Presented at CCS'16 Vienna, Austria

Drammer

Deterministic Rowhammer Attacks on Mobile Platforms

V. v. d. Veen, Y. Fratantonio, M. Lindorfer, D. Gruss, C. Maurice, G. Vigna, H. Bos, K. Razavi, C. Giuffrida

Vrije Universiteit Amsterdam **UC Santa Barbara**

Graz University of Technology

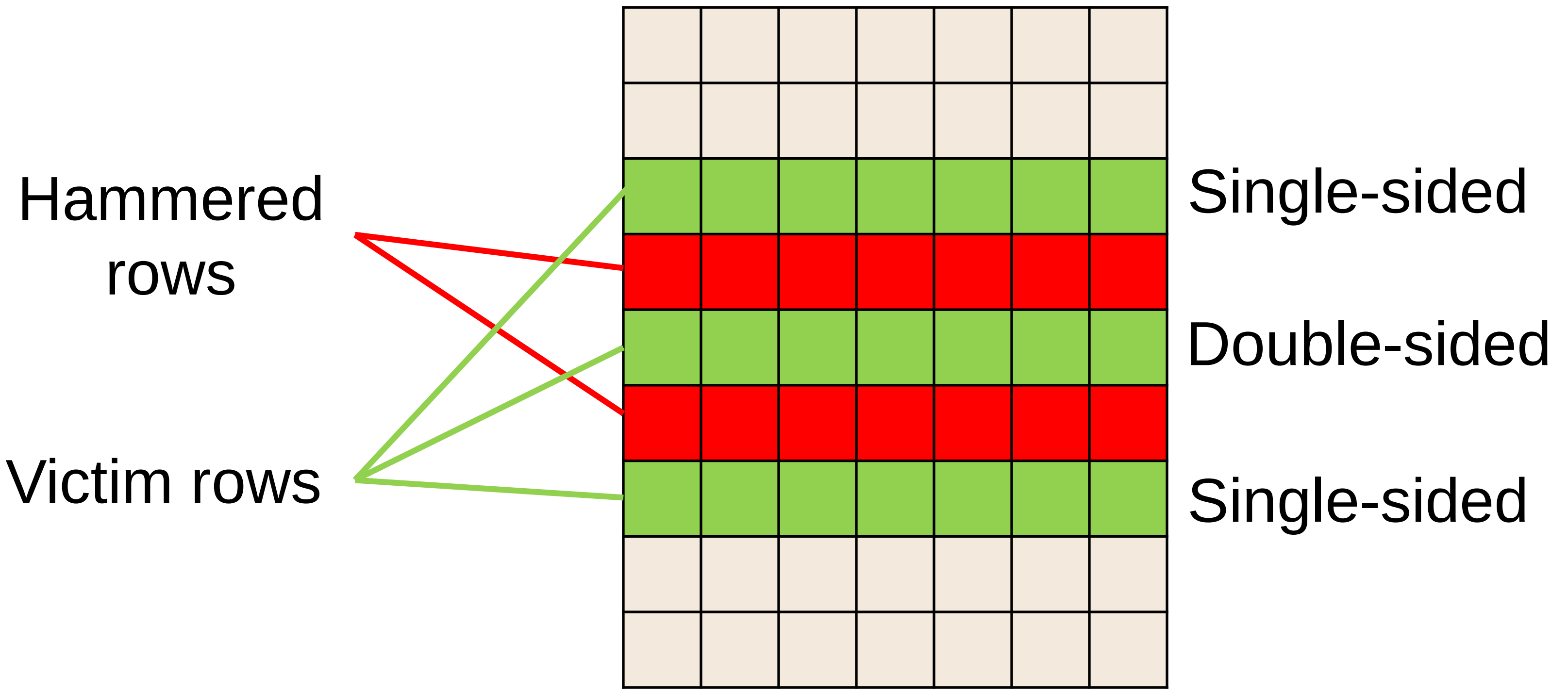
Presented by David Enderlin

ETH Zürich, 17 October 2019

Background, Problems & Goal

Rowhammer

- Flip bits in adjacent memory rows by “hammering” [1]



[1] Y. Kim et al. Flipping Bits in Memory Without Accessing Them, ISCA ‘14

Current Rowhammer Exploits

- Current exploits are either
  - Probabilistic [2]
  - Rely on special memory management features [3,4]
- Probabilistic attacks are especially problematic
- Only target x86
- There was doubt whether Rowhammer is even possible on ARM

[2] D. Gruss, et al. Rowhammer.js: A Remote Software-Induced Fault Attack in Javascript, DIMVA ’16
[3] K. Razavi, et al. Flip Feng Shui: Hammering a Needle in the Software Stack, USENIX ’16
[4] Y. Xiao, et al. One Bit Flips, One Cloud Flops: Cross-VM Row Hammer Attacks, USENIX ‘16

Rowhammer Exploits in General

- Triggering the Rowhammer bug is different than using it

- We need three things:

1. Physical Memory Addressing
    - To attack a specific row we have to know which rows are next to it

2. Fast Uncached Memory Access
    - Hammering fast enough to trigger the Rowhammer bug

3. Physical Memory Massaging
    - Some way to get the sensitive data into the attacked row

Goal

- Show that Rowhammer is possible on ARM/Android
- Implement the first deterministic Rowhammer-based Android root exploit
  - Without requiring special memory management features
  - Without requiring any permissions

Novelty, Key Ideas and Attack Overview

Novelty

The paper makes two important contributions:

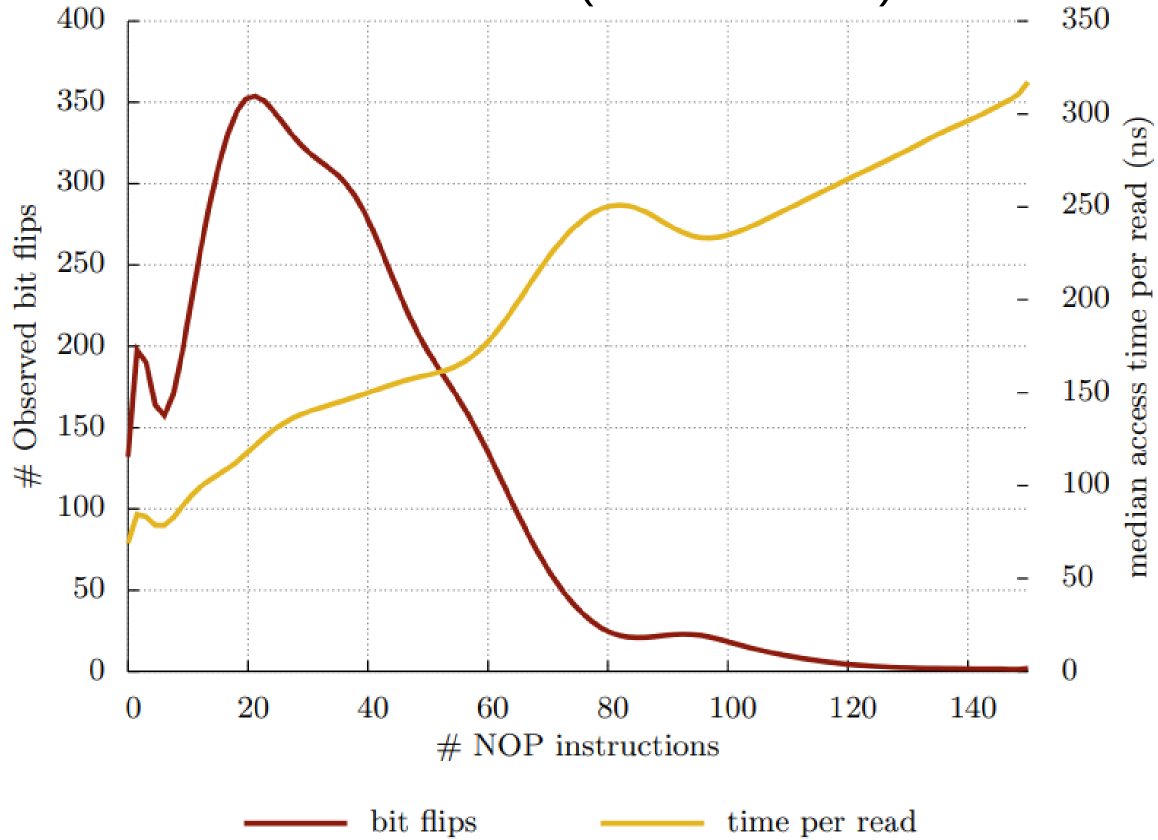
1. Phys Feng Shui: A generic technique for deterministic Rowhammer exploitation
   - Using commodity features offered by the OS
   - Abusing the predictable behavior of the memory allocator

2. Using this technique to implement an Android root exploit: Drammer

RowhARMer

- Rowhammer on mobile devices is possible!

Device: LG Nexus 5 (Android 6.0.1)



How To Exploit Rowhammer

1. Physical memory addressing
2. Fast uncached memory access
3. Physical memory massaging

How To Exploit Rowhammer

1. Physical memory addressing
2. Fast uncached memory access
3. Physical memory massaging

1. Physical Memory Addressing

- Physical memory layout is unknown to userspace
- Problem: We need to know the mapping from virtual to physical memory pages to exploit Rowhammer
- Current methods in x86:
  - Pagemap interface
  - Huge pages

2. Fast Uncached Memory Access

- Prerequisite to trigger the Rowhammer bug

- Problems:
  - Memory controller might not be fast enough
  - CPU cache masks out all memory reads after the first

- We need to bypass the cache somehow

- Current methods in x86:
  - Explicit cache flush using `clflush`
  - Cache eviction sets
  - Non-temporal access instructions (e.g. `MOVNTI`, `MOVNTDQA`)

DMA Buffer Management

- Modern (mobile) devices have many different hardware components:
  - e.g. GPU, Display Controller, Camera, Sensors, ...

- OS needs to provide direct memory access (DMA) to support efficient memory sharing between components

- Most devices perform DMA operations on contiguous physical memory pages

- Without DMA the CPU would have to stall for all memory accesses from all hardware components

DMA provides all we need

- DMA bypasses the cache ✓
- DMA gives us physically contiguous memory ✓
  - This provides us with at least relative physical memory addressing
- On Android: ION memory allocator

How To Exploit Rowhammer

1. Physical memory addressing
2. Fast uncached memory access
3. Physical memory massaging

3. Physical Memory Massaging

- Trick the victim into using a memory cell that is vulnerable to Rowhammer
- Victim should store security-sensitive data (e.g. page table) into vulnerable cell
- Current methods in x86:
  - Page-table spraying
  - Memory deduplication
  - MMU paravirtualization

Phys Feng Shui

Flip Feng Shui: Hammering a Needle in the Software Stack

Kaveh Razavi*
Vrije Universiteit Amsterdam

Ben Gras*
Vrije Universiteit Amsterdam

Erik Bosman
Vrije Universiteit Amsterdam

Bart Preneel
Katholieke Universiteit Leuven

Cristiano Giuffrida
Vrije Universiteit Amsterdam

Herbert Bos
Vrije Universiteit Amsterdam

1. Allocate “everything”
2. Free a page which is vulnerable
3. The victim has to use the vulnerable page for its data

x86 vs. ARM

	x86 Platforms	ARMv7/ARMv8
Physical Memory Addressing		
Pagemap interface	●	○
Huge pages	●	-
Fast Uncached Memory Access		
Explicit cache flush	●	○ / ◐
Cache eviction sets	●	- / -
Non-temporal access instructions	●	- / ◐
Physical Memory Massaging		
Page-table spraying	●	◐
Memory deduplication	●	-
MMU paravirtualization	●	-

●: Available in unprivileged mode
○: Available in privileged mode
◐: Not practical enough

Attack Overview

1. **Memory Templating**
   Scan memory for useful bit flips
2. **Land sensitive data**
   Store a page table on a vulnerable location
3. **Reproduce the bit flip**
   Modify the page table to get root access

Mechanisms

Attack Procedure in Detail

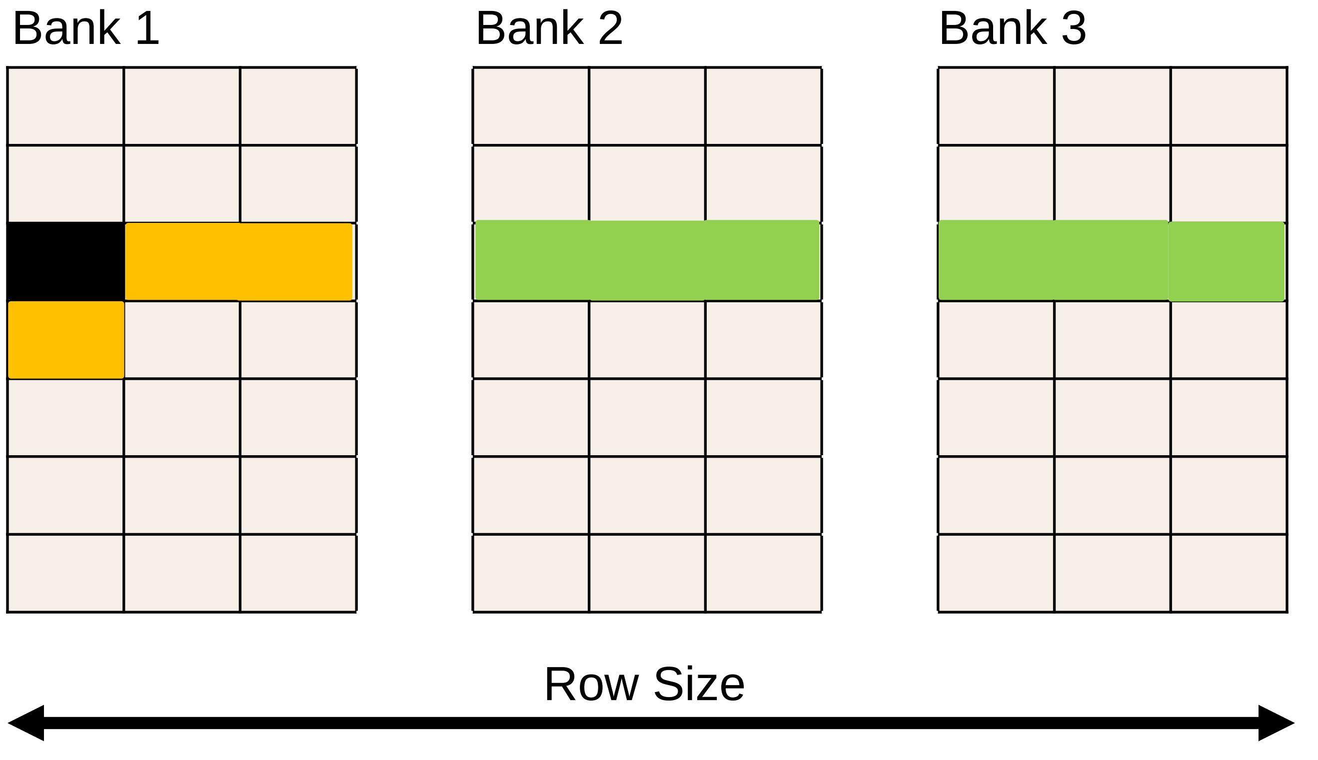
1. Probe DRAM row size
2. Phys Feng Shui
3. Hammering the page-table
4. Exploiting

