TLB;DR: Enhancing TLB-based Attacks with TLB Desynchronized Reverse Engineering

Andrei Tatar

<u>Daniël Trujillo</u>

Cristiano Giuffrida

Herbert Bos

VRIJE UNIVERSITEIT AMSTERDAM

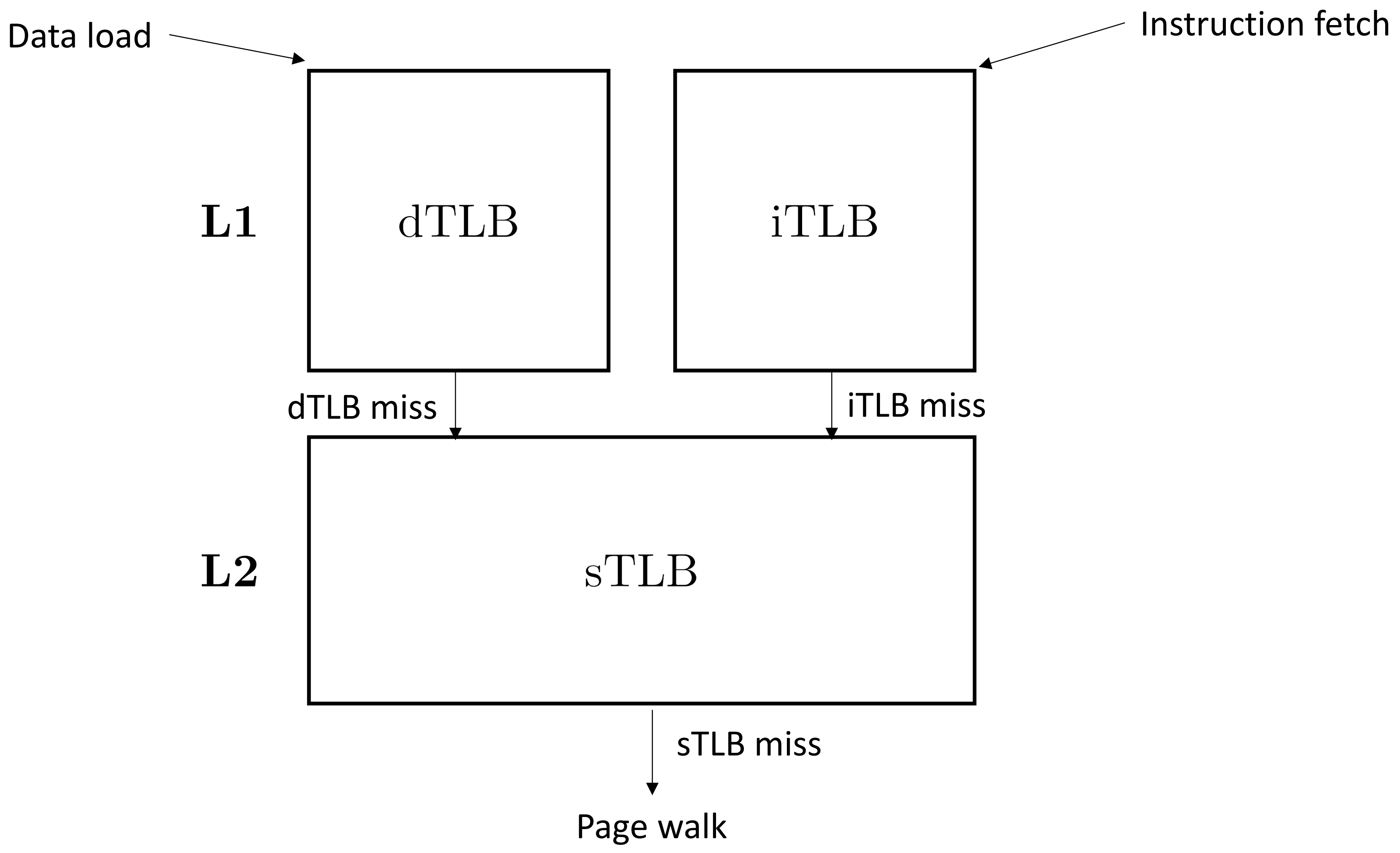
Teaser

- Introduce a novel way to reverse engineer Translation Lookaside Buffers (TLBs)
- Previously undocumented TLB properties on Intel® CPUs
  - Including a novel replacement policy!
- Improve previously proposed TLB-based attacks

Virtual Memory & the TLB

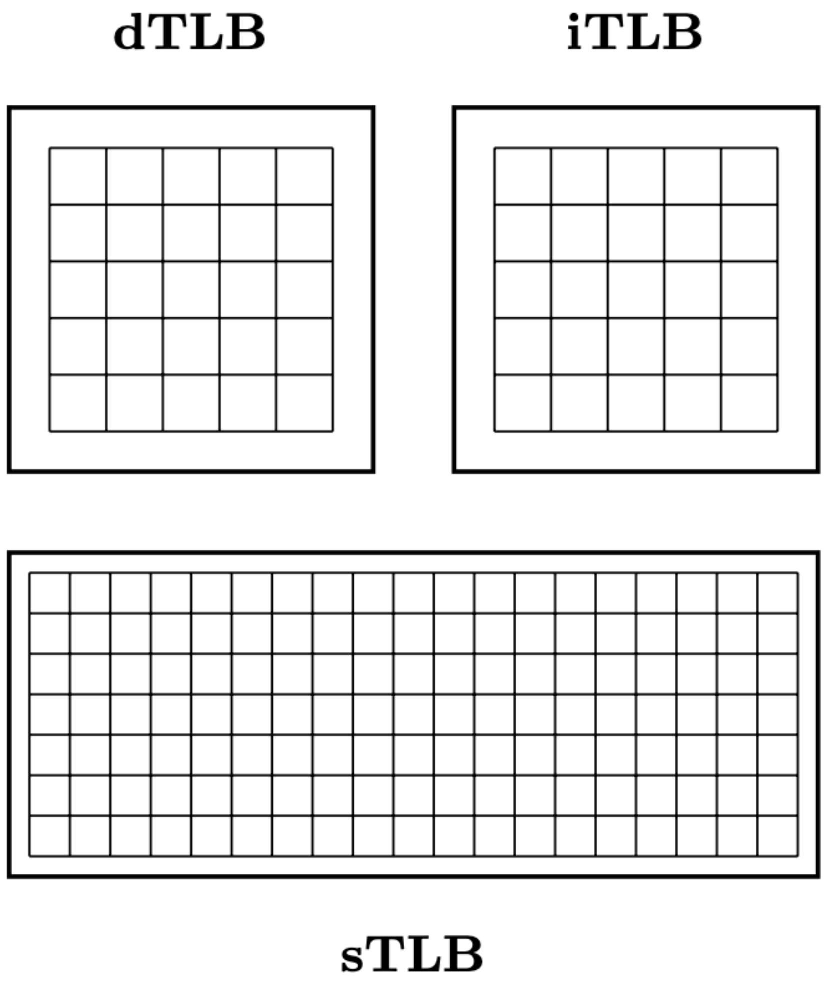
- CPUs support virtual memory
- Translation to physical memory on page granularity (e.g. 4KB)
  - Page tables denote virtual memory to physical memory mappings
- MMU performs page walk
  - Involves up to 4 memory accesses to page tables
- **Translation Lookaside Buffer** (**TLB**) caches recent translations (PTEs)
  - If **TLB hit**, we avoid expensive page walk
  - If **TLB miss**, we still have to do page walk

Intel® TLBs: Topology



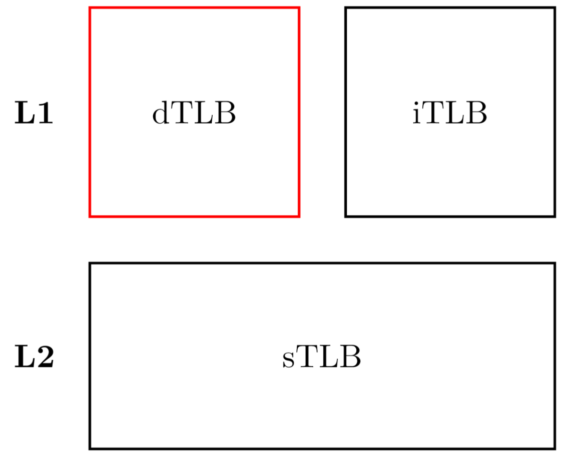
TLB Sets

- TLBs are organized in sets
- Each set has *W* ways
- Translation entries can occupy any of the *W* ways
- Each virtual address deterministically maps to one set using a **hash function**



Previous work by Gras et al.

