

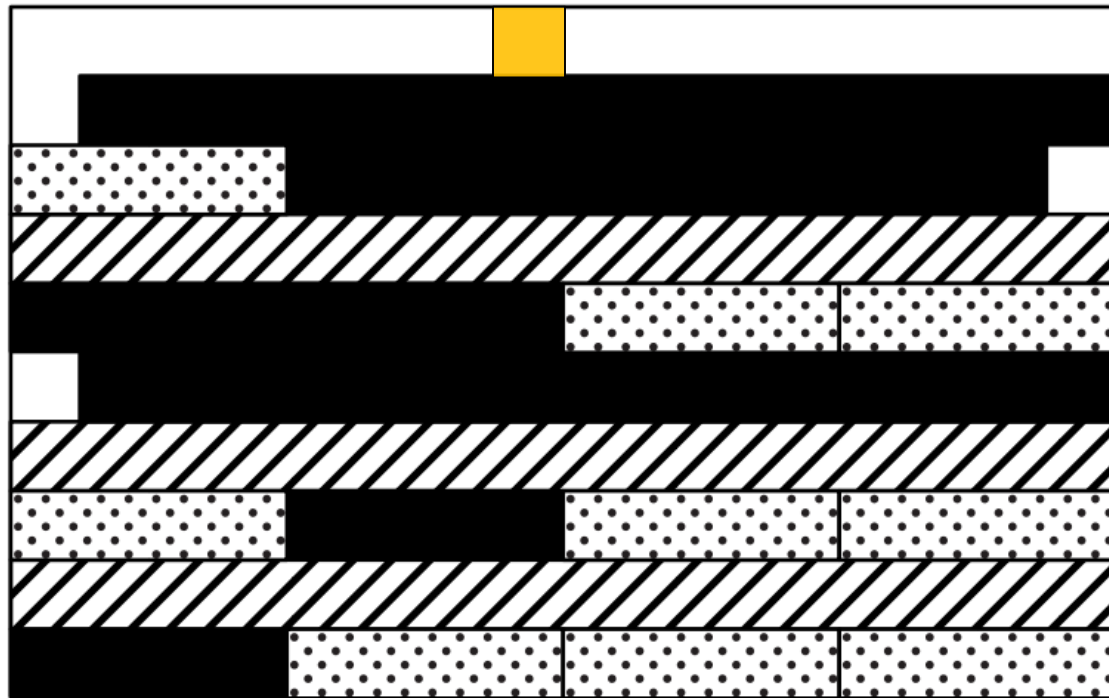
- Exhaust chunks of size L and probe them for vulnerable templates.

Phys Feng Shui – Exhaust(M)



-  **M chunk:** Medium sized chunk with the site set to row size
- Blocks of size M or larger are no longer available.

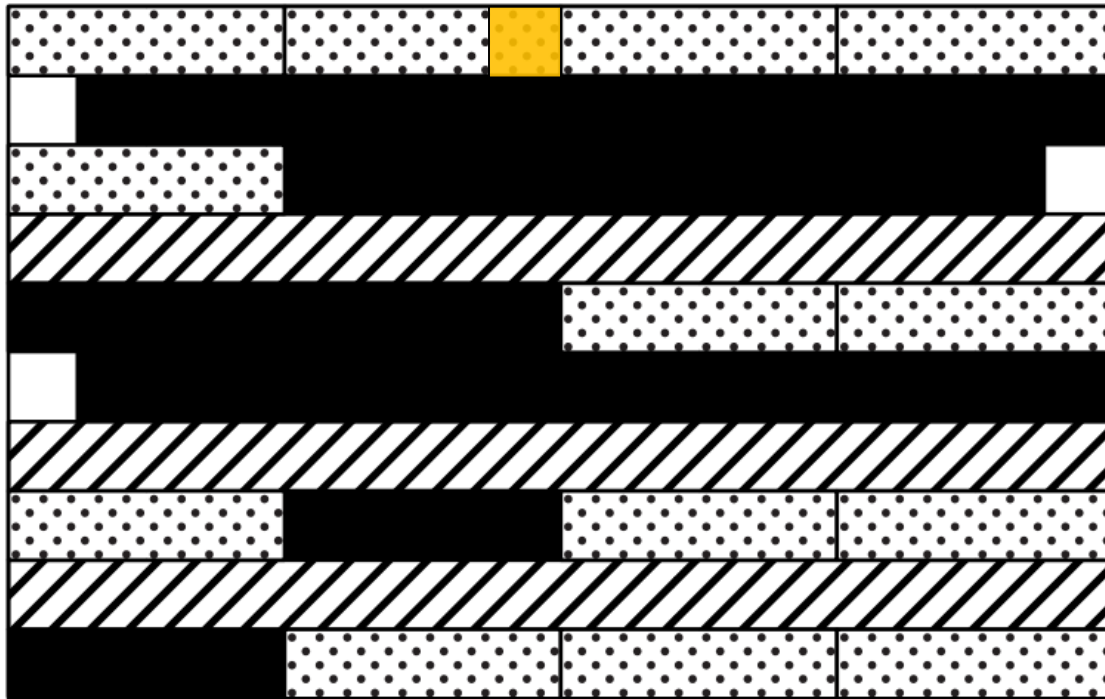
Phys Feng Shui – Free(L)



STEP 3

- Release block of size L with exploitable template.