Attack Procedure in Detail

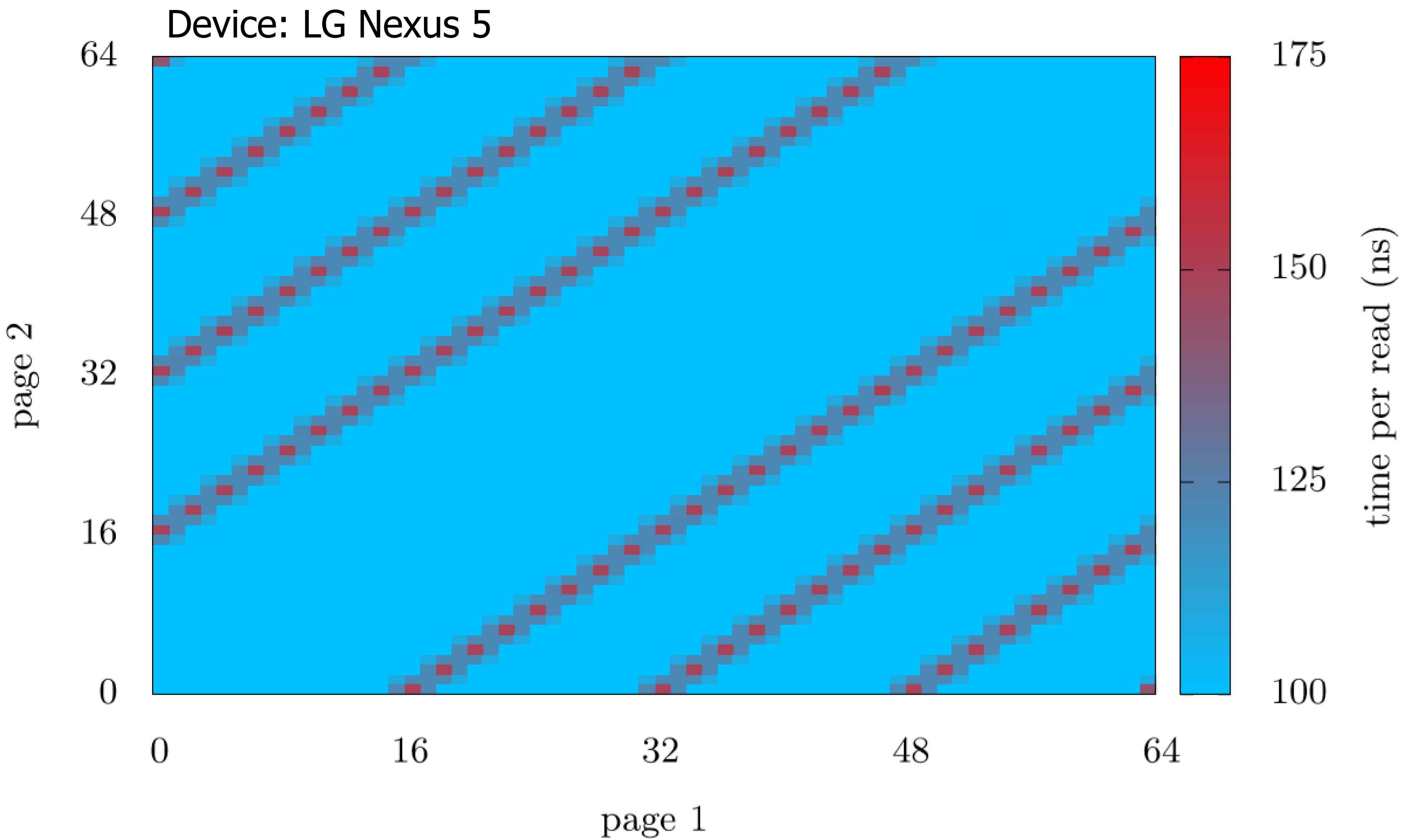
1. Probe DRAM row size
2. Phys Feng Shui
3. Hammering the page-table
4. Exploiting

Probing DRAM Row Size

- We have to know the DRAM row size to apply Rowhammer
- Two page reads from the same bank are slower than from different banks



Probing DRAM Row Size in Practice

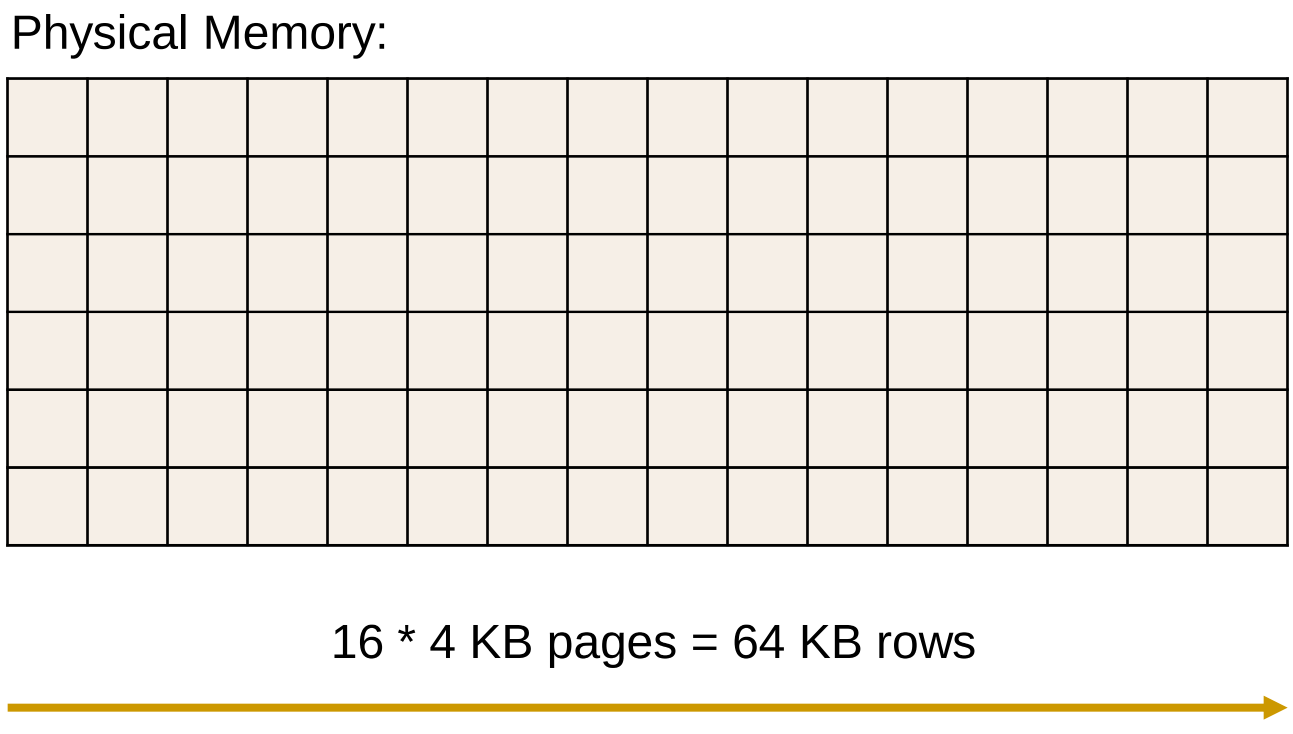


Attack Procedure in Detail

1. Probe DRAM row size
2. Phys Feng Shui
3. Hammering the page-table
4. Exploiting

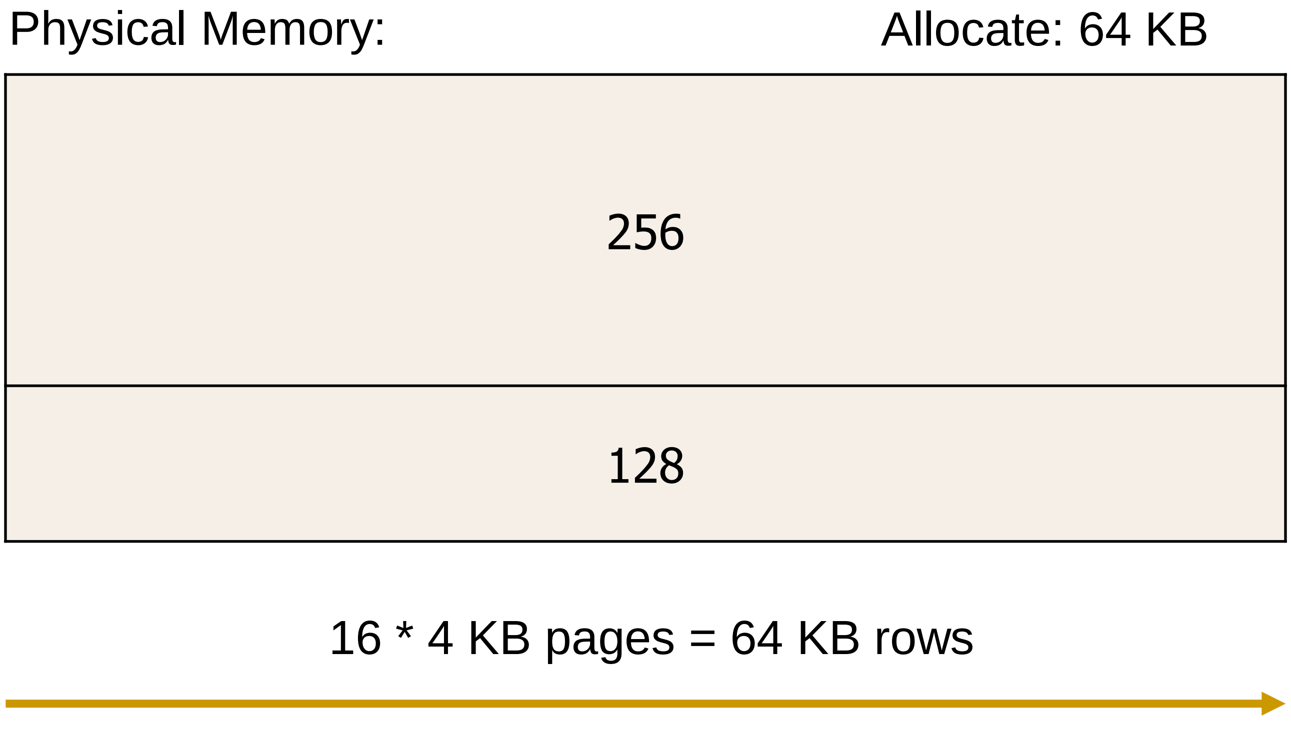
Phys Feng Shui – Buddy Allocator

- Exploit predictable behavior of the Linux Buddy Allocator
  - Split smallest chunk until it fits the requested allocation
  - On free: Merge chunks back into bigger chunks



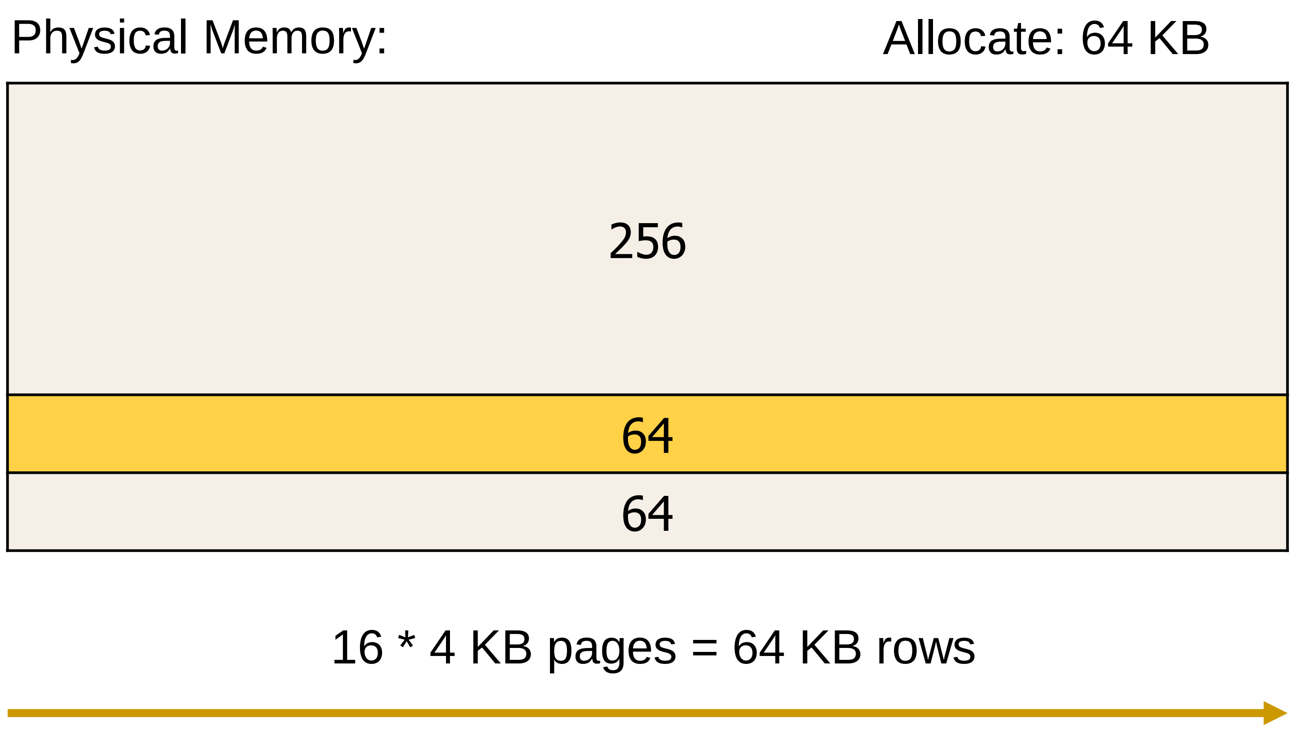
Phys Feng Shui – Buddy Allocator

- Exploit predictable behavior of the Linux Buddy Allocator
  - Split smallest chunk until it fits the requested allocation
  - On free: Merge chunks back into bigger chunks



