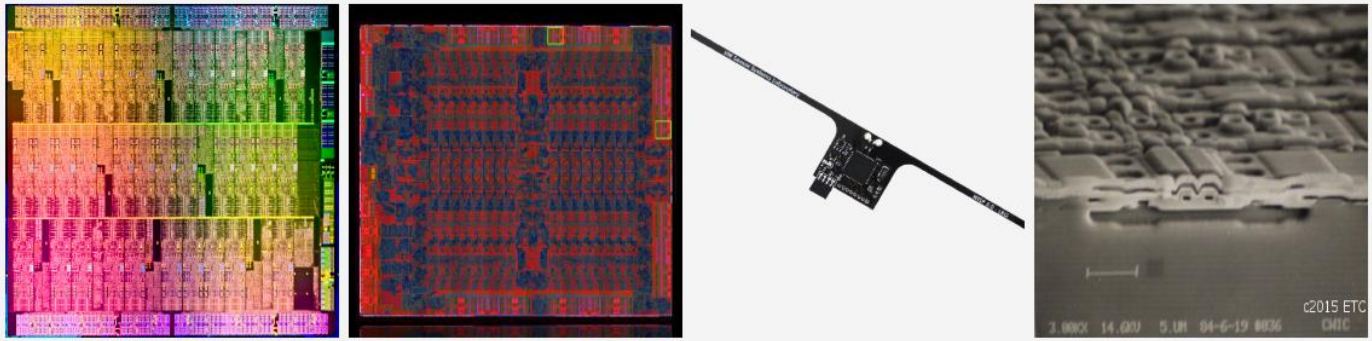
18-344: Computer Systems and the Hardware-Software Interface

Fall 2025



Course Description

Lecture 22: Row Hammer

This course covers the design and implementation of computer systems from the perspective of the hardware software interface. The purpose of this course is for students to understand the relationship between the operating system, software, and computer architecture. Students that complete the course will have learned operating system fundamentals, computer architecture fundamentals, compilation to hardware abstractions, and how software actually executes from the perspective of the hardware software/boundary. The course will focus especially on understanding the relationships between software and hardware, and how those relationships influence the design of a computer system's software and hardware. The course will convey these topics through a series of practical, implementation-oriented lab assignments.

Introduction to Row Hammering …with DRAM Refresher

Credit:

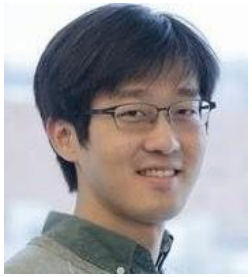
This portion of the lecture is an abridged version of the following lecture:

Prof. Phil Gibbons
18-742: Computer Architecture & Systems
Spring 2025, Lecture 19

"Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors"

Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, Onur Mutlu 2014

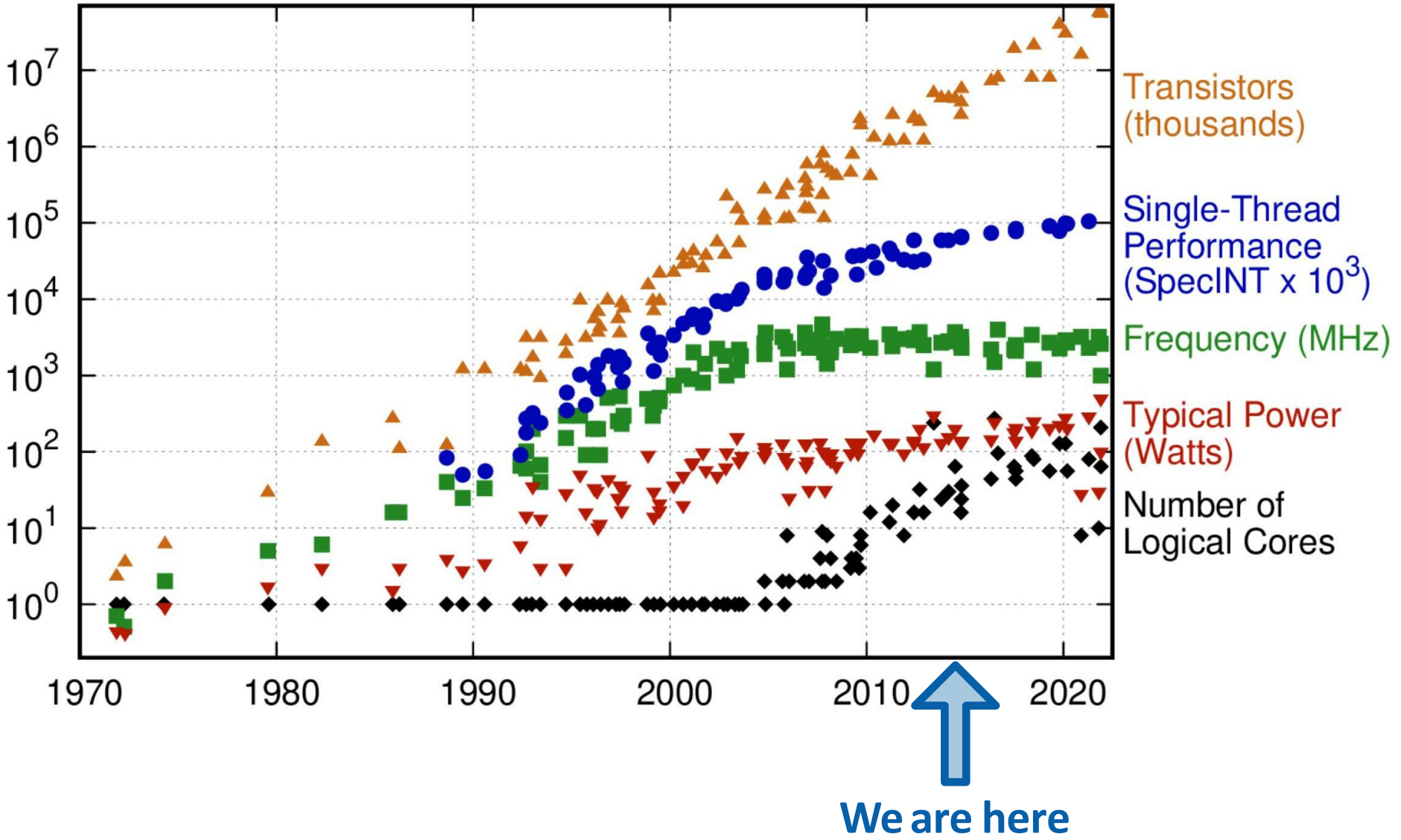
- **Yoongu:** CMU PhD, now Google
- **Ross:** CMU student, now Stanford PhD
- **Jeremie:** CMU MS, now CMU PhD
- **Chris F:** CMU PhD, now Fastly
- **Ji-Hye:** CMU student, now ??
- **Donghyuk:** CMU PhD, now Nvidia
- **Chris W:** Intel Principal Eng., CMU MS
- **Konrad:** ex-Intel
- **Onur:** CMU prof, now ETH
  - Young Architect Award, Maurice Wilkes Award, ACM/IEEE Fellow







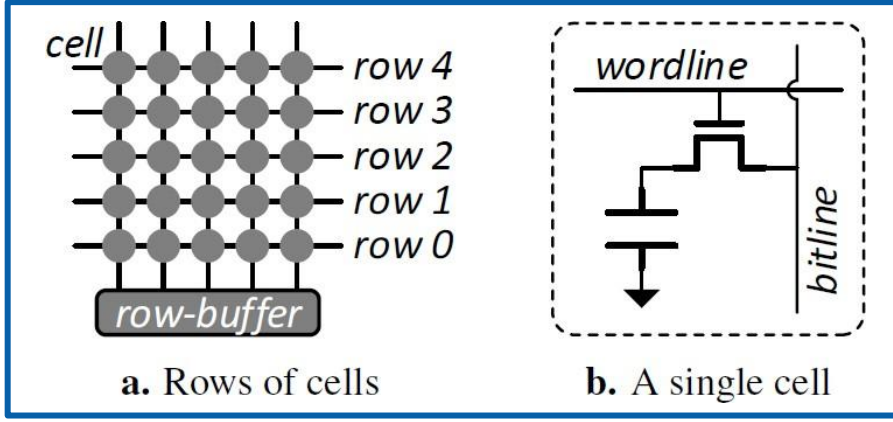

50 Years of Microprocessor Trend Data

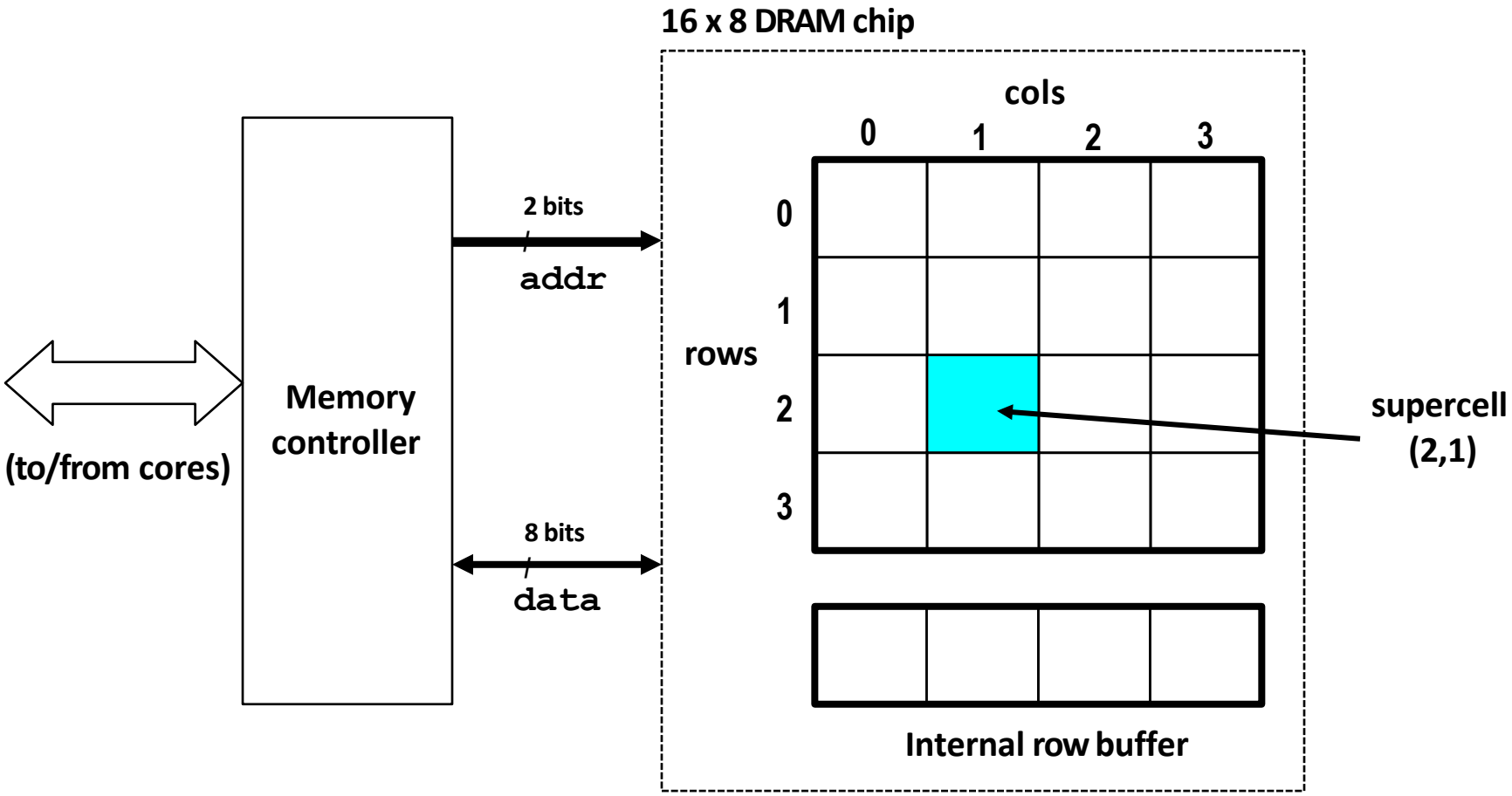


DRAM Organization

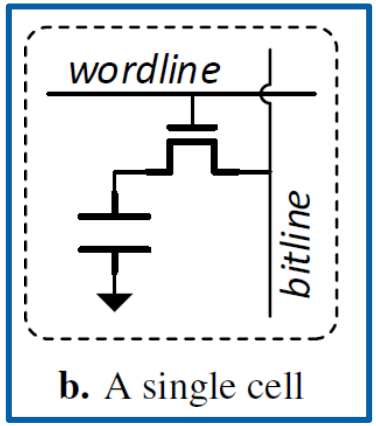
d x *w* DRAM:

- $d \cdot w$ total bits organized as d supercells of size w bits



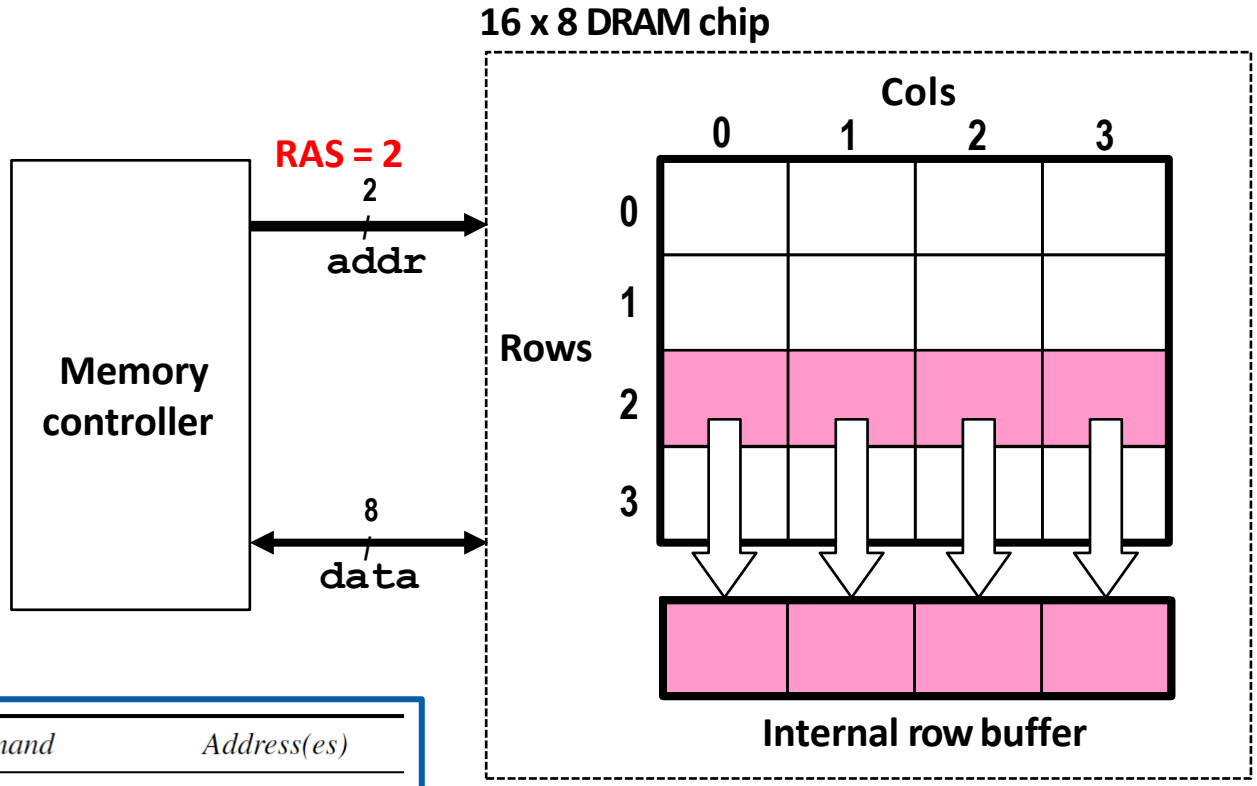


Reading DRAM Supercell (2,1)



Step 1(a): Row access strobe (RAS) selects row 2.

Step 1(b): Raising wordline causes Row 2 to be copied from DRAM array to row buffer.

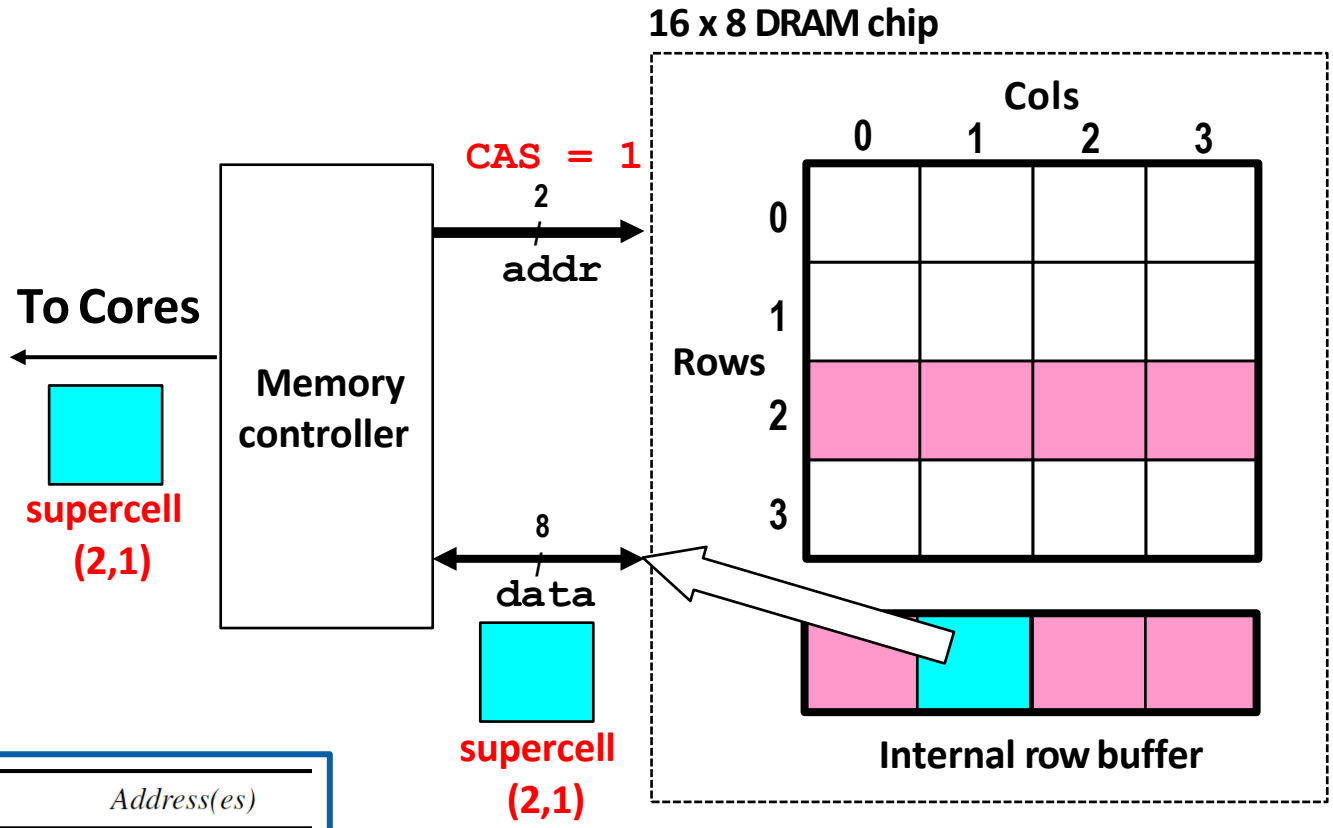


Operation	*Command*	*Address(es)*
1. Open Row	`ACTIVATE (ACT)`	Bank, Row
2. Read/Write Column	`READ/WRITE`	Bank, Column
3. Close Row	`PRECHARGE (PRE)`	Bank
Refresh (Section 2.4)	`REFRESH (REF)`	—

Reading DRAM Supercell (2,1)

Step 2(a): Column access strobe (CAS) selects column 1.

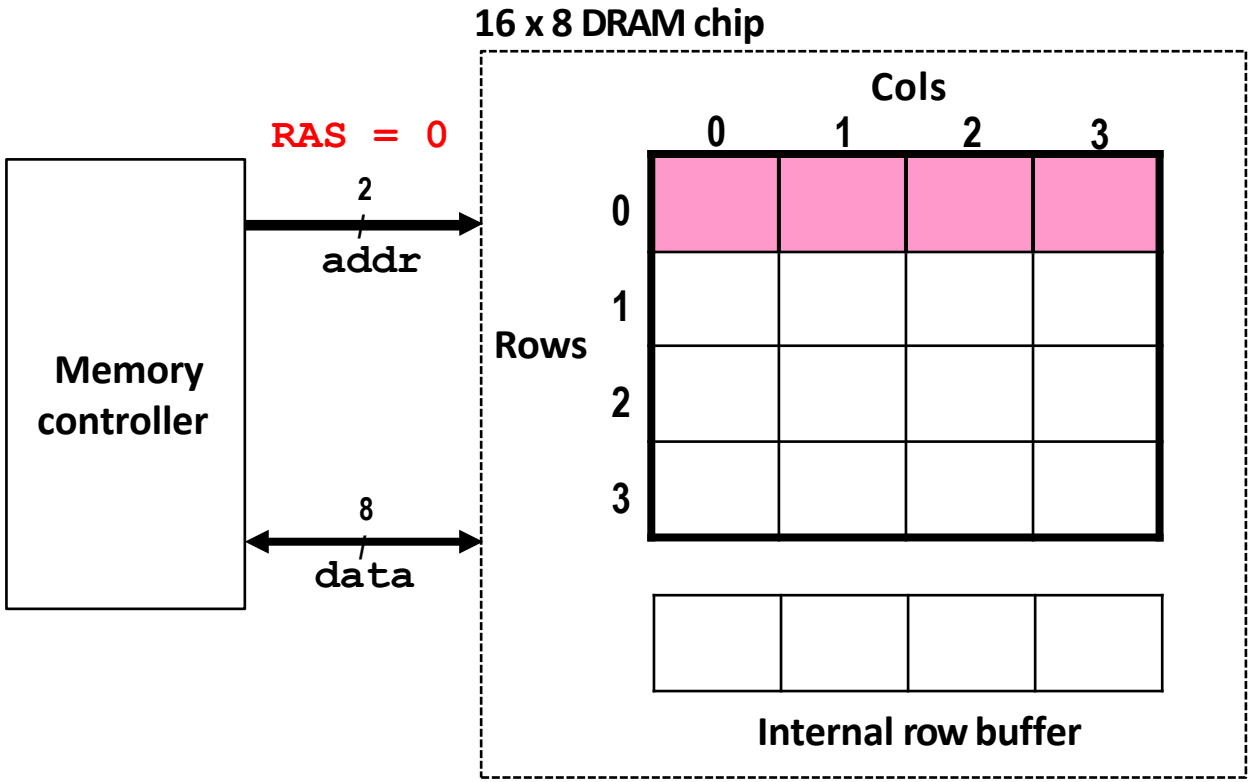
Step 2(b): Supercell (2,1) copied from buffer to data lines, and back to CPU.