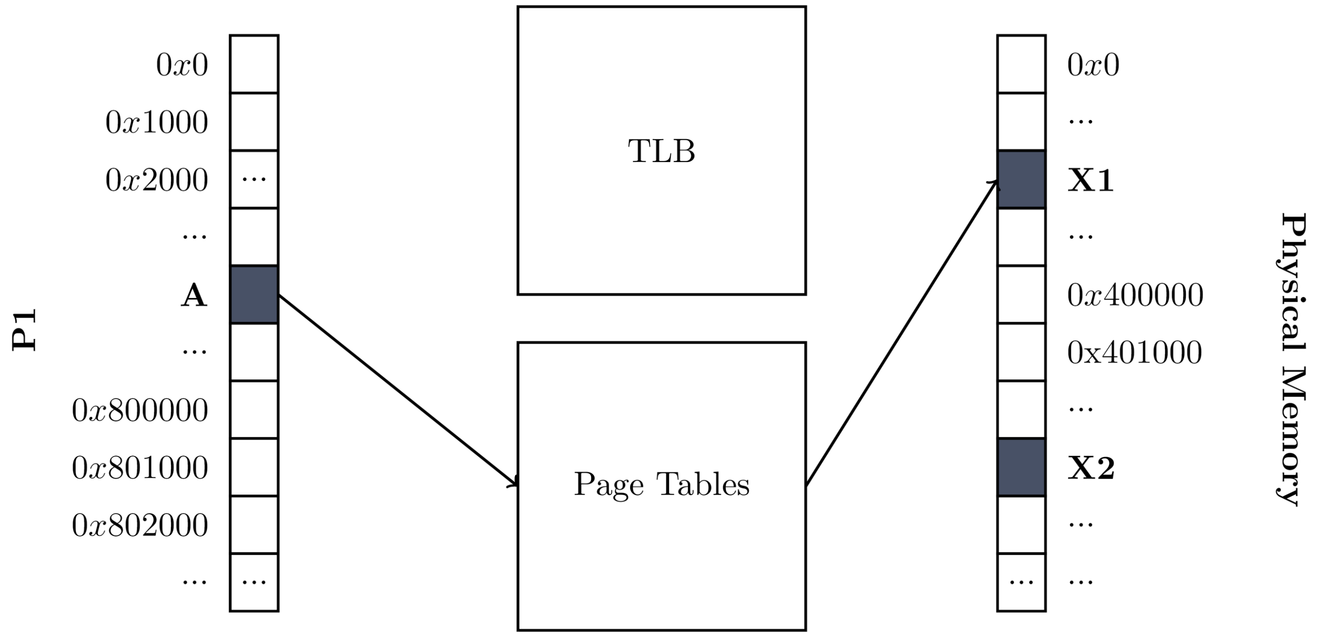
- TLB is potentially shared between mutually distrusting parties!
  - TLBs are typically even shared across hyperthreads
- If TLB state depends on secret, then this secret can be leaked
  - The TLB state can be sampled by timing accesses
- TLBleed: leaks cryptographic key using dTLB
- However, many TLB properties remained unknown



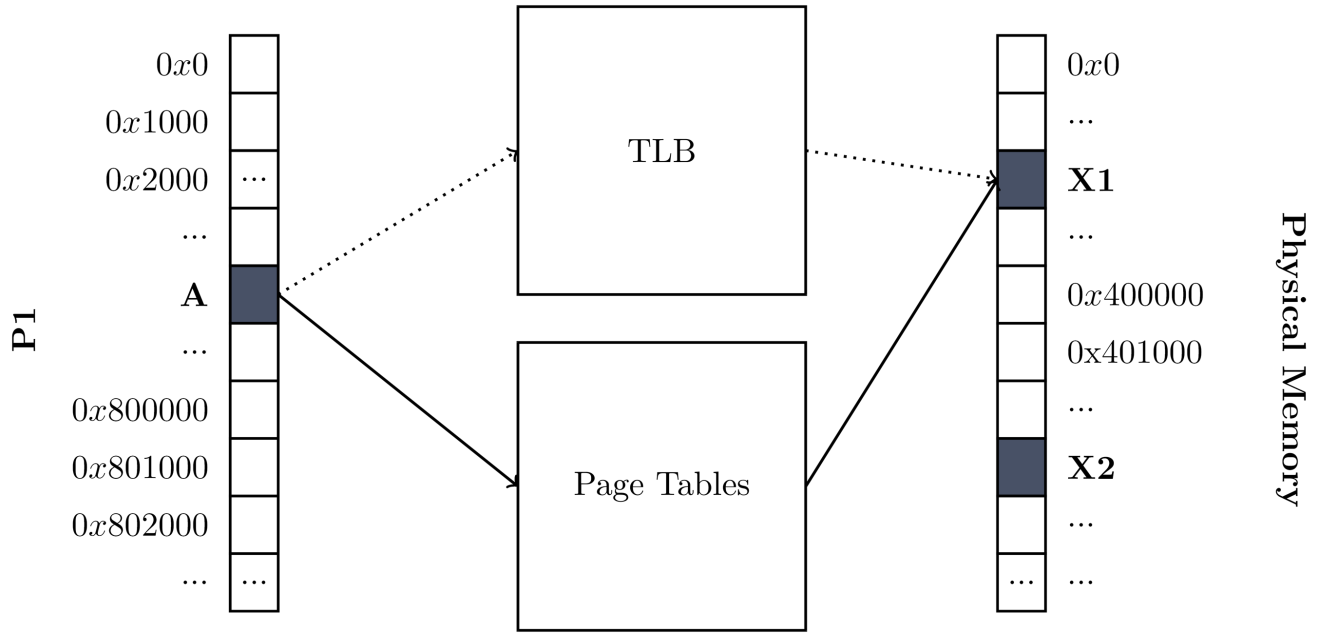
TLB Desynchronization

- When page tables are changed, TLB requires invalidation
  - TLBs are non-coherent with in-memory page tables
- Failing to invalidate could lead to serious bugs
  - Read or execute from the wrong physical address!

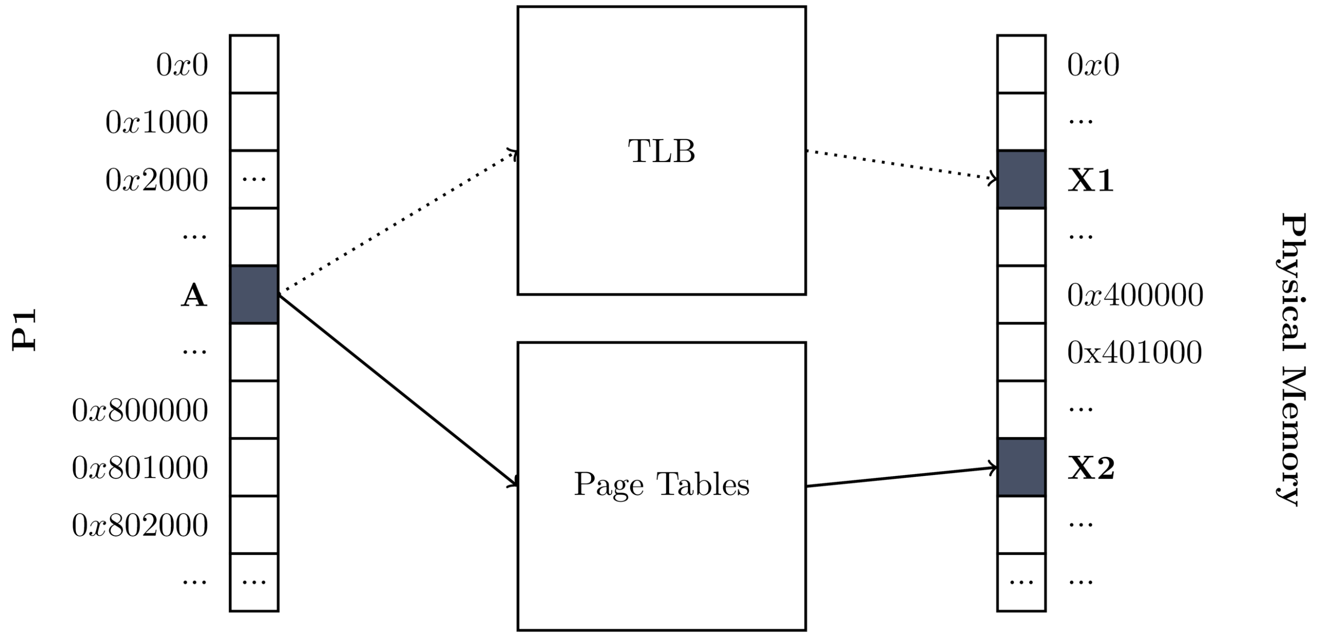
TLB Desynchronization: Leverage Desynced TLBs