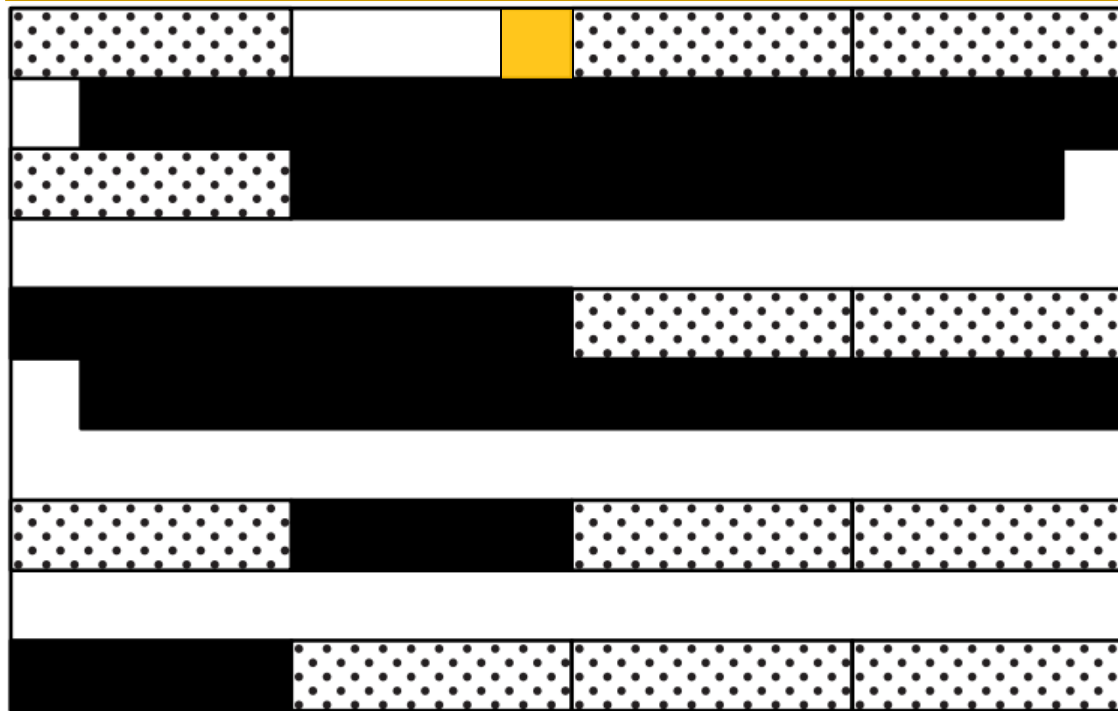
Phys Feng Shui – Exhaust(M)



STEP 4

- After exhausting M chunks again, we have a M chunk that holds exploitable template.

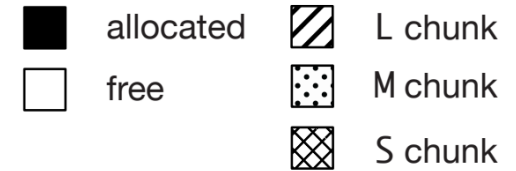
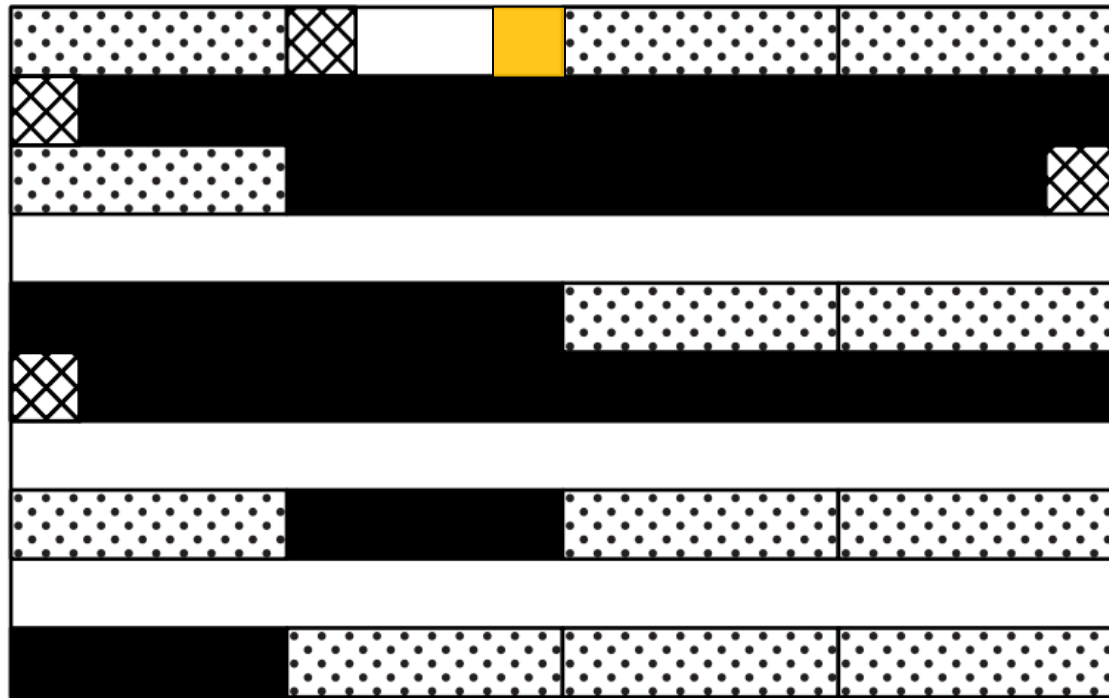
Phys Feng Shui –Free(M)+FreeAll(L)



STEP 5

- Release memory pressure due to lack of swap space in mobile devices.