Operation	*Command*	*Address(es)*
1. Open Row	`ACTIVATE (ACT)`	Bank, Row
2. Read/Write Column	`READ/WRITE`	Bank, Column
3. Close Row	`PRECHARGE (PRE)`	Bank
Refresh (Section 2.4)	`REFRESH (REF)`	—

Reading DRAM Supercell (2,1)

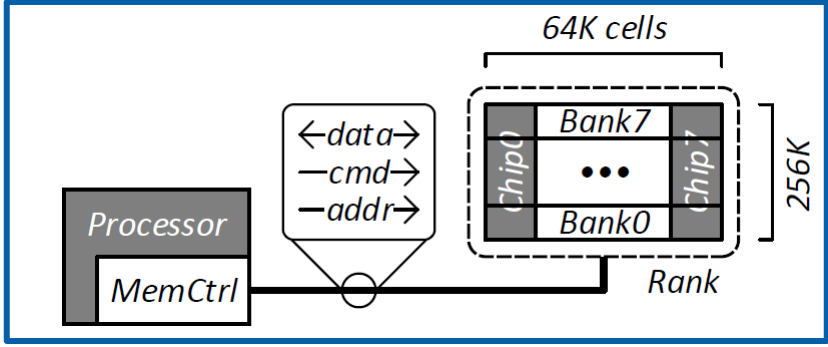
Step 3: To serve a different RAS, first lower wordline to close row 2 and empty row buffer.

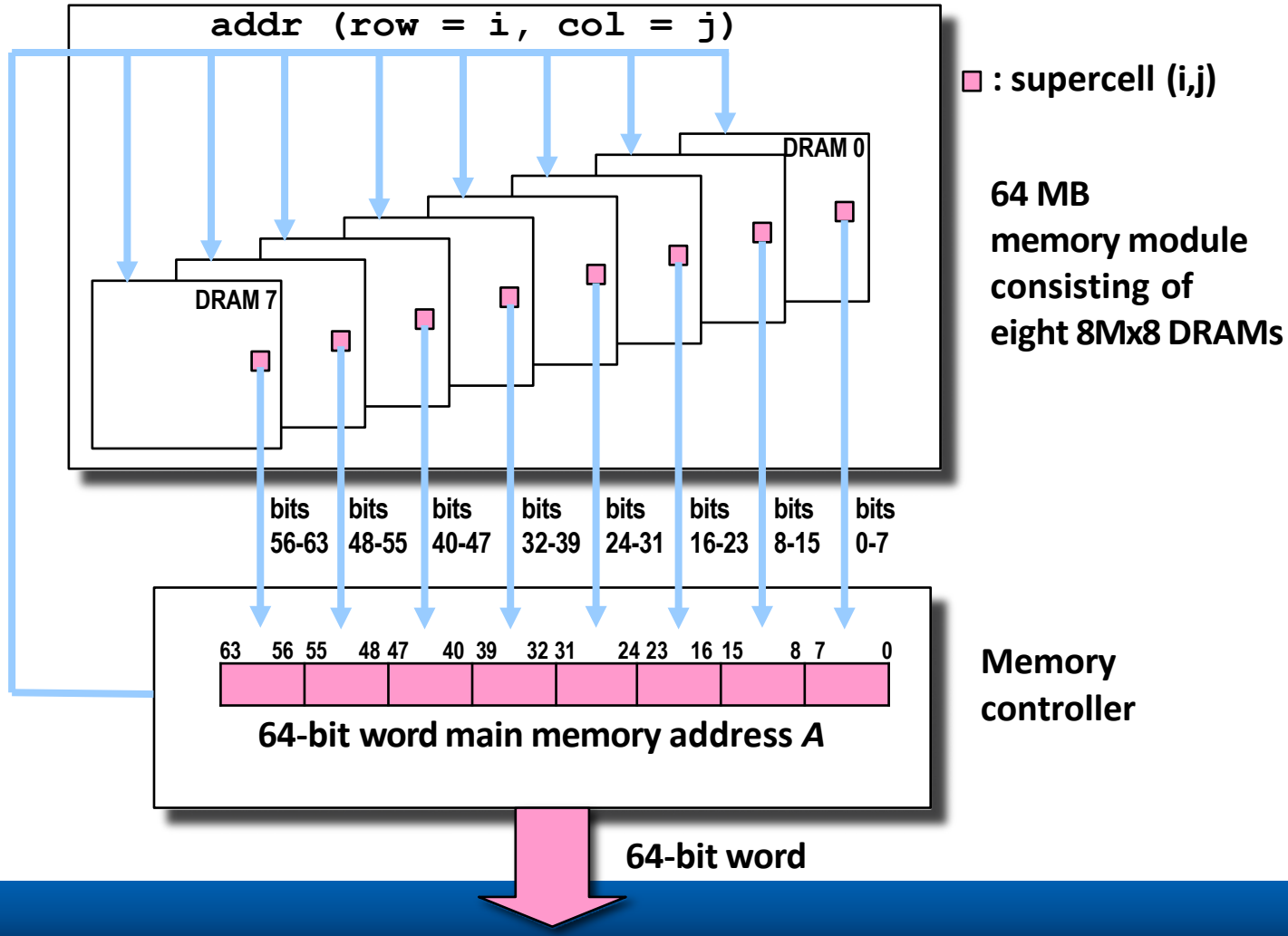


Operation	*Command*	*Address(es)*
1. Open Row	ACTIVATE (ACT)	Bank, Row
2. Read/Write Column	READ/WRITE	Bank, Column
3. Close Row	PRECHARGE (PRE)	Bank
Refresh (Section 2.4)	REFRESH (REF)	—

t_{RRR} ~ 50 nanosecs between reopening same row

DRAM Rank





: supercell (i,j)

64 MB
memory module
consisting of
eight 8Mx8 DRAMs

Memory
controller

Shrinking DRAM Process Technology

- **Benefits**
  - Reduces cost-per-bit
  - Desired capacity w/fewer DIMMs: smaller form factor
- **Reliability challenges**
  - Holds limited charge: reduces noise margin, vulnerable to loss
  - Proximity: electromagnetic coupling effects
  - High variation: more cells susceptible to inter-cell crosstalk

Key Findings

- **Disturbance errors are widespread in commodity DRAM**
  - Disturbable cells exist in 110 of 129 tested modules (all of 2012-13)
  - Intel filed Row Hammer patents in 2014 (paper was under review)
- **Simple user-level programs can induce such errors**
- **Root cause is the repeated toggling of a row's wordline**
  - Voltage fluctuation causes nearby rows to rapidly lose charge
  - As few as 139K times suffice
- **Up to 1 in 1.7K cells is disturbable**
- **Propose probabilistic adjacent row refresh as mitigation**

Assembly Code on Intel/AMD Machines

```
1 code1a:
2   mov (X), %eax
3   mov (Y), %ebx
4   clflush (X)
5   clflush (Y)
6   mfence
7   jmp code1a
```

a. Induces errors

```
1 code1b:
2   mov (X), %eax
3   clflush (X)
4
5
6   mfence
7   jmp code1b
```

b. Does not induce errors

Bit-Flip	Sandy Bridge	Ivy Bridge	Haswell	Piledriver
'0' → '1'	7,992	10,273	11,404	47
'1' → '0'	8,125	10,449	11,467	12

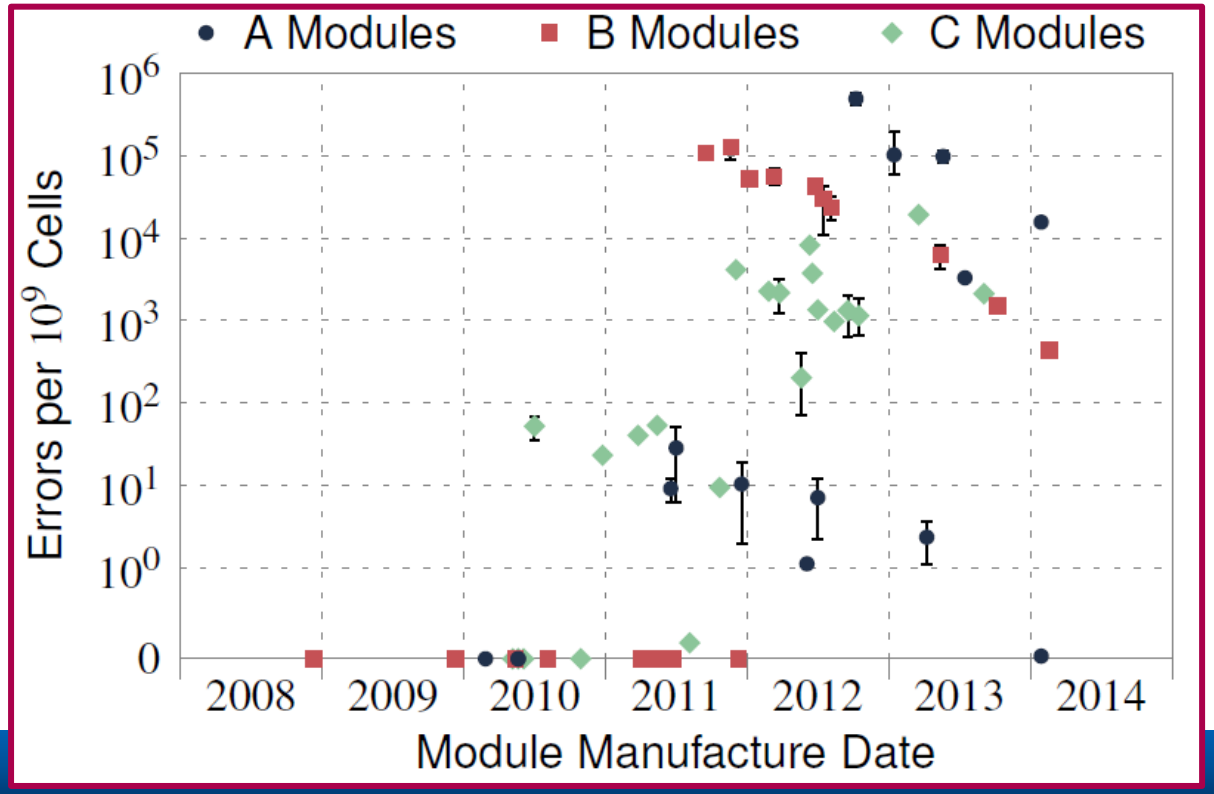
Errors vs. Manufacturing Date

```
1  TestBulk(AI, RI, DP)
2    setAI(AI)
3    setRI(RI)
4    N ← (2 × RI)/AI
5
6    writeAll(DP)
7    for r ← 0 ··· ROW_MAX
8      for i ← 0 ··· N
9        ACT r^th row
10       READ 0^th col.
11       PRE r^th row
12   readAll()
13   findErrors()
```

a. Test all rows at once

Access Pattern	Disturbance Errors?
1. $(open–read–close)^N$	**Yes**
2. $(open–write–close)^N$	**Yes**
3. $open–read^N–close$	No
4. $open–write^N–close$	No

FPGA experiments to raw DRAM
(Activation Interval, Refresh Interval, Data Pattern)



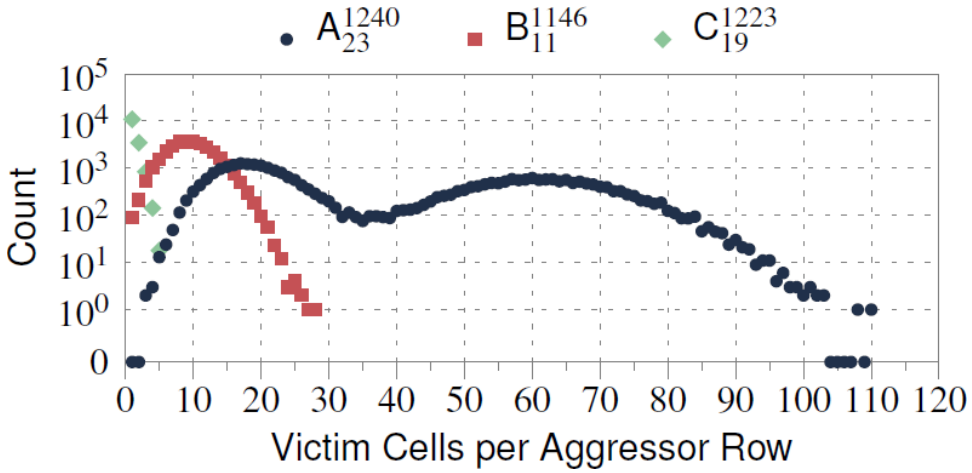
Uncorrectable Multi-Bit Errors

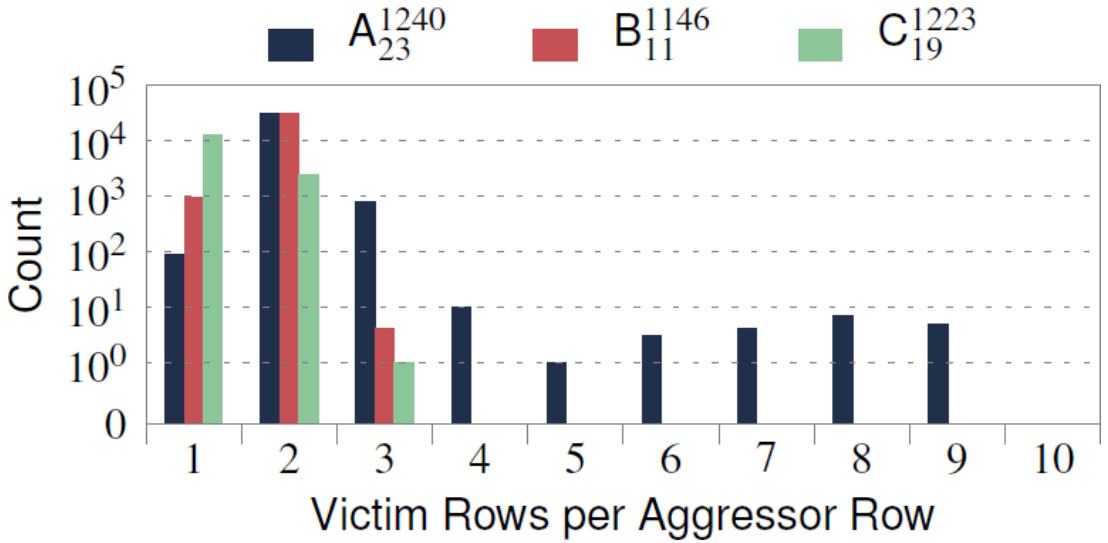
For single error-correction codes on 64-bit words

Module	*Number of 64-bit words with X errors*			
	$X = 1$	$X = 2$	$X = 3$	$X = 4$
A_{23}	9,709,721	**181,856**	**2,248**	**18**
B_{11}	2,632,280	**13,638**	**47**	0
C_{19}	141,821	**42**	0	0

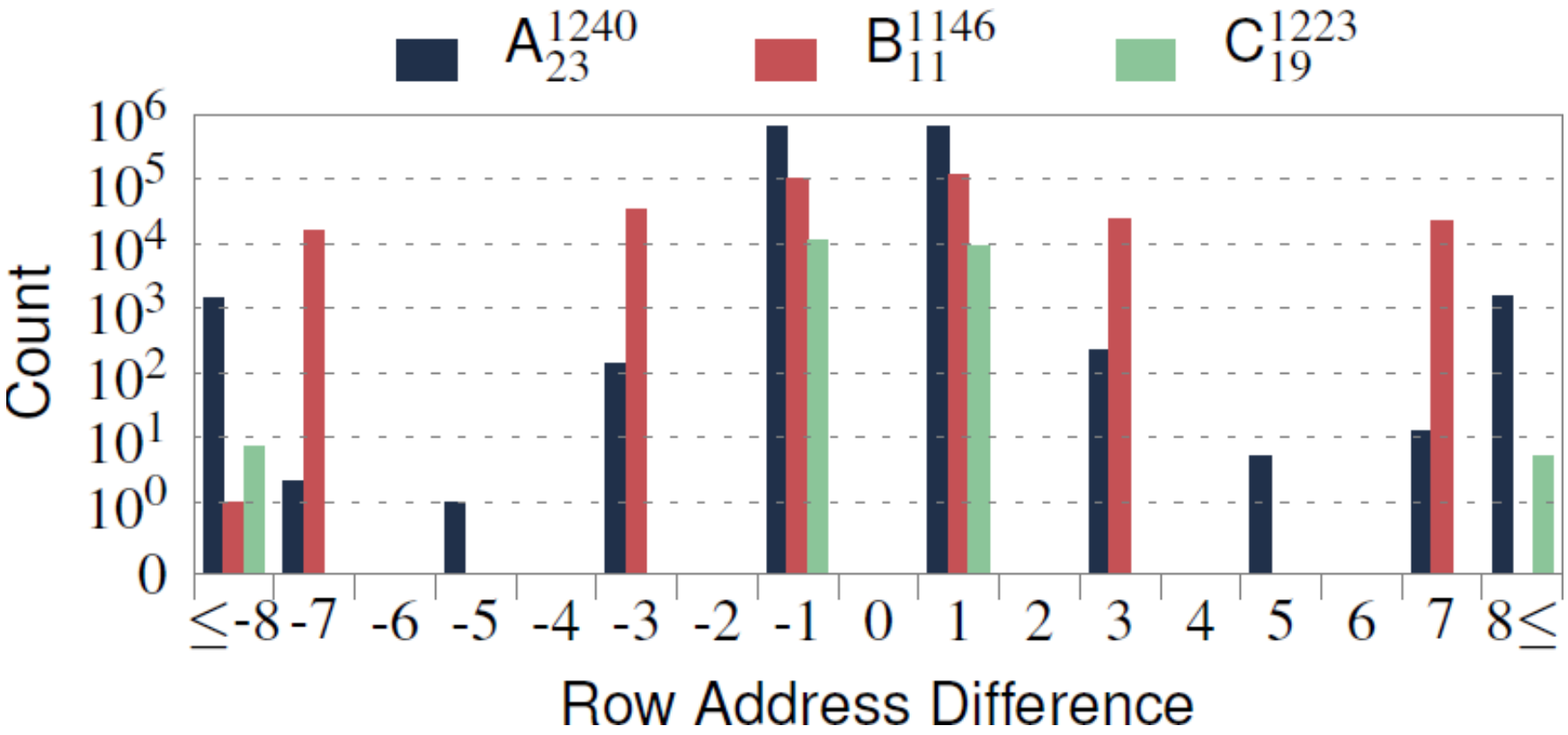
Table 5. Uncorrectable multi-bit errors (in bold)

Victim Cells/Rows per Aggressor Row





Which Rows Affected by Aggressor Row



Causes for ^{-1,1}: Manufacturer-dependent mappings, remappings

Num Errors for Different Data Patterns

Module	TestBulk(*DP*) + TestBulk(~*DP*)			
	Solid	RowStripe	ColStripe	Checkered
A_{23}	112,123	**1,318,603**	763,763	934,536
B_{11}	12,050	**320,095**	9,610	302,306
C_{19}	57	20,770	130	**29,283**

Error is always in discharge direction,
which can be 1->0 (for true-cells) or 0->1 (for anti-cells).

DP matters in some complicated way.

Sensitivity Results

- Error are mostly repeatable
- Victim cells ≠ Weak (leakier) cells
- Not strongly effected by temperature

Possible Row Hammer Mitigations

- Make better chips
- Correct errors
- Refresh all rows frequently
- Retire cells (manufacturer)
- Retire cells (end-user)
- Identify “hot” rows & refresh neighbors
- PARA – next slide

PARA: Probabilistic Adjacent Row Refresh

Duration	N_{th}=50K	N_{th}=100K	N_{th}=200K
64*ms*	1.4×10^{-11}	1.9×10^{-22}	3.6×10^{-44}
1 *year*	6.8×10^{-3}	9.4×10^{-14}	1.8×10^{-35}

Table 7. Error probabilities for PARA when p=0.001

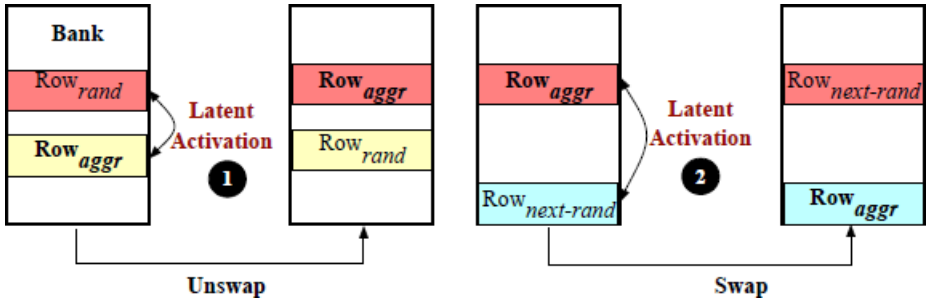
Requires manufacturers to expose their mappings & remappings

Higher & higher *p* to keep up with lower NN_{th} =>
Frequent refreshes are an attack on their neighbors!