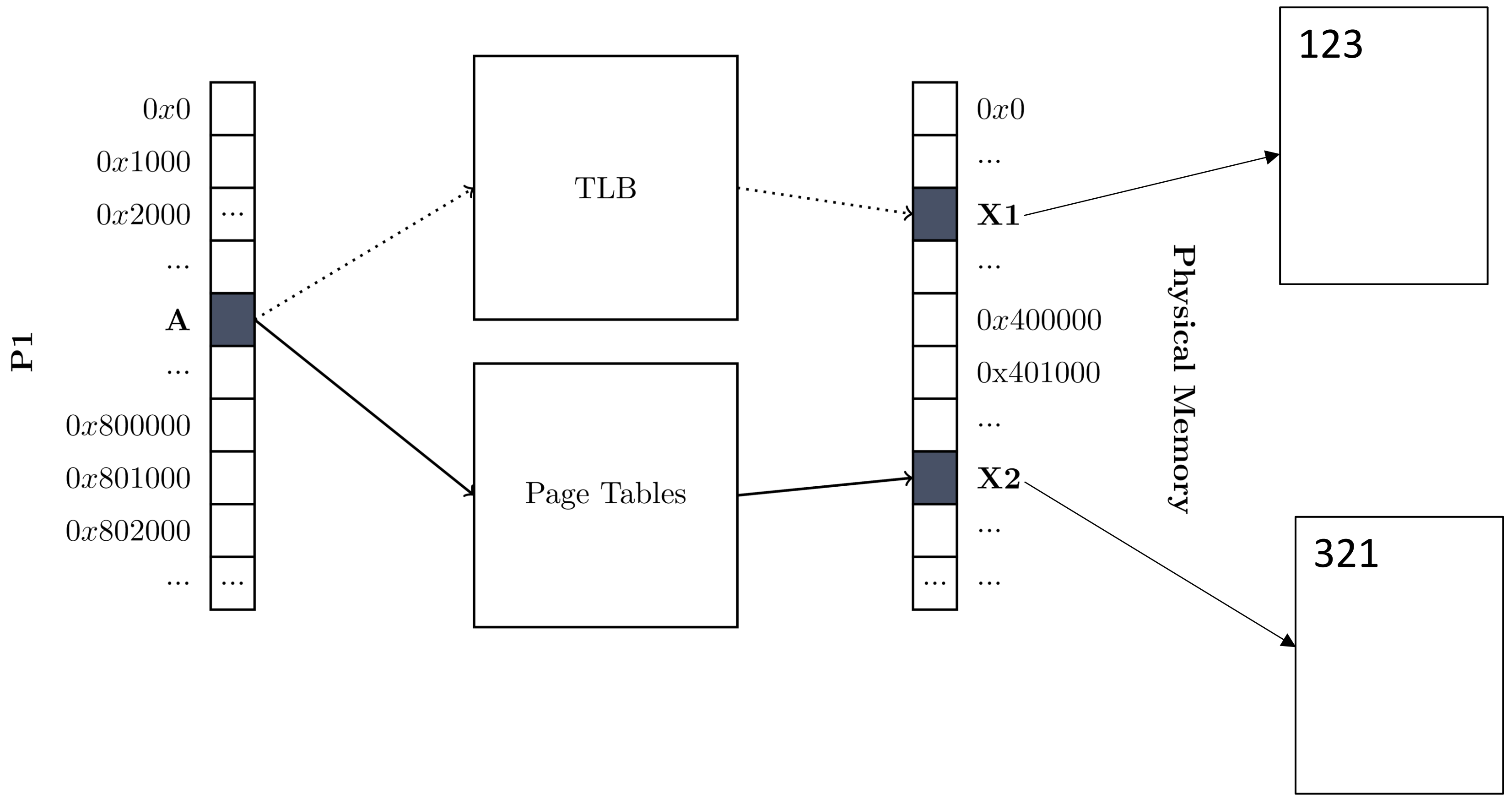
TLB Desynchronization: Leverage Desynced TLBs



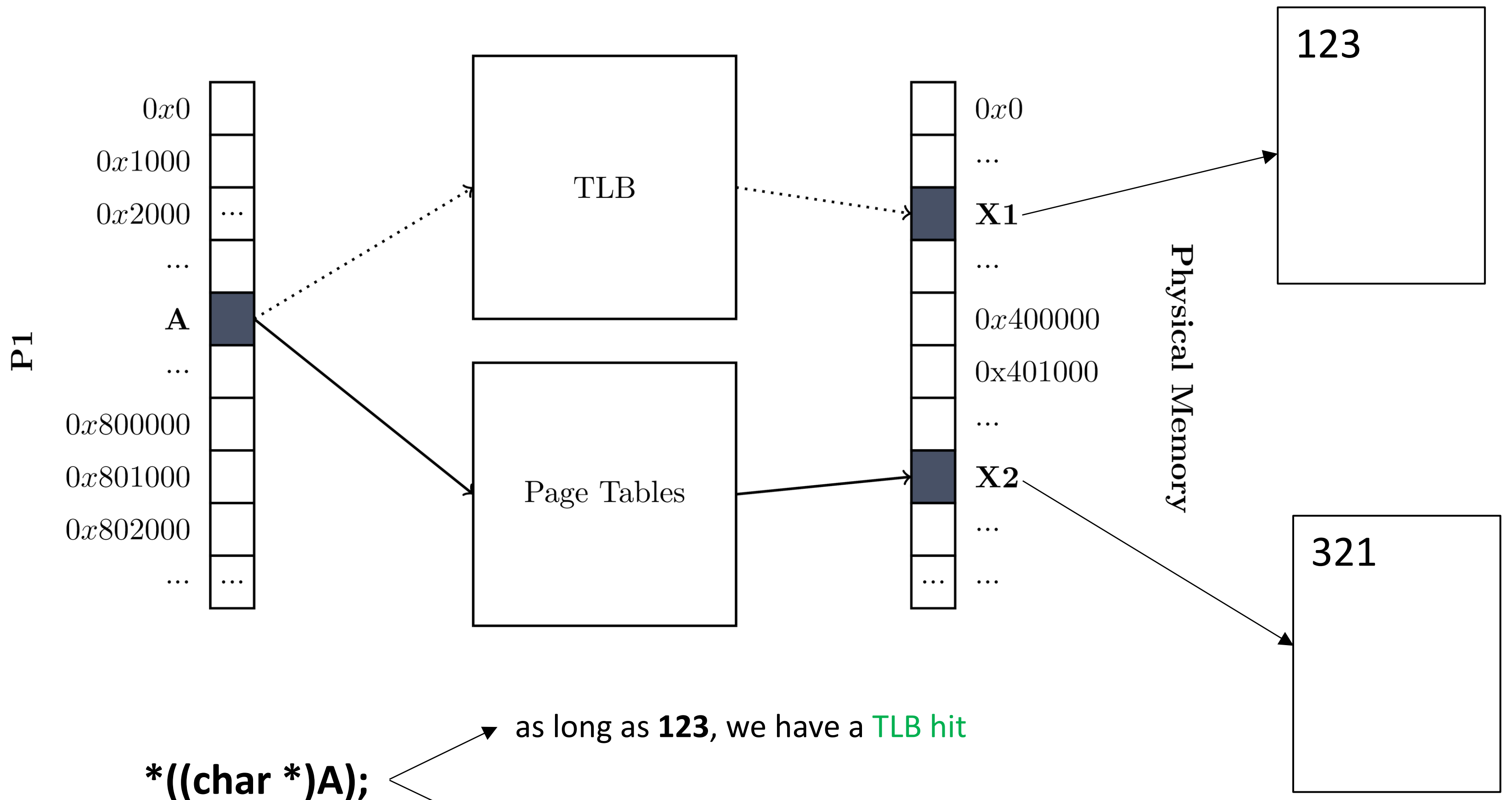
TLB Desynchronization: Leverage Desynced TLBs