Phys Feng Shui – Land(S)

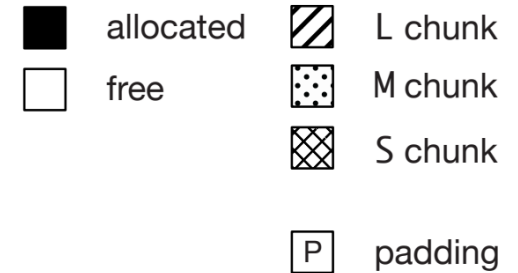
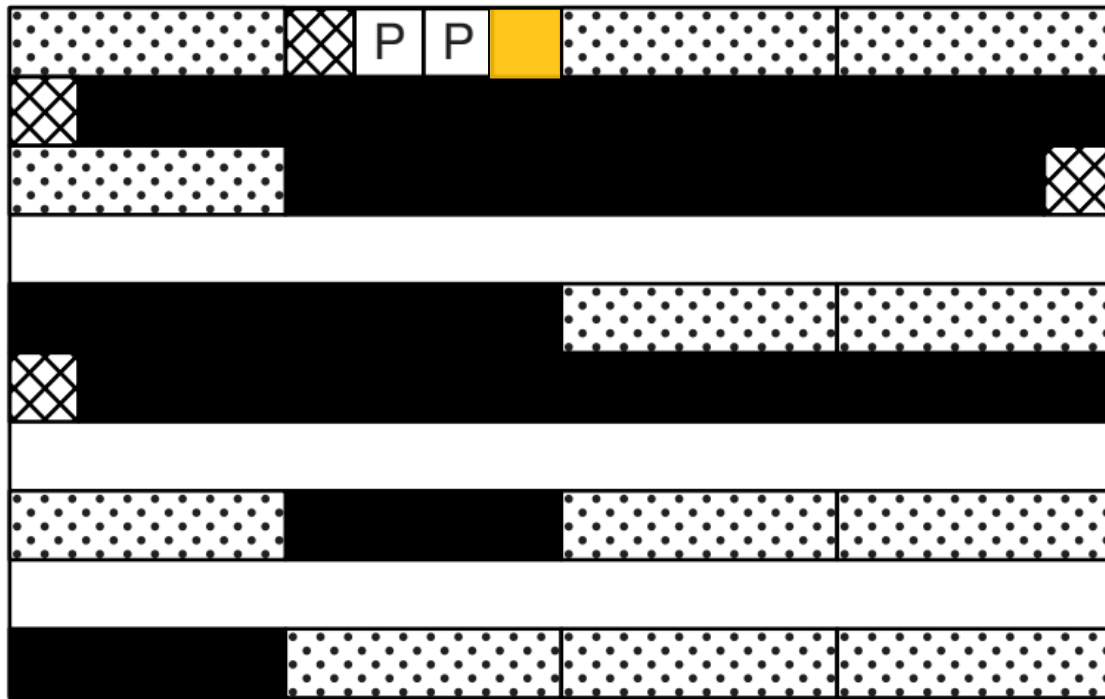


STEP 6

 **S chunk:** Small sized chunk with the size fixed at [page size](#)

- Repeatedly allocate S chunks until we land in M with exploitable chunk.

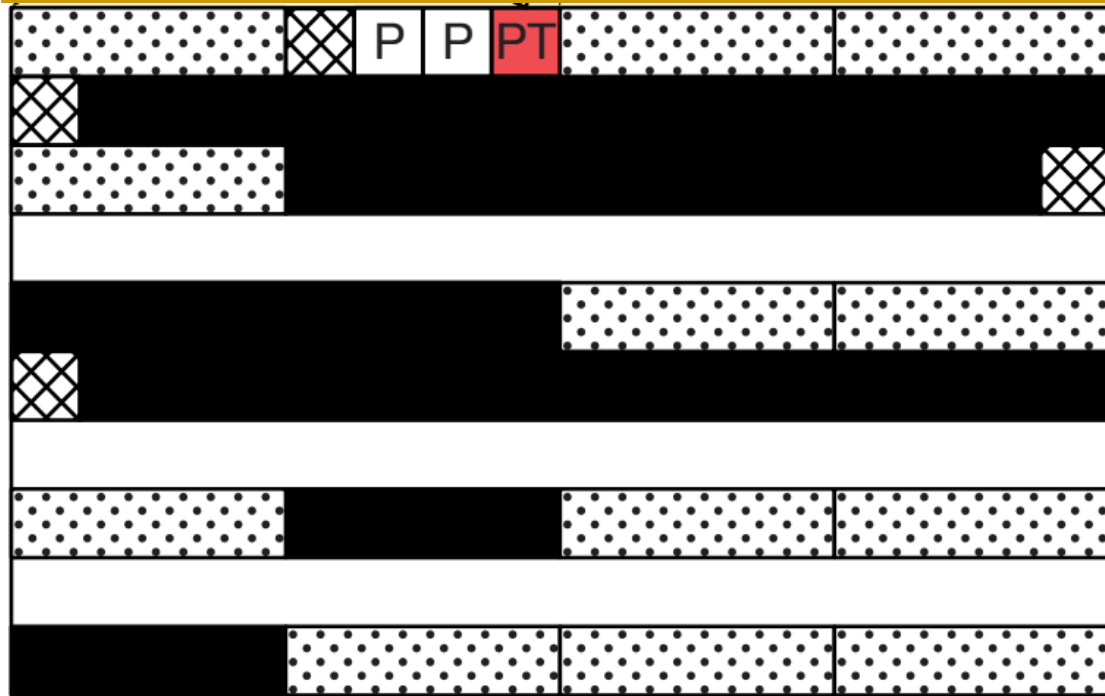
Phys Feng Shui – Padding(S)



STEP 7

- Add padding using S chunks

Phys Feng Shui – Map(M)