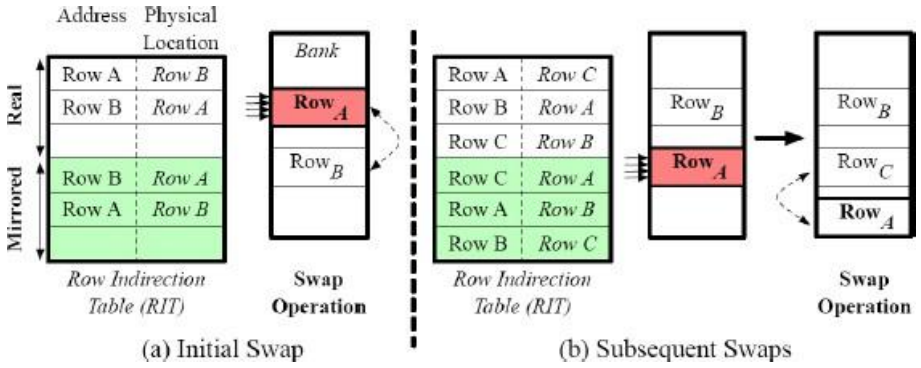
"Scalable and Secure Row-Swap: Efficient and Safe Row Hammer Mitigation in Memory Systems"

Jeonghyun Woo, Gururaj Saileshwar, Prashant J. Nair 2023

Randomized Row-Swap (prior SOTA) is not secure



Instead, don't unswap

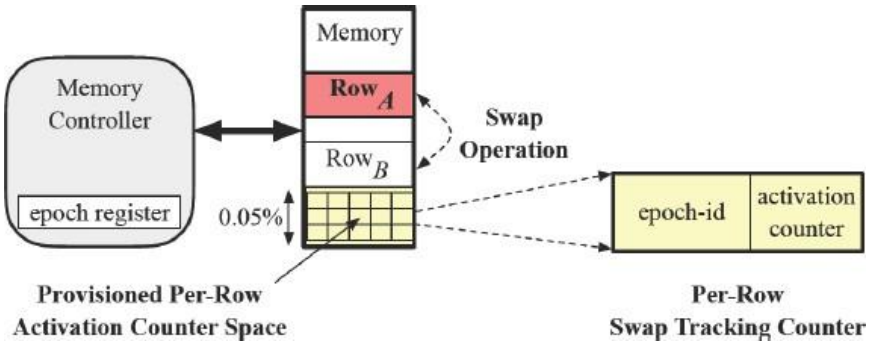


Then do periodic lazy eviction of RIT

"Scalable and Secure Row-Swap: Efficient and Safe Row Hammer Mitigation in Memory Systems"

Jeonghyun Woo, Gururaj Saileshwar, Prashant J. Nair 2023

Finally, track swap counts



When too much swapping, pin row in LLC for refresh epoch

All accesses hit in LLC and don't go to memory

Diving Deeper:

Drammer: Deterministic Rowhammer Attacks on Mobile Platforms

Credit:

This portion of the lecture is an abridged version of the original presentation

Drammer: Deterministic Rowhammer Attacks on Mobile Platforms

Victor van der Veen[1], Yanick Fratantonio[2], Martina Lindorfer[2], Daniel Gruss[3], Clémentine Maurice[3], Giovanni Vigna[2], Herbert Bos[1], Kaveh Razavi[1], and Cristiano Giuffrida[1]

[1]Vrije Universiteit Amsterdam, [2]UC Santa Barbara, [3]TU Graz

Drammer: Deterministic Rowhammer Attacks on Mobile Platforms

Victor van der Veen[1], *Yanick Fratantonio*[2], *Martina Lindorfer*[2], *Daniel Gruss*[3], *Clémentine Maurice*[3], *Giovanni Vigna*[2], *Herbert Bos*[1], *Kaveh Razavi*[1], *and Cristiano Giuffrida*[1]

[1]Vrije Universiteit Amsterdam, [2]UC Santa Barbara, [3]TU Graz

Overview

1. ## Memory Templating

   Scan memory for useful bit flips

2. ## Land sensitive data

   Store a crucial data structure on a vulnerable page

3. ## Reproduce the bit flip

   Modify the data structure and get root acces

Templating

Uncached memory access

- **`clflush`**
- cache eviction
- non-temporal access instructions

Determining the physical addresses aggressor/victim rows

- **`/proc/self/pagemap`**
- 2MB huge pages (relative)

But does it work on ARM?

None of them

Templating on ARM

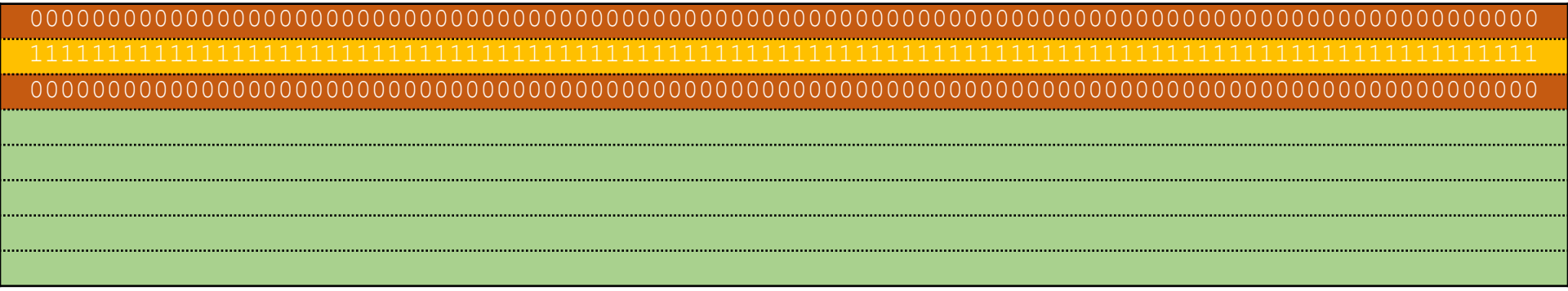
DMA

Direct Memory Access

Android's DMA memory allocator provides everything we need:

- Uncached memory (no **`clflush`** required)
- Physically contiguous memory

Physical memory:



Templating on ARM

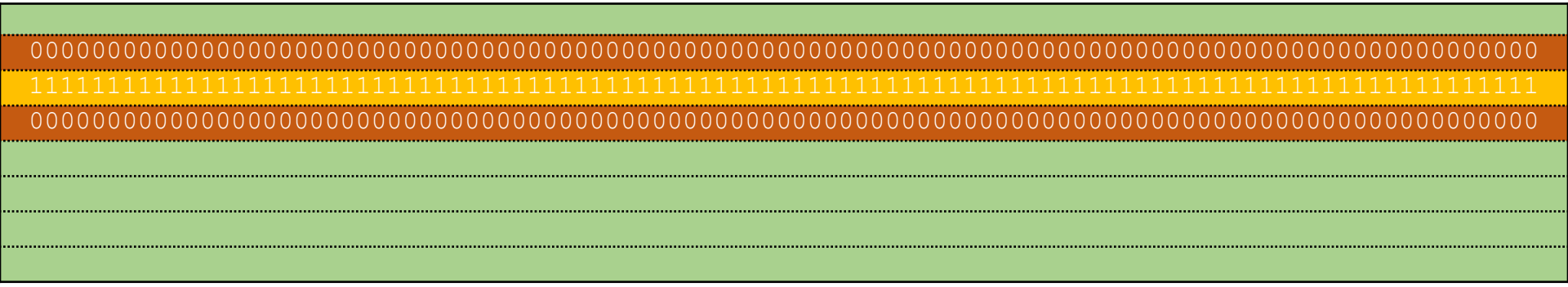
DMA

Direct Memory Access

Android's DMA memory allocator provides everything we need:

- Uncached memory (no **`clflush`** required)
- Physically contiguous memory

Physical memory:



Templating on ARM

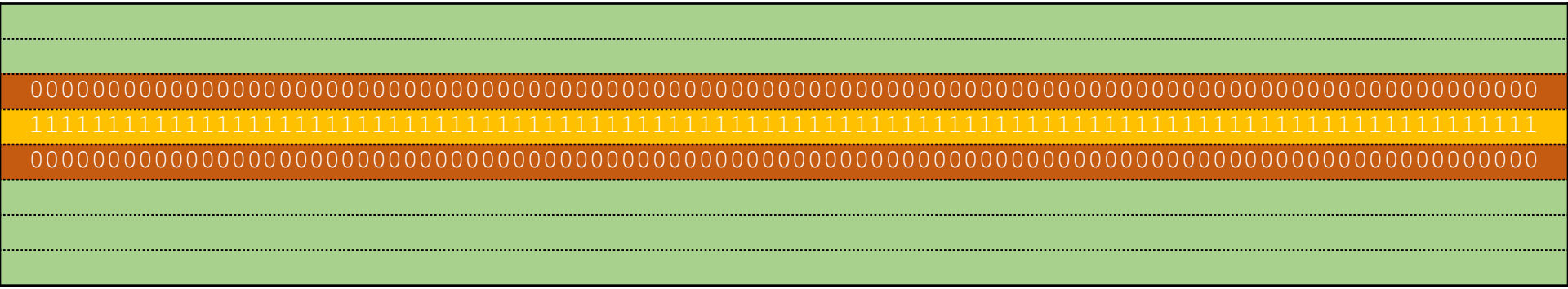
DMA

Direct Memory Access

Android's DMA memory allocator provides everything we need:

- Uncached memory (no **`clflush`** required)
- Physically contiguous memory

Physical memory:



Templating on ARM

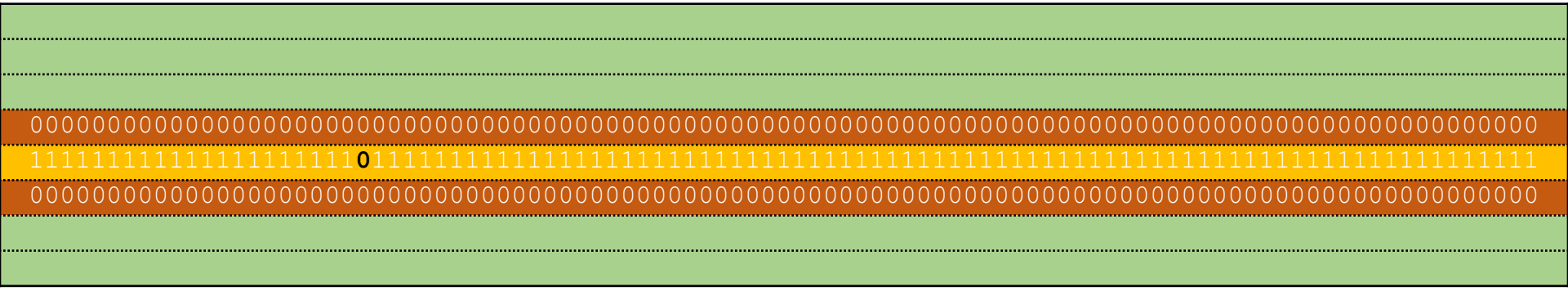
DMA

Direct Memory Access

Android's DMA memory allocator provides everything we need:

- Uncached memory (no **`clflush`** required)
- Physically contiguous memory

Physical memory:



Templating on ARM

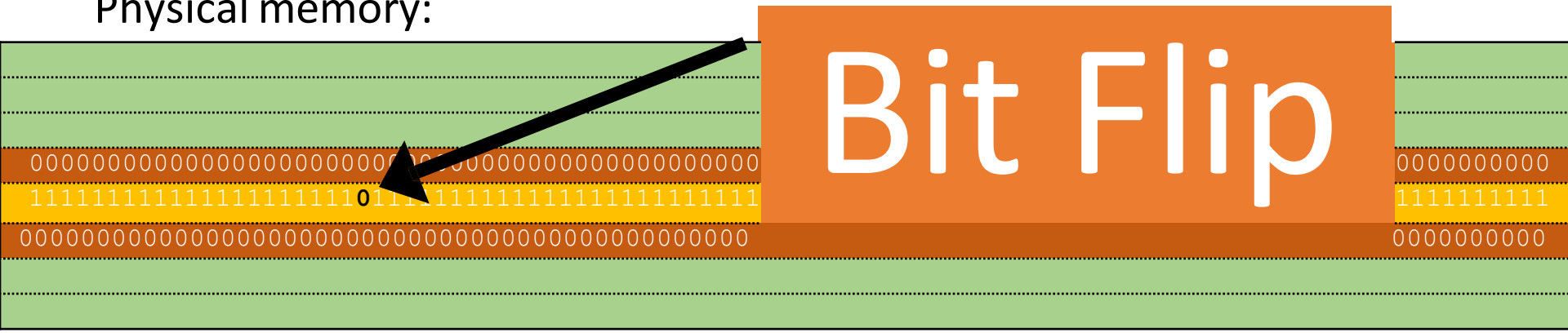
DMA

Direct Memory Access

Android's DMA memory allocator provides everything we need:

- Uncached memory (no **`clflush`** required)
- Physically contiguous memory

Physical memory:



Overview

1. Memory Templating

   Scan memory for useful bit flips

2. **Land sensitive data**

   Store a crucial data structure on a vulnerable page

Overview

1. Memory Templating

   Scan memory for useful bit flips

2. Land a Page Table

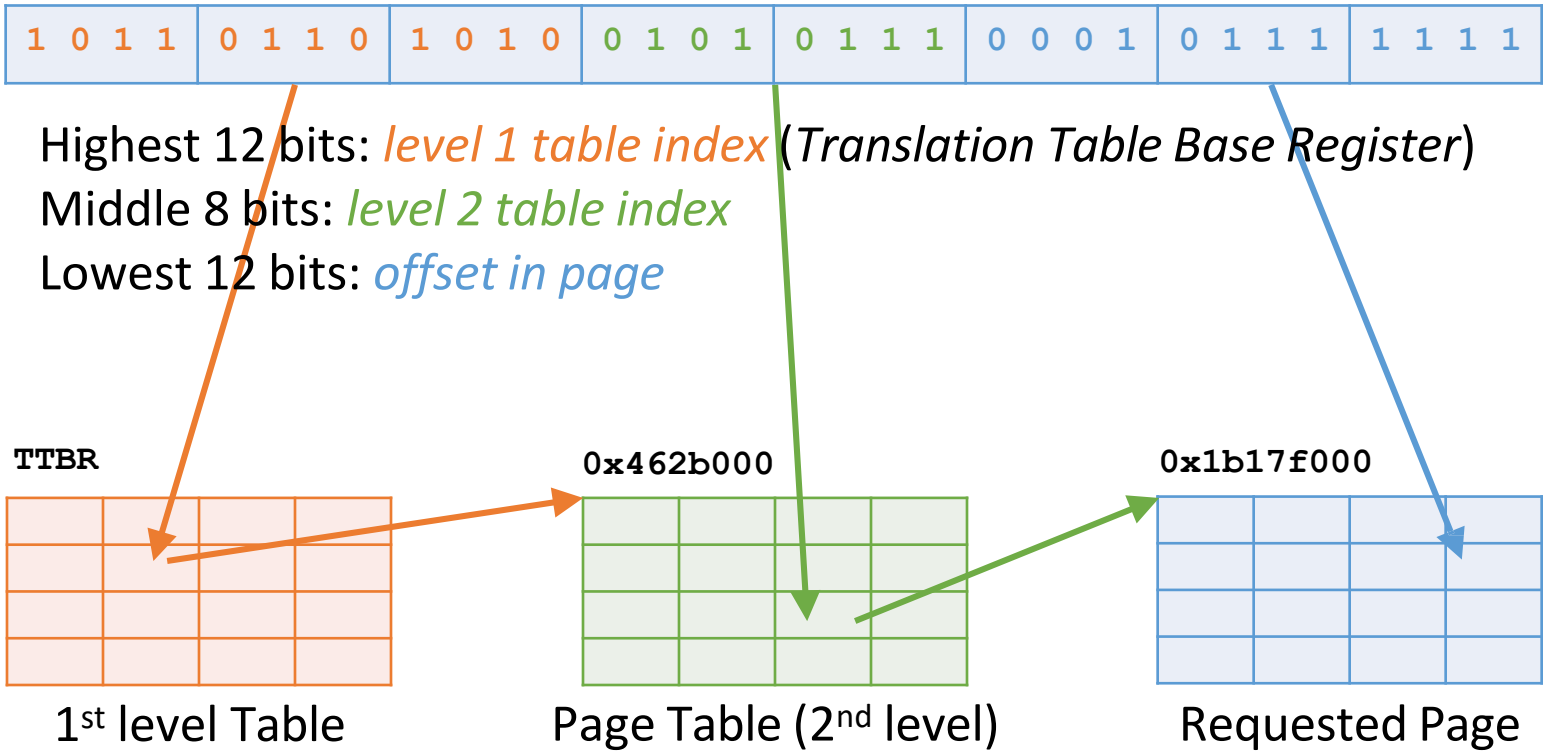
   Store a page table on a vulnerable page

But why?

Page Tables

Mapping virtual addresses to physical addresses

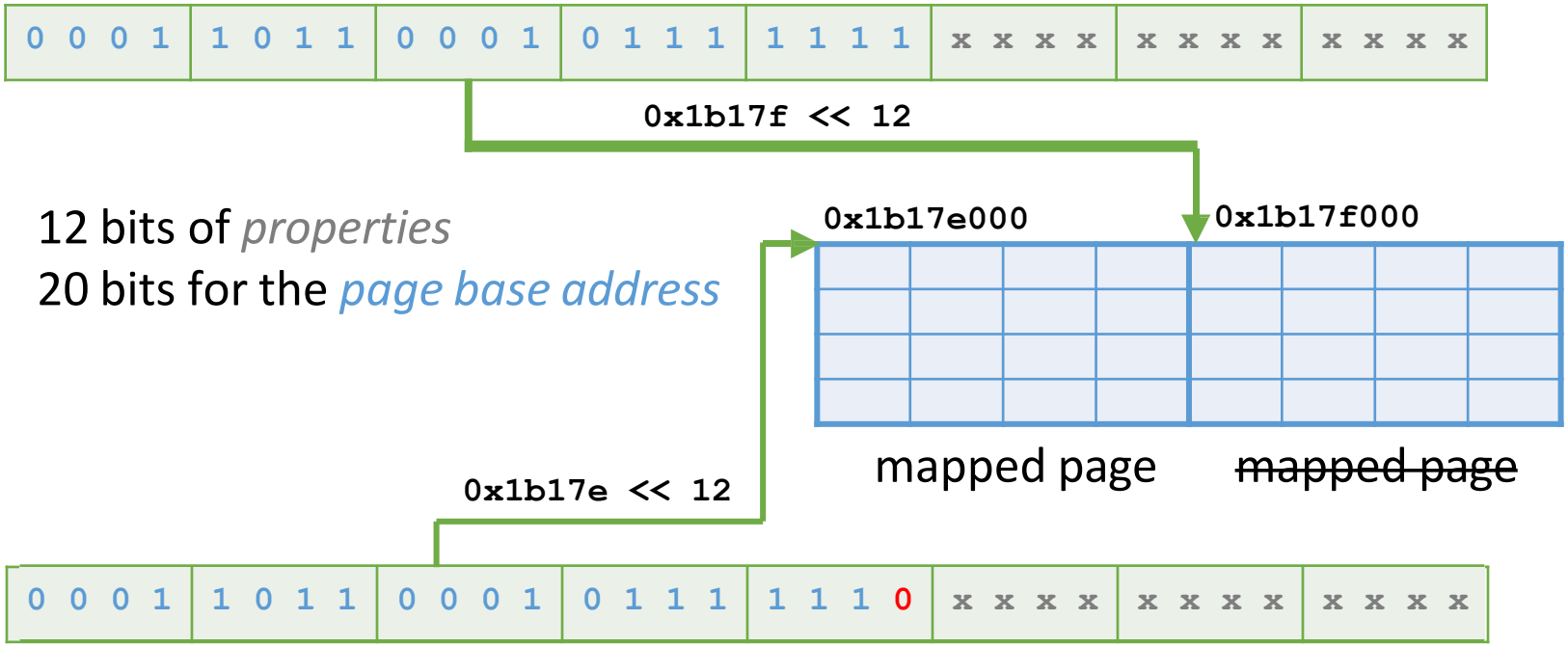
Example lookup for input virtual address `0xb6a5717f`



- Highest 12 bits: *level 1 table index* (*Translation Table Base Register*)
- Middle 8 bits: *level 2 table index*
- Lowest 12 bits: *offset in page*

Rowhammer Attacks on
Page Table Entries

Entry in the (2nd level) Page Table

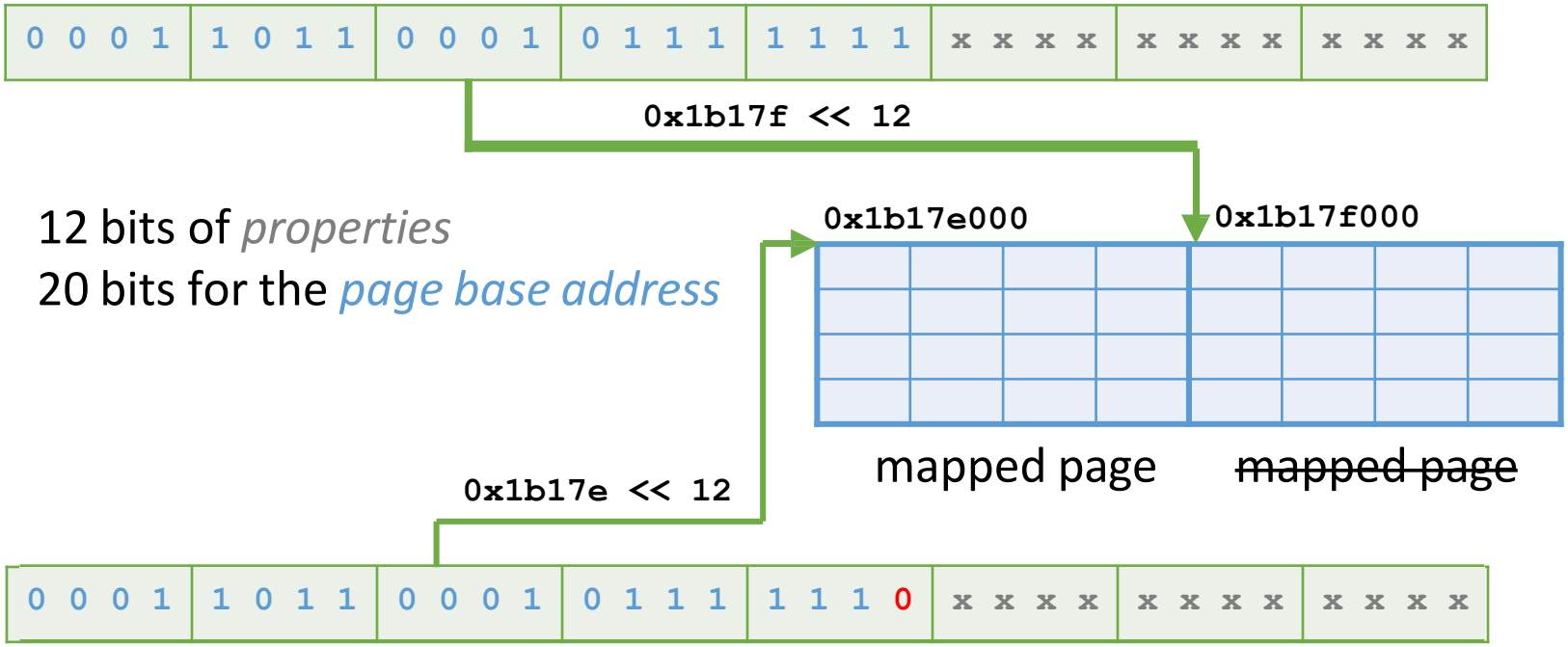


- 12 bits of *properties*
- 20 bits for the *page base address*

What if we flip a bit in the entry?