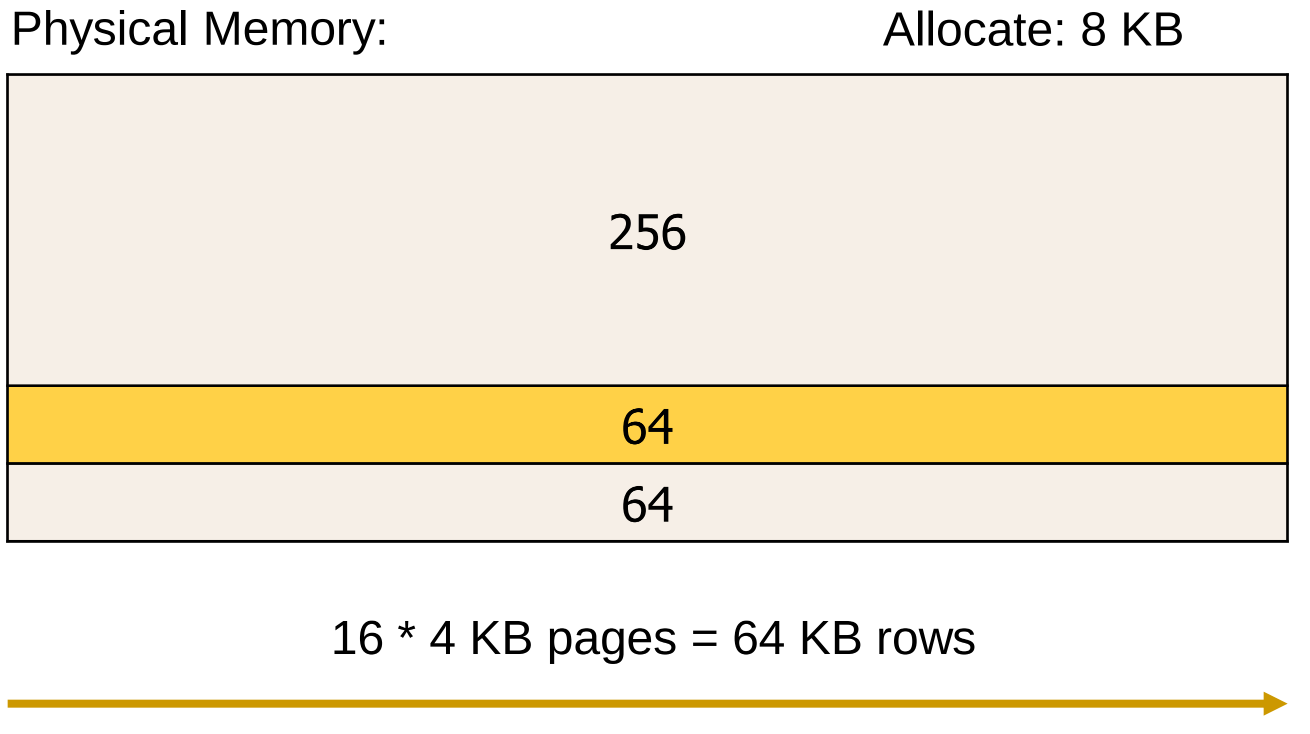
Phys Feng Shui – Buddy Allocator

- Exploit predictable behavior of the Linux Buddy Allocator
  - Split smallest chunk until it fits the requested allocation
  - On free: Merge chunks back into bigger chunks



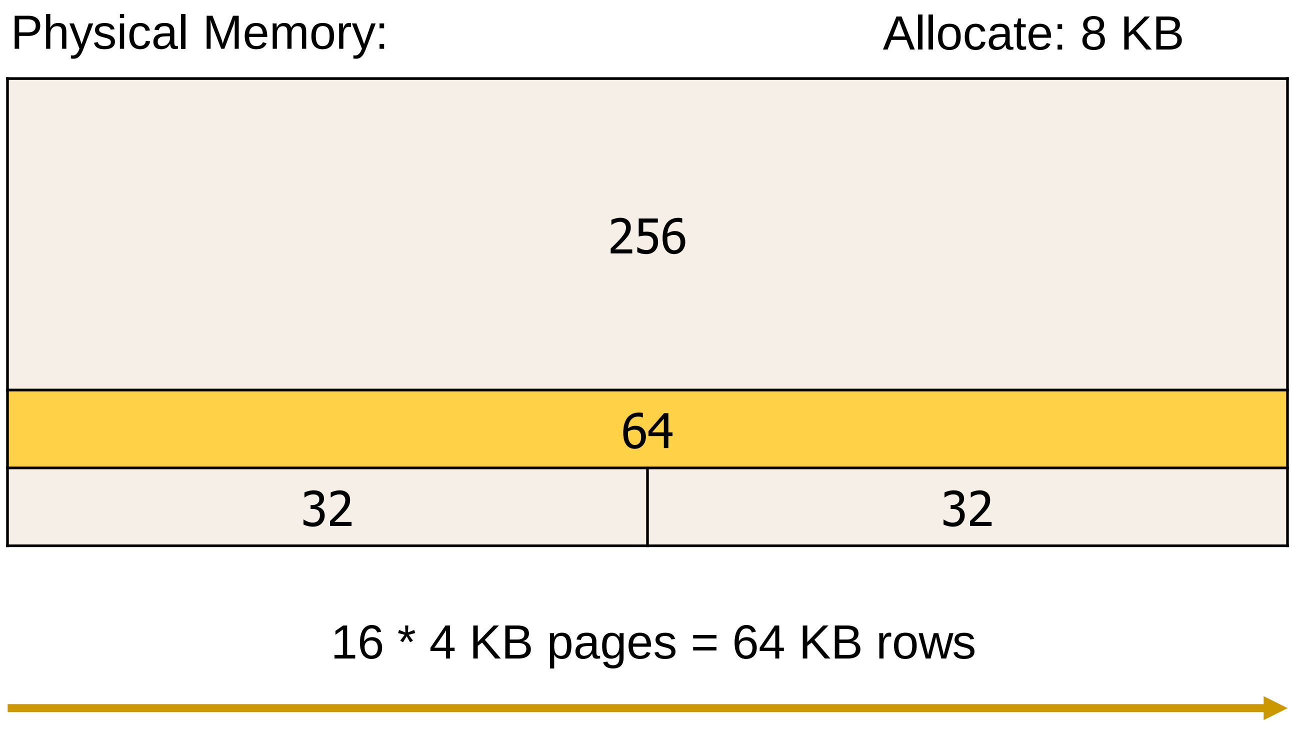
Phys Feng Shui – Buddy Allocator

- Exploit predictable behavior of the Linux Buddy Allocator
  - Split smallest chunk until it fits the requested allocation
  - On free: Merge chunks back into bigger chunks



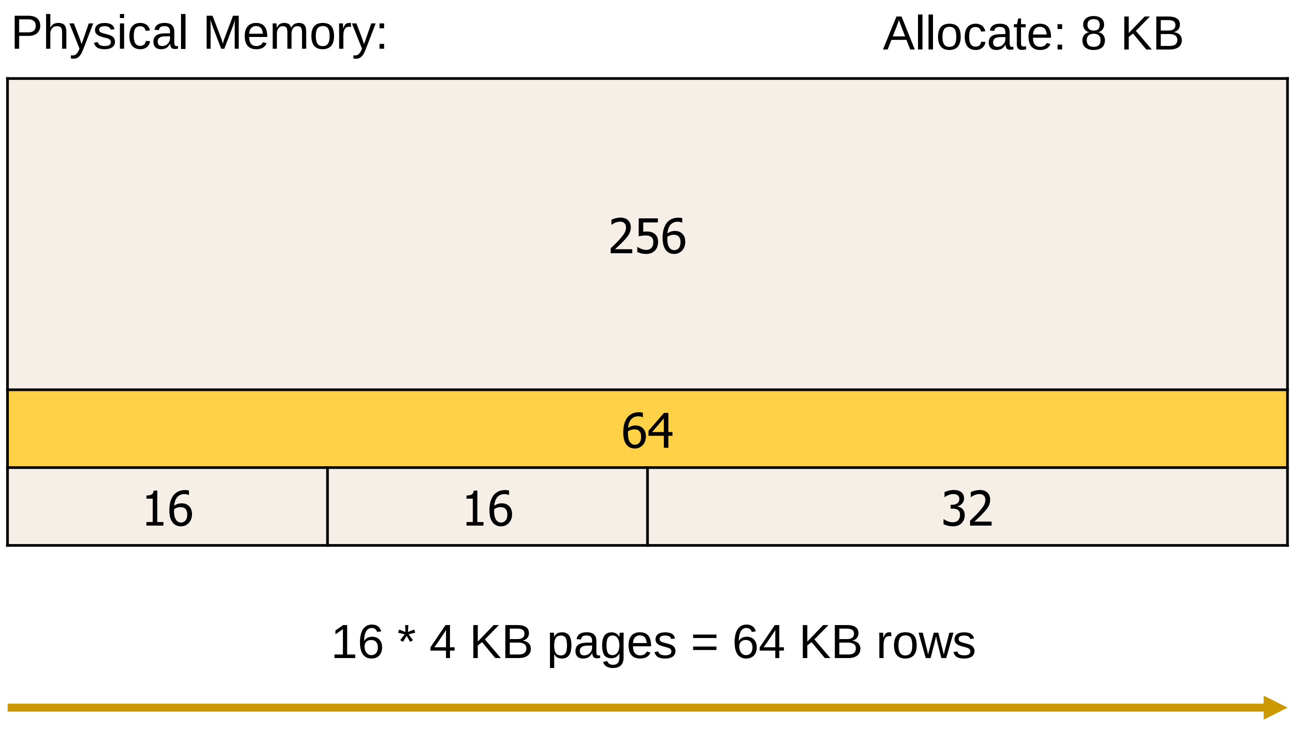
Phys Feng Shui – Buddy Allocator

- Exploit predictable behavior of the Linux Buddy Allocator
  - Split smallest chunk until it fits the requested allocation
  - On free: Merge chunks back into bigger chunks



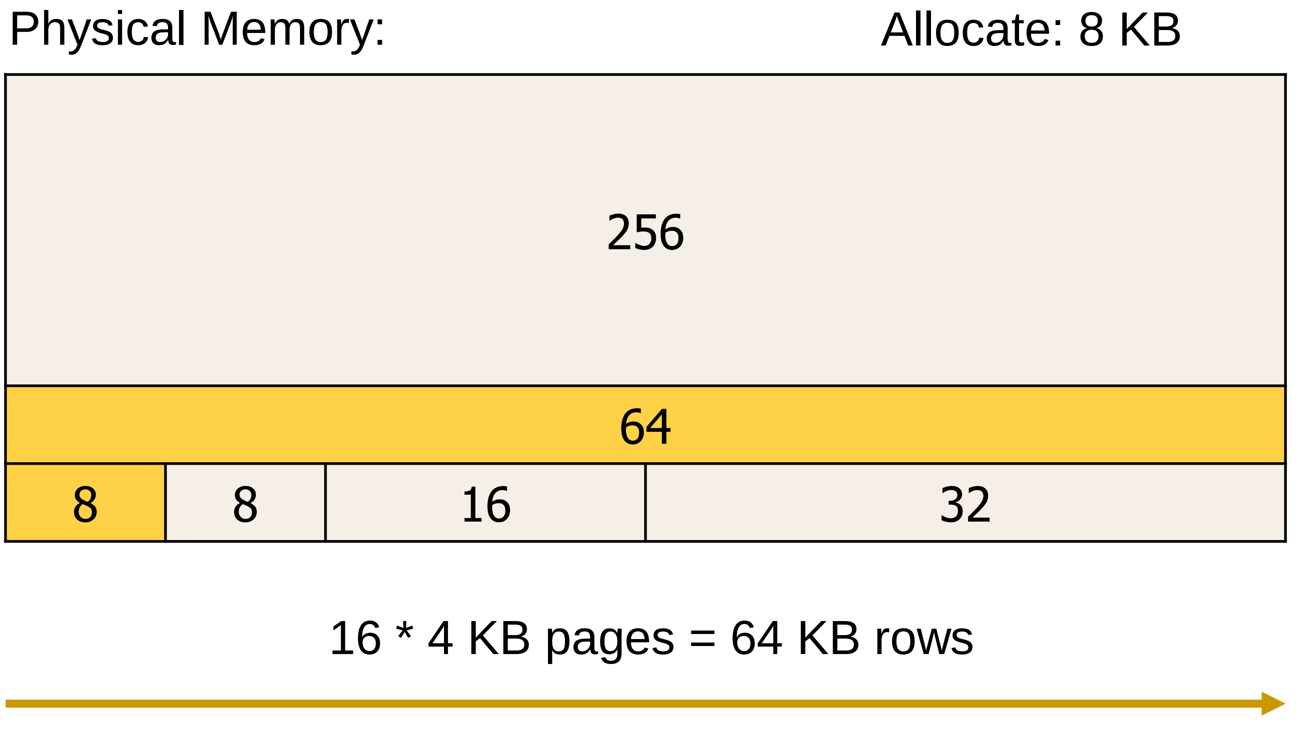
Phys Feng Shui – Buddy Allocator

- Exploit predictable behavior of the Linux Buddy Allocator
  - Split smallest chunk until it fits the requested allocation
  - On free: Merge chunks back into bigger chunks



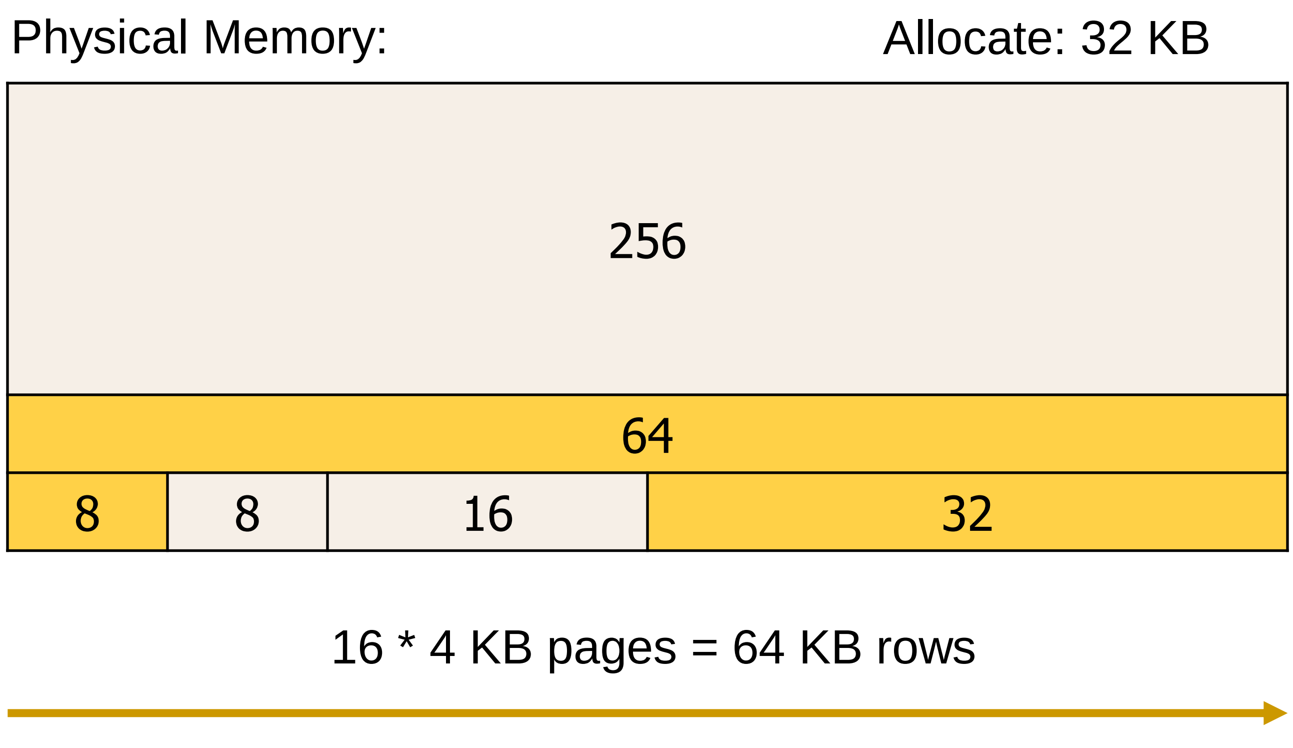
Phys Feng Shui – Buddy Allocator

- Exploit predictable behavior of the Linux Buddy Allocator
  - Split smallest chunk until it fits the requested allocation
  - On free: Merge chunks back into bigger chunks