STEP 8

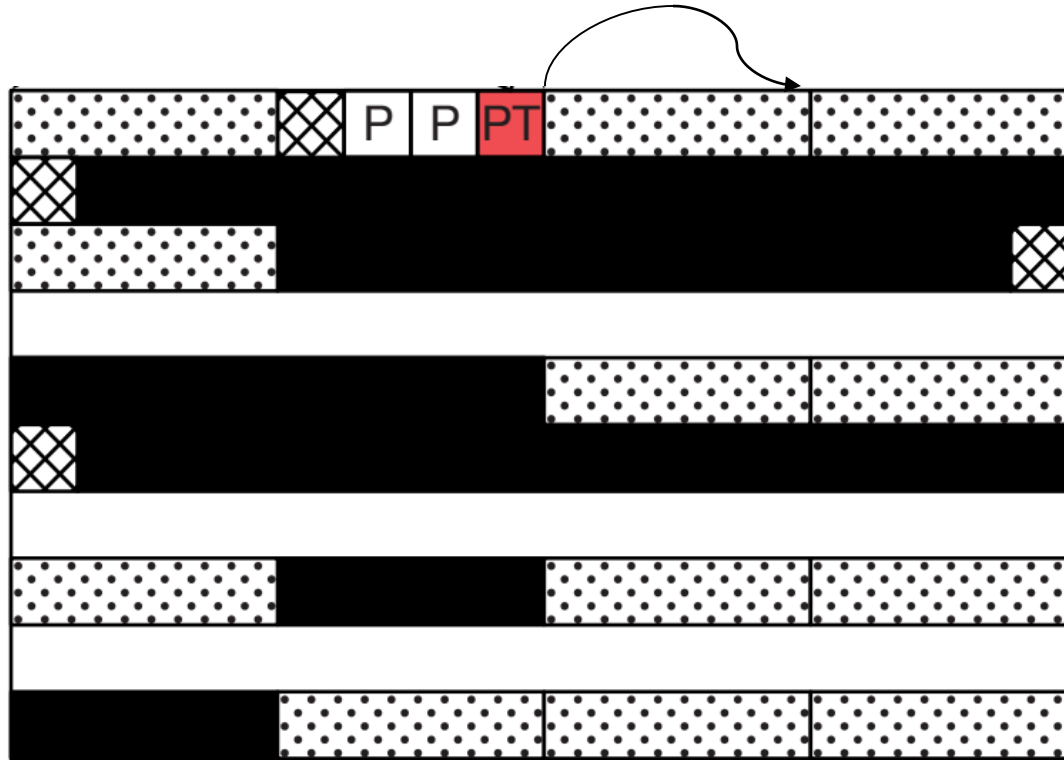
- Force **page table** allocation



P2. Physical Memory Massaging

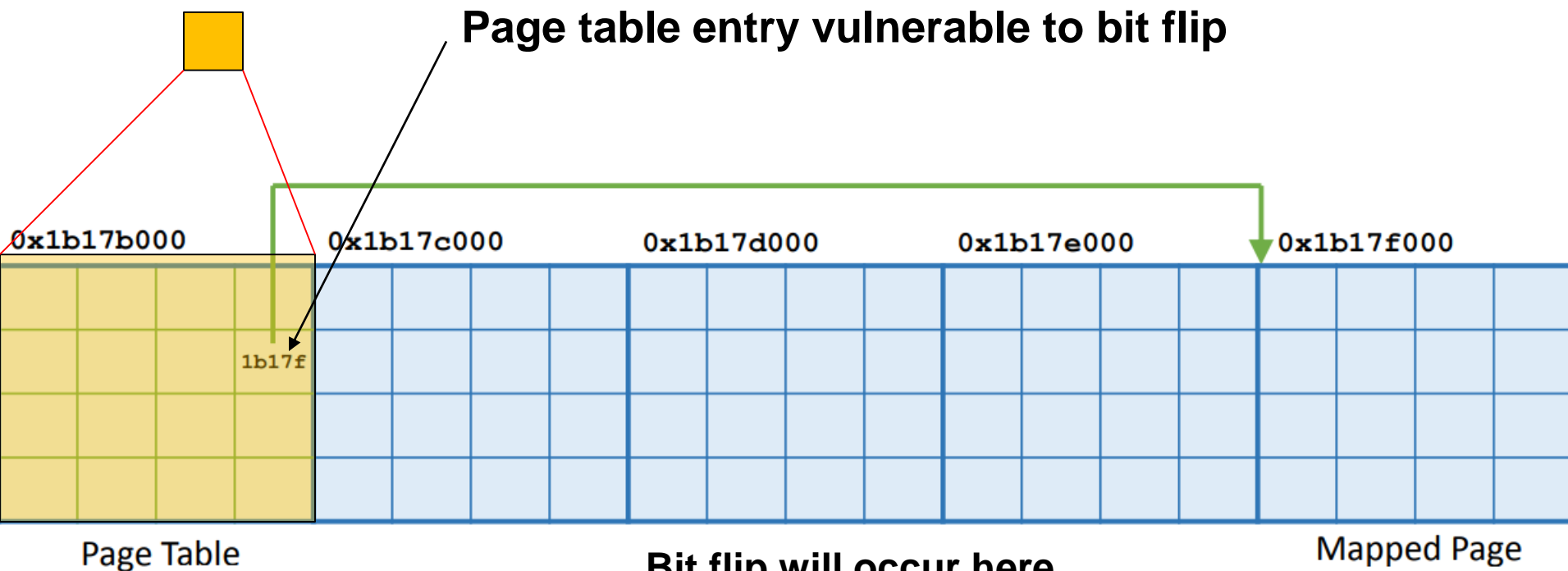
- ❑ **Determine DRAM chip's row size** to understand memory model for later **templating**
- ❑ **Memory Templating** to find memory locations **susceptible** to Rowhammer
- ❑ **Land sensitive data** in **susceptible location**
- ❑ **Reproduce the bit flip** with **security-sensitive** data now in susceptible location

Reproduce Bit Flip



- Map the **PTE** with a bit flip at offset bit n to a location 2^n pages away from the PT.
- The bit-flip in PT will cause the page table entry to **point to PT itself**.