Rowhammer Attacks on
Page Table Entries

Entry in the (2nd level) Page Table



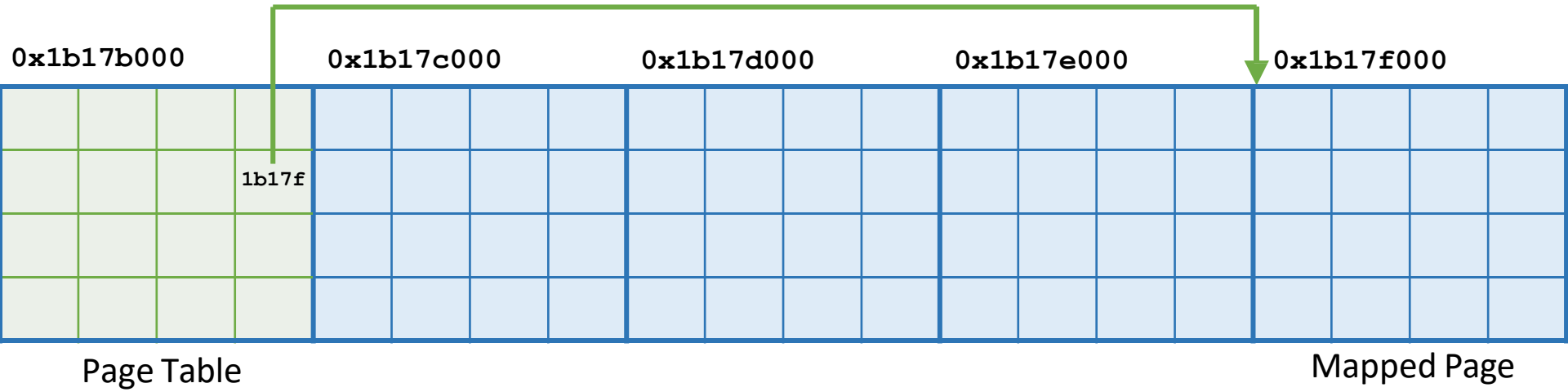
- 12 bits of *properties*
- 20 bits for the *page base address*

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

Deterministic Attacks on
Page Table Entries

1. Map a page 4 pages ‘away’ from its page table



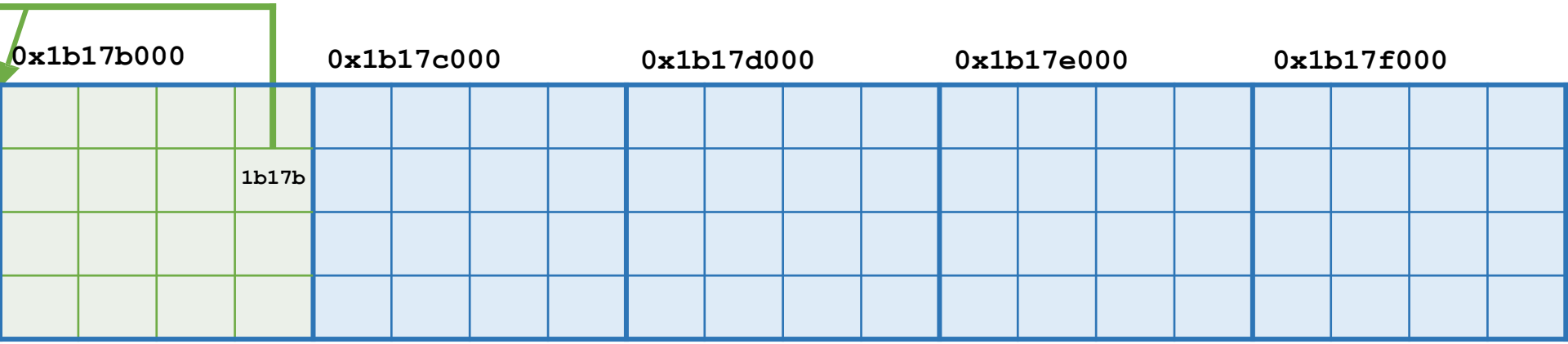
Virtual address `0xb6a57000` maps to Page Table Entry:

0 0 0 1	1 0 1 1	0 0 0 1	0 1 1 1	1 1 1 1	x x x x	x x x x	x x x x

which translates to physical page **`0x1b17f000`**

Deterministic Attacks on
Page Table Entries

1. Map a page 4 pages ‘away’ from its page table
2. Flip bit 4 in the page table entry



Mapped Page Table

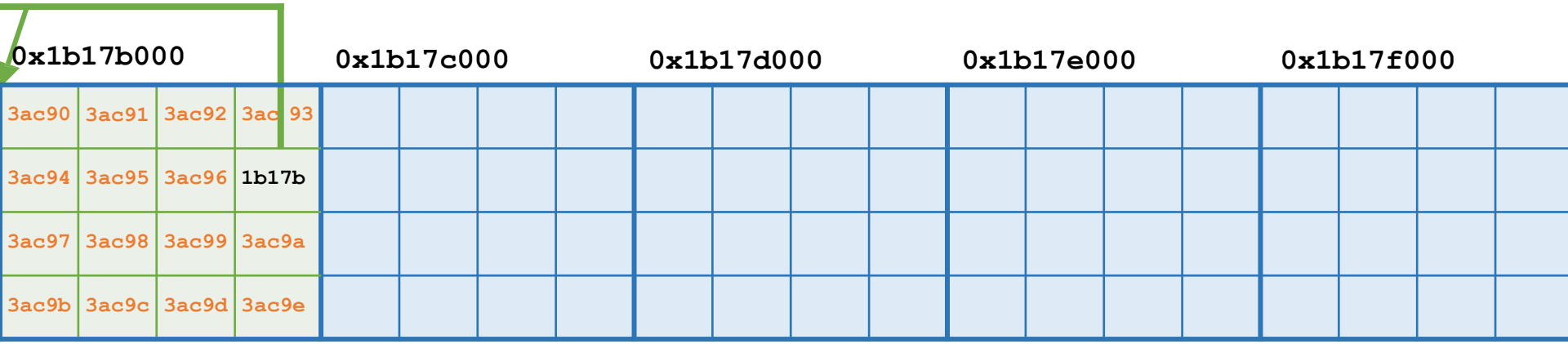
Virtual address `0xb6a57000` maps to Page Table Entry:

0 0 0 1	1 0 1 1	0 0 0 1	0 1 1 1	1 0 1 1	x x x x	x x x x	x x x x

which translates to physical page **0x1b17b000**

Deterministic Attacks on Page Table Entries

1. Map a page 4 pages ‘away’ from its page table
2. Flip bit 4 in the page table entry
3. Write page table entries



Mapped Page Table

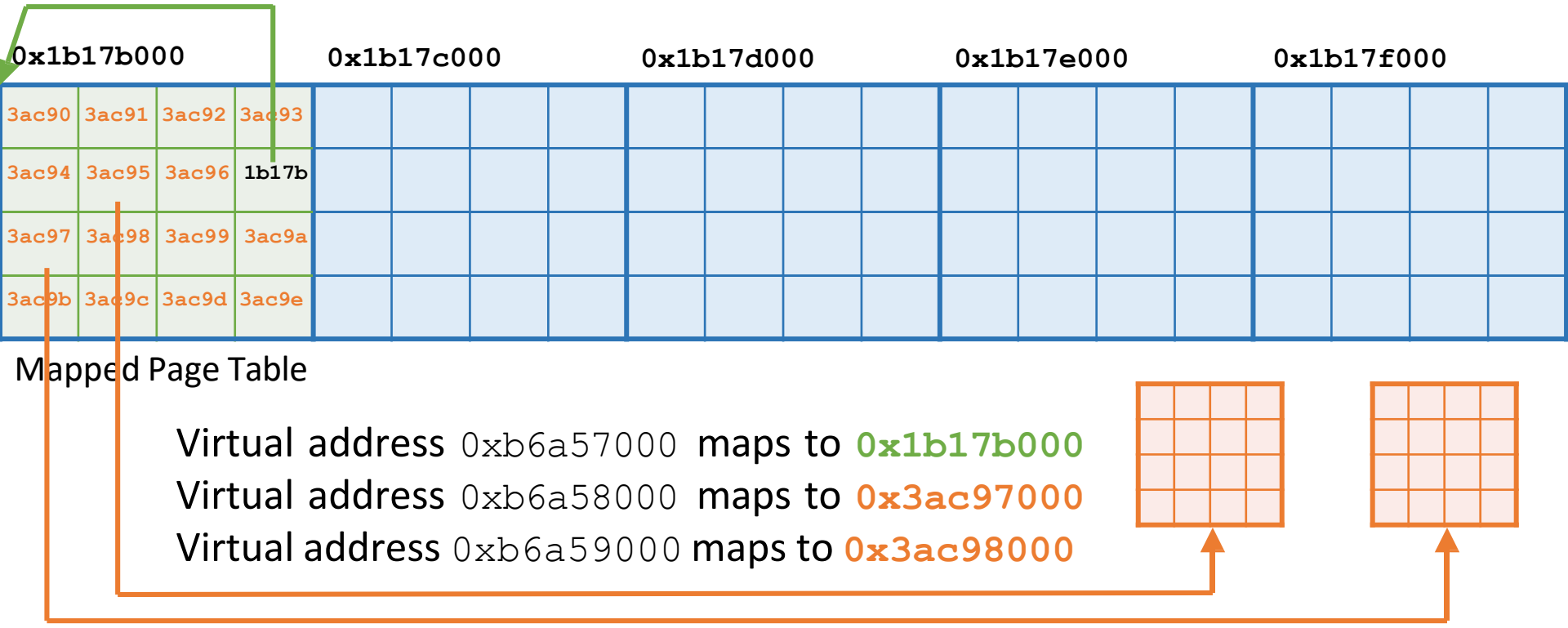
Virtual address `0xb6a57000` maps to Page Table Entry:

0 0 0 1	1 0 1 1	0 0 0 1	0 1 1 1	1 0 1 1	x x x x	x x x x	x x x x

which translates to physical page **0x1b17b000**

Deterministic Attacks on Page Table Entries

1. Map a page 4 pages ‘away’ from its page table
2. Flip bit 2 in the page table entry
3. Write page table entries
4. Read/write kernel memory



Overview

1. Memory Templating

   Scan memory for useful bit flips

2. Land a Page Table

   Store a page table on a vulnerable page

But how?

Landing a Page Table

- No access to `pagemap` (virtual – physical address mapping)
- No fancy memory management features (deduplication)

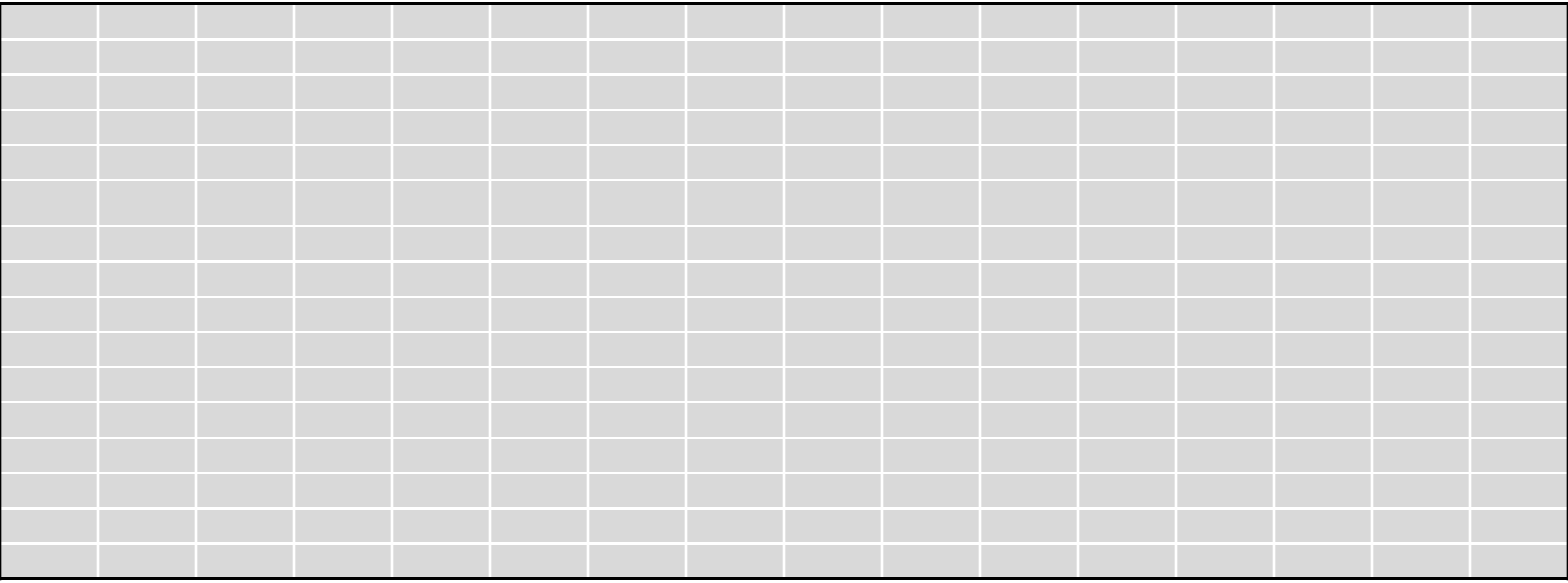
Phys Feng Shui

Landing a Page Table

- No access to `pagemap` (virtual – physical address mapping)
- No fancy memory management features (deduplication)

Phys Feng Shui

Physical memory:



Landing a Page Table

- No access to `pagemap` (virtual – physical address mapping)
- No fancy memory management features (deduplication)

Phys Feng Shui

Physical memory:

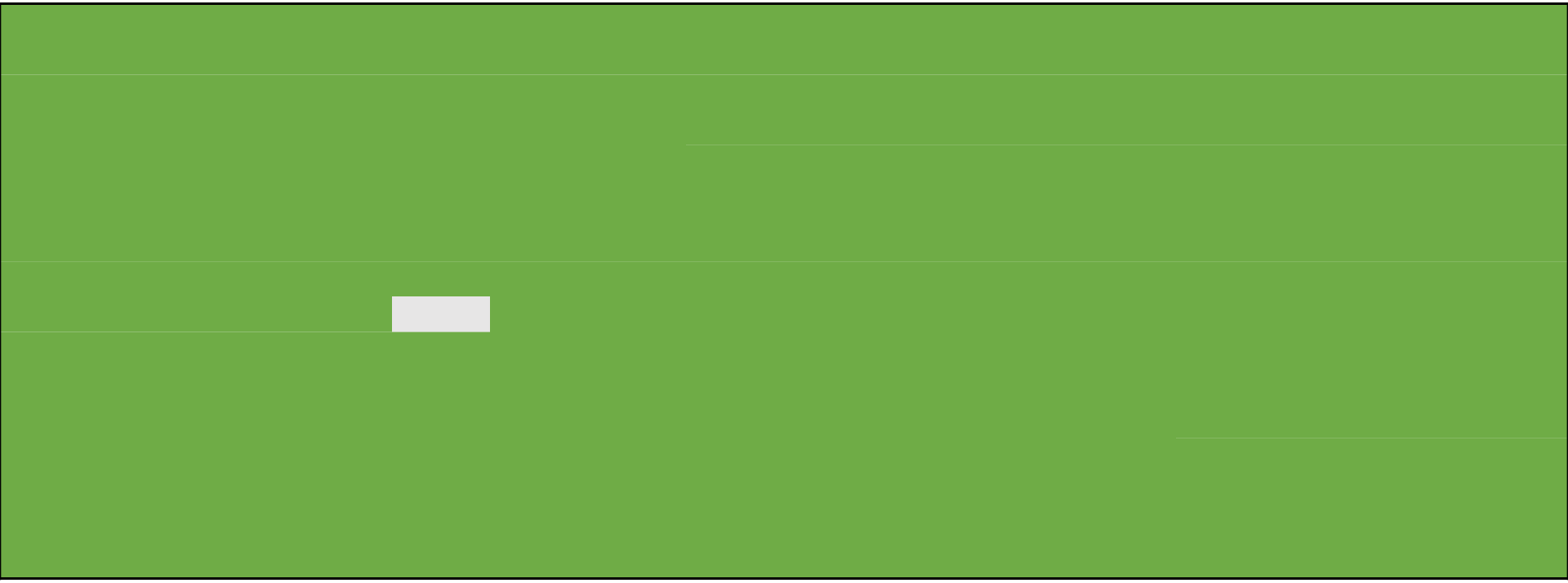
Exhaust all memory

Landing a Page Table

- No access to `pagemap` (virtual – physical address mapping)
- No fancy memory management features (deduplication)

Phys Feng Shui

Physical memory:



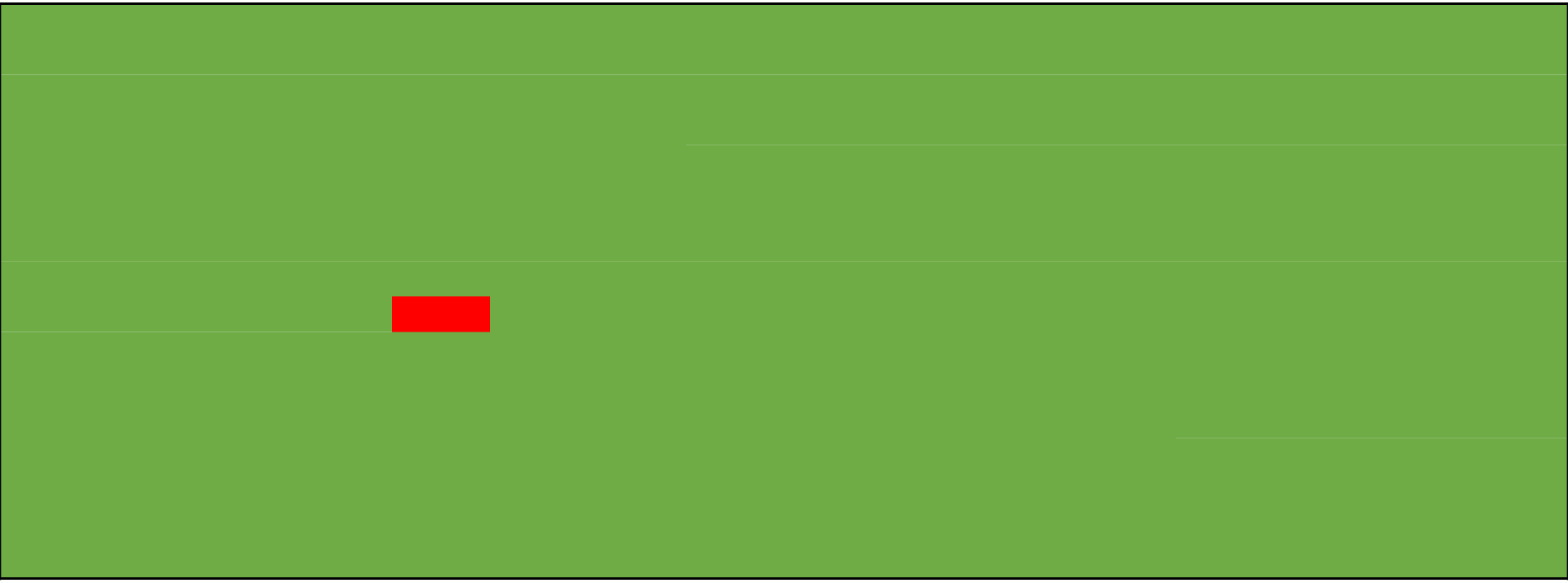
Release the vulnerable page

Landing a Page Table

- No access to `pagemap` (virtual – physical address mapping)
- No fancy memory management features (deduplication)

Phys Feng Shui

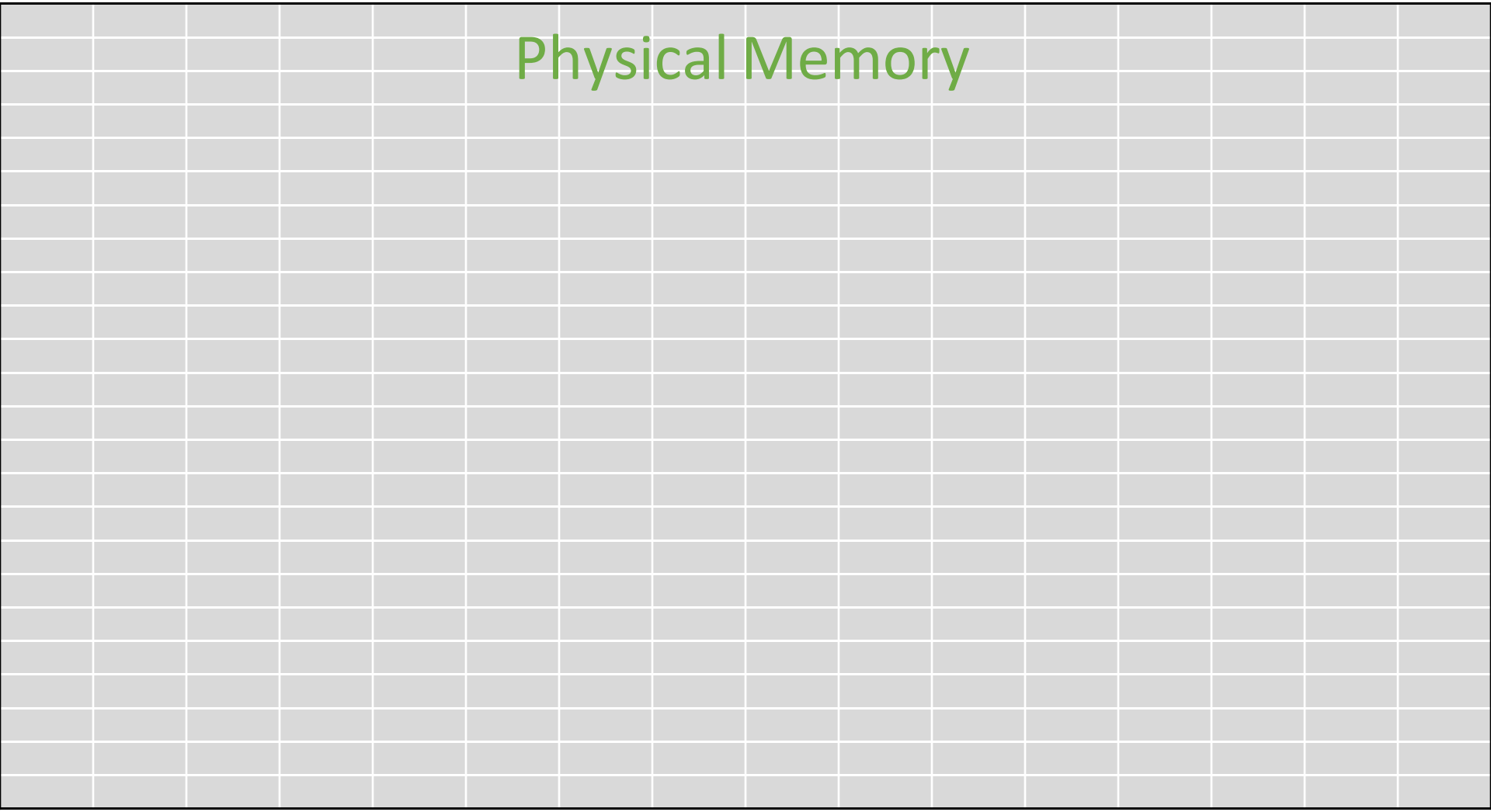
Physical memory:



Trigger a Page Table Allocation

Phys Feng Shui

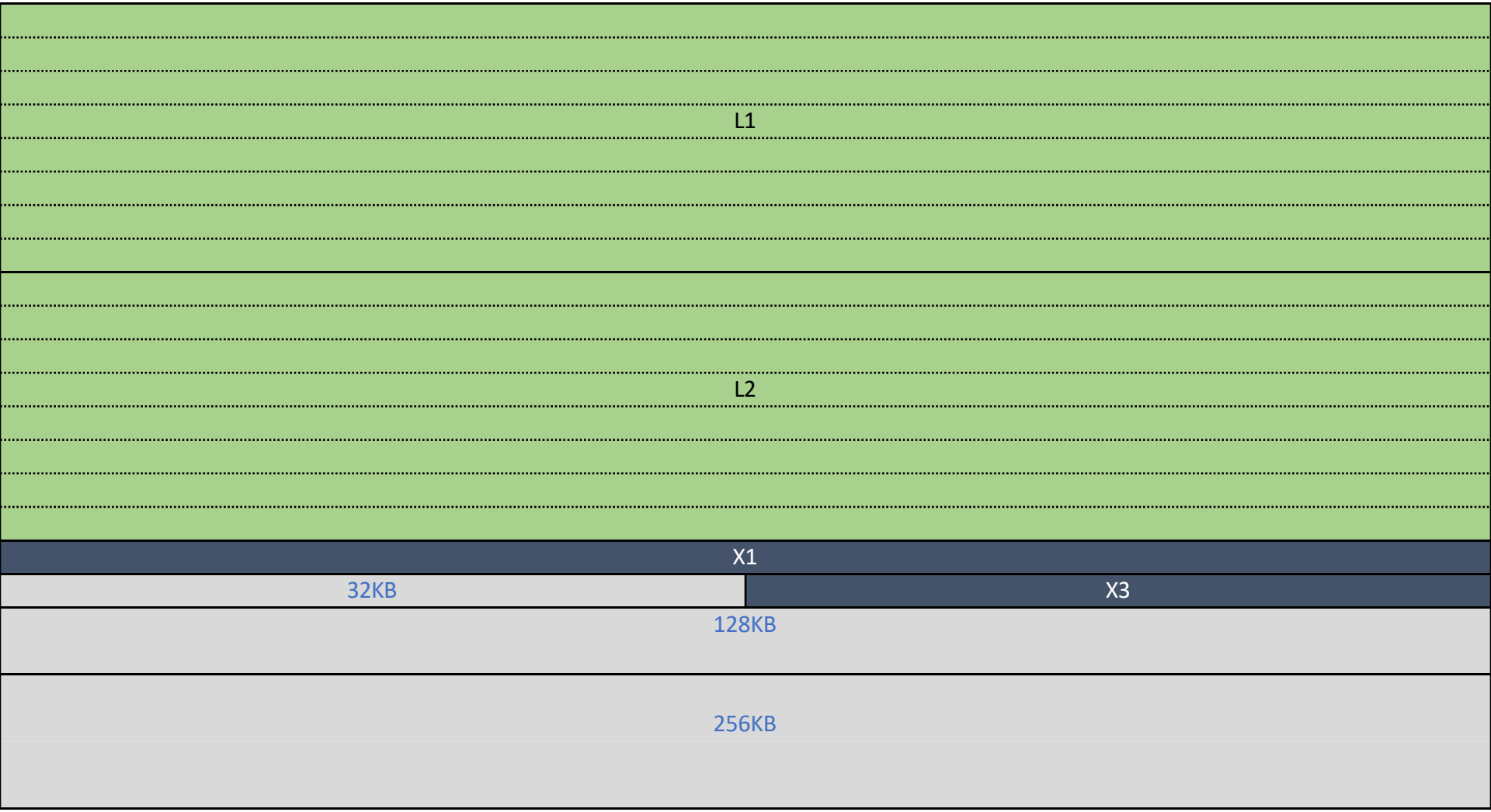
Exploit the predictable behavior of the Buddy Allocator



Phys Feng Shui step 1/8

Exhaust + Template *Large* chunks

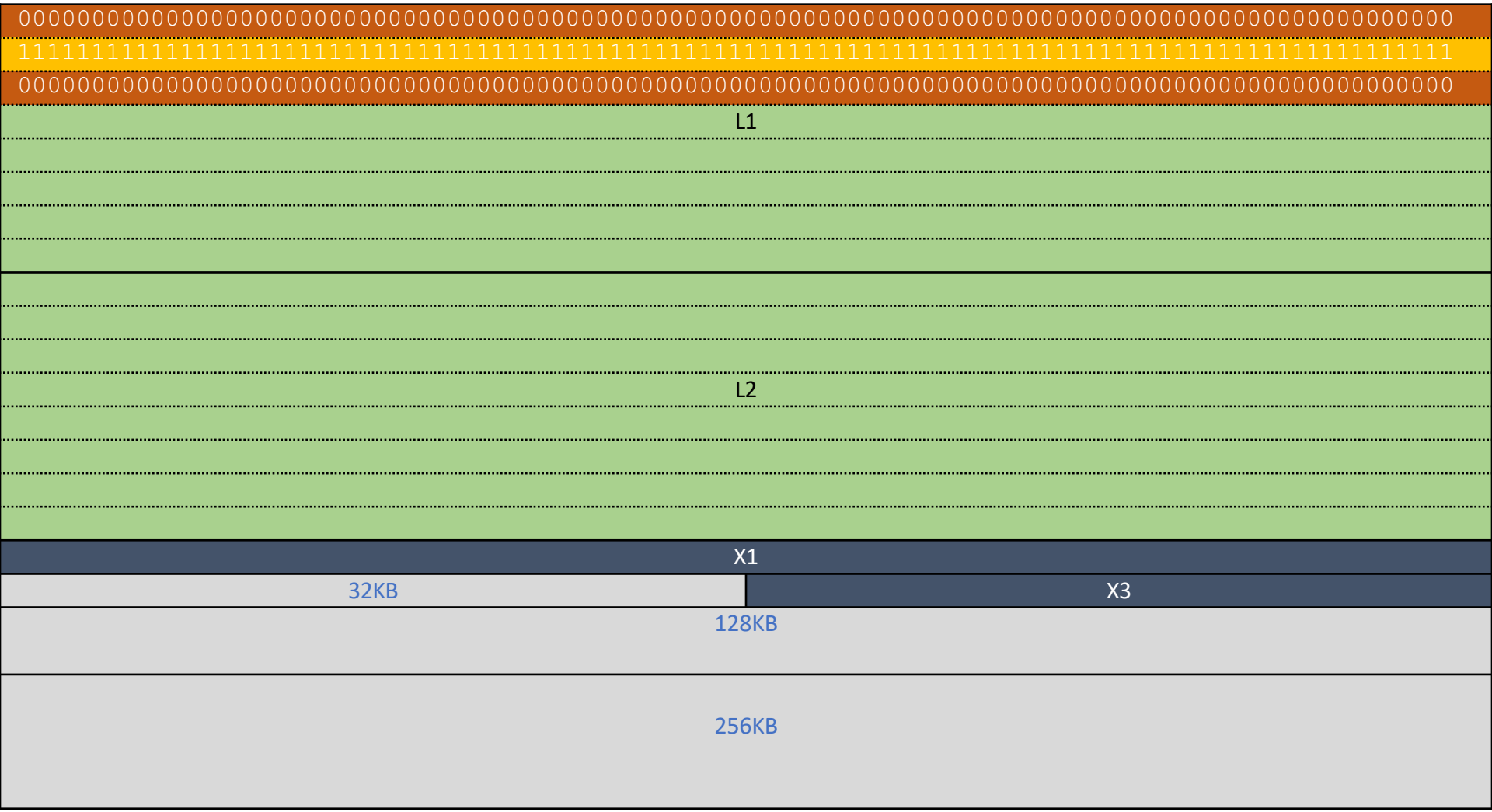
```
L1, L2, …, Ln = exhaust(L); // get all 2^9 = 512KB chunks
```



Phys Feng Shui step 1/8

Exhaust + Template *Large* chunks

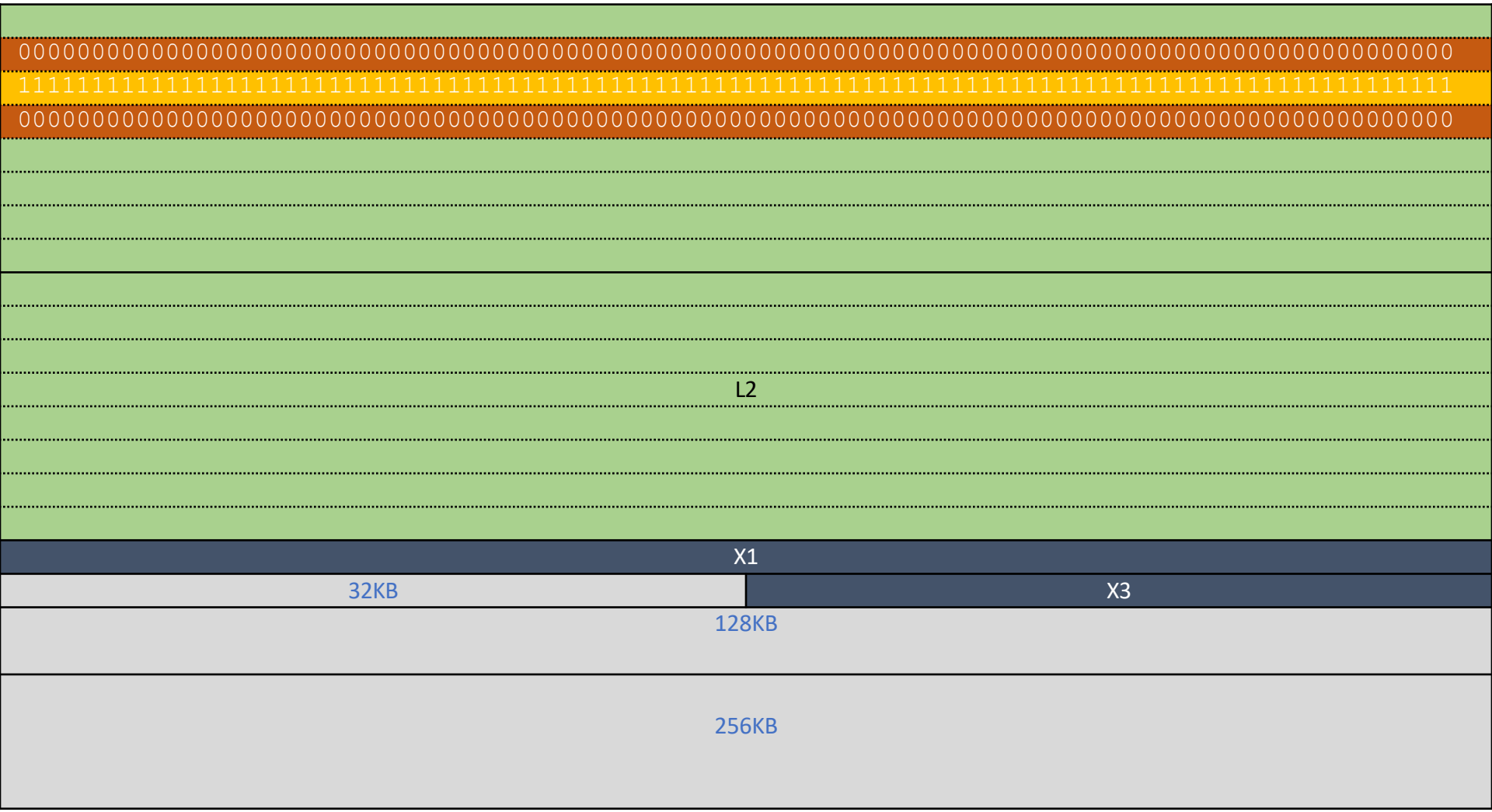
```
Hammer(L1, 2); // hammer row 2 of chunk L1
```



Phys Feng Shui step 1/8

Exhaust + Template *Large* chunks

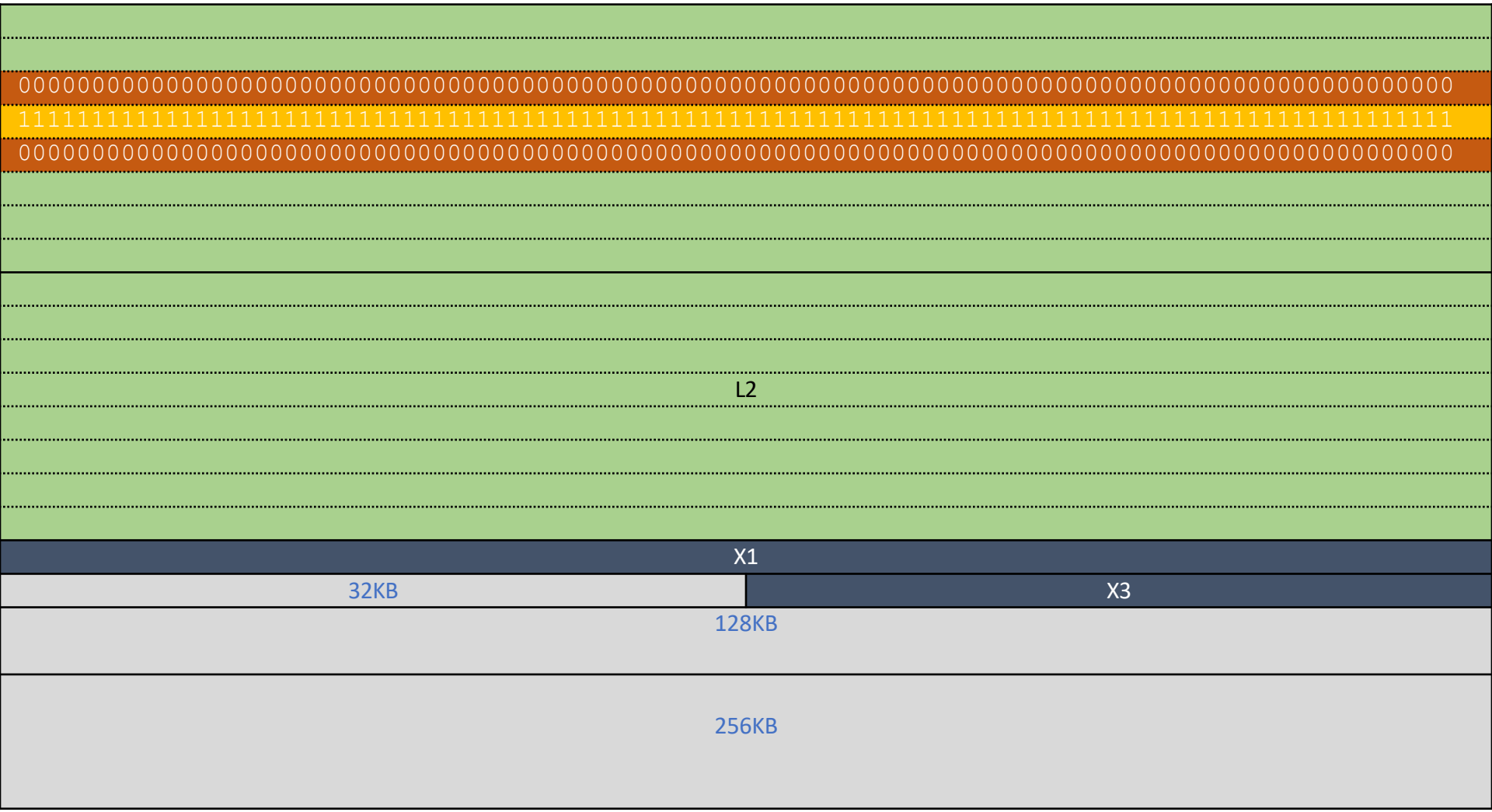
```
Hammer(L1, 3); // hammer row 3 of chunk L1
```



Phys Feng Shui step 1/8

Exhaust + Template *Large* chunks

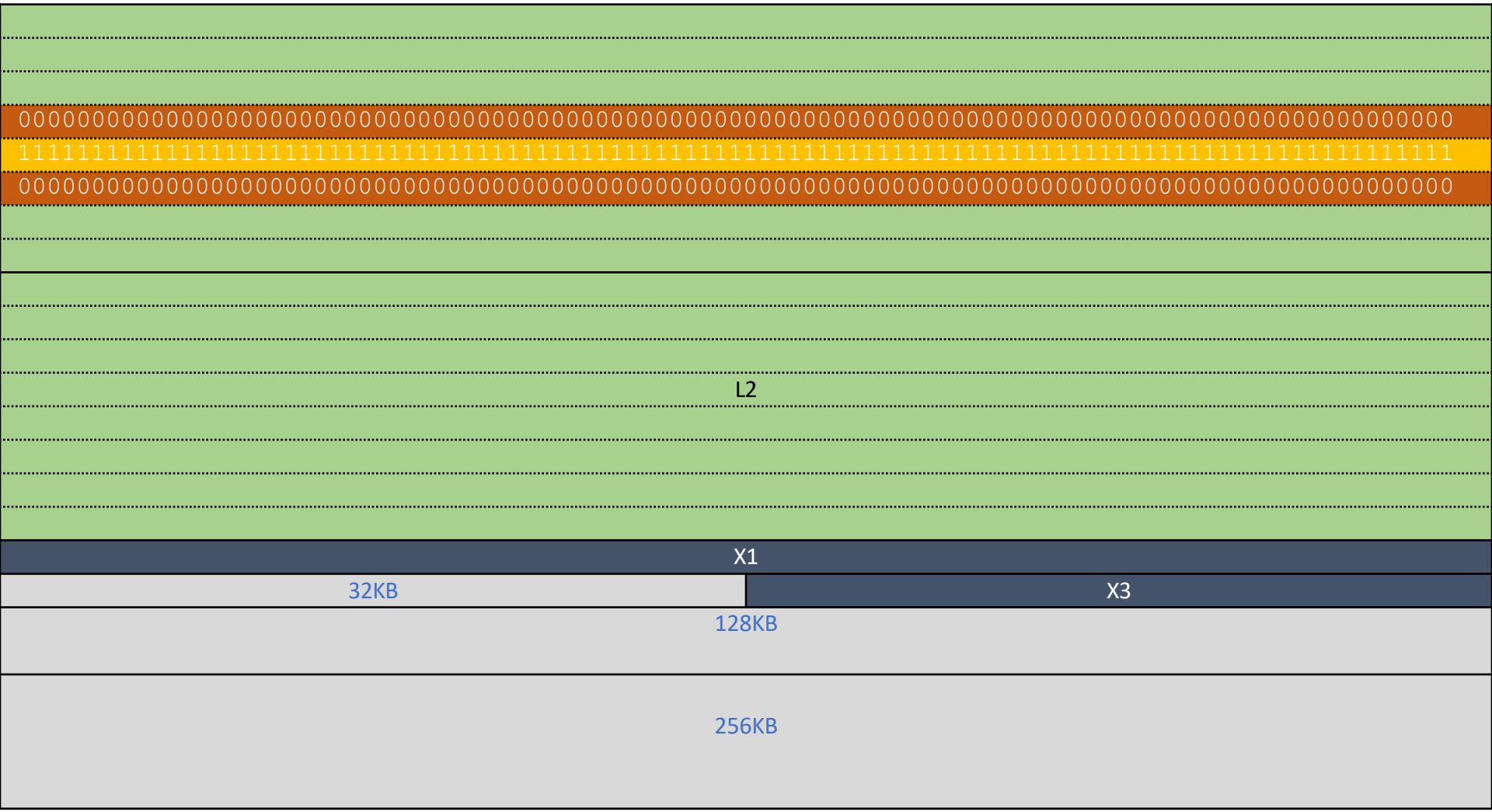
```
Hammer(L1, 4); // hammer row 4 of chunk L1
```