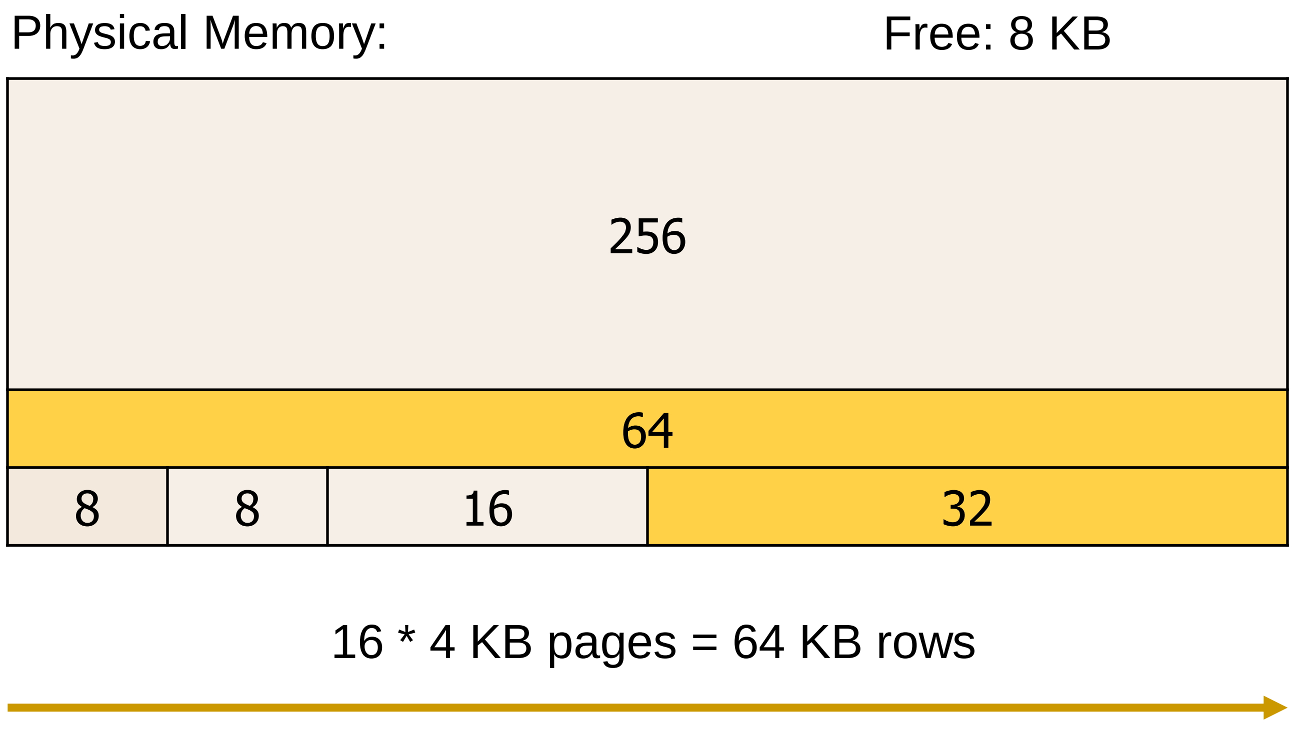
Phys Feng Shui – Buddy Allocator

- Exploit predictable behavior of the Linux Buddy Allocator
  - Split smallest chunk until it fits the requested allocation
  - On free: Merge chunks back into bigger chunks



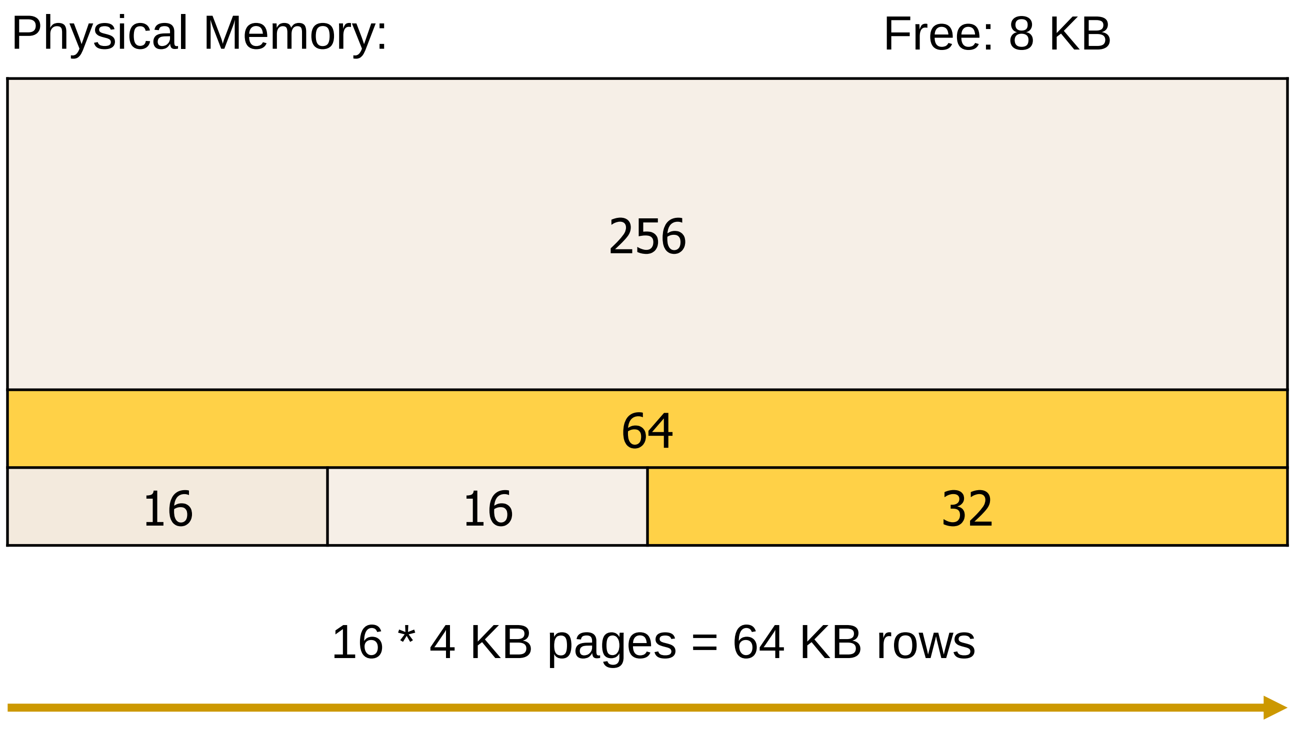
Phys Feng Shui – Buddy Allocator

- Exploit predictable behavior of the Linux Buddy Allocator
  - Split smallest chunk until it fits the requested allocation
  - On free: Merge chunks back into bigger chunks



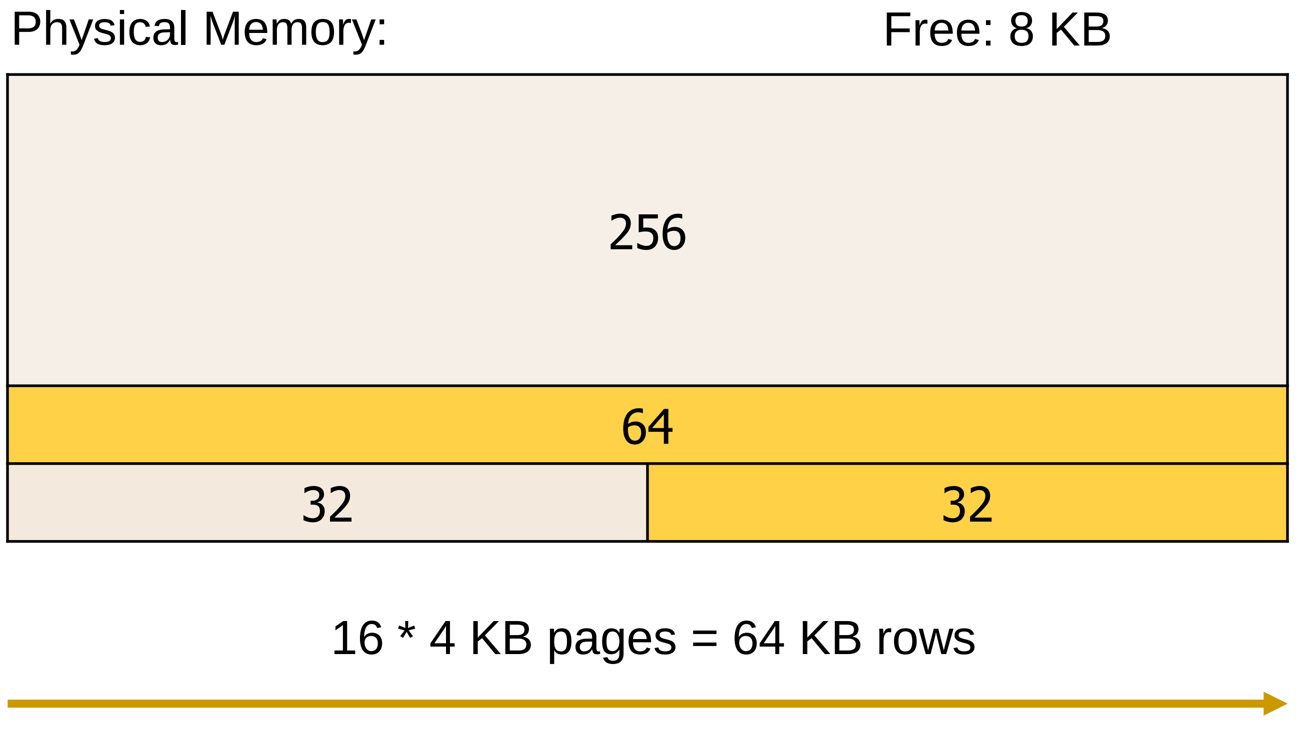
Phys Feng Shui – Buddy Allocator

- Exploit predictable behavior of the Linux Buddy Allocator
  - Split smallest chunk until it fits the requested allocation
  - On free: Merge chunks back into bigger chunks



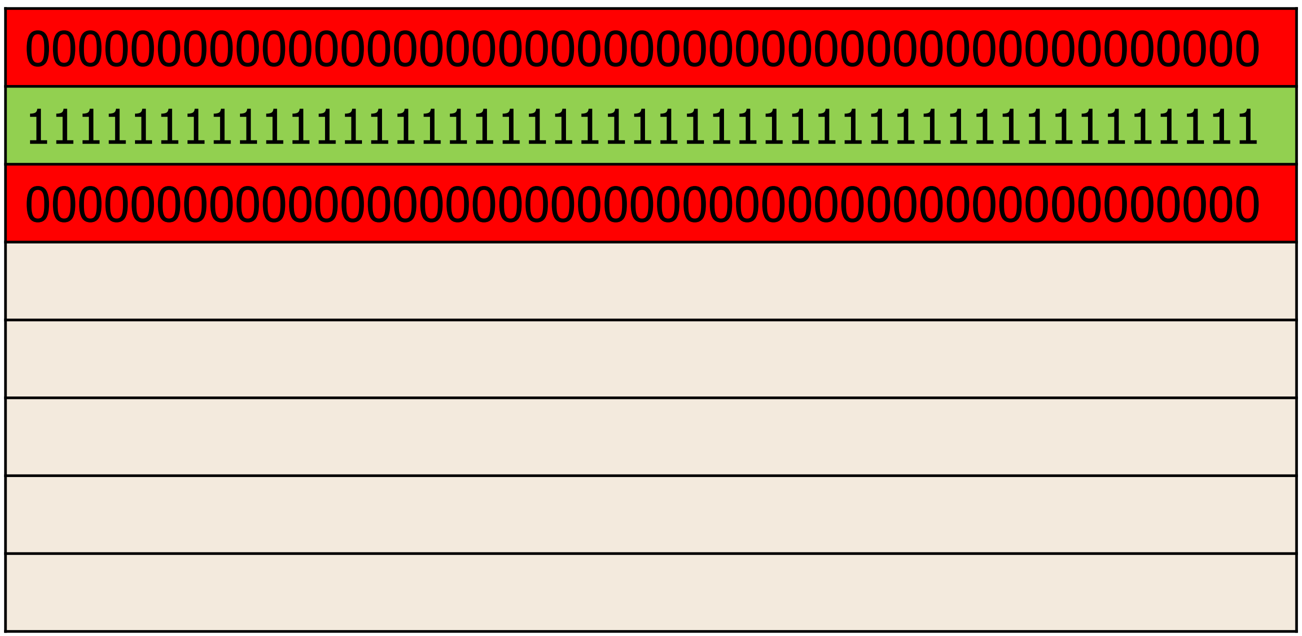
Phys Feng Shui – Buddy Allocator

- Exploit predictable behavior of the Linux Buddy Allocator
  - Split smallest chunk until it fits the requested allocation
  - On free: Merge chunks back into bigger chunks



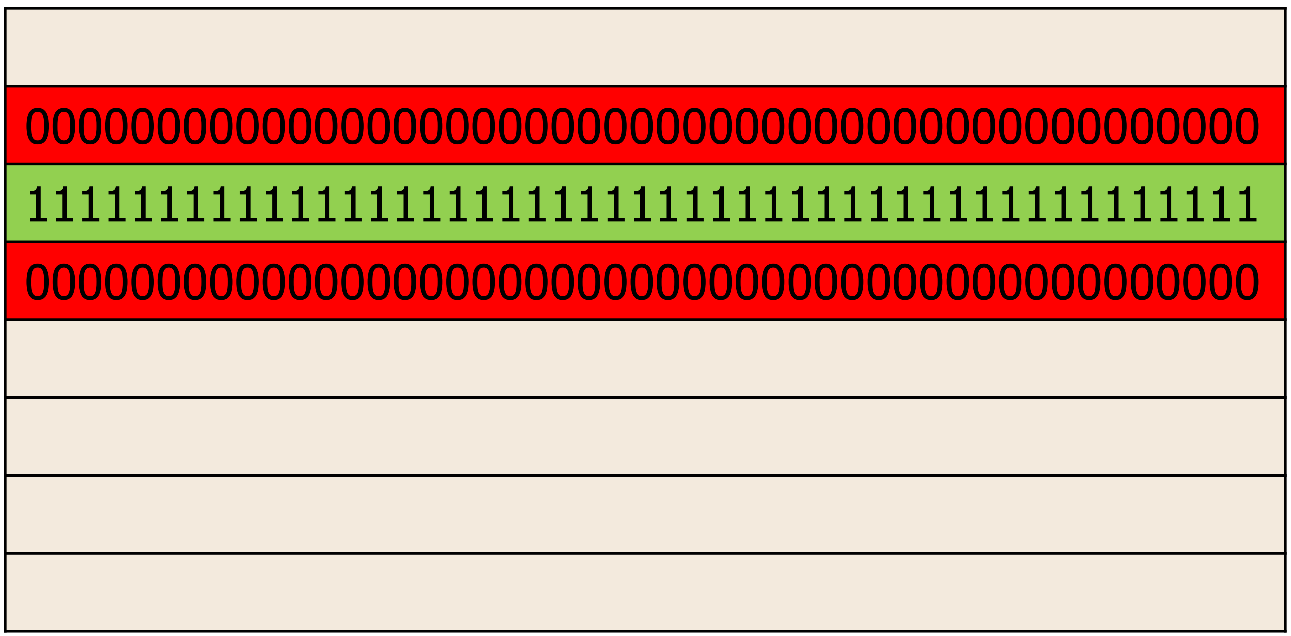
Phys Feng Shui - Memory Templating

- Scan memory for vulnerable rows and keep track of which bits flipped in which row



Phys Feng Shui - Memory Templating

- Scan memory for vulnerable rows and keep track of which bits flipped in which row