Phys Feng Shui



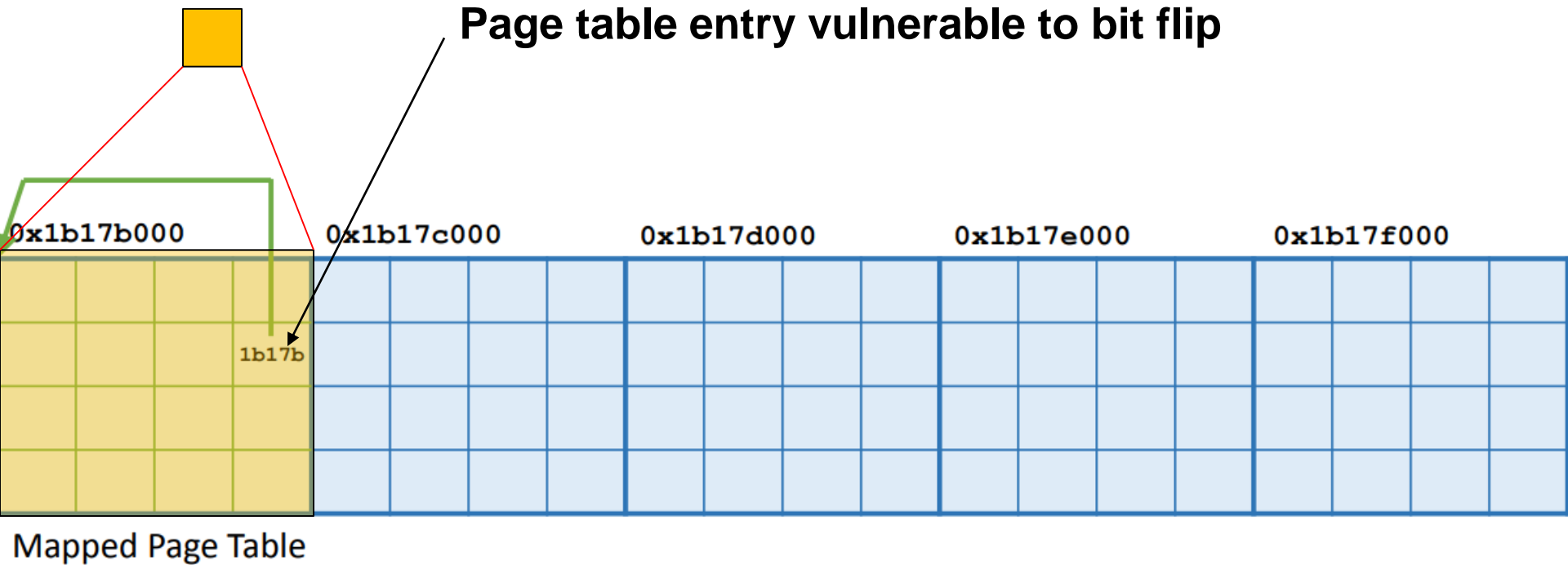
Bit flip will occur here

Virtual address 0xb6a57000 maps to Page Table Entry:

0	0	0	1	1	0	1	1	0	0	0	1	0	1	1	1	1	1	x	x	x	x	x	x	x	x	x	x	x	x	x
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

which translates to physical page 0x1b17f000

Phys Feng Shui

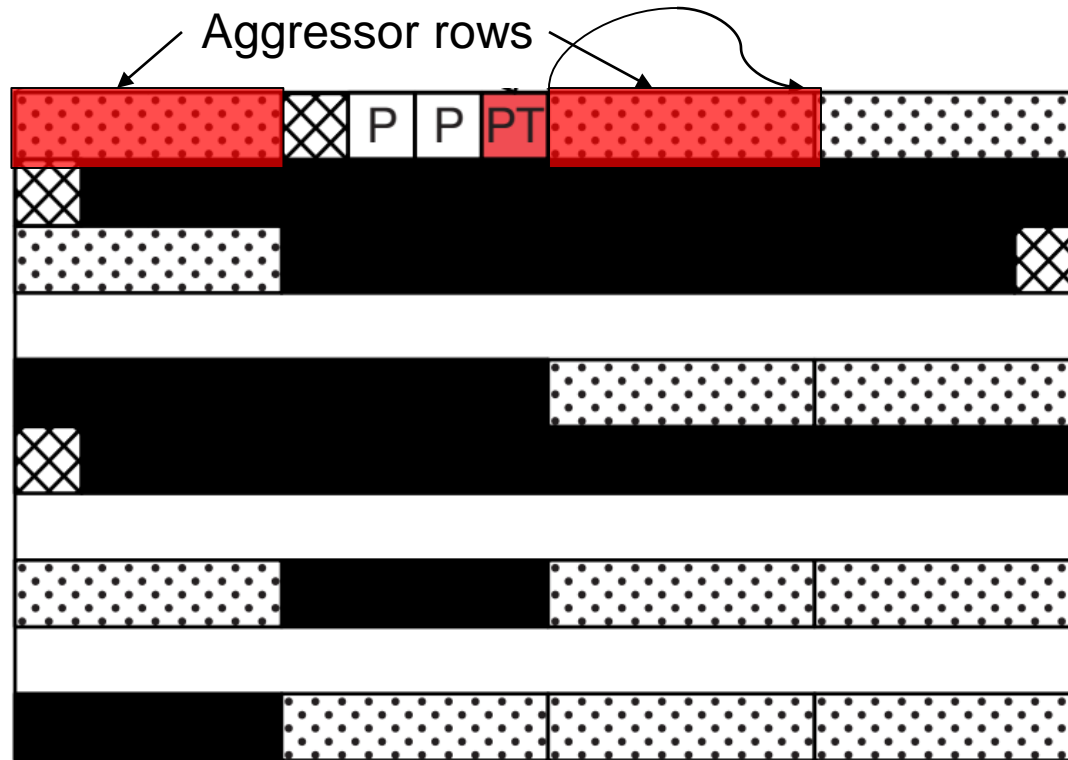


Virtual address 0xb6a57000 maps to Page Table Entry:

0	0	0	1	1	0	1	1	0	0	0	1	0	1	1	1	0	1	1	x	x	x	x	x	x	x	x	x	x	x	x
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

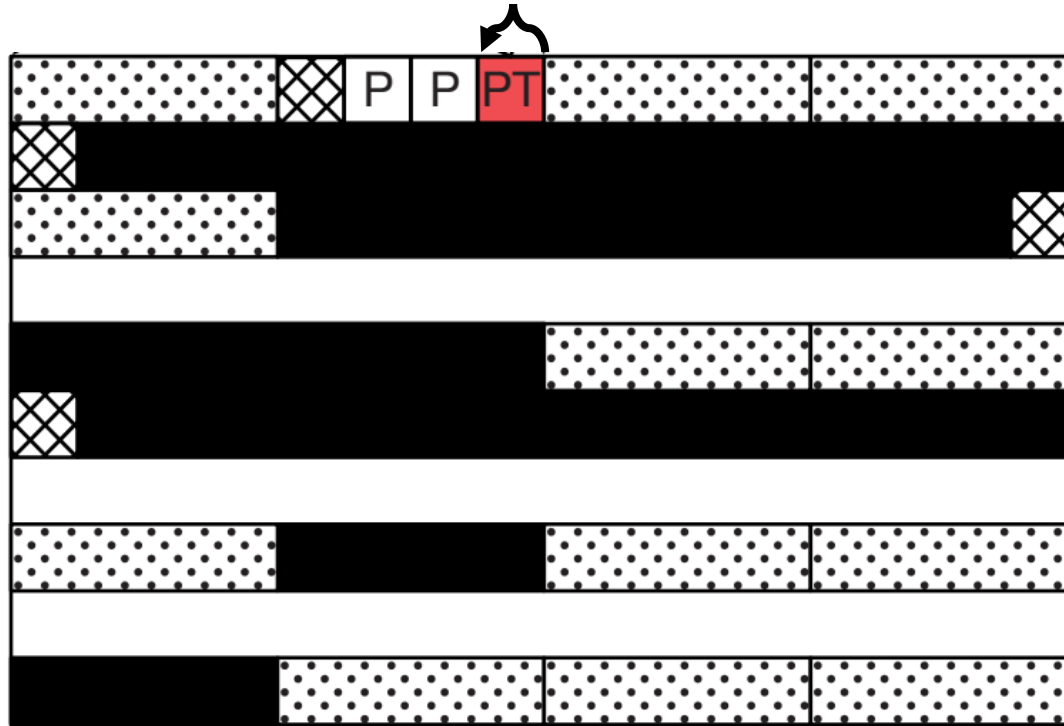
which translates to physical page 0x1b17f000

Reproduce Bit Flip



- Using the two aggressor rows we start the **double-sided Rowhammer attack** to reproduce the bit-flip in PT.

Reproduce Bit Flip



- Now the page table entry in PT points to the **physical address of the virtual page table**.

Root Privilege Escalation

0x1b17b000				0x1b17c000				0x1b17d000				0x1b17e000				0x1b17f000			
3ac90	3ac91	3ac92	3ac93																
3ac94	3ac95	3ac96	1b17b																
3ac97	3ac98	3ac99	3ac9a																
3ac9b	3ac9c	3ac9d	3ac9e																

Mapped Page Table

- Having control over our own PTP allows us to repeatedly map different physical pages to scan kernel memory.
- With that, we can find our own “struct cred” structure (representing a process’ security context) on our UID.
- With access to the struct cred, we can give our process root privilege.

Evaluation of Attack

Empirical Analysis Results

Device		DRAM	# flips	# 1-to-0	# 0-to-1
ARMv7	Nexus 5 ₁	2 GB	1,058	1,011	47
	Nexus 5 ₂	2 GB	284,428	261,232	23,196
	Nexus 5 ₃	2 GB	547,949	534,695	13,254
	Nexus 5 ₄	2 GB	0	—	—
	Nexus 5 ₅	2 GB	747,013	704,824	42,189
	Nexus 5 ₆	2 GB	215,233	207,856	7,377
	Nexus 5 ₈	2 GB	32,328	28,500	3,828
	Nexus 5 ₉	2 GB	476,170	434,086	42,084
	Nexus 5 ₁₀	2 GB	160,245	150,485	9,760
	Nexus 5 ₁₁	2 GB	0	—	—
	Nexus 5 ₁₂	2 GB	17,384	16,767	617
	Nexus 5 ₁₃	2 GB	161,514	160,473	1,041
	Nexus 5 ₁₄	2 GB	295,537	277,708	17,829
	Nexus 5 ₁₅	2 GB	38,969	35,515	3,454
	Nexus 5 ₁₇	2 GB	0	—	—
	Galaxy S5	2 GB	0	—	—
	OnePlus One ₁	3 GB	3,981	2,924	1,057
	OnePlus One ₂	3 GB	1,992	942	1,050
ARMv8	Moto G ₂₀₁₃	1 GB	429	419	10
	Moto G ₂₀₁₄	1 GB	1,577	1,523	54
	Nexus 4	2 GB*	1,328	1,061	267
	Nexus 5x	2 GB	0	—	—
	Galaxy S6	3 GB ^o	0	—	—
	K3 Note	2 GB	0	—	—
	Mi 4i	2 GB	0	—	—
	Desire 510	1 GB	0	—	—
	G4	3 GB	117,496	117,260	236

*LPDDR2 ^oLPDDR4

- Number of flips varies a lot even when comparing the same devices.
- 1-to-0 and 0-to-1 flips not symmetric.
- ARMv8 seems somewhat more resilient to flips.