TLB Desynchronization: Leverage Desynced TLBs



TLB Desynchronization: Leverage Desynced TLBs

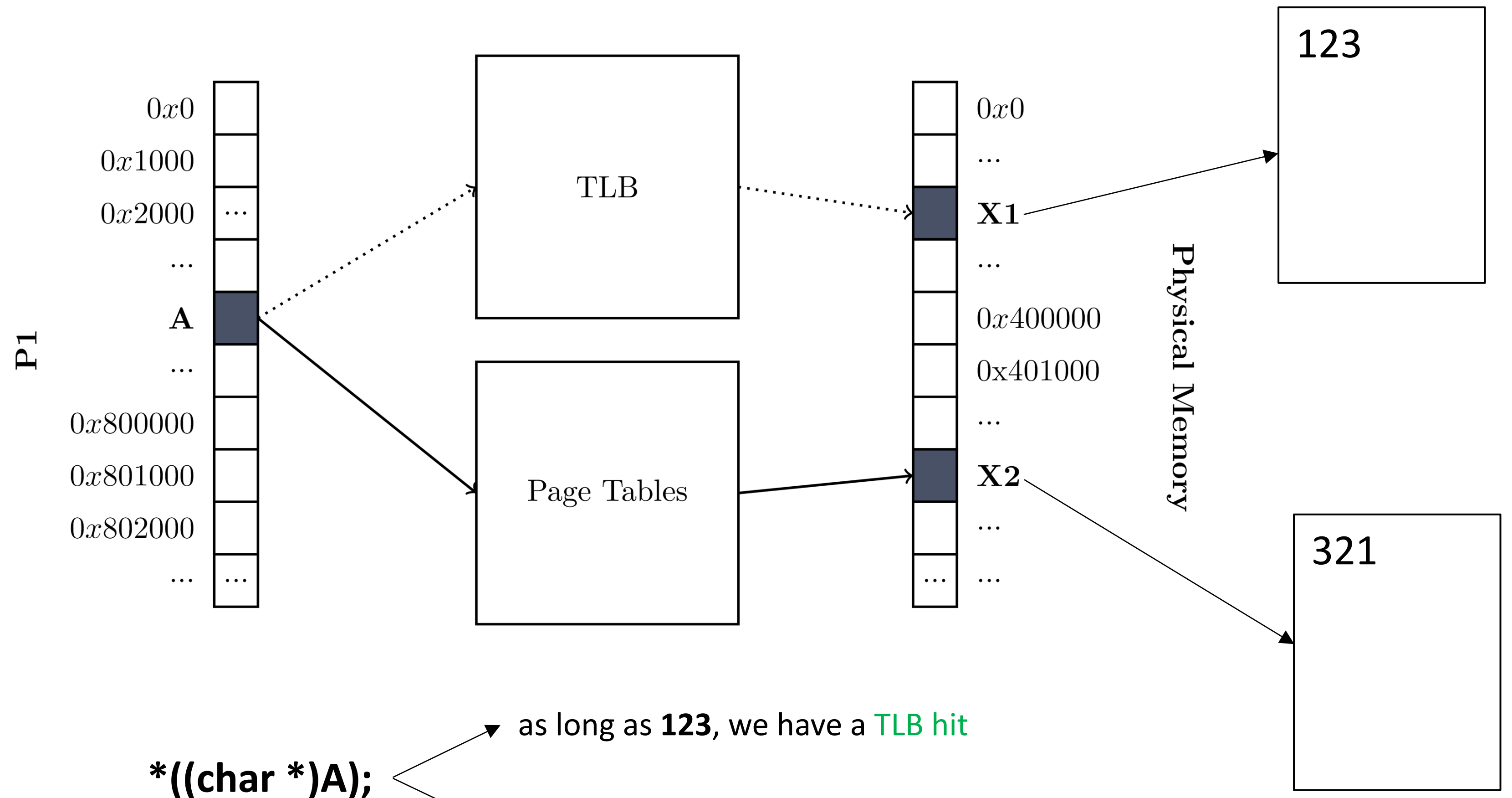


***((char *)A);**

as long as **123**, we have a TLB hit

as soon as **321**, we have a TLB miss

TLB Desynchronization: Leverage Desynced TLBs



***((char *)A);**

- as long as **123**, we have a TLB hit
- as soon as **321**, we have a TLB miss

TLB Desynchronization allows reliable TLB hit detection!

We can reverse engineer the TLB with this primitive

Reverse Engineering: Results on Intel®

- Inclusion policies
  - Intel TLBs are non-inclusive & non-exclusive
- Insertion policies
  - After page walk: inserts in both L1 and L2
  - After L2 sTLB hit: inserts in L1
  - After L1 eviction: no L2 sTLB insertion (not a victim cache)
- Set Sizes & Set Mapping
  - Two types of hash functions (linear and XOR)
  - Mostly in line with previous work
- Replacement policies

Reverse Engineering: Results on Intel®

- Inclusion policies
  - Intel TLBs are non-inclusive & non-exclusive

- Insertion policies
  - After page walk: inserts in both L1 and L2
  - After L2 sTLB hit: inserts in L1
  - After L1 eviction: no L2 sTLB insertion (not a victim cache)

- Set Sizes & Set Mapping
  - Two types of hash functions (linear and XOR)
  - Mostly in line with previous work

- Replacement policies

Reverse Engineering: Replacement Policies

- TLB sets eventually become full

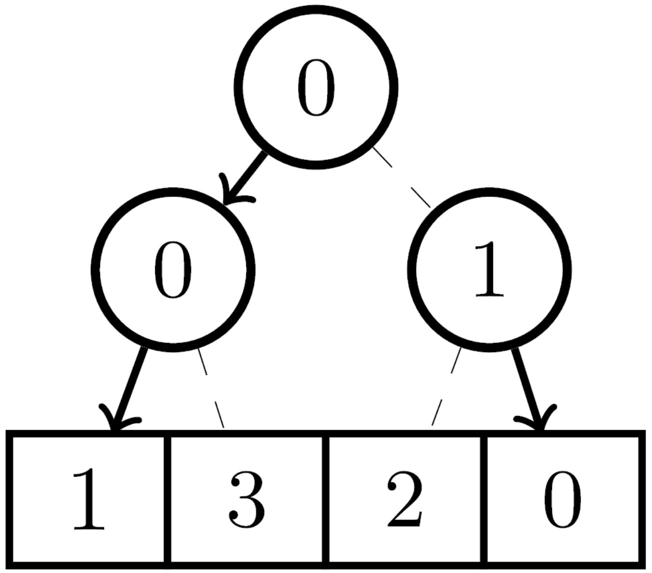
- Replacement policy decides the **victim** for eviction
  - Its goal is to maximize future TLB hits

- Three replacement policies active on Intel® TLBs
  - LRU
  - Tree-PLRU
  - $(\text{MRU}+1)_{\%3}\text{PLRU}_4$

Reverse Engineering: LRU & Tree-PLRU

- Least-Recently-Used (LRU)
  - Past is often a good approximation of the future
  - Found active on Ivy Bridge's iTLB

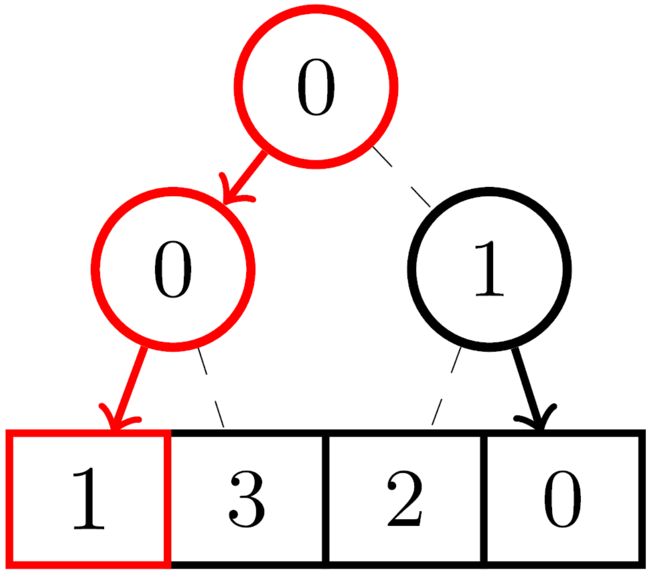
- Tree Pseudo LRU (Tree-PRLU)
  - Approximation of LRU
  - Binary tree with $W - 1$ bits
  - Victim pointed to by arrows
  - Found active on all dTLBs
  - Found active on sTLBs of older Intel® CPUs



Reverse Engineering: LRU & Tree-PLRU

- Least-Recently-Used (LRU)
  - Past is often a good approximation of the future
  - Found active on Ivy Bridge’s iTLB