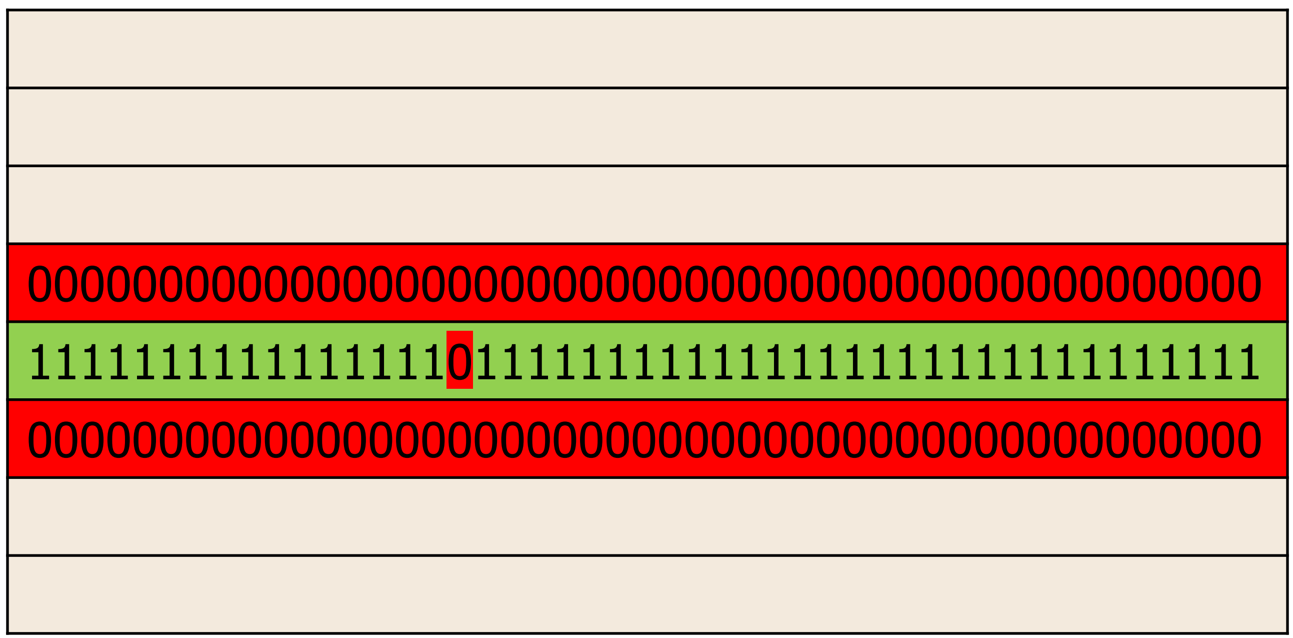


Phys Feng Shui - Memory Templating

- Scan memory for vulnerable rows and keep track of which bits flipped in which row

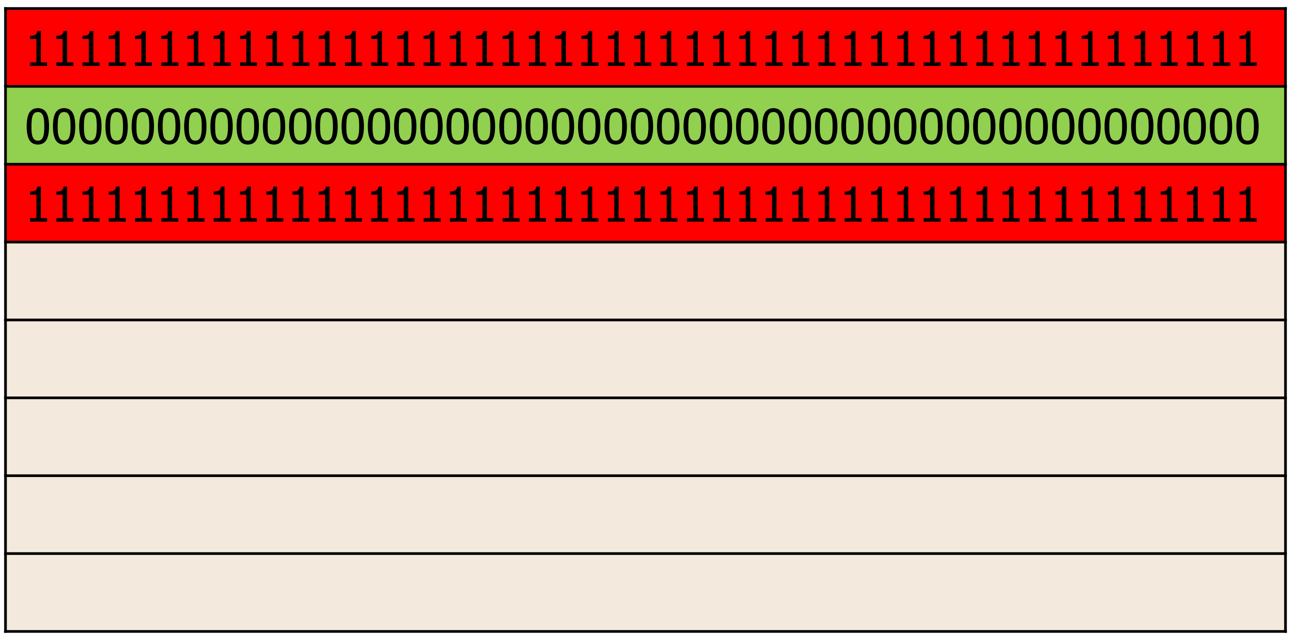
Phys Feng Shui - Memory Templating

- Scan memory for vulnerable rows and keep track of which bits flipped in which row



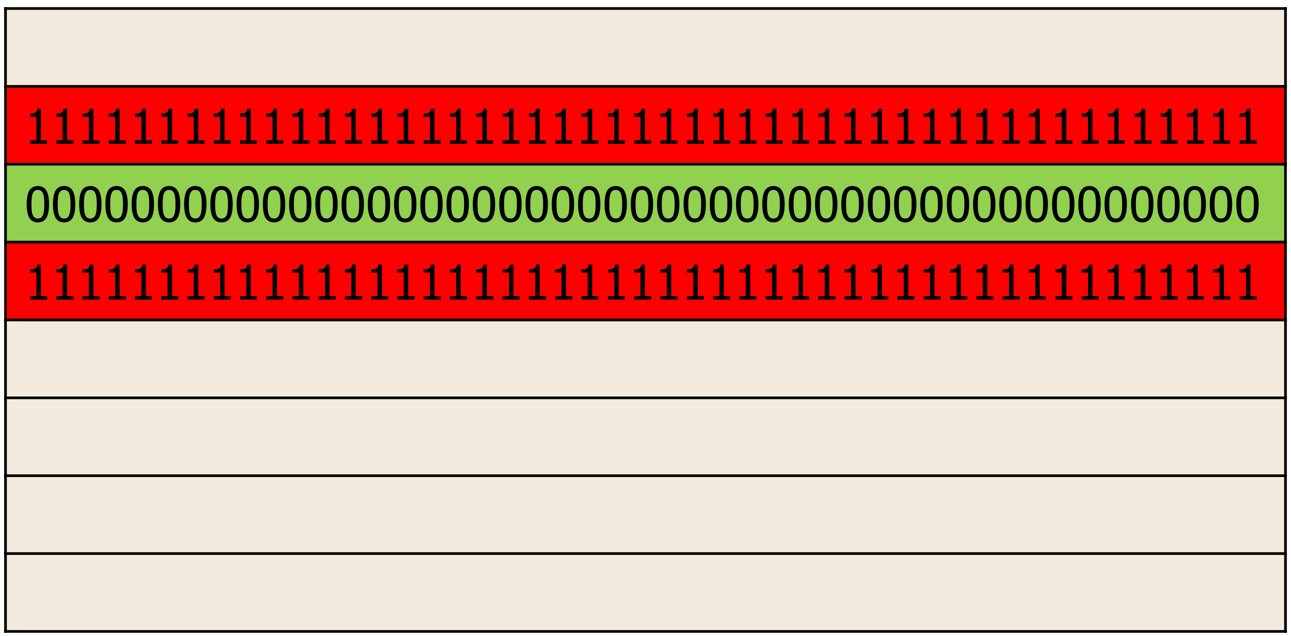
Phys Feng Shui - Memory Templating

- Scan memory for vulnerable rows and keep track of which bits flipped in which row



Phys Feng Shui - Memory Templating

- Scan memory for vulnerable rows and keep track of which bits flipped in which row



Phys Feng Shui - Memory Templating

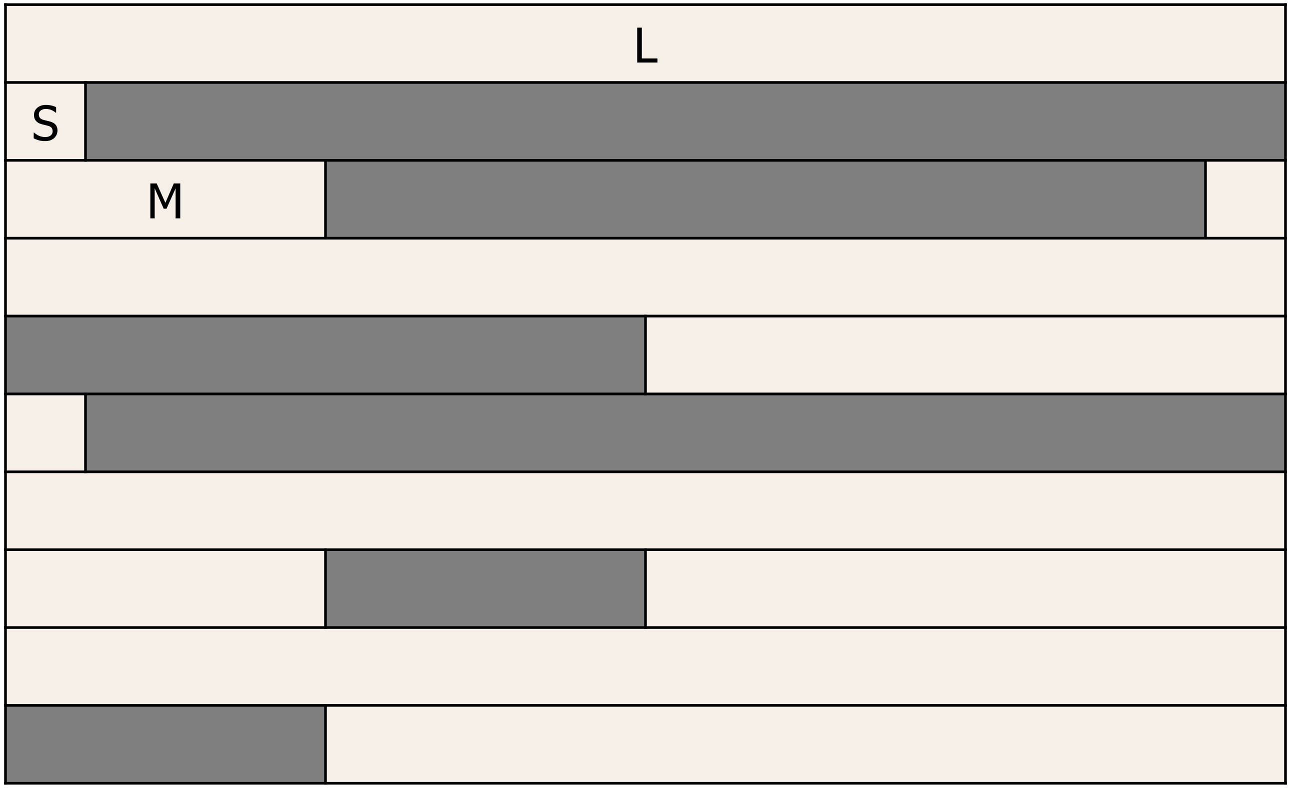
- Scan memory for vulnerable rows and keep track of which bits flipped in which row

Phys Feng Shui - Memory Templating

- Scan memory for vulnerable rows and keep track of which bits flipped in which row

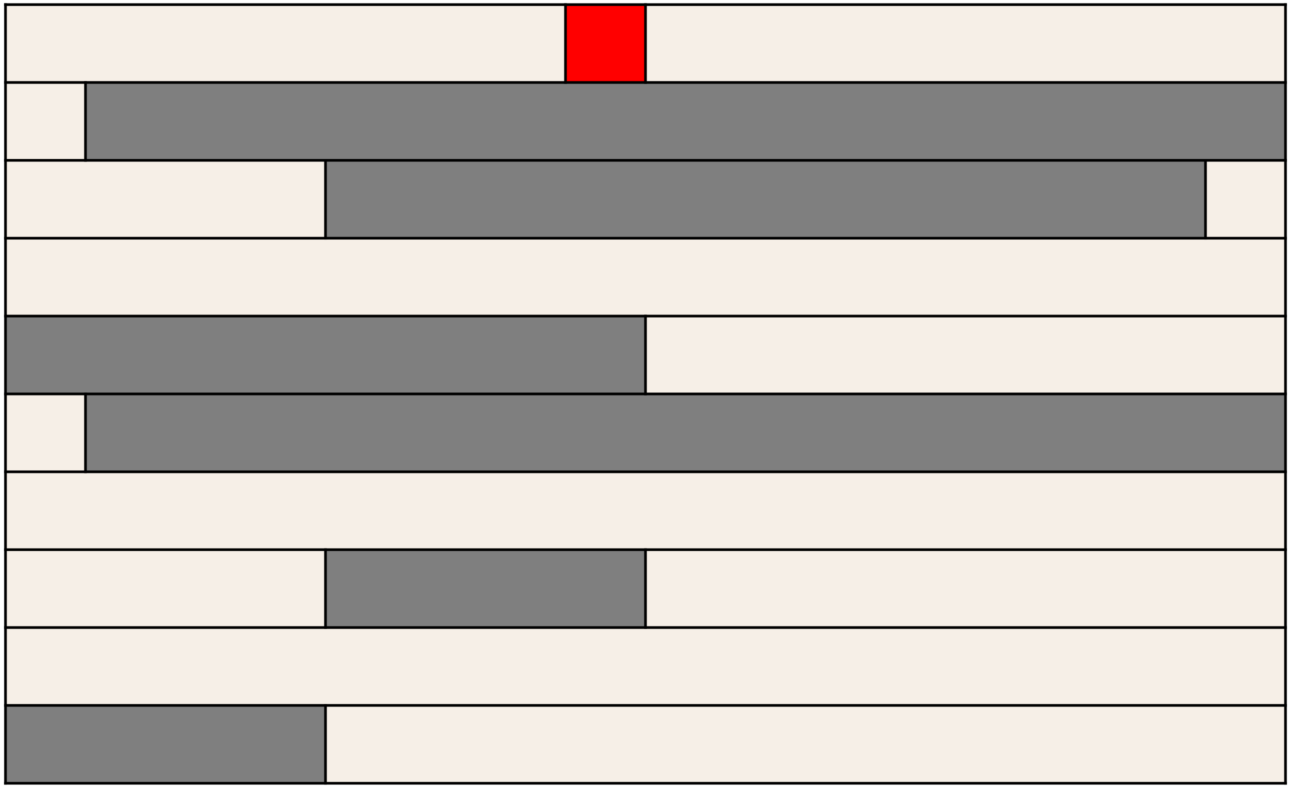
Phys Feng Shui

- L-Chunks: Largest possible contiguous chunk = 64 KB
- M-Chunks: Row Size = 16 KB
- S-Chunks: Page Size = 4 KB