Empirical Analysis Results

		<i>DRAM</i>	<i># flips</i>	<i># exploitable</i>	<i>1st</i>
ARMv7	Nexus 5 ₁	2 GB	1,058	62 (5.86%)	116s
	Nexus 5 ₂	2 GB	284,428	14,852 (5.22%)	1s
	Nexus 5 ₃	2 GB	547,949	32,715 (5.97%)	1s
	Nexus 5 ₄	2 GB	0	—	—
	Nexus 5 ₅	2 GB	747,013	46,609 (6.24%)	1s
	Nexus 5 ₆	2 GB	215,233	13,365 (6.21%)	3s
	Nexus 5 ₈	2 GB	32,328	1,894 (5.86%)	4s
	Nexus 5 ₉	2 GB	476,170	30,190 (6.34%)	0s
	Nexus 5 ₁₀	2 GB	160,245	8,701 (5.43%)	1s
	Nexus 5 ₁₁	2 GB	0	—	—
	Nexus 5 ₁₂	2 GB	17,384	1,241 (7.14%)	16s
	Nexus 5 ₁₃	2 GB	161,514	10,378 (6.43%)	355s
	Nexus 5 ₁₄	2 GB	295,537	18,900 (6.40%)	1s
	Nexus 5 ₁₅	2 GB	38,969	2,775 (7.12%)	11s
	Nexus 5 ₁₇	2 GB	0	—	—
	Galaxy S5	2 GB	0	—	—
	OnePlus One ₁	3 GB	3,981	242 (6.08%)	942s
	OnePlus One ₂	3 GB	1,992	94 (4.72%)	326s
ARMv8	Moto G ₂₀₁₃	1 GB	429	30 (6.99%)	441s
	Moto G ₂₀₁₄	1 GB	1,577	71 (4.66%)	92s
	Nexus 4	2 GB*	1,328	104 (7.83%)	7s
	Nexus 5x	2 GB	0	—	—
	Galaxy S6	3 GB ^o	0	—	—
	K3 Note	2 GB	0	—	—
	Mi 4i	2 GB	0	—	—
	Desire 510	1 GB	0	—	—
	G4	3 GB	117,496	6,560 (5.58%)	5s

*LPDDR2 ^oLPDDR4

- Around 6% of all observed flips are exploitable.
- “1st” describes the time it takes until first exploitable bit is found. The longest measured time is over 15min. After that, escalating privilege only takes around 22s on average.

Mitigation Techniques

Existing Rowhammer Defenses

■ Software-based:

- ❑ Instruction “blacklisting”: disallowing instructions such as CLFLUSH.
- ❑ Restrict access to pagemap interface
- ❑ Detection of Rowhammer attacks by monitoring cache miss rate

Because of DMA, DRAMMER does not:

- Rely on comparable instructions
- Need access to pagemap
- Use cache, therefore not cause any misses

Existing Rowhammer Defenses

■ **Hardware-based:**

- ❑ Memory with **Error Correcting Codes (ECC)**
- ❑ **Doubling DRAM refresh rates**
- ❑ **Detection of Activation Patterns** to **refresh targeted rows** (e.g. Probabilistic Adjacent Row Activation)

- ECCs have **not been reliable**.
- Doubling refresh rates has severe consequences for power consumption and performance

Countermeasures Against DRAMMER

■ **Restriction of userland interface:**

- ❑ Rethink how **DMA-support** should be implemented, maybe with a restricted interface
- ❑ Possible improvement is to adopt **constraint-based allocations**

■ **Memory isolation and integrity:**

- ❑ Isolate ION regions controlled by **userland from kernel** memory
- ❑ Treat cells with **security-critical data** differently, e.g. have regions with higher refresh rates.

■ **Prevention of memory exhaustion:**

- ❑ Per-process **memory limits**

Executive Summary

Executive Summary

- **Motivation:** Current Rowhammer exploits are mostly **probabilistic** and most studies only focus on **x86**.
- **Goal:** Implement **deterministic** Rowhammer attack on **ARM** devices.
- **Challenges:**
 - ❑ Fast uncached memory access
 - ❑ Put page table into Rowhammer exploitable memory
 - ❑ Find aggressor rows
- **Key Ideas:**
 - ❑ Use Android's DMA buffers to get uncached contiguous memory
 - ❑ Use predictability of memory allocator to put page table into exploitable memory
- **Result:** Many **ARMv7** and one **ARMv8** device have been **successfully exploited** using DRAMMER.

Strengths

Strengths

- Novel solution to make use of the [buddy allocator](#) and [DMA](#) for physical memory massaging.
- First Rowhammer exploit on [Android/ARM](#).
- Potentially motivates the research for much needed [mitigation strategies](#) for the billions of vulnerable mobile devices.
- Releasing their codebase as [open source project](#) and building a public database of known vulnerable devices further enables future research.
- The demonstrated techniques for deterministic Rowhammer are [generic enough](#) to be applicable to other devices.