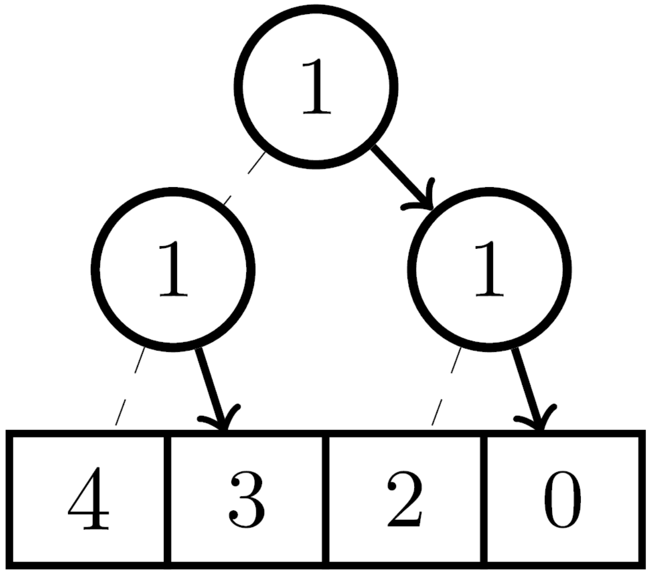
- Tree Pseudo LRU (Tree-PRLU)
  - Approximation of LRU
  - Binary tree with $W - 1$ bits
  - Victim pointed to by arrows
  - Found active on all dTLBs
  - Found active on sTLBs of older Intel® CPUs



Reverse Engineering: LRU & Tree-PLRU

- Least-Recently-Used (LRU)
  - Past is often a good approximation of the future
  - Found active on Ivy Bridge's iTLB

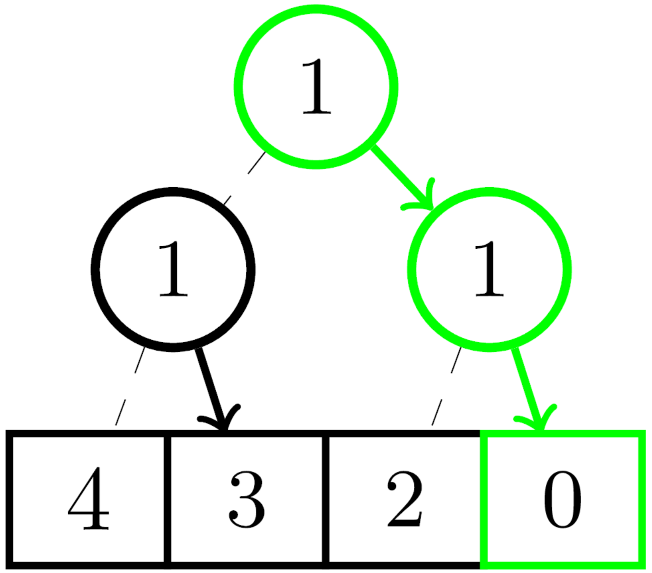
- Tree Pseudo LRU (Tree-PRLU)
  - Approximation of LRU
  - Binary tree with $W - 1$ bits
  - Victim pointed to by arrows
  - Found active on all dTLBs
  - Found active on sTLBs of older Intel® CPUs



Reverse Engineering: LRU & Tree-PLRU

- Least-Recently-Used (LRU)
  - Past is often a good approximation of the future
  - Found active on Ivy Bridge's iTLB

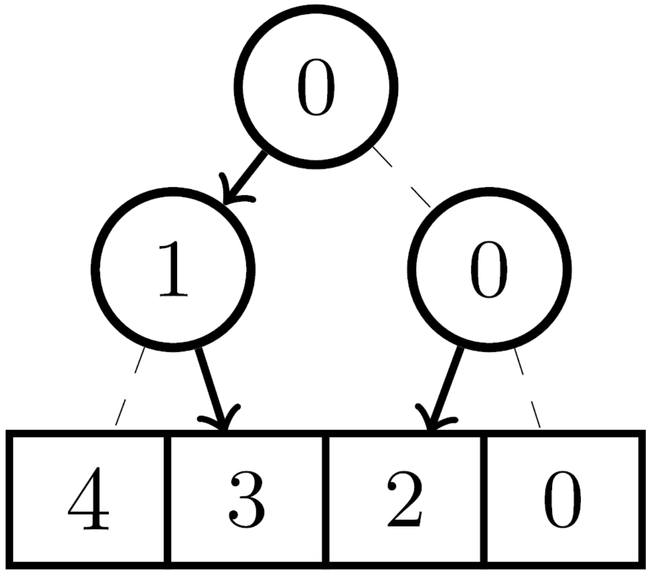
- Tree Pseudo LRU (Tree-PRLU)
  - Approximation of LRU
  - Binary tree with $W - 1$ bits
  - Victim pointed to by arrows
  - Found active on all dTLBs
  - Found active on sTLBs of older Intel® CPUs



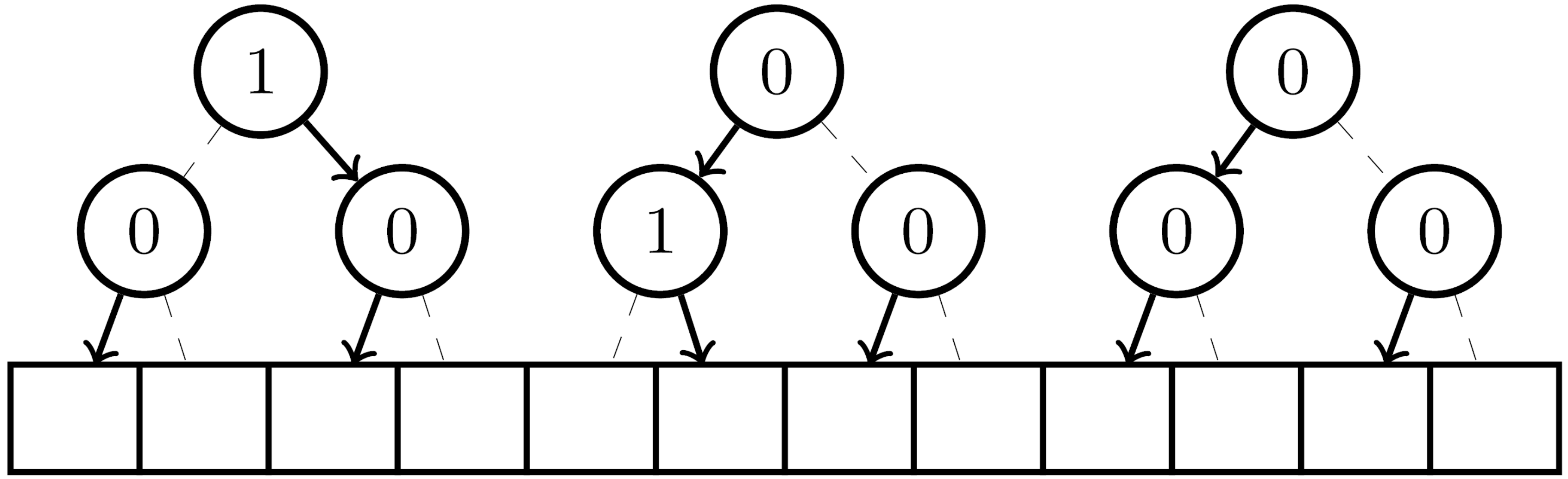
Reverse Engineering: LRU & Tree-PLRU

- Least-Recently-Used (LRU)
  - Past is often a good approximation of the future
  - Found active on Ivy Bridge's iTLB