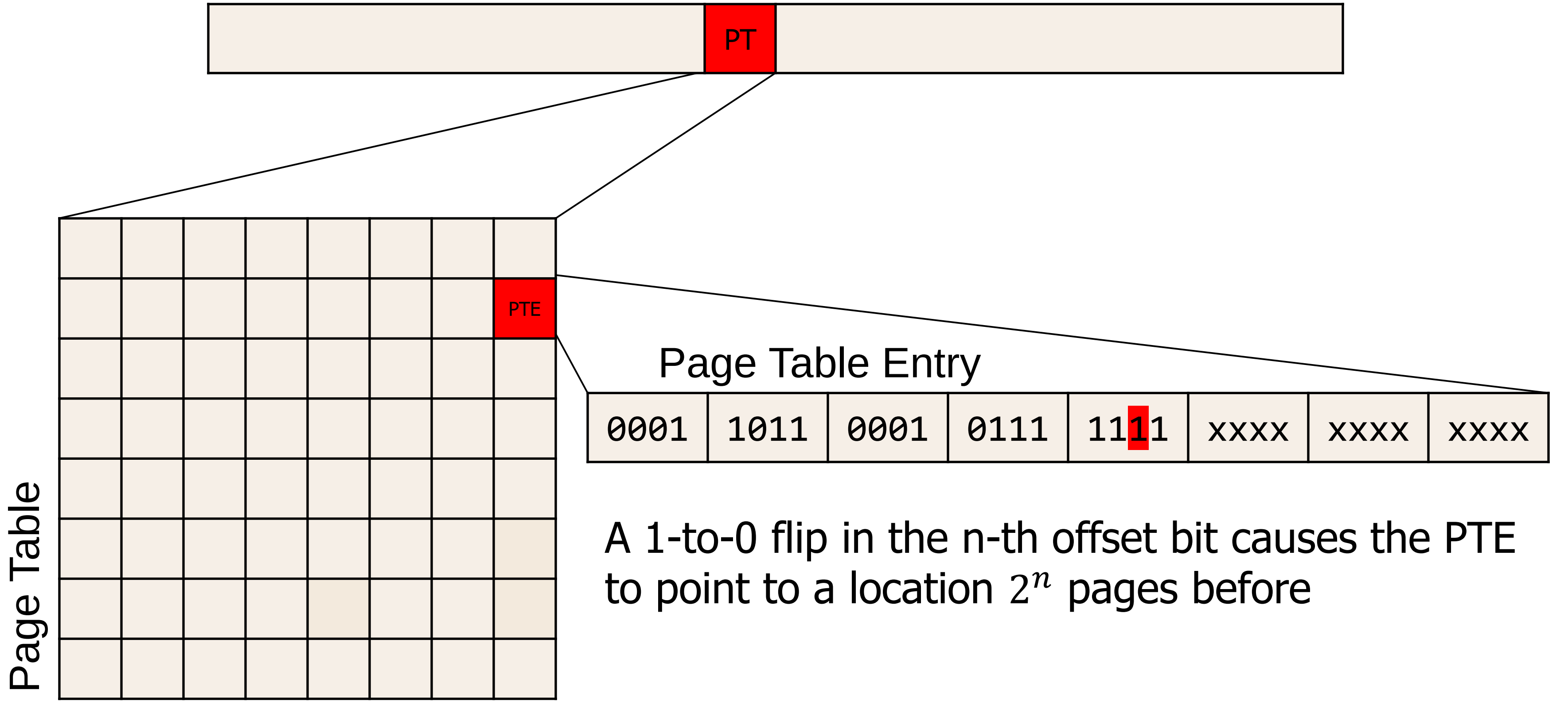
Phys Feng Shui

- Assume we have an exploitable bit-flip in the red location
- Trick the OS to place a page table in that location



Phys Feng Shui

- Trick the OS to place a page table in the red location



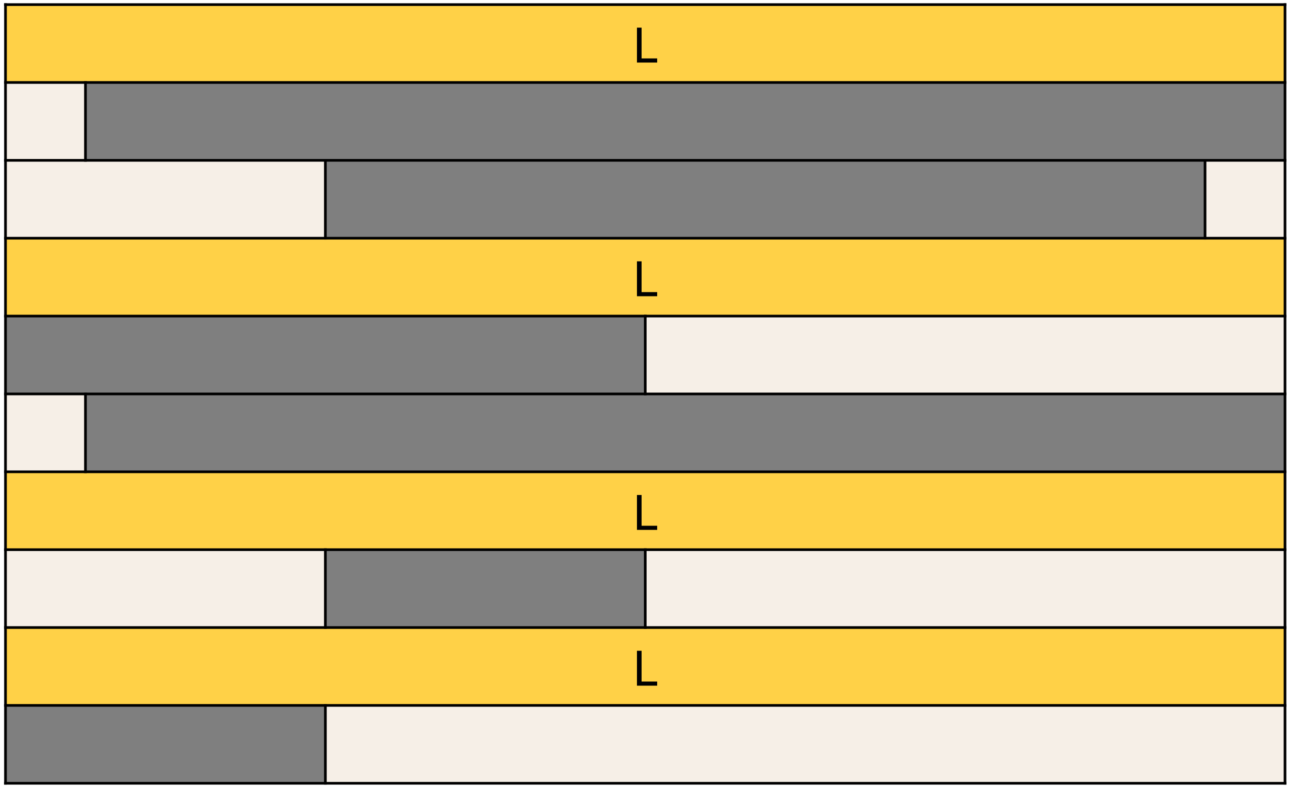
A 1-to-0 flip in the n-th offset bit causes the PTE to point to a location 2^n pages before

Phys Feng Shui - Steps

1. Exhaust(L) + Template(L)
2. Exhaust(M)
3. Free(L*)
4. Exhaust(M)
5. Free(M*) + FreeAll(L)
6. Land(S)
7. Padding(S)
8. Map(M)

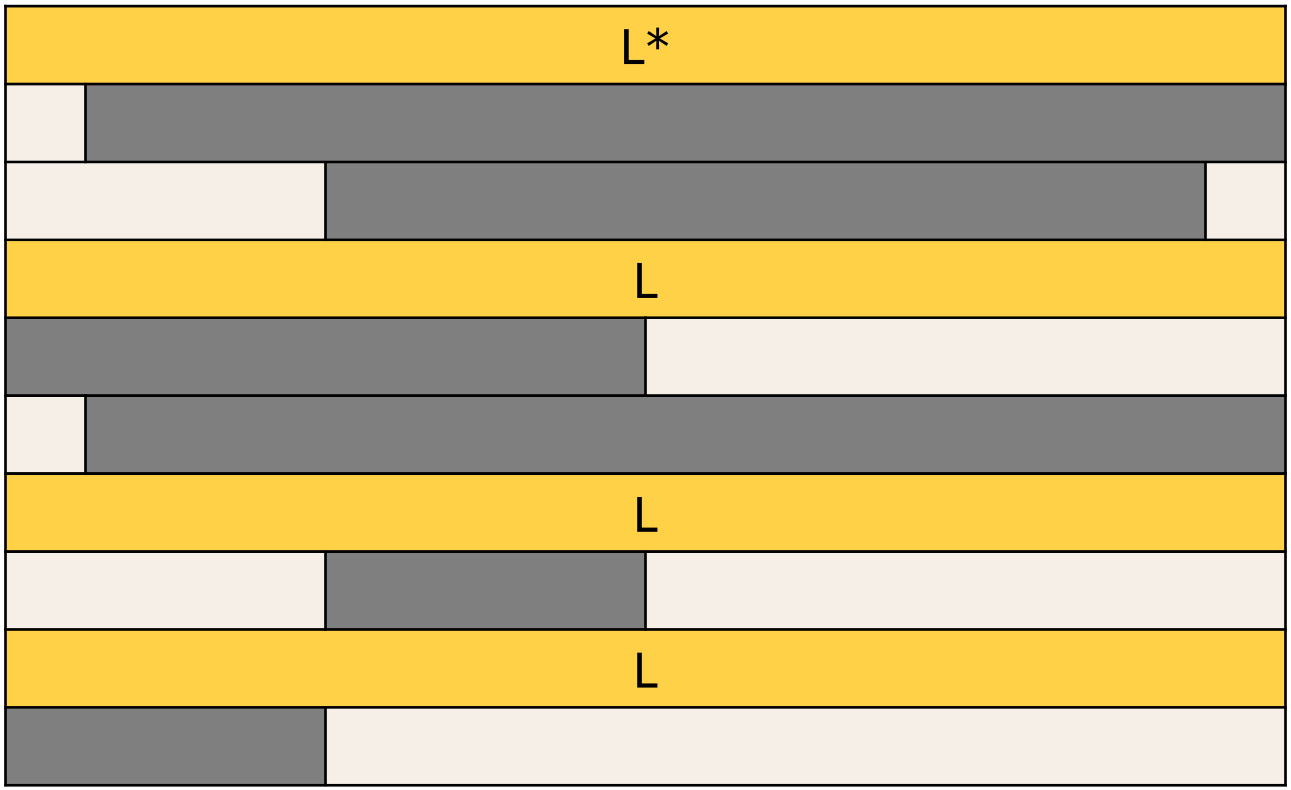
Phys Feng Shui – Exhaust(L) + Template(L)

- Allocate as many L-Chunks as possible



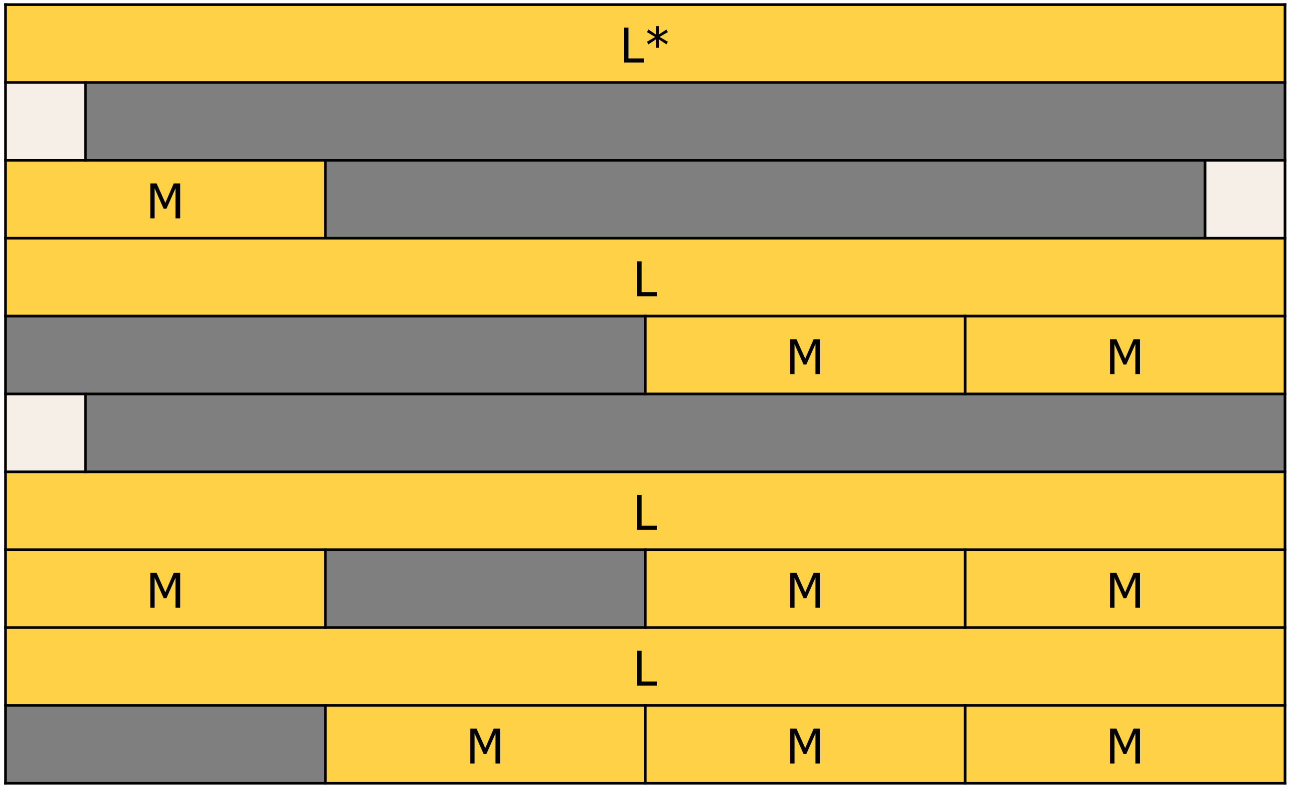
Phys Feng Shui – Exhaust(L) + Template(L)

- Allocate as many L-Chunks as possible
- Scan rows in L-Chunks for vulnerable rows (Templating)