Weaknesses

Weaknesses

- The results of the empirical analysis are **barely analyzed**.
- Very few **ARMv8** testing and results. Especially the existing results are conspicuous and demand more testing.
- The paper covers a wide range of topics. Certain topics and discussion points are **poorly explained**.
- The choice of **Android devices** seems a bit random.
- The **structure** and **writing** of the paper is confusing.

Thoughts and Ideas

Thoughts and Ideas

- How does introducing a **per-process limit** memory mitigate the exhaustion of memory?
- How do **external conditions** and **DRAM wearing** affect the number of bit flips induced by Rowhammer on ARM devices?
- Have these issues been addressed in **LPDDR4**?
- Have these issues been addressed in **ARMv8** and newer?
- Could DRAMMER be used to create an App to “**root**” an Android device?

Takeaways

Takeaways

- Implementing **DMA** efficiently while assuring security is difficult.
- **Memory allocators** such as the Linux buddy allocator lack security measurements.
- **System design** has to be done with security precautions in mind.

Open Discussion

Open Discussion

- Why do the results of the **empirical analysis** vary so much between different and even same devices?
- Could an attack similar to DRAMMER also be possible for an **iOS device**?
- Can you think of any other means to **mitigate** Rowhammer exploits on Android/ARM?
- Should hardware vulnerabilities have the same **disclosure deadline** as software vulnerabilities? For example, Google Project Zero has a 90-day deadline.
- Do you think Rowhammer is still a problem in **2020**?
 - Jeremie S. Kim, Minesh Patel, A. Giray Yağlıkçı, Hasan Hassan, Roknoddin Azizi, Lois Orosa, Onur Mutlu, [Revisiting RowHammer: An Experimental Analysis of Modern DRAM Devices and Mitigation Techniques](#)
- Have these issues been addressed in **LPDDR4**?

Backup Slides

	Device	Hardware Details			Analysis Results							
		SoC	DRAM	RS	MB	ns	# flips	KB	# 1-to-0	# 0-to-1	# exploitable	1 st
ARMv7	Nexus 5 ₁	MSM8974 [†]	2 GB	64	441	70	1,058	426	1,011	47	62 (5.86%)	116s
	Nexus 5 ₂	MSM8974 [†]	2 GB	64	472	69	284,428	2	261,232	23,196	14,852 (5.22%)	1s
	Nexus 5 ₃	MSM8974 [†]	2 GB	64	461	69	547,949	1	534,695	13,254	32,715 (5.97%)	1s
	Nexus 5 ₄	MSM8974 [†]	2 GB	64	616	71	0	—	—	—	—	—
	Nexus 5 ₅	MSM8974 [†]	2 GB	64	630	69	747,013	1	704,824	42,189	46,609 (6.24%)	1s
	Nexus 5 ₆	MSM8974 [†]	2 GB	64	512	69	215,233	3	207,856	7,377	13,365 (6.21%)	3s
	Nexus 5 ₈	MSM8974 [†]	2 GB	64	485	70	32,328	15	28,500	3,828	1,894 (5.86%)	4s
	Nexus 5 ₉	MSM8974 [†]	2 GB	64	569	69	476,170	2	434,086	42,084	30,190 (6.34%)	0s
	Nexus 5 ₁₀	MSM8974 [†]	2 GB	64	406	69	160,245	3	150,485	9,760	8,701 (5.43%)	1s
	Nexus 5 ₁₁	MSM8974 [†]	2 GB	64	613	70	0	—	—	—	—	—
	Nexus 5 ₁₂	MSM8974 [†]	2 GB	64	600	70	17,384	35	16,767	617	1,241 (7.14%)	16s
	Nexus 5 ₁₃	MSM8974 [†]	2 GB	64	575	69	161,514	4	160,473	1,041	10,378 (6.43%)	355s
	Nexus 5 ₁₄	MSM8974 [†]	2 GB	64	576	69	295,537	2	277,708	17,829	18,900 (6.40%)	1s
	Nexus 5 ₁₅	MSM8974 [†]	2 GB	64	573	69	38,969	15	35,515	3,454	2,775 (7.12%)	11s
	Nexus 5 ₁₇	MSM8974 [†]	2 GB	64	621	70	0	—	—	—	—	—
	Galaxy S5	MSM8974 [‡]	2 GB	64	207	82	0	—	—	—	—	—
	OnePlus One ₁	MSM8974 [‡]	3 GB	64	292	71	3,981	75	2,924	1,057	242 (6.08%)	942s
	OnePlus One ₂	MSM8974 [‡]	3 GB	64	1189	69	1,992	611	942	1,050	94 (4.72%)	326s
Moto G ₂₀₁₃	MSM8226	1 GB	32	134	127	429	275	419	10	30 (6.99%)	441s	
Moto G ₂₀₁₄	MSM8226	1 GB	32	151	127	1,577	98	1,523	54	71 (4.66%)	92s	
Nexus 4	APQ8064	2 GB*	64	82	18	1,328	64	1,061	267	104 (7.83%)	7s	
ARMv8	Nexus 5x	MSM8992	2 GB	64	271	63	0	—	—	—	—	—
	Galaxy S6	Exynos7420	3 GB [°]	128	234	82	0	—	—	—	—	—
	K3 Note	MT6752	2 GB	64	423	218	0	—	—	—	—	—
	Mi 4i	MSM8939	2 GB	64	327	159	0	—	—	—	—	—
	Desire 510	MSM8916	1 GB	32	186	122	0	—	—	—	—	—
	G4	MSM8992	3 GB	64	833	64	117,496	8	117,260	236	6,560 (5.58%)	5s

[†]MSM8974AA [‡]MSM8974AC *LPDDR2 [°]LPDDR4