Phys Feng Shui step 1/8

Exhaust + Template *Large* chunks

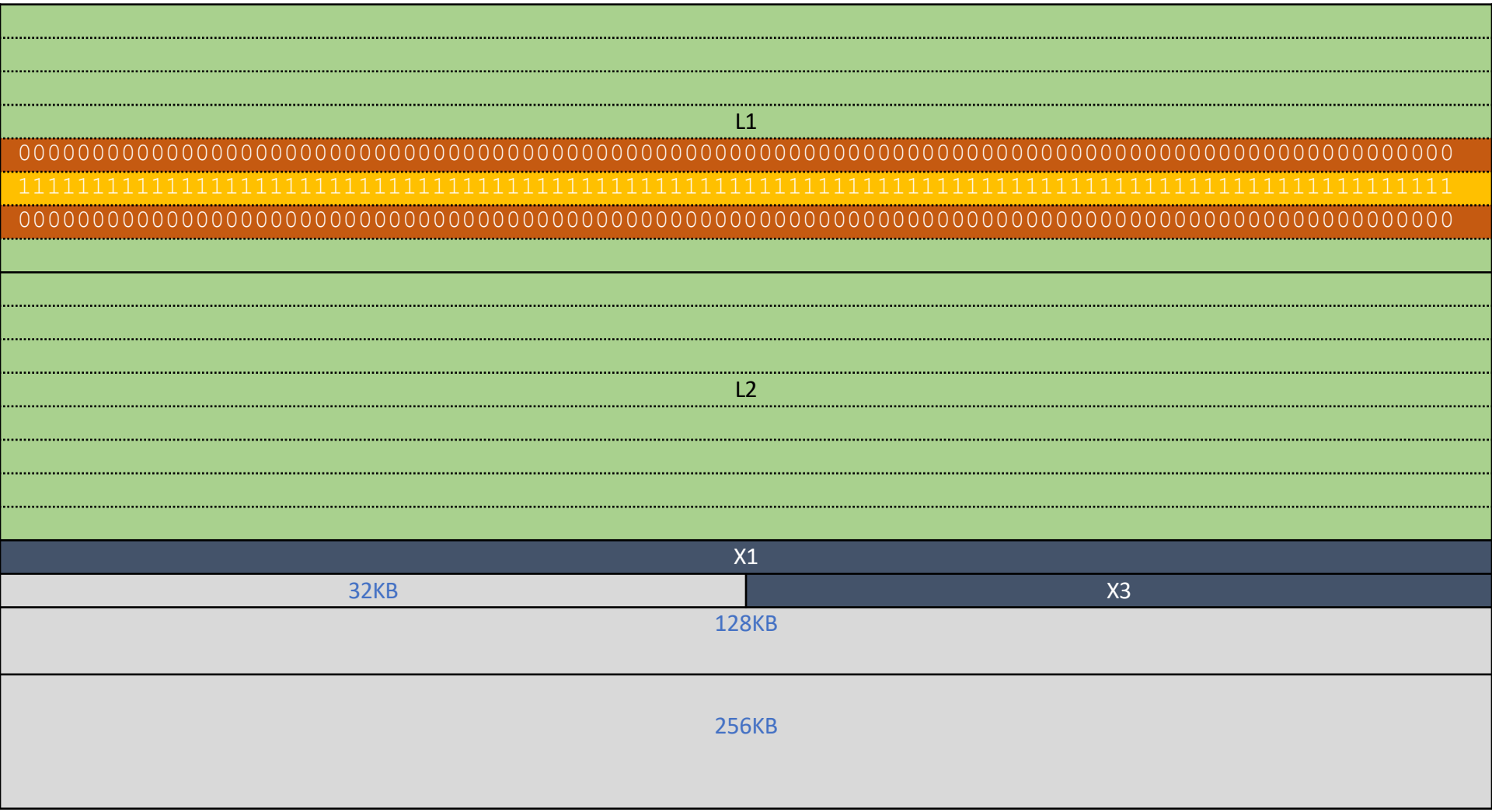
```
Hammer(L1, 5); // hammer row 5 of chunk L1
```



Phys Feng Shui step 1/8

Exhaust + Template *Large* chunks

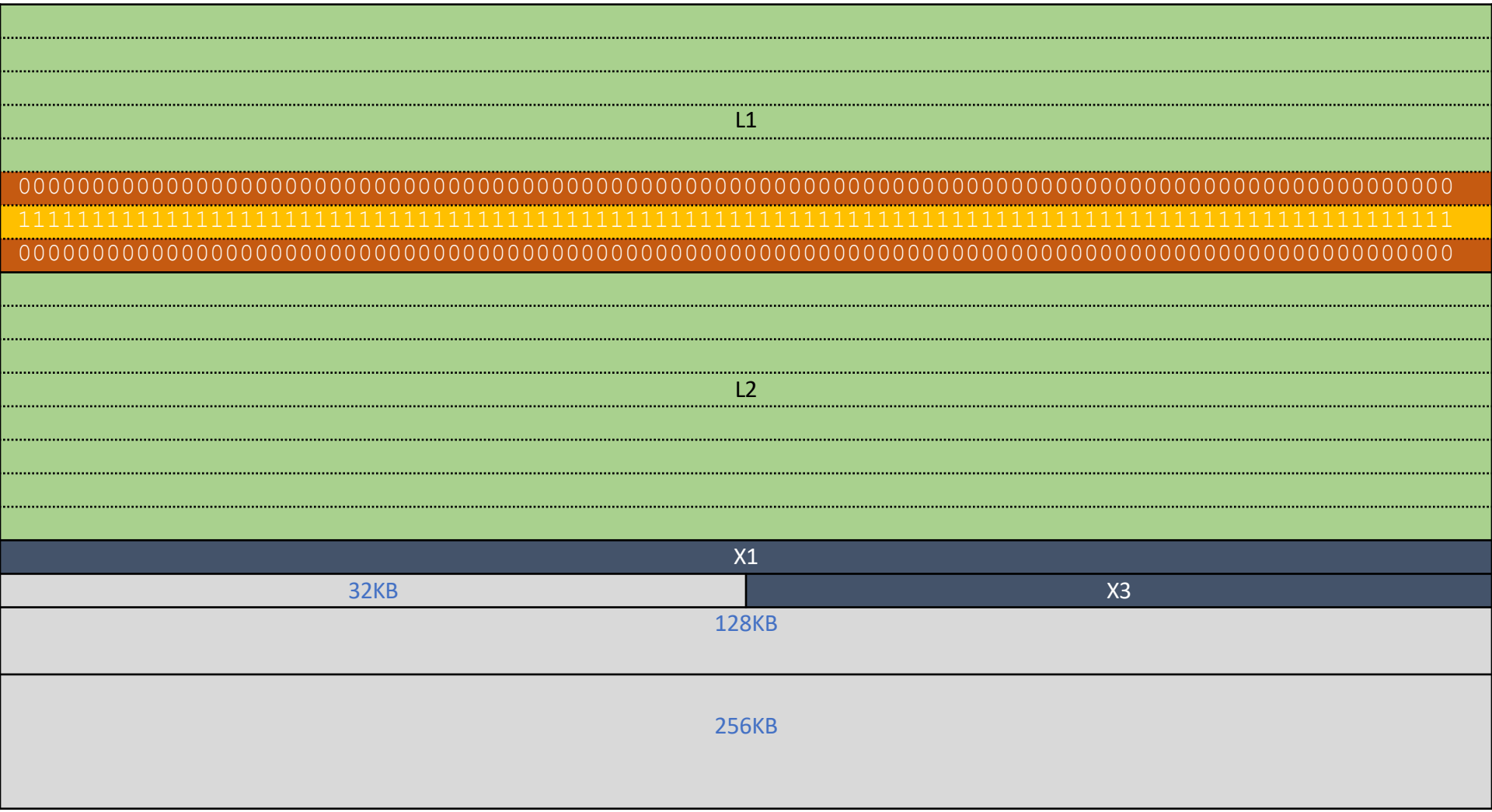
```
Hammer(L1, 6); // hammer row 6 of chunk L1
```



Phys Feng Shui step 1/8

Exhaust + Template *Large* chunks

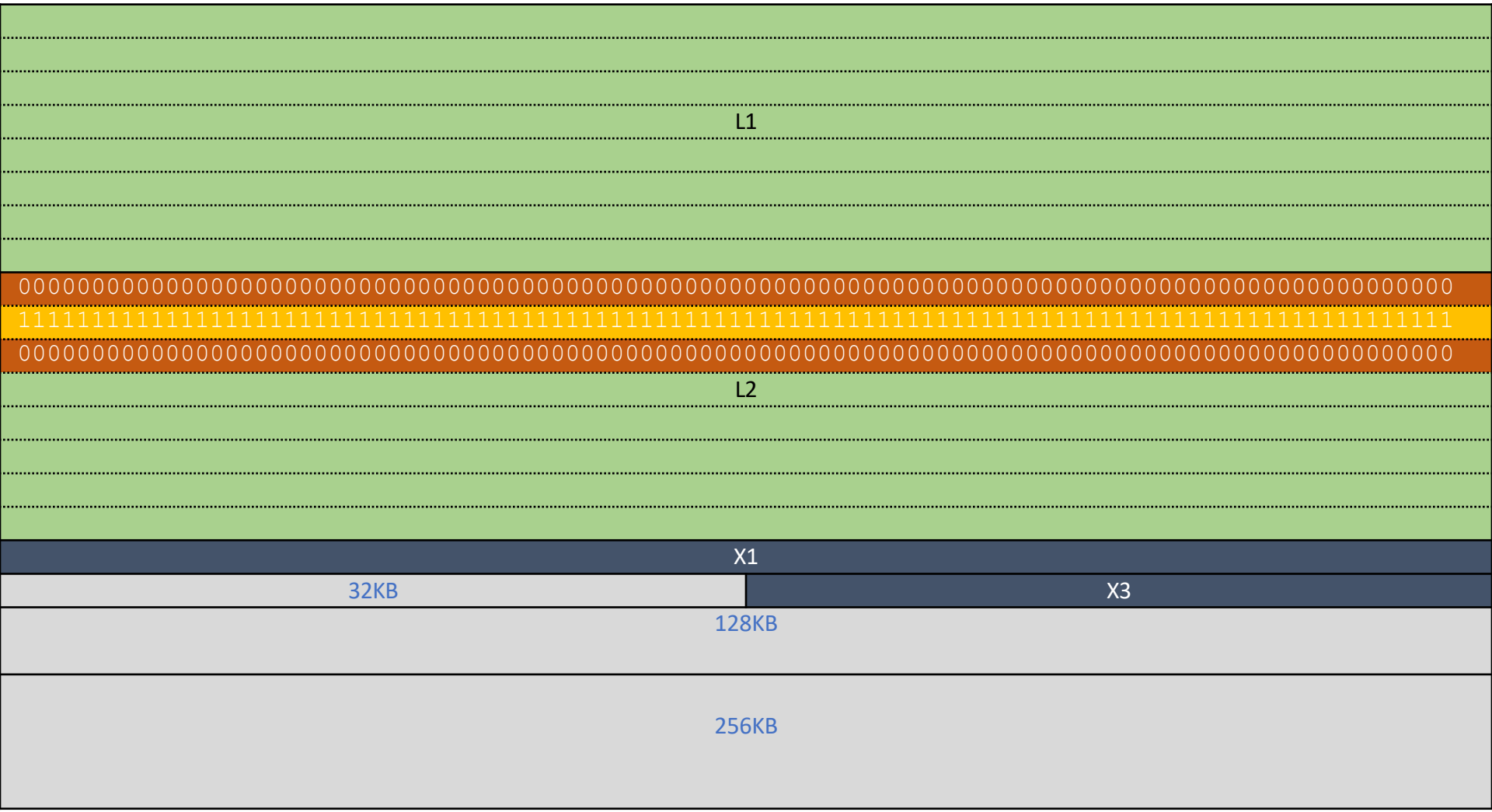
```
Hammer(L1, 7); // hammer row 7 of chunk L1
```



Phys Feng Shui step 1/8

Exhaust + Template *Large* chunks

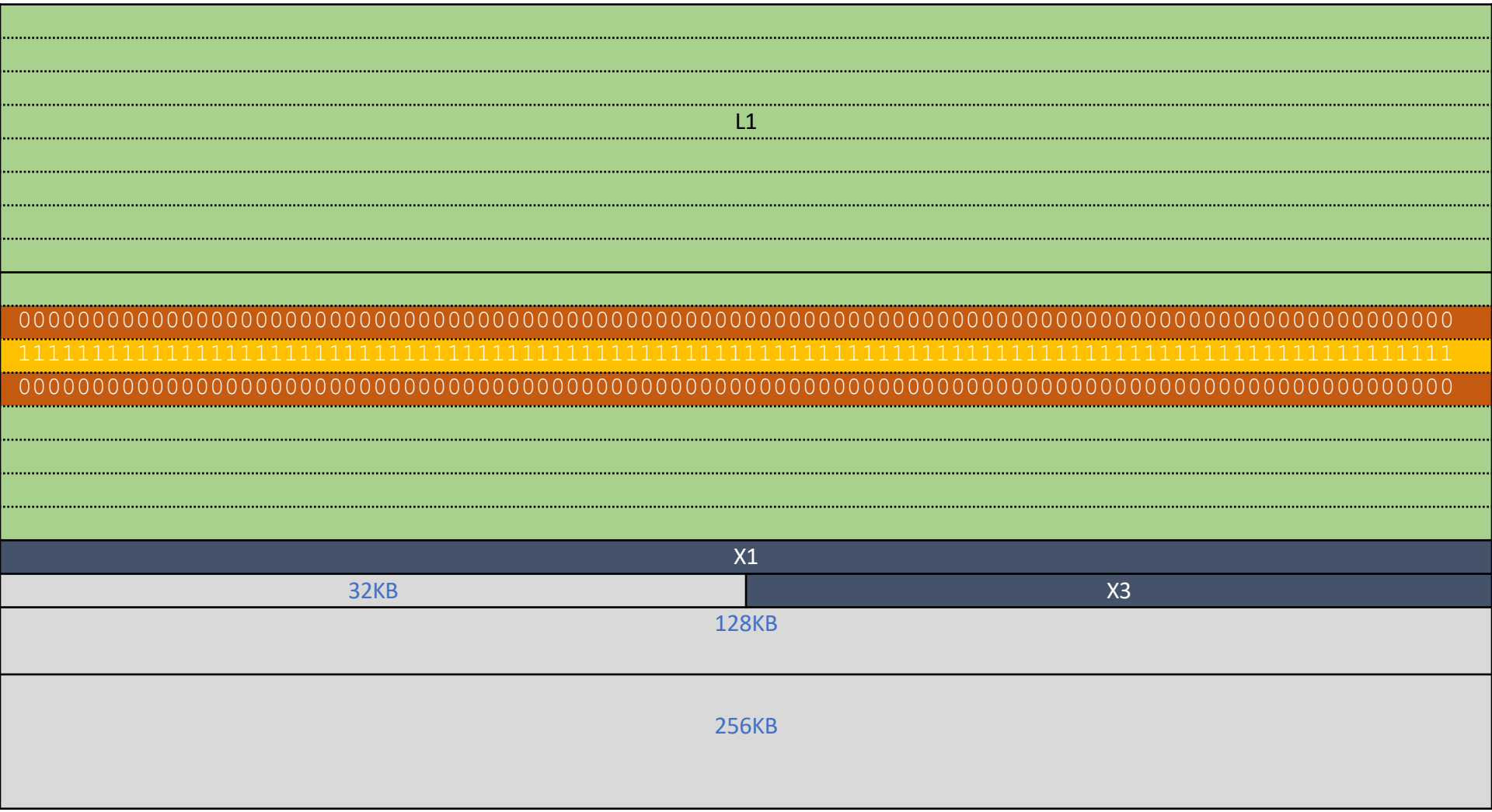
```
Hammer(L2, 2); // hammer row 2 of chunk L2
```



Phys Feng Shui step 1/8

Exhaust + Template *Large* chunks

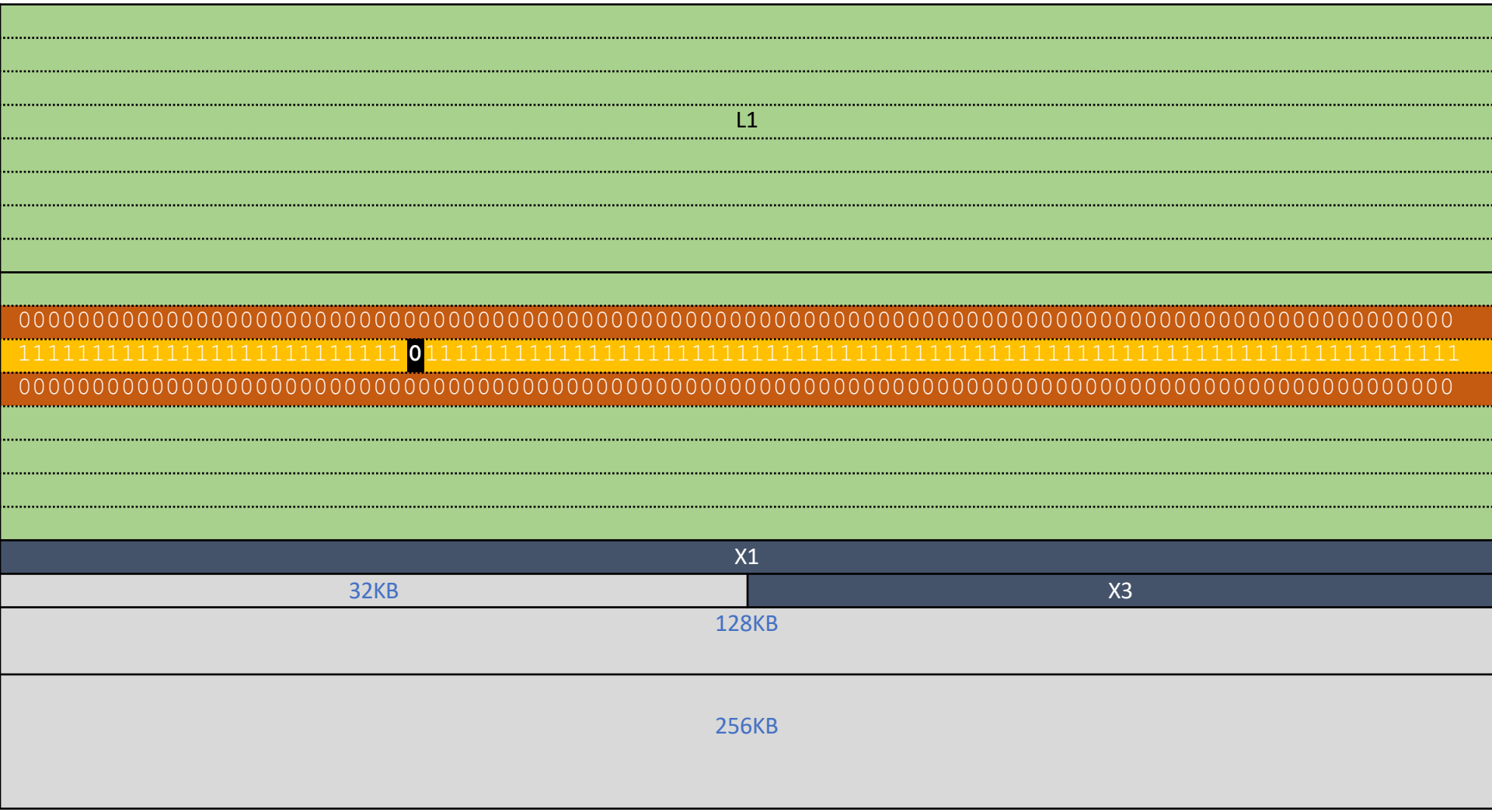
```
Hammer(L2, 3); // hammer row 3 of chunk L2
```



Phys Feng Shui step 1/8

Exhaust + Template *Large* chunks

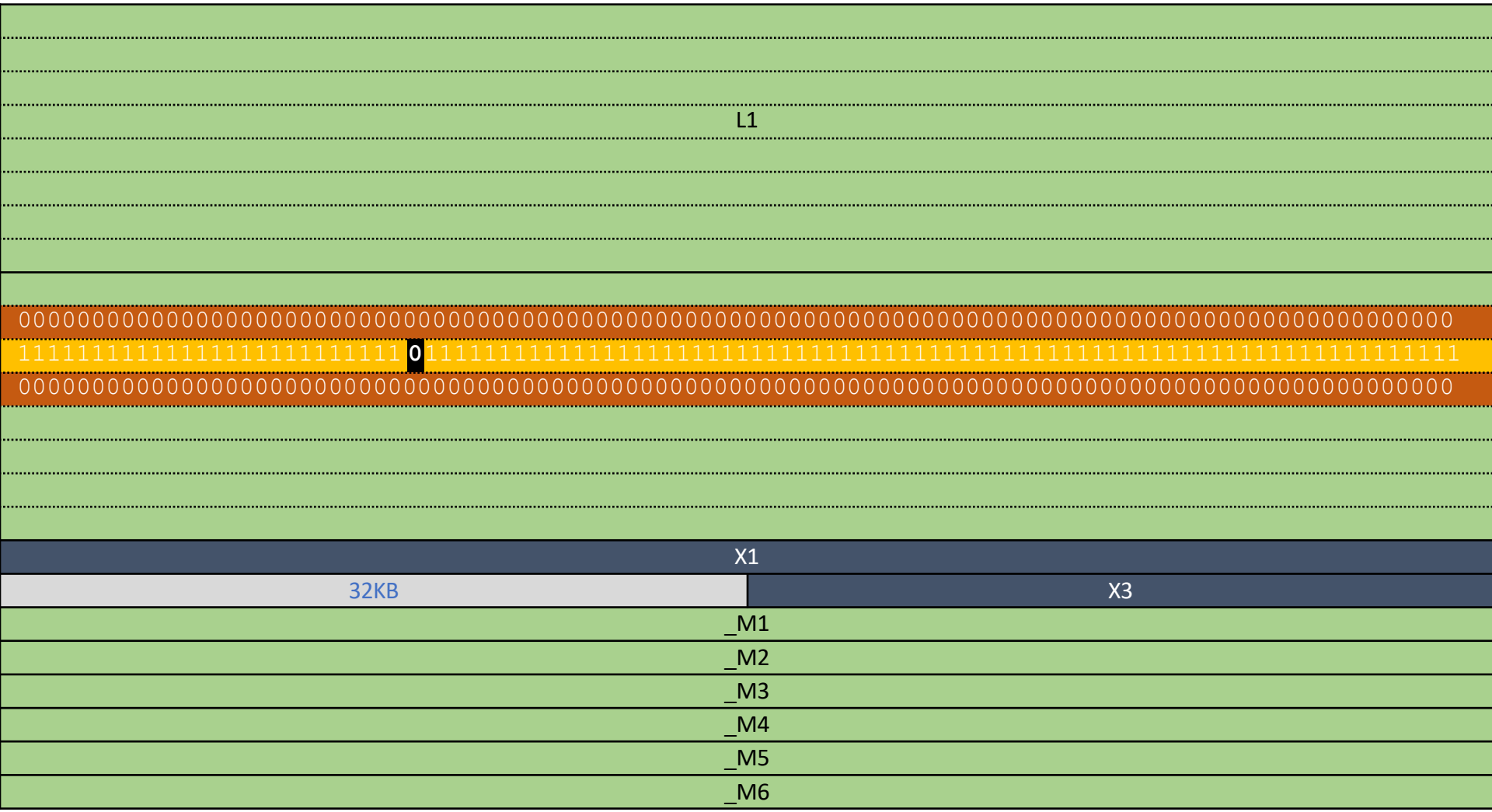
"exploitable flip found in page 5 of virtual row 3 of L2!"



Phys Feng Shui step 3/8

Release *Large* chunk with vulnerable page

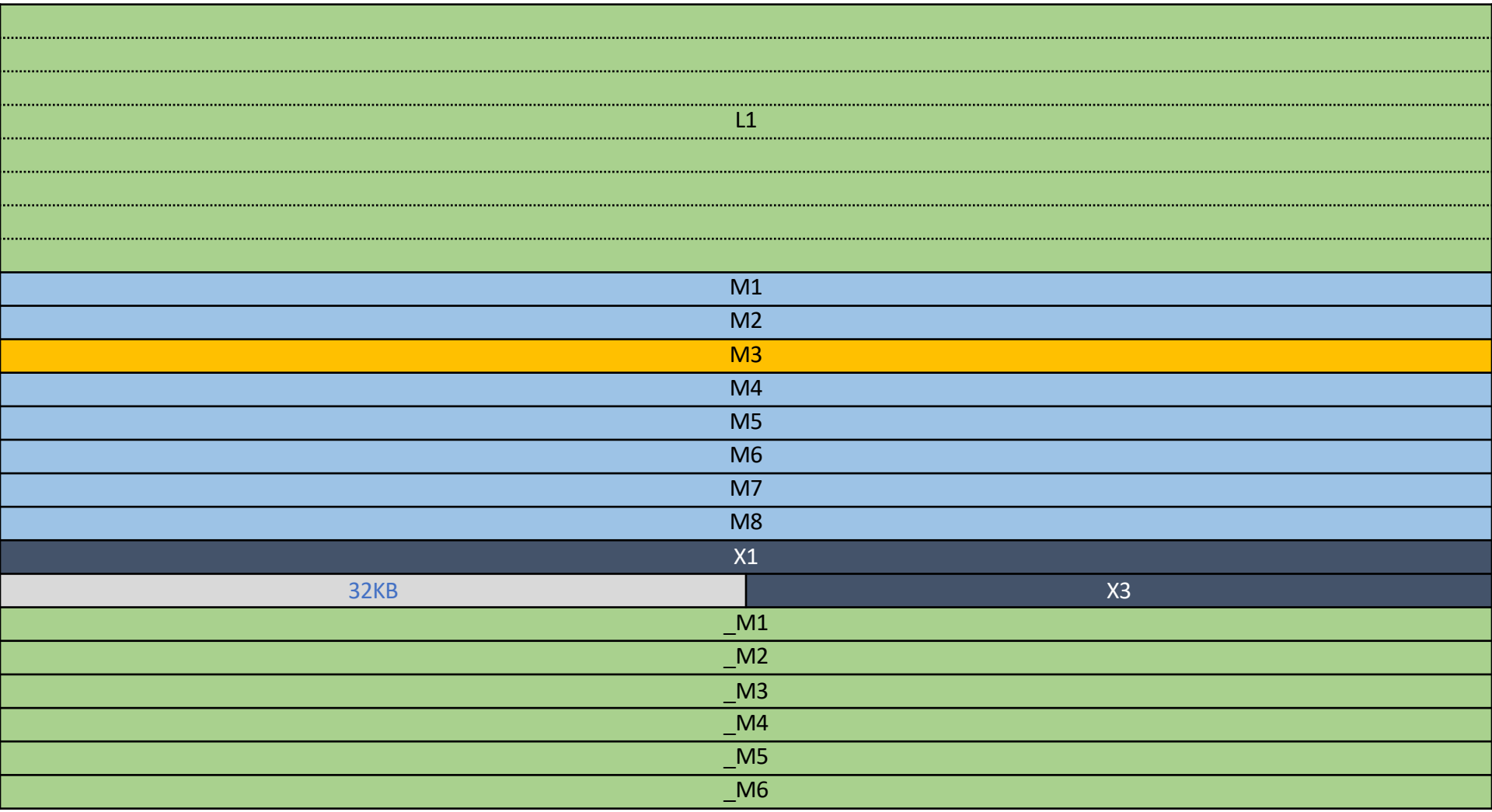
```
Release(L2); // L chunk with vulnerable page
```



Phys Feng Shui step 4/8

Exhaust *Medium-sized* chunks (again)

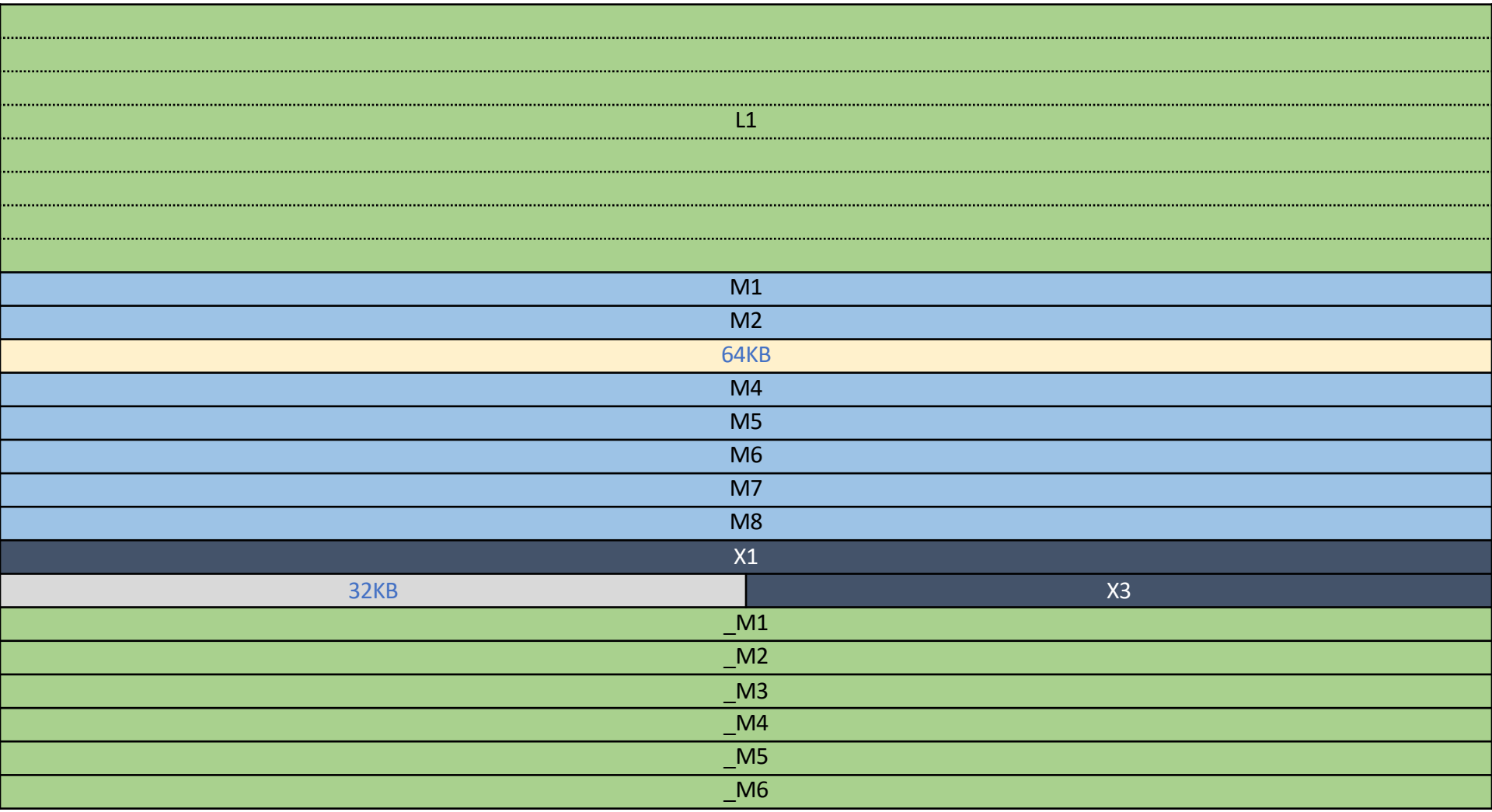
```
M1, M2, …, Mn = exhaust(6); // get all 2^6 = 64KB chunks
```



Phys Feng Shui step 5/8

Release vulnerable *Medium-sized* chunk + Release all Large chunks

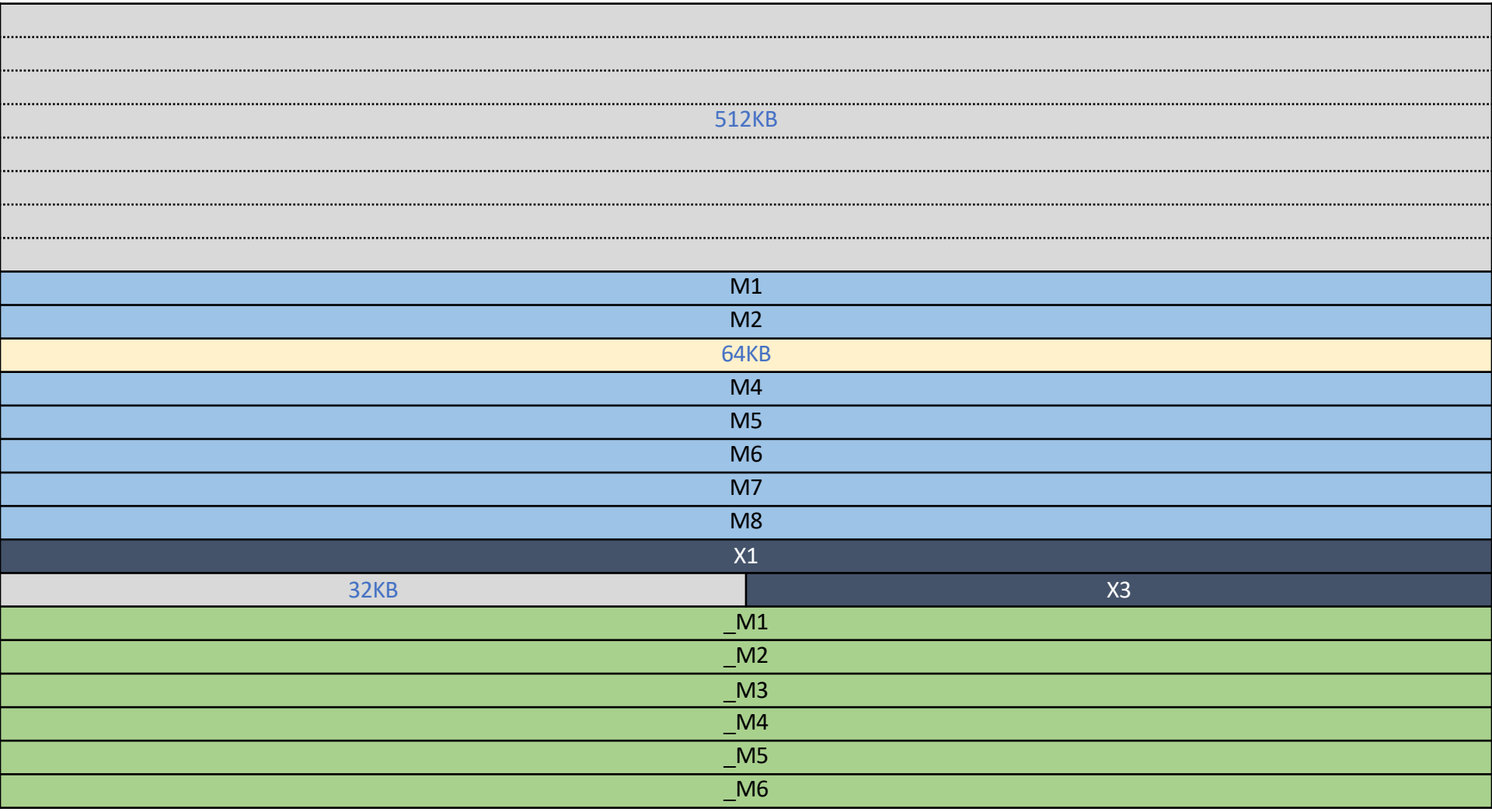
```
Release(M3); // releases the vulnerable row
```



Phys Feng Shui step 5/8

Release vulnerable *Medium-sized* chunk + Release all Large chunks

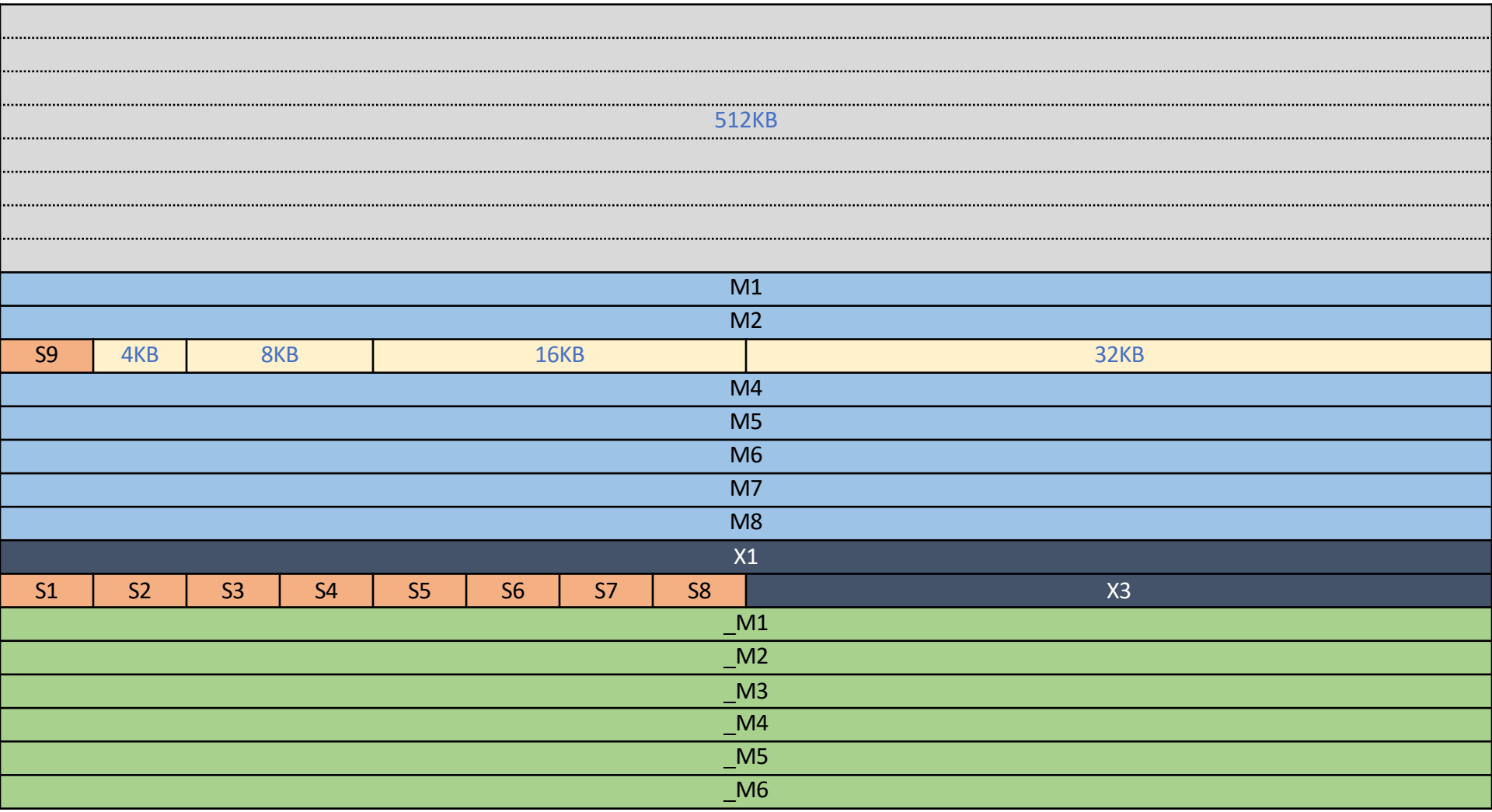
```
ReleaseAll(L); // to avoid going out-of-memory later
```



Phys Feng Shui step 6/8

Land a *small* chunk in the vulnerable 64 KB row

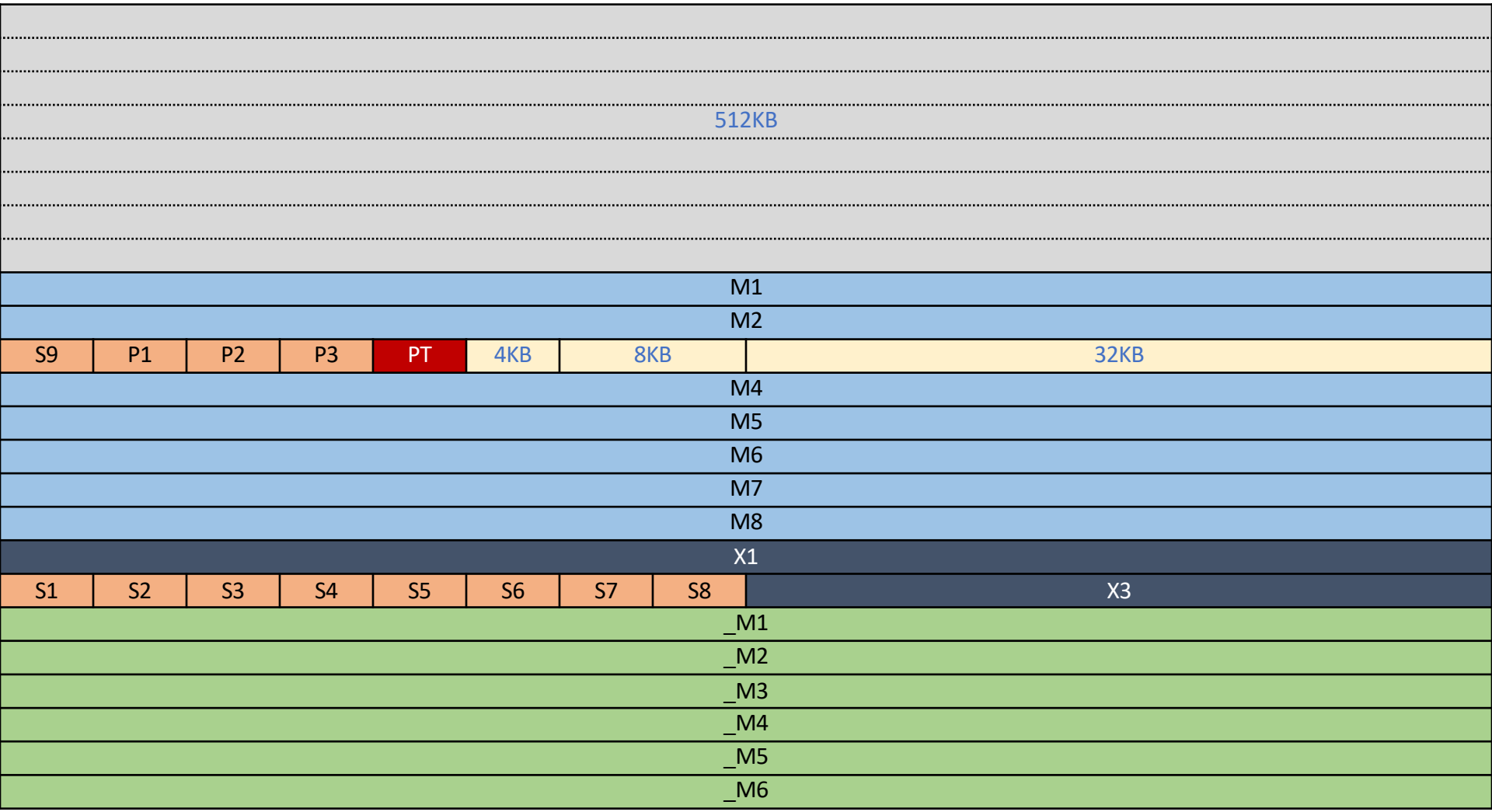
```
Land(S); // allocate 4KB pages until the 64KB is used
```



Phys Feng Shui step 8/8

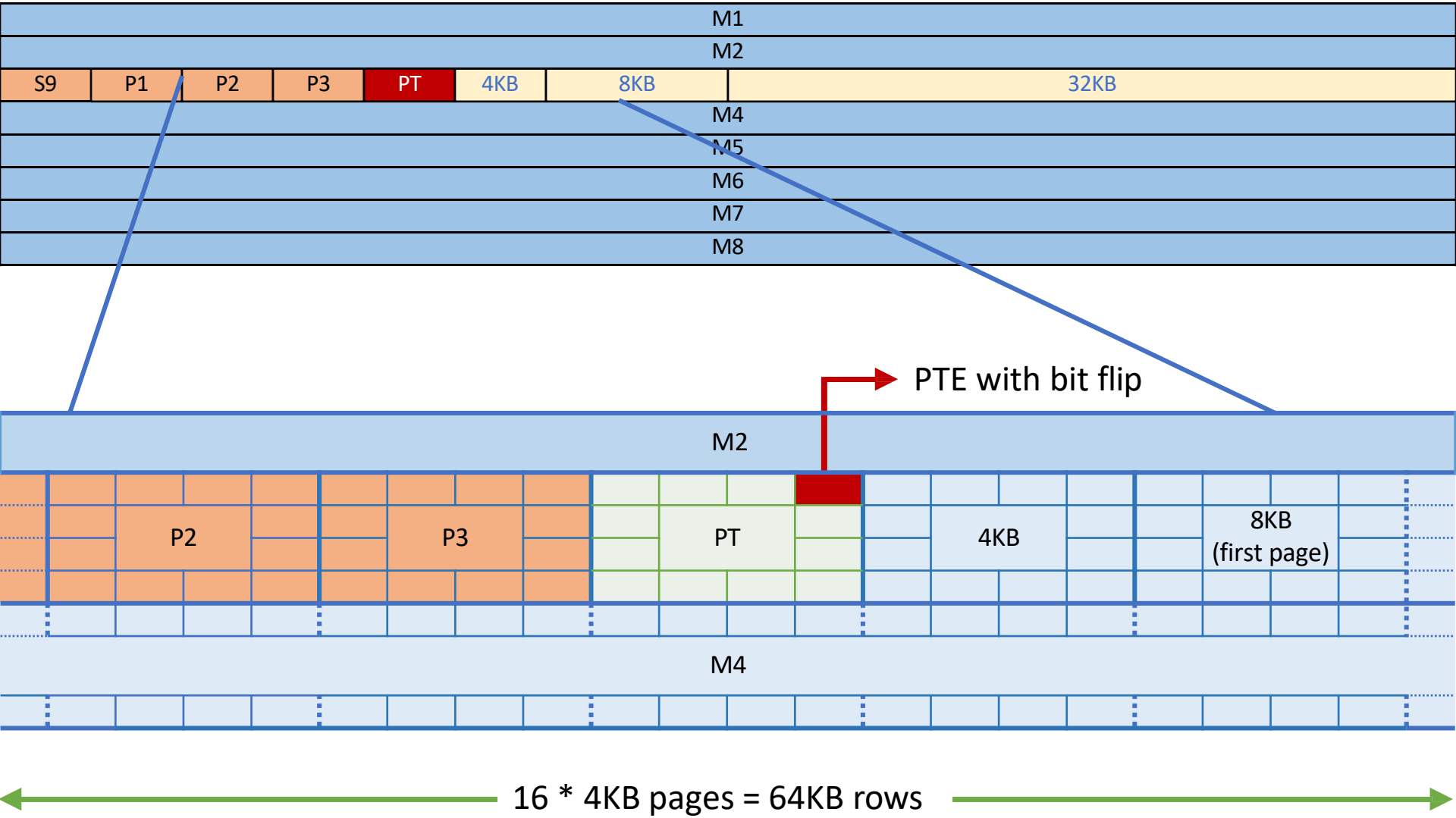
Force a Page Table allocation + map the vulnerable PTE

```
PT = mmap(MAP_FIXED); // Force a Page Table allocation
```



Phys Feng Shui step 8/8

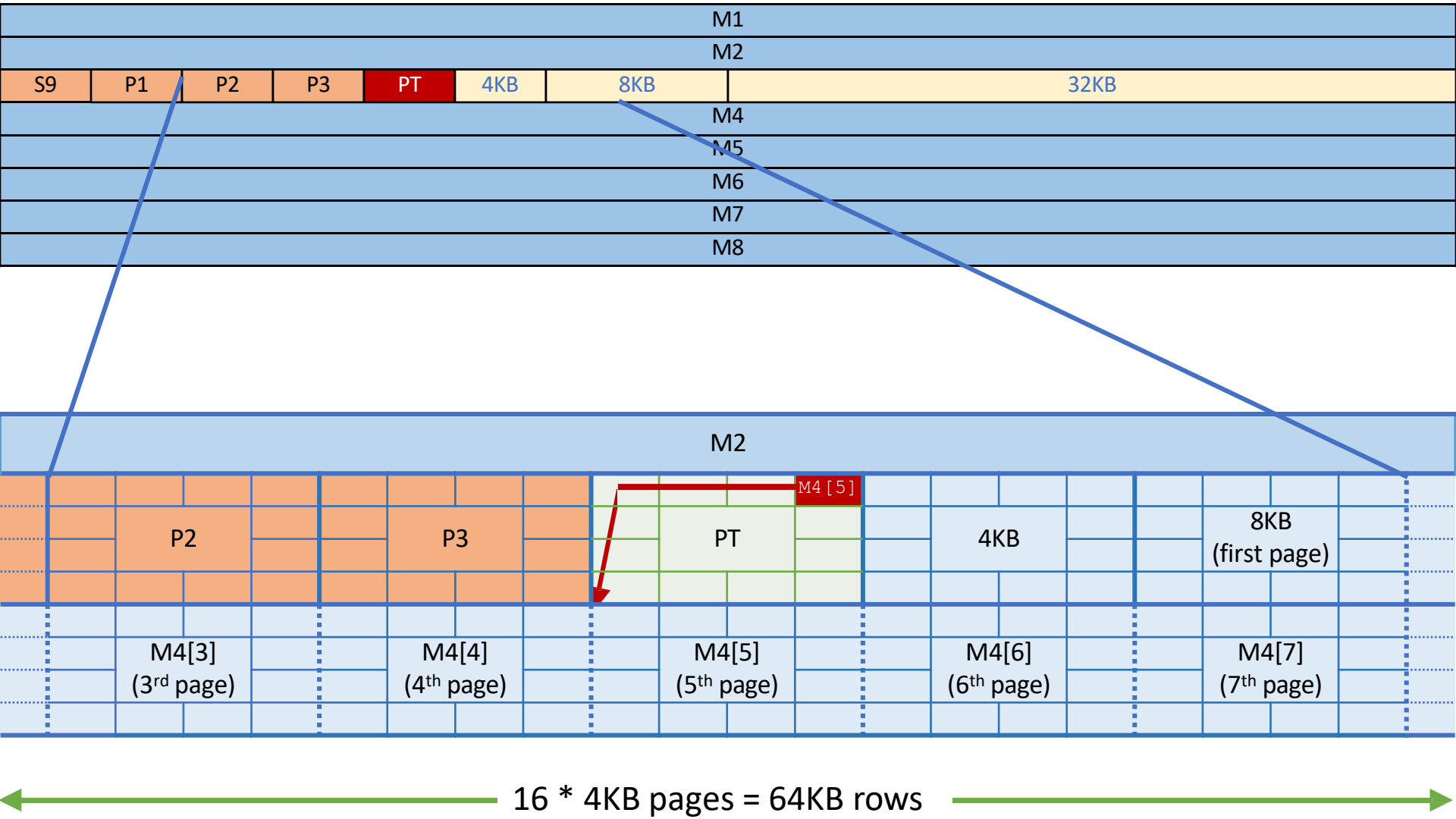
Force a Page Table allocation + map the vulnerable PTE



Phys Feng Shui step 8/8

Force a Page Table allocation + map the vulnerable PTE

```
mmap(M4[5], MAP_FIXED); // map vulnerable PTE 64KB 'away'
```



Overview

1. Memory Templating

   Scan memory for useful bit flips

2. Land a Page Table

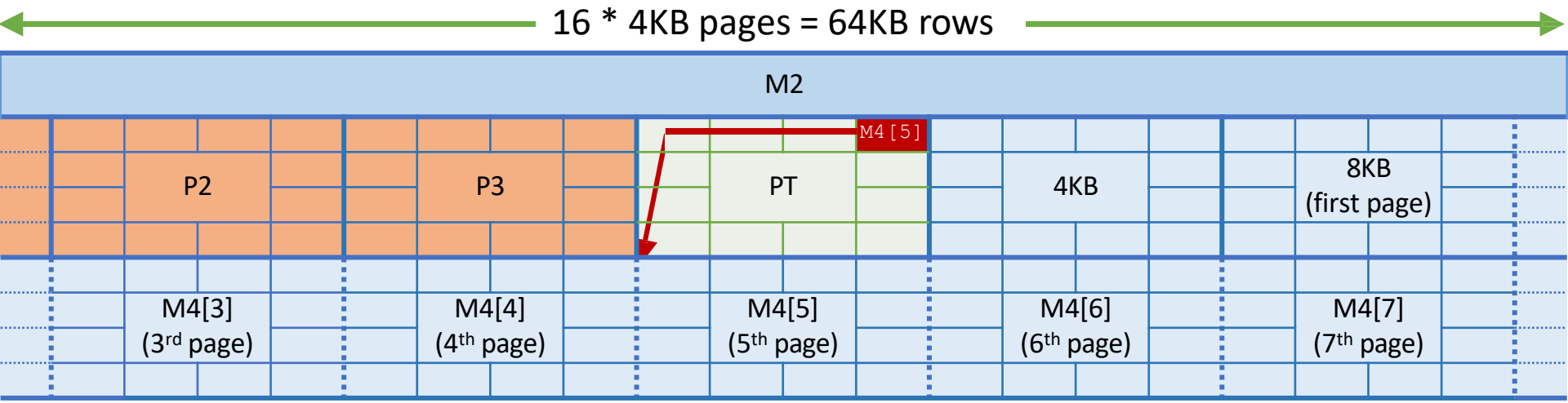
   Store a page table on a vulnerable page

3. **Reproduce the bit flip**

   Modify the data structure and get root acces

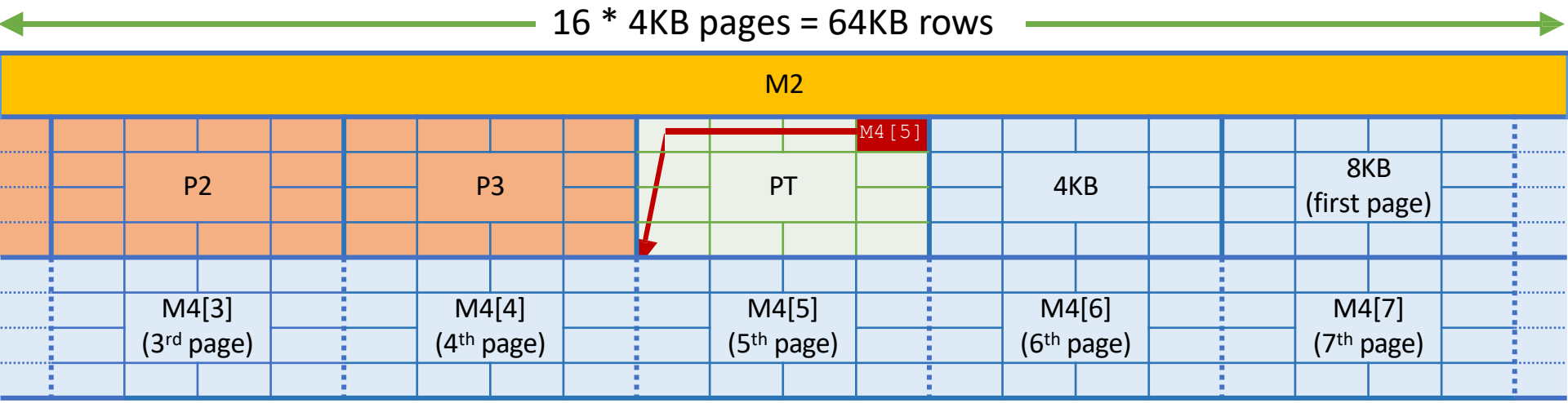
Drammer

Perform double-sided rowhammer to flip a bit in the PTE



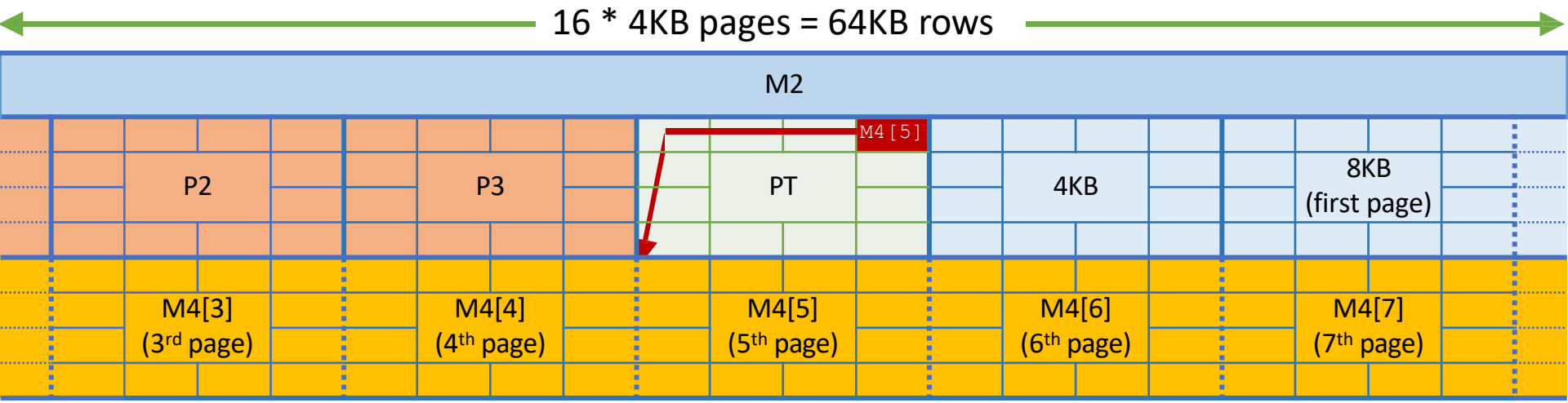
Drammer

Perform double-sided rowhammer to flip a bit in the PTE



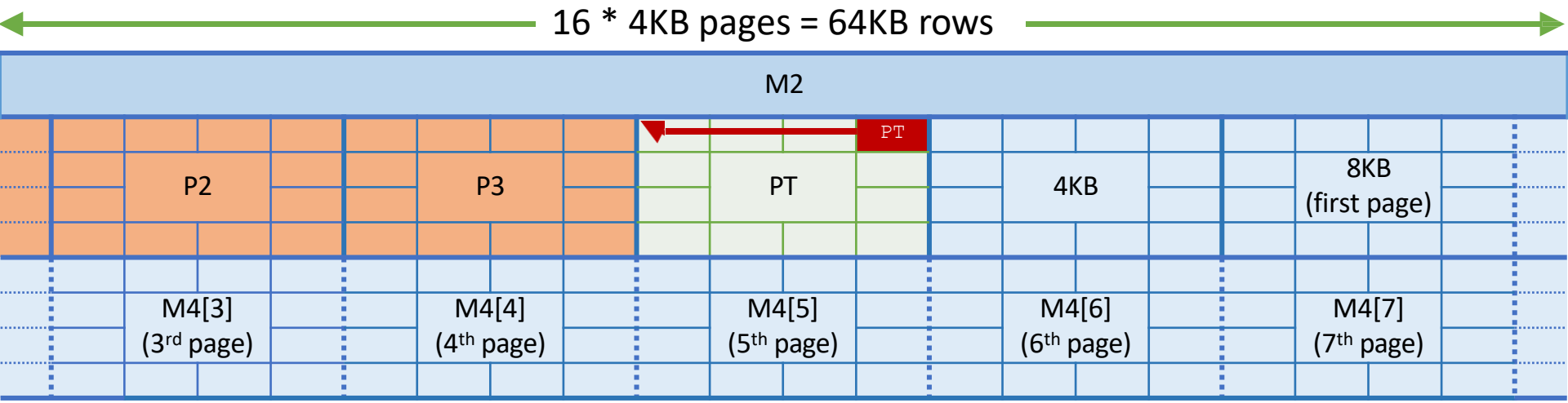
Drammer

Perform double-sided rowhammer to flip a bit in the PTE



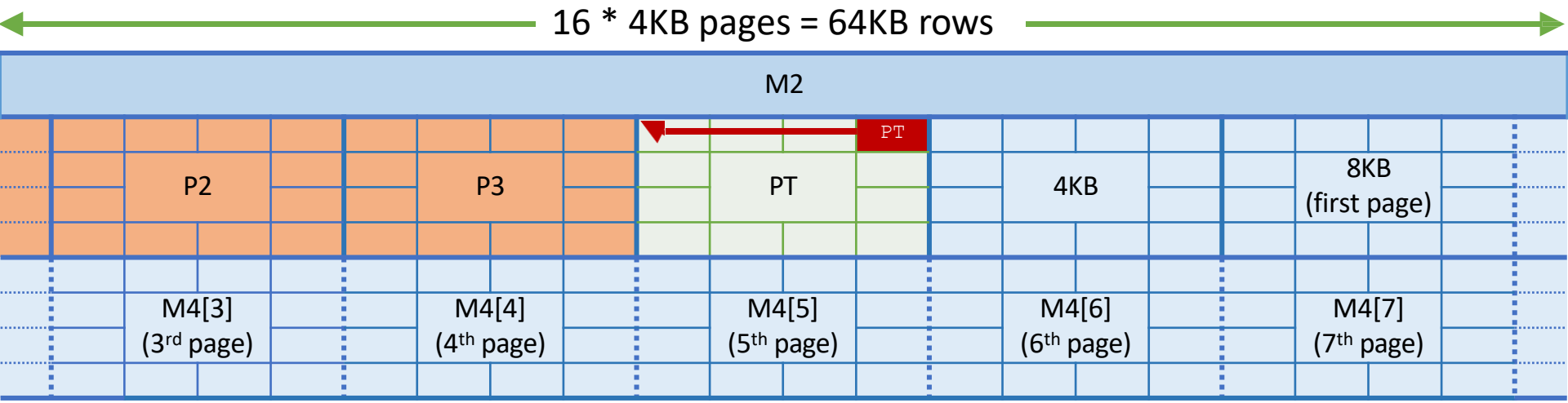
Drammer

Write access to a Page Table



Drammer

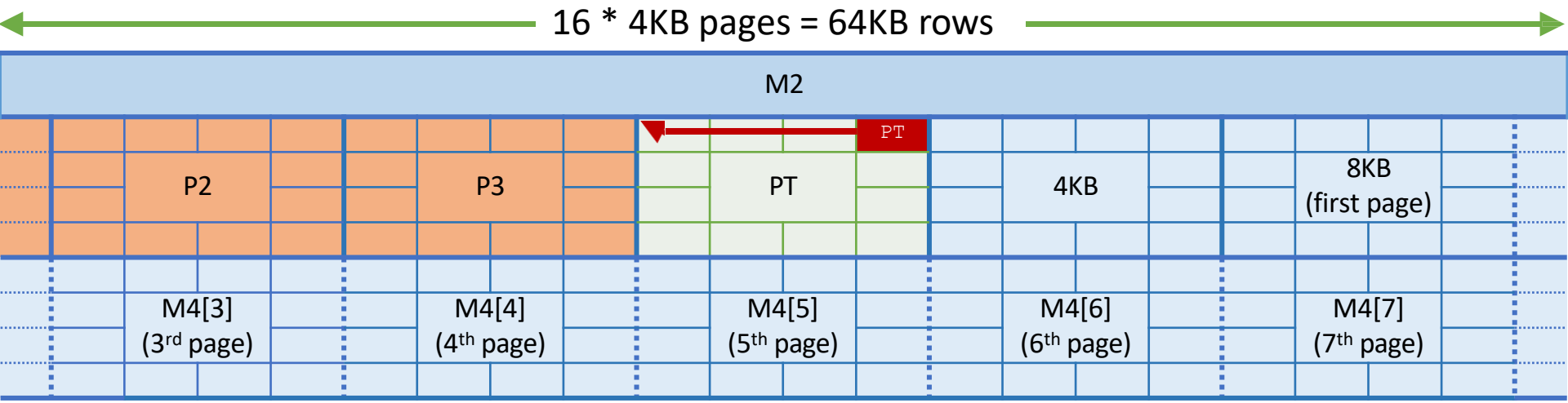
Write access to a Page Table



1. Fill PT with Page Table Entries to kernel memory

Drammer

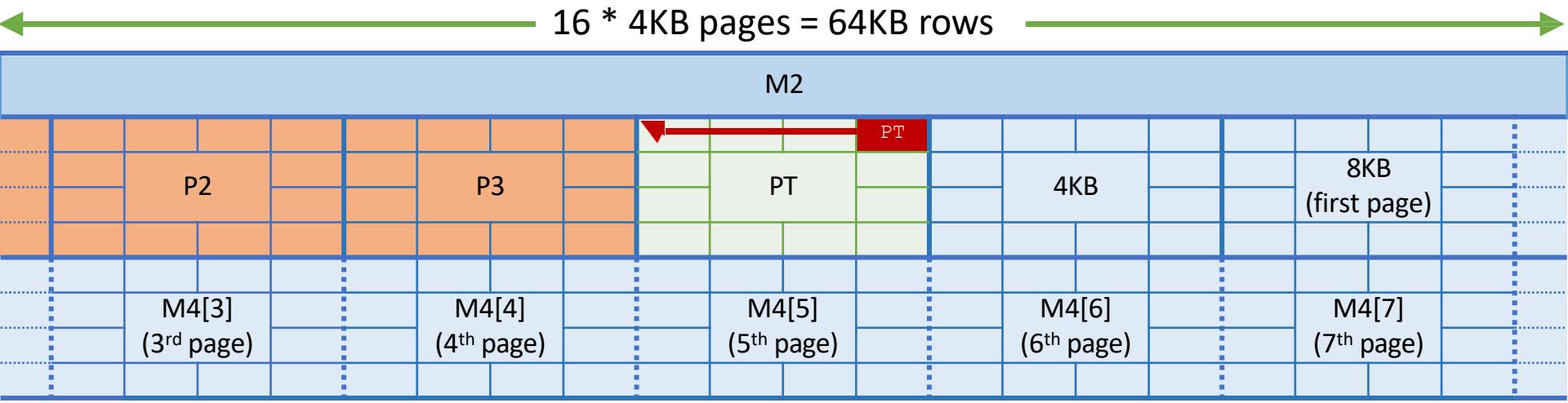
Write access to a Page Table



1. Fill PT with Page Table Entries to kernel memory
2. Search kernel memory for our **`struct cred`**

Drammer

Write access to a Page Table



1. Fill PT with Page Table Entries to kernel memory
2. Search kernel memory for our **`struct cred`**
3. Overwrite our **`uid`** and **`gid`** to get root privileges

Evaluation

Device	#flips	1st exploitable flip after
LG Nexus 5[1]	1058	116s
LG Nexus 5[4]	0	-
LG Nexus 5[5]	747,013	1s
LG Nexus 4	1,328	7s
OnePlus One	3,981	942s
Motorola Moto G (2013)	429	441s
LG G4 (ARMv8 – 64-bit)	117,496	5s

Bit flips on 18 out of 27 tested devices

Evaluation

Device	#flips	1st exploitable flip after
LG Nexus 5[1]	1058	116s
LG Nexus 5[4]	0	-
LG Nexus 5[5]	747,013	1s
LG Nexus 4	1,328	7s
OnePlus One	3,981	942s
Motorola Moto G (2013)	429	441s
LG G4 (ARMv8 – 64-bit)	117,496	5s

Evaluation

Device	#flips	1st exploitable flip after
LG Nexus 5[1]	1058	116s
LG Nexus 5[4]	0	-
LG Nexus 5[5]	747,013	1s
LG Nexus 4	1,328	7s
OnePlus One	3,981	942s
Motorola Moto G (2013)	429	441s
LG G4 (ARMv8 – 64-bit)	117,496	5s

Evaluation

Device	#flips	1st exploitable flip after
LG Nexus 5[1]	1058	116s
LG Nexus 5[4]	0	-
LG Nexus 5[5]	747,013	1s
LG Nexus 4	1,328	7s
OnePlus One	3,981	942s
Motorola Moto G (2013)	429	441s
LG G4 (ARMv8 – 64-bit)	117,496	5s

Evaluation

Device	#flips	1st exploitable flip after
LG Nexus 5[1]	1058	116s
LG Nexus 5[4]	0	-
LG Nexus 5[5]	747,013	1s
LG Nexus 4	1,328	7s
OnePlus One	3,981	942s
Motorola Moto G (2013)	429	441s
LG G4 (ARMv8 – 64-bit)	117,496	5s

After the 1st exploitable flip, exploitation takes at most 22 seconds

Evaluation

Device	#flips	1st exploitable flip after
LG Nexus 5[1]	1058	116s
LG Nexus 5[4]	0	-
LG Nexus 5[5]	747,013	1s
LG Nexus 4	1,328	7s
OnePlus One	3,981	942s
Motorola Moto G (2013)	429	441s
LG G4 (ARMv8 – 64-bit)	117,496	5s

After the 1st exploitable flip, exploitation takes at most 22 seconds

Drammer test app reported bit flips on:

Google Pixel, OnePlus 3, Galaxy Note 7, HTC One M8, …

Conclusion

- Deterministic Rowhammer exploitation
- No special memory management features required (e.g., deduplication)
- ARM memory controllers are fast enough to do Rowhammer
- LPDDR* found vulnerable
- No easy software fix
- Using DMA bypasses state-of-the-art defenses (e.g., ANVIL)

- More details
  - Demos, statistics and test app: https://vusec.net/projects/drammer
  - Open source: https://github.com/vusec/drammer