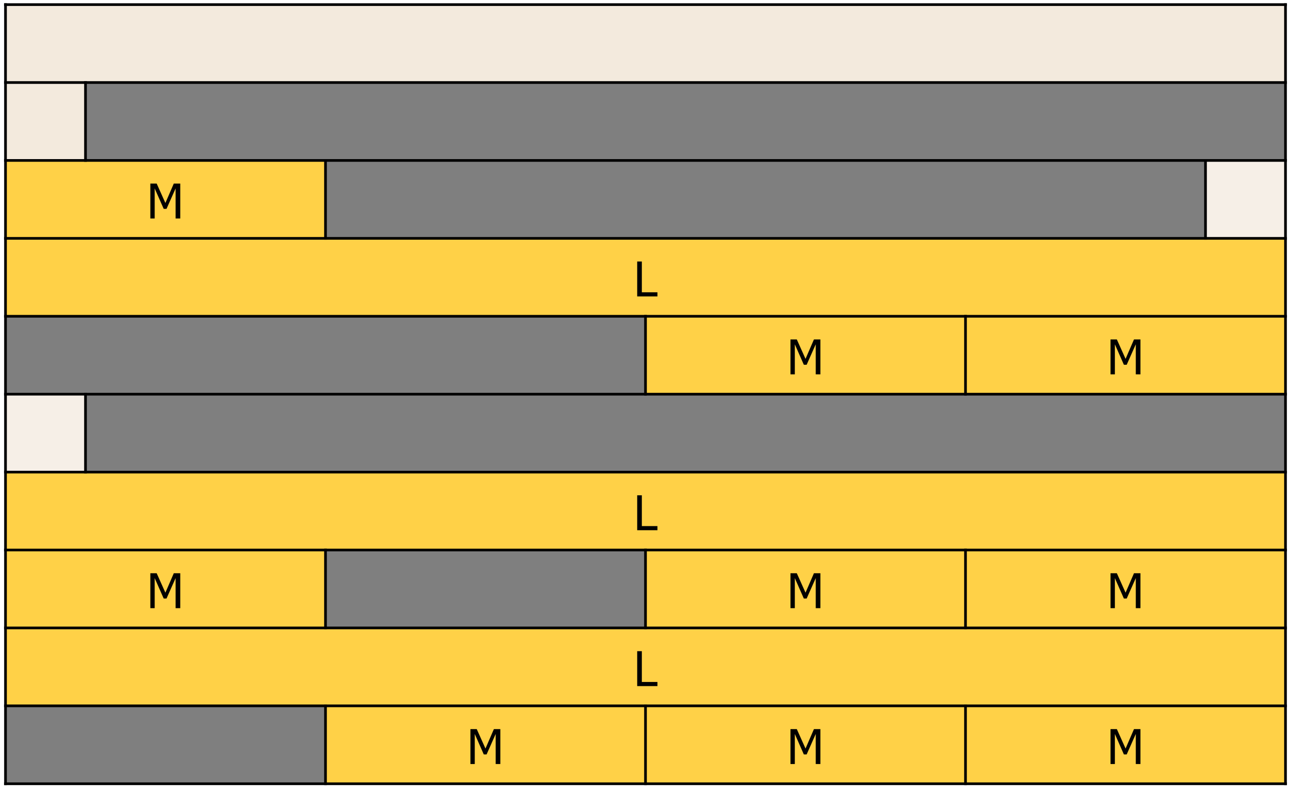
Phys Feng Shui – Exhaust(M)

- Allocate as many M-Chunks as possible



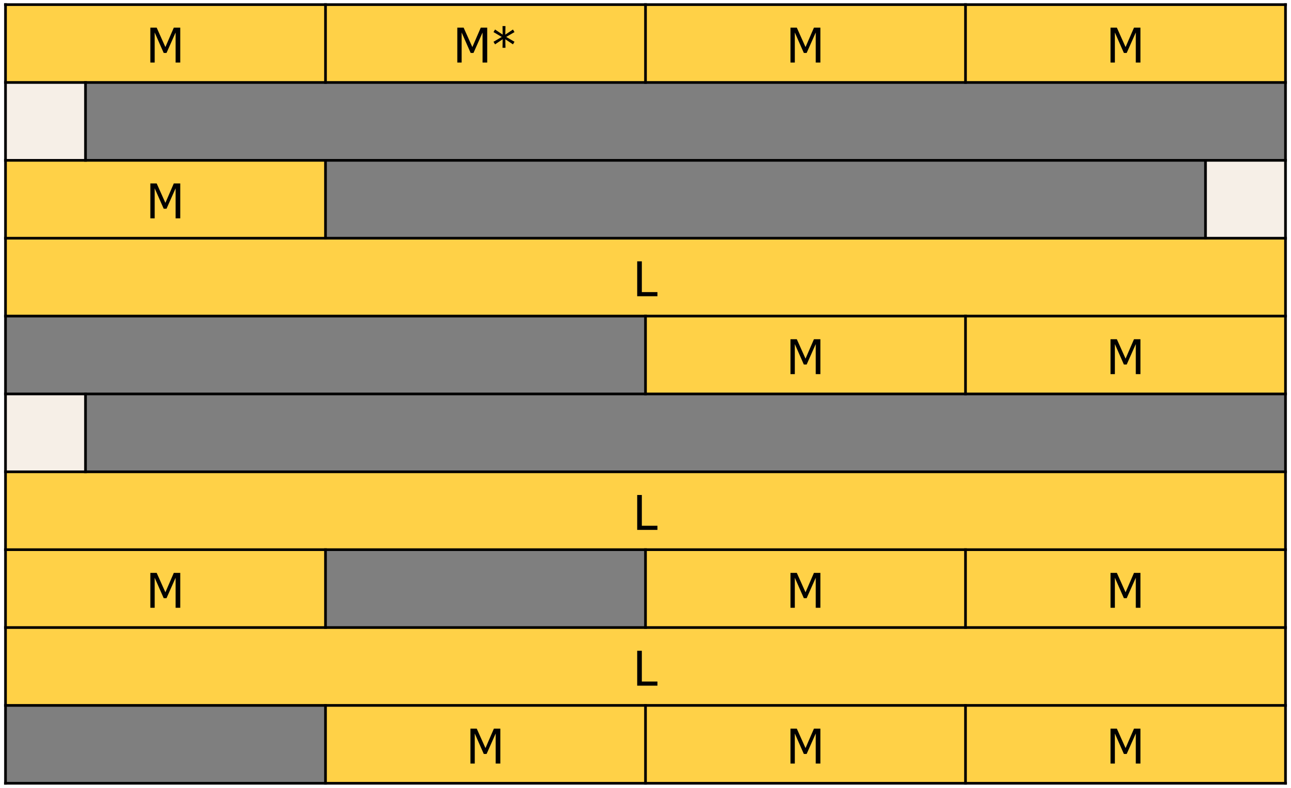
Phys Feng Shui – Free(L*)

- Free the L-Chunk with the vulnerable row



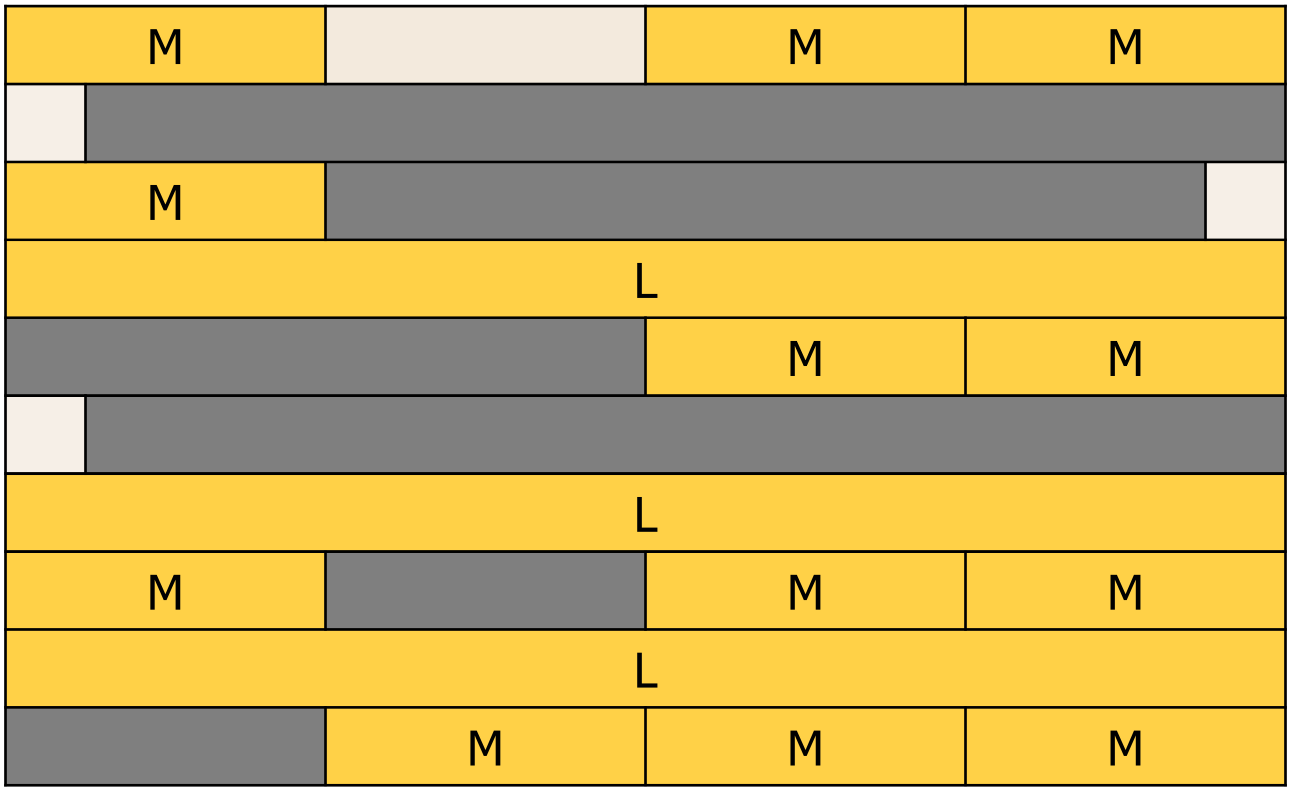
Phys Feng Shui – Free(L*) + Exhaust(M)

- Free the L-Chunk with the vulnerable row
- Allocate as many M-Chunks as possible



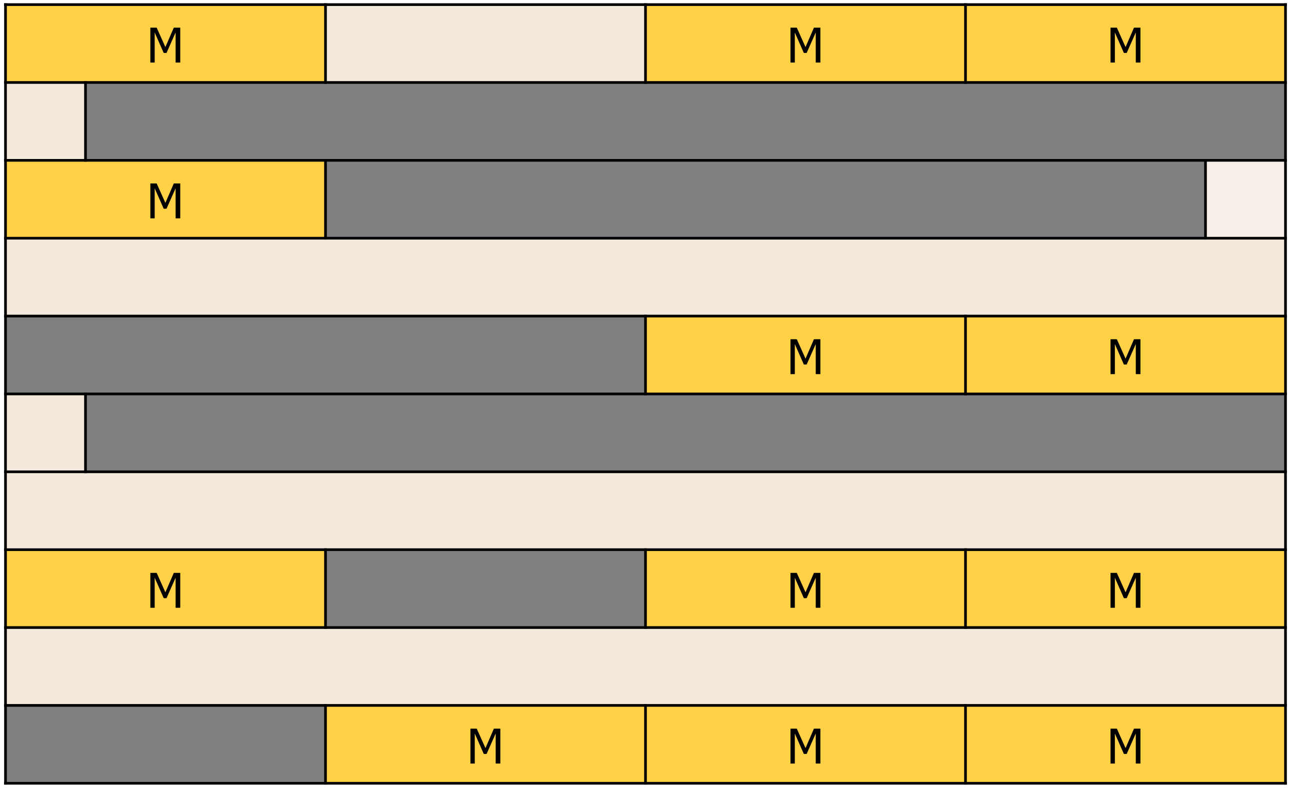
Phys Feng Shui – Free(M*) + FreeAll(L)

- Free the M-Chunk with the vulnerable row



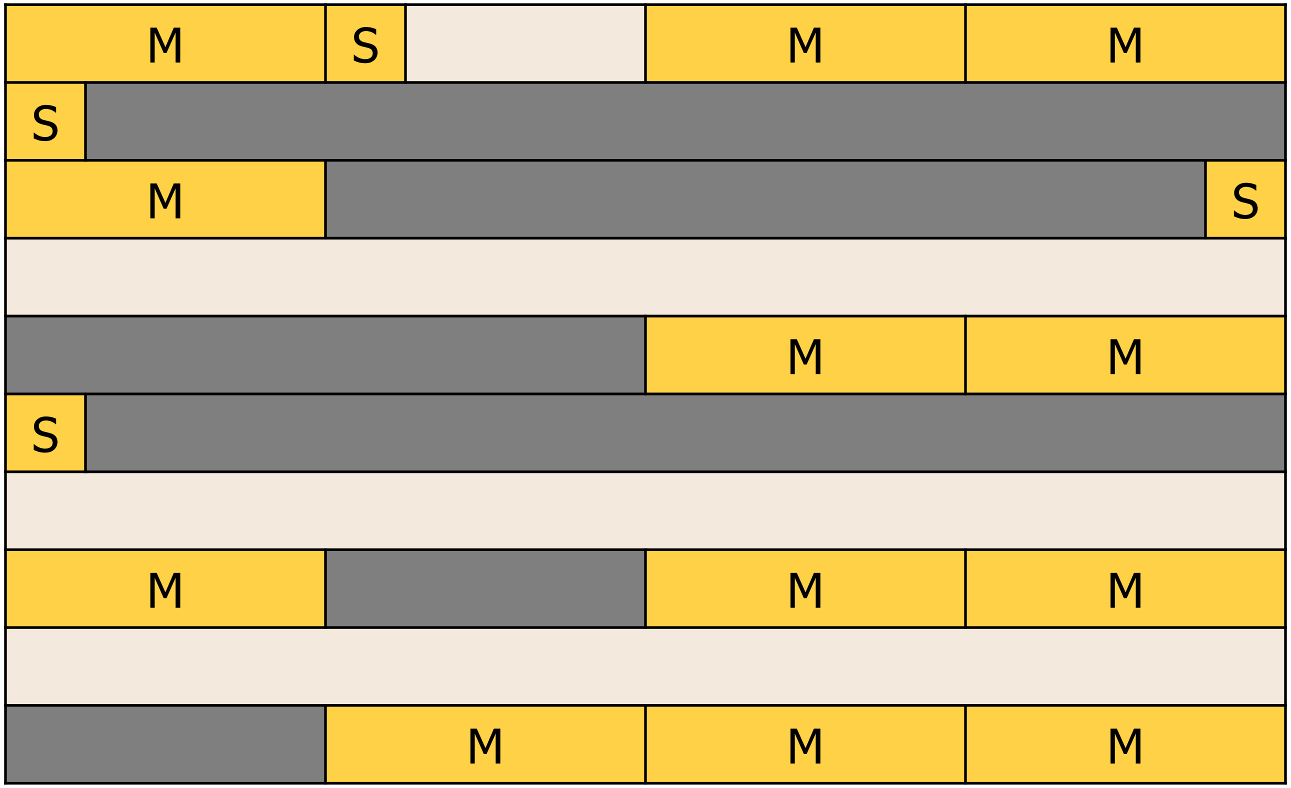
Phys Feng Shui – Free(M*) + FreeAll(L)