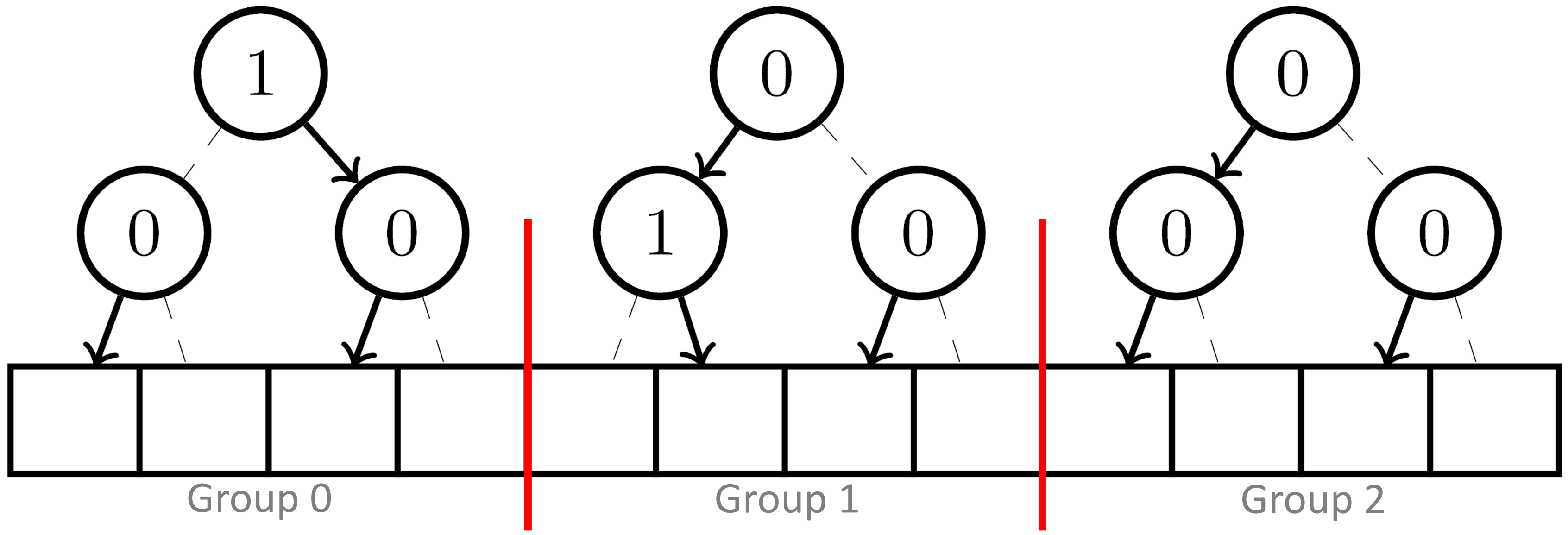
- Tree Pseudo LRU (Tree-PRLU)
  - Approximation of LRU
  - Binary tree with $W - 1$ bits
  - Victim pointed to by arrows
  - Found active on all dTLBs
  - Found active on sTLBs of older Intel® CPUs



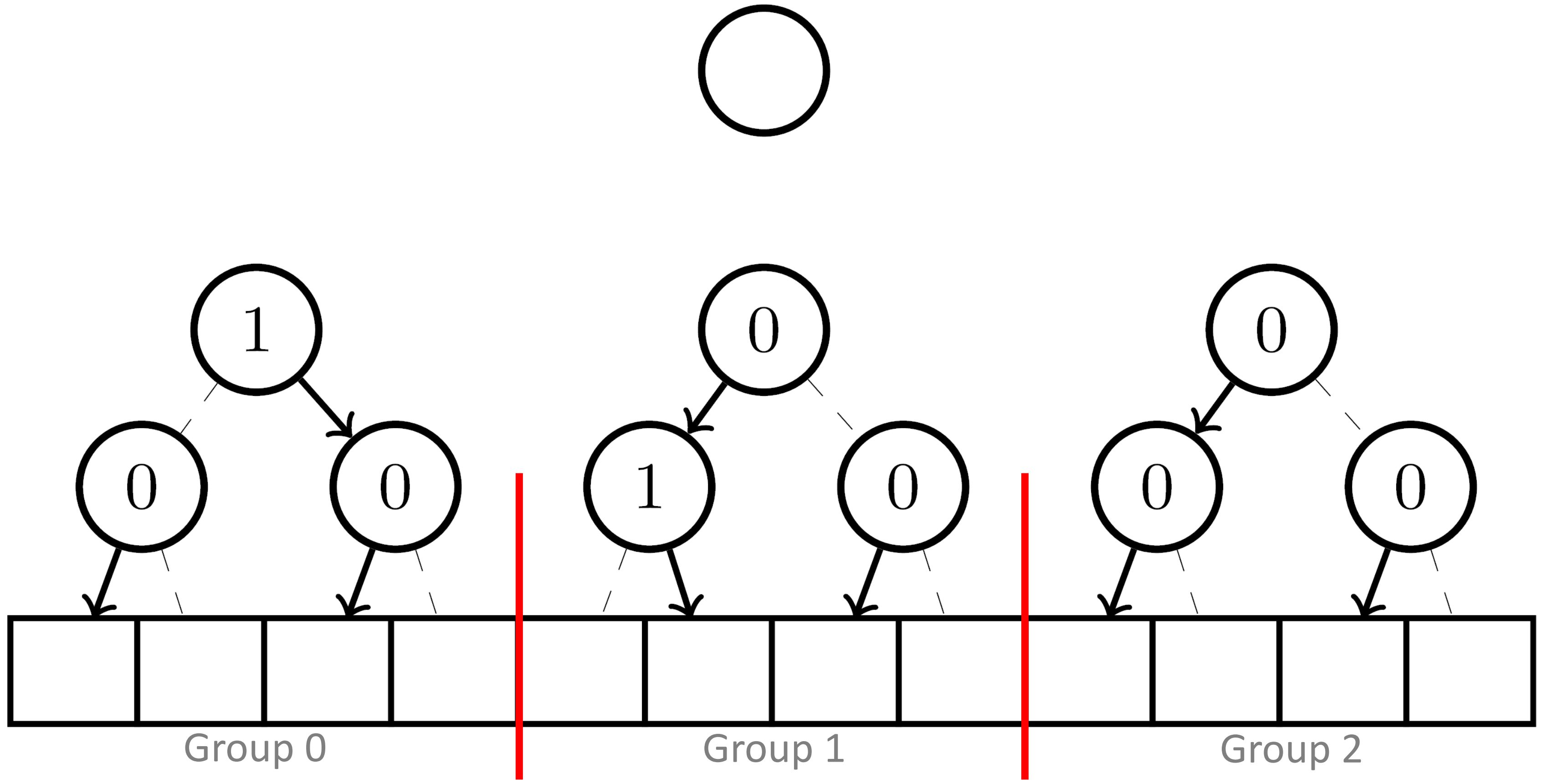
Reverse Engineering: $(\text{MRU}+1)_{\%3}\text{PLRU}_4$



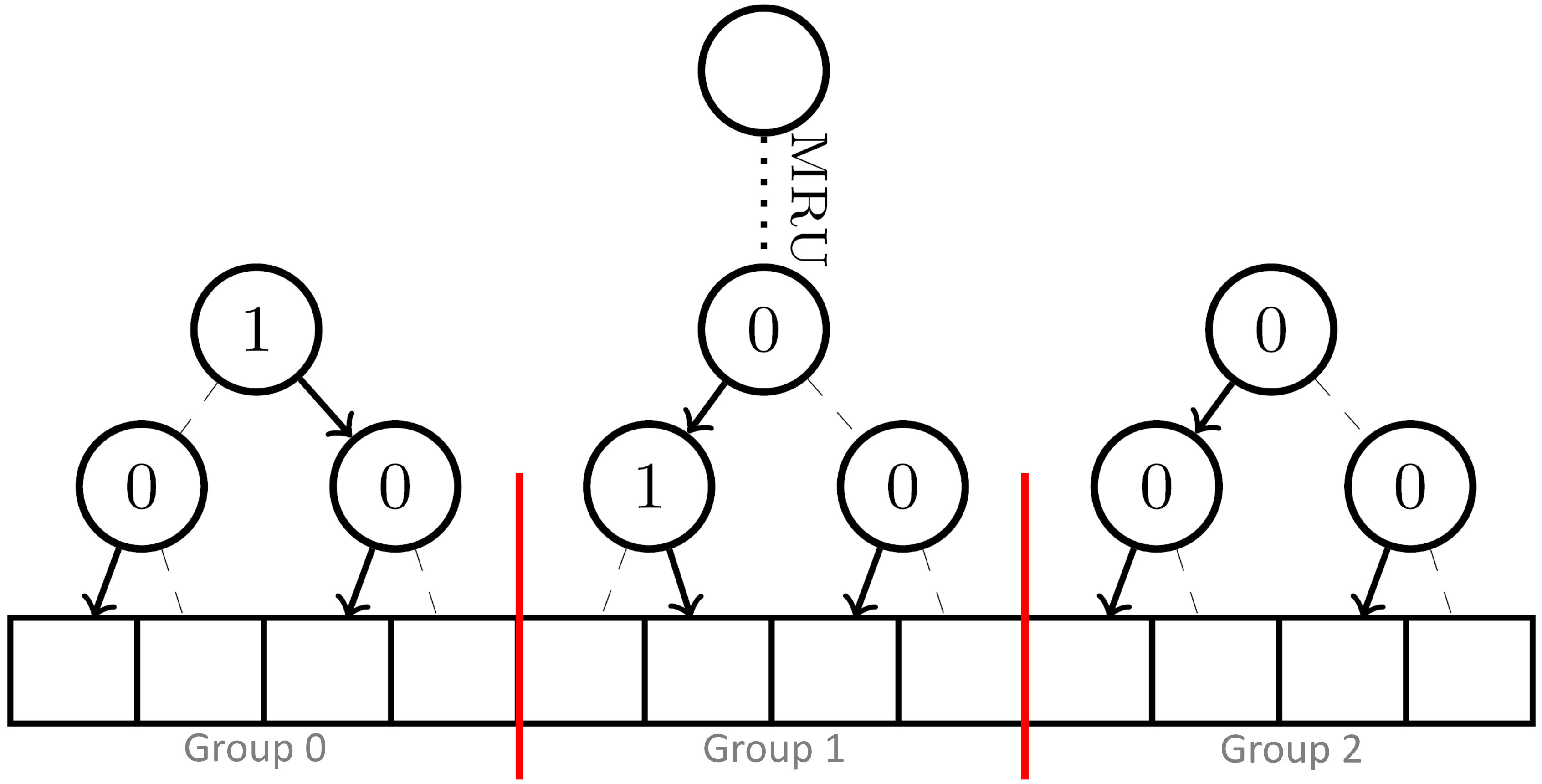
Reverse Engineering: $(MRU+1)_{\%3}PLRU_4$



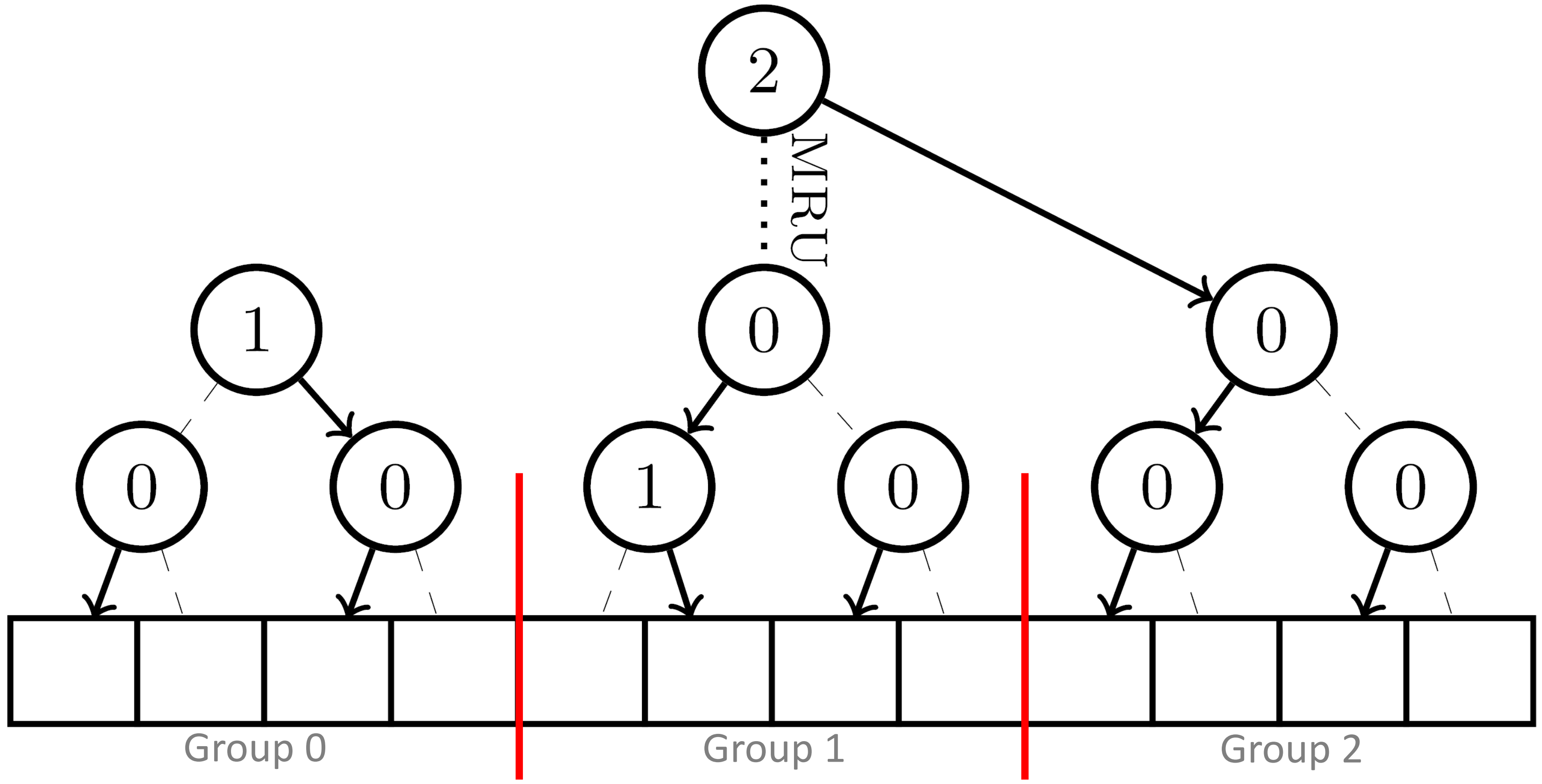
Reverse Engineering: $(\mathrm{MRU}+1)_{\%3}\mathrm{PLRU}_4$



Reverse Engineering: $(MRU+1)_{\%3}PLRU_4$

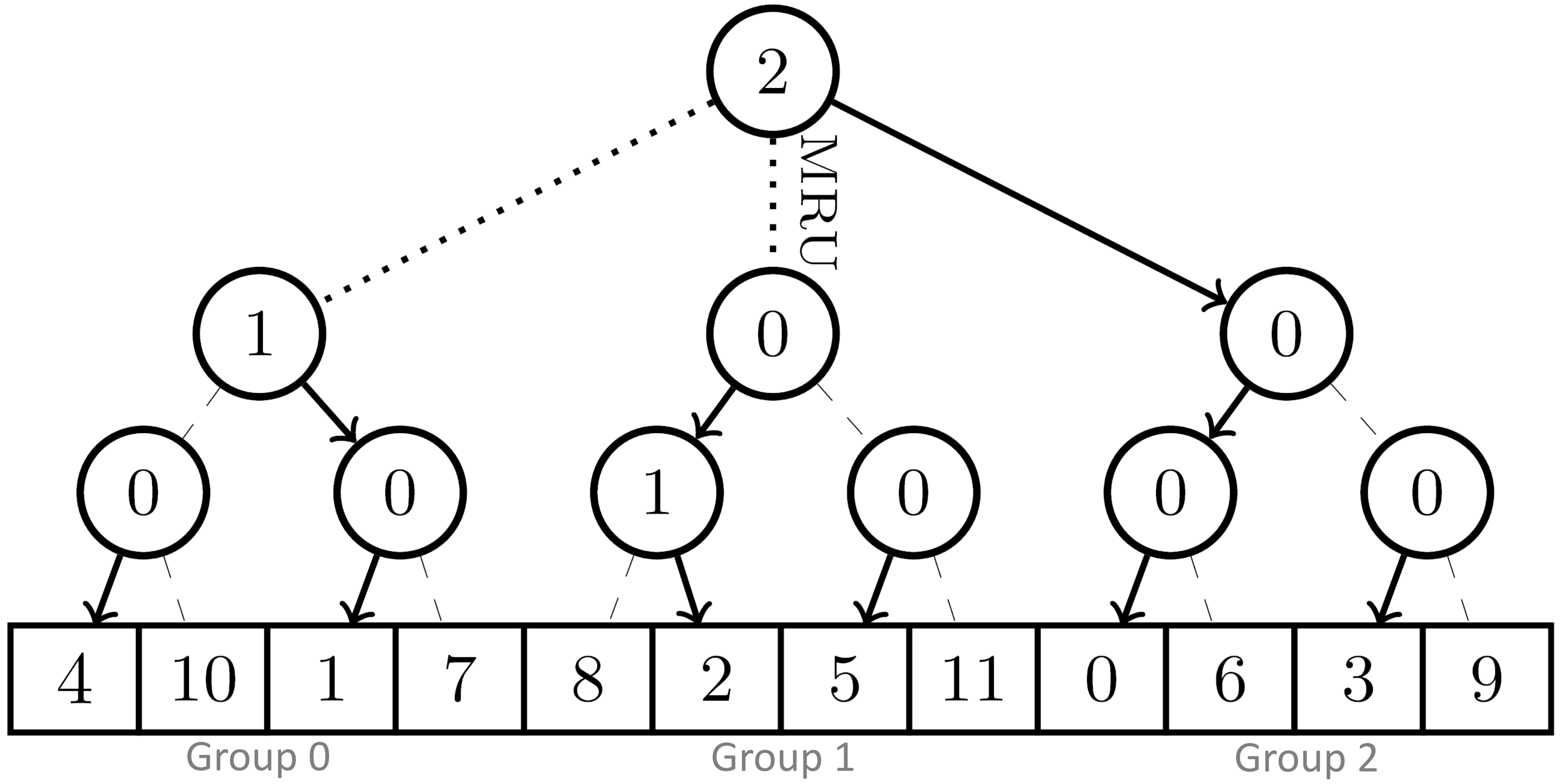


Reverse Engineering: $(\mathrm{MRU}+1)_{\%3}\mathrm{PLRU}_4$



Reverse Engineering: $(\text{MRU}+1)_{\%3}\text{PLRU}_4$

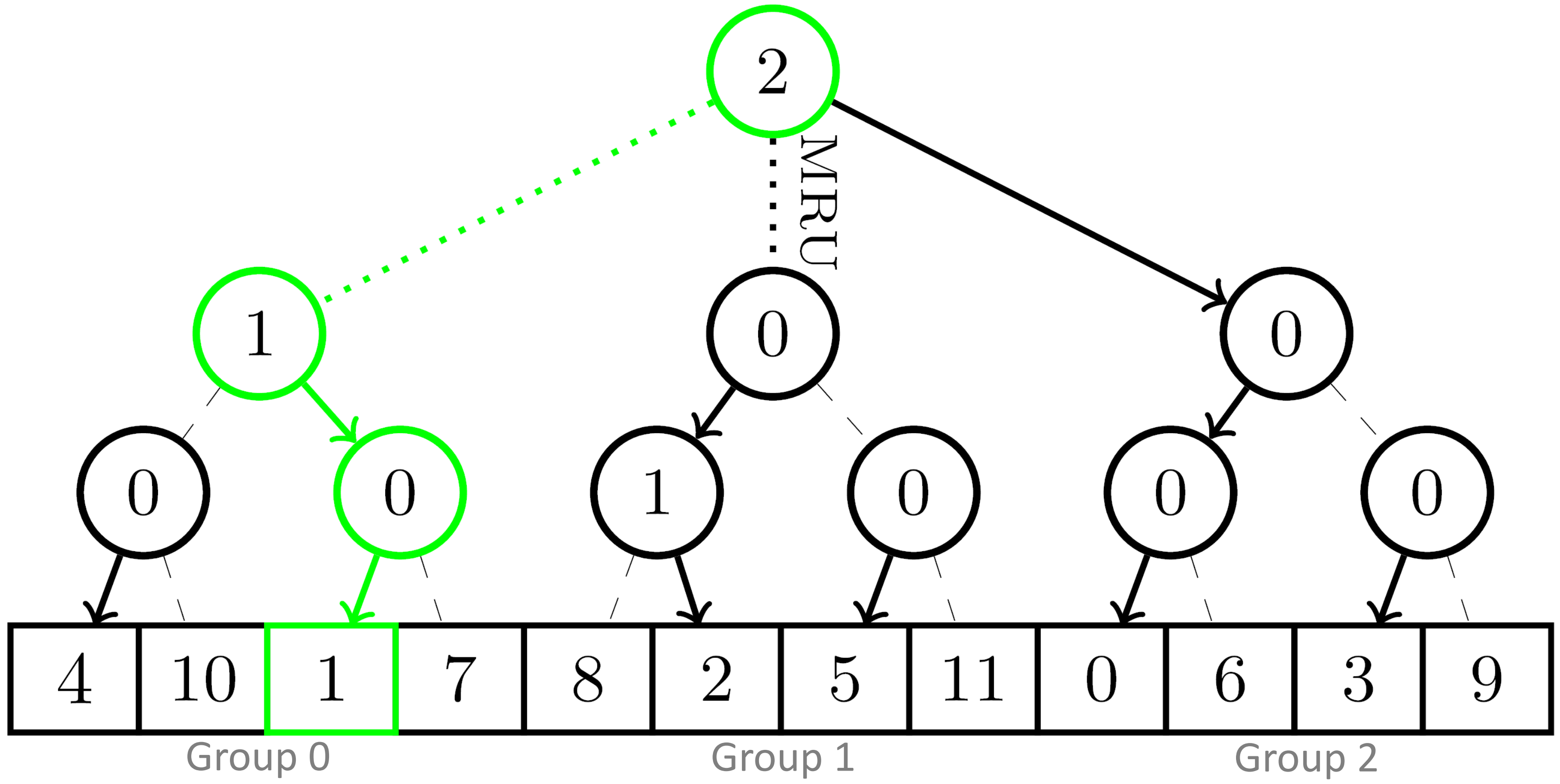
Example sequence: 1, 12



Reverse Engineering: $(\mathrm{MRU}+1)_{\%3}\mathrm{PLRU}_4$

$\Downarrow$

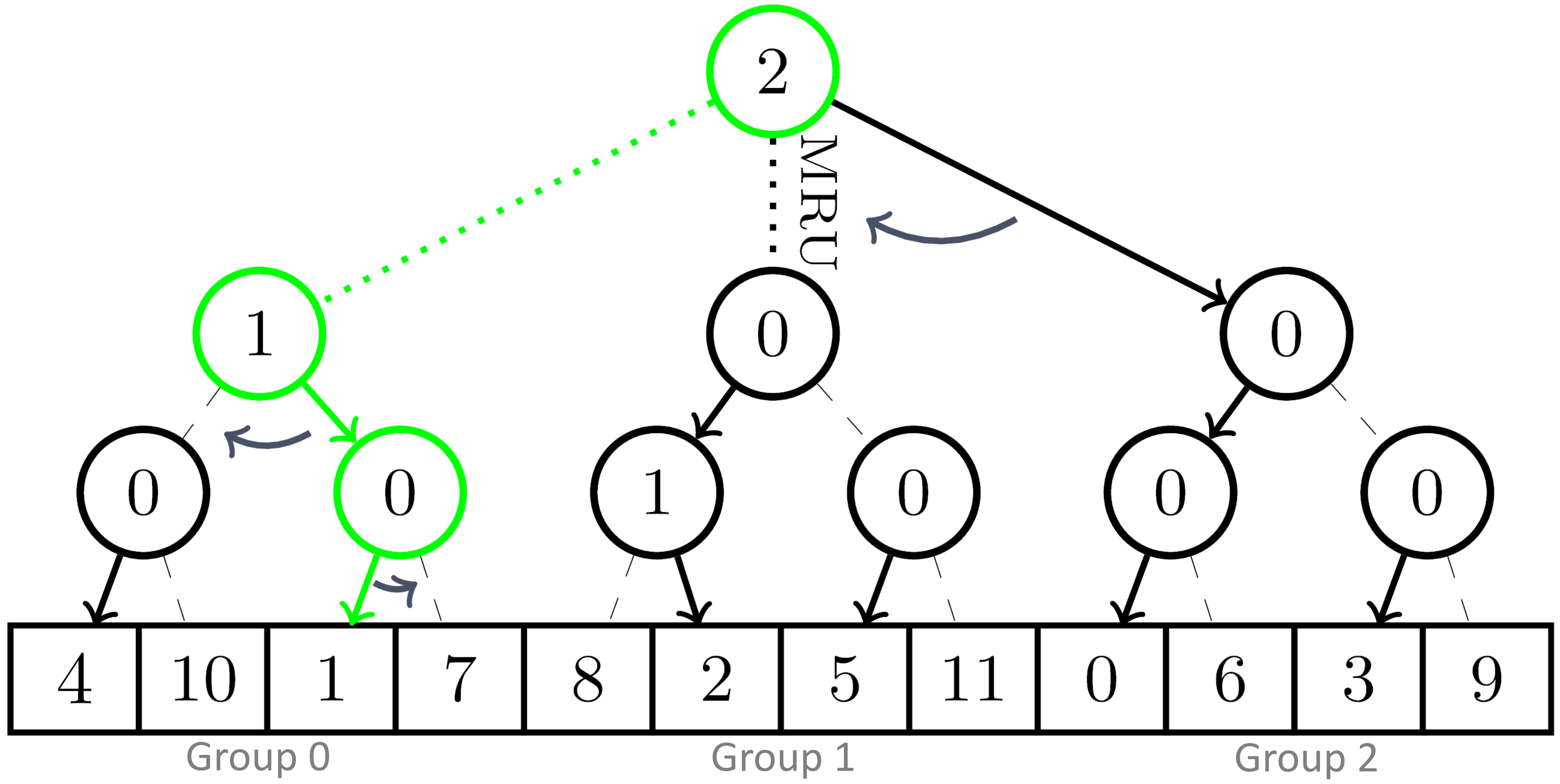
Example sequence: 1, 12



Reverse Engineering: $(\text{MRU}+1)_{\%3}\text{PLRU}_4$

$\Downarrow$