- Free the M-Chunk with the vulnerable row
- Free all remaining L-Chunks



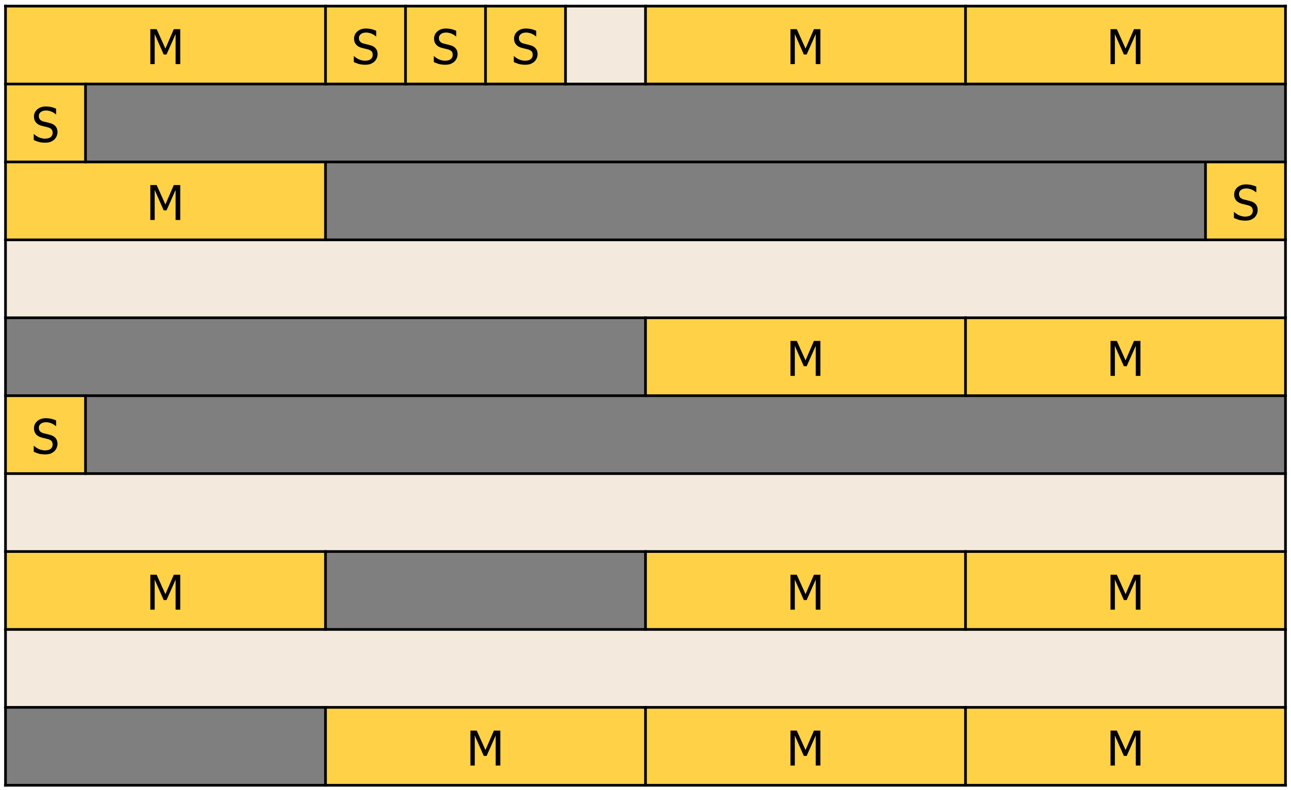
Phys Feng Shui – Land(S)

- Allocate S-Chunks until they land in the vulnerable region
  - We can use `/proc/zone-info` and `/proc/pagetypeinfo` to determine when we reach the vulnerable region



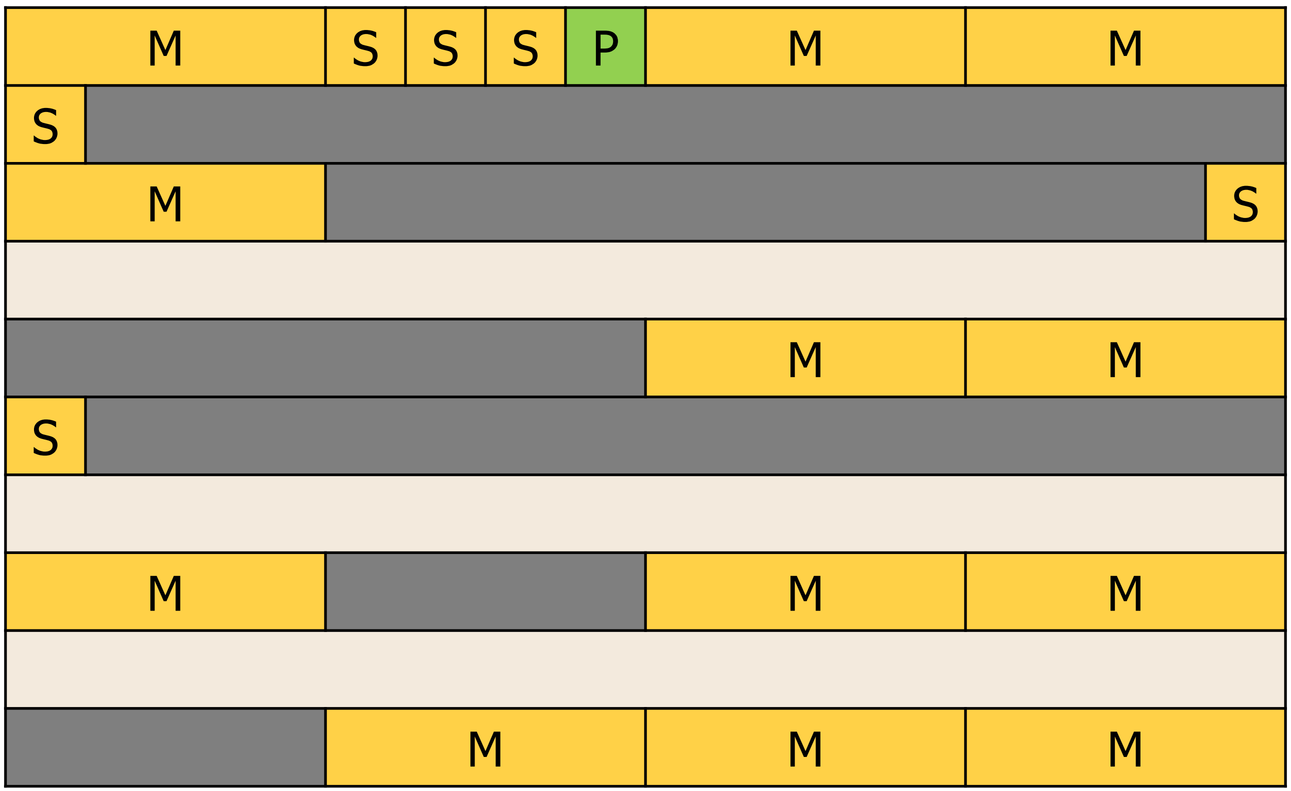
Phys Feng Shui – Padding(S)

- Insert some padding so that the next allocated page-table will be placed in the vulnerable page



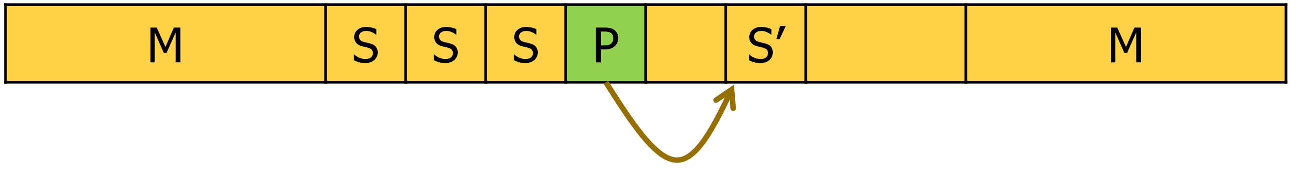
Phys Feng Shui – Map(M)

- Force another page-table allocation
- Map the PTE with a bit flip at offset bit n to a location 2^n pages away from the PT

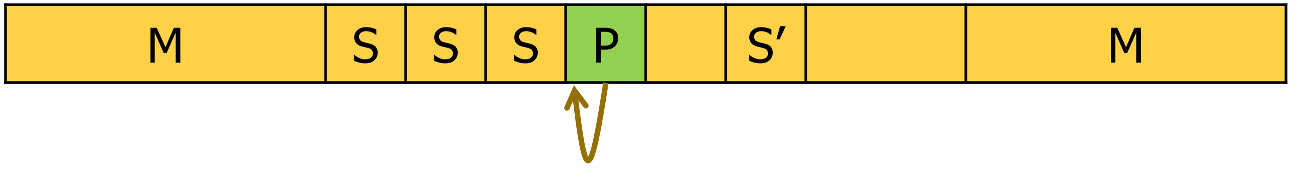


Phys Feng Shui – Map(M)

- Force another page-table allocation
- Map the PTE with a bit flip at offset bit n to a location 2^n pages away from the PT



- Map the vulnerable PTE to M’ which is $2 = 2^1$ pages away



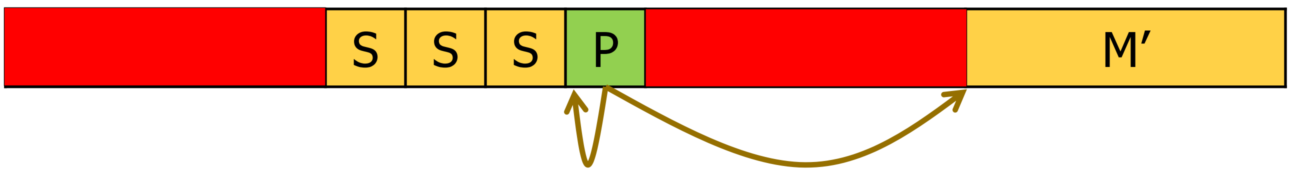
- A 1-to-0 flip in the 2nd offset bit of the PTE would result in the PTE mapping to the PT itself

Attack Procedure in Detail

1. Probe DRAM row size
2. Phys Feng Shui
3. Hammering the page-table
4. Exploiting

Hammering

- Hammer until we reproduce the bit-flip from the templating stage



- Our PTE now points to the PT itself and we can effectively access the whole memory including kernel pages.

Attack Procedure in Detail

1. Probe DRAM row size
2. Phys Feng Shui
3. Hammering the page-table
4. Exploiting

Exploitation

1. Fill PT with PTE's to kernel memory

2. Search for the security context of our own process stored in a `struct cred`

3. Overwrite our `uid` and `gid` to get root privileges

Methodology and Evaluation

Methodology

- Only Android devices were tested

- Architectures:
  - ARMv7
  - ARMv8
- DRAM types:
  - LPDDR2/3/4

- Metrics:
  - Time until first bit-flip
  - Number of bit-flips
  - Number of exploitable bit-flips

Analysis

		Hardware Details			Analysis Results							
	Device	*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

Analysis Summary

- 80% of ARMv7 devices vulnerable
- 16% of ARMv8 devices vulnerable
  - Seems more robust
- The same device can sometimes be vulnerable and sometimes not
  - 20% of Nexus 5 devices were not vulnerable

- Time until first bit flip can vary greatly
- Percentage of exploitable bit-flips always around 7%

- LPDDR2/3 is vulnerable
- LPDDR4 maybe vulnerable (only 1 device tested)

Mitigation

Software Mitigation

- Disallowing `clflush` and non-temporal access instructions
- Disallowing `pagemap` interface
- ANVIL [5]
  - Detect Rowhammer attack by observing cache misses

[5] Z.B. Aweke, et al. ANVIL: Software-Based Protection Against Next-Generation Rowhammer Attacks ACM '16

Hardware Mitigation

- Increase refresh rate
  - Needs 8x refresh rate for complete mitigation
- ECC Memory
- Target Row Refresh
  - LPDDR4 supports this
- PARA[1] & ARMOR[7]

[1] Y. Kim, et al. Flipping Bits in Memory Without Accessing Them, ISCA '14
[7] M. Ghasempour, et al. ARMOR: A Run-Time Memory Hot-Row Detector, 2015

Drammer Mitigation

- Restricting the DMA interface
- Isolate DMA-able memory from other regions
  - We can currently allocate memory in low memory regions used for kernel and page tables
- Introduce per-process memory limits
- All mitigations that prevent bit-flips are effective

Summary

Summary

- First effort to show that Rowhammer is possible on a platform other than x86
- Implemented a deterministic Rowhammer attack that grants root privileges using DMA and Phys Feng Shui
  - Even without using special OS features
  - Shown by implementing it on ARM/Android
- Many devices are vulnerable
  - If there are bit-flips, the device is vulnerable

Strengths

Strengths

- Novel and elegant solution to exploiting Rowhammer
- Does not rely on special OS features
- It is hard to mitigate if Rowhammer is possible on the device
- Well structured paper
- Most of it is well explained

Weaknesses

Weaknesses

- Assumes that bit-flips are always reproducible
- Not well tested on ARMv8
- Not tested outside of Android
- Some parts are not good explained
- Paper proposed some mitigation options which are not useful (Flikker[8], RAPID[9])

[8] S. Liu, et al. Flikker: Saving DRAM Refresh-power through Critical Data Partitioning, ASPLOS ‘11

[9] R. K. Venkatesan, et al. Retention-aware placement in DRAM (RAPID), HPCA ‘06

Related Work

Related Work

- ARMageddon [10]
  - Demonstrated cache eviction on ARM
- DRAMA [11]
  - Demonstrated that reverse engineering can reduce search time for Rowhammer bit flips
- Android ION Hazard: the Curse of Customizable Memory Management [12]
  - Shows security flaws of Android ION memory allocator

[10] M. Lipp, et al. ARMageddon: Cache Attacks on Mobile Devices, USENIX '16
[11] P. Pessl, et al. DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks, USENIX '16
[12] H. Zhang, et al. Android ION Hazard: the Curse of Customizable Memory Management System, CCS '16

Takeaways

Takeaways

- Prior x86 Rowhammer exploitation methods cannot be used on ARM/Android
- ARM Memory controllers are fast enough to do Rowhammer
- Drammer is a novel deterministic method to exploit the Rowhammer bug
- Bypasses defenses like ANVIL using DMA
- No easy software fix
- Simple and effective

Open Discussion

Open Discussion

- Thoughts on the previous ideas to exploit Rowhammer?
- Will the problem stay relevant even with recent efforts of mitigation?
- Can you think of any additional mitigation for this attack?
- Could you think of other applications for this attack?

Additional Thoughts

- Can this be applied to iOS?
  - If we can use DMA from userspace, probably
- It could be used to root your Android Phone with a simple app for your own use
  - No bootloader unlocking needed

More Questions or Suggestions?

https://moodle-app2.let.ethz.ch/mod/forum/discuss.php?d=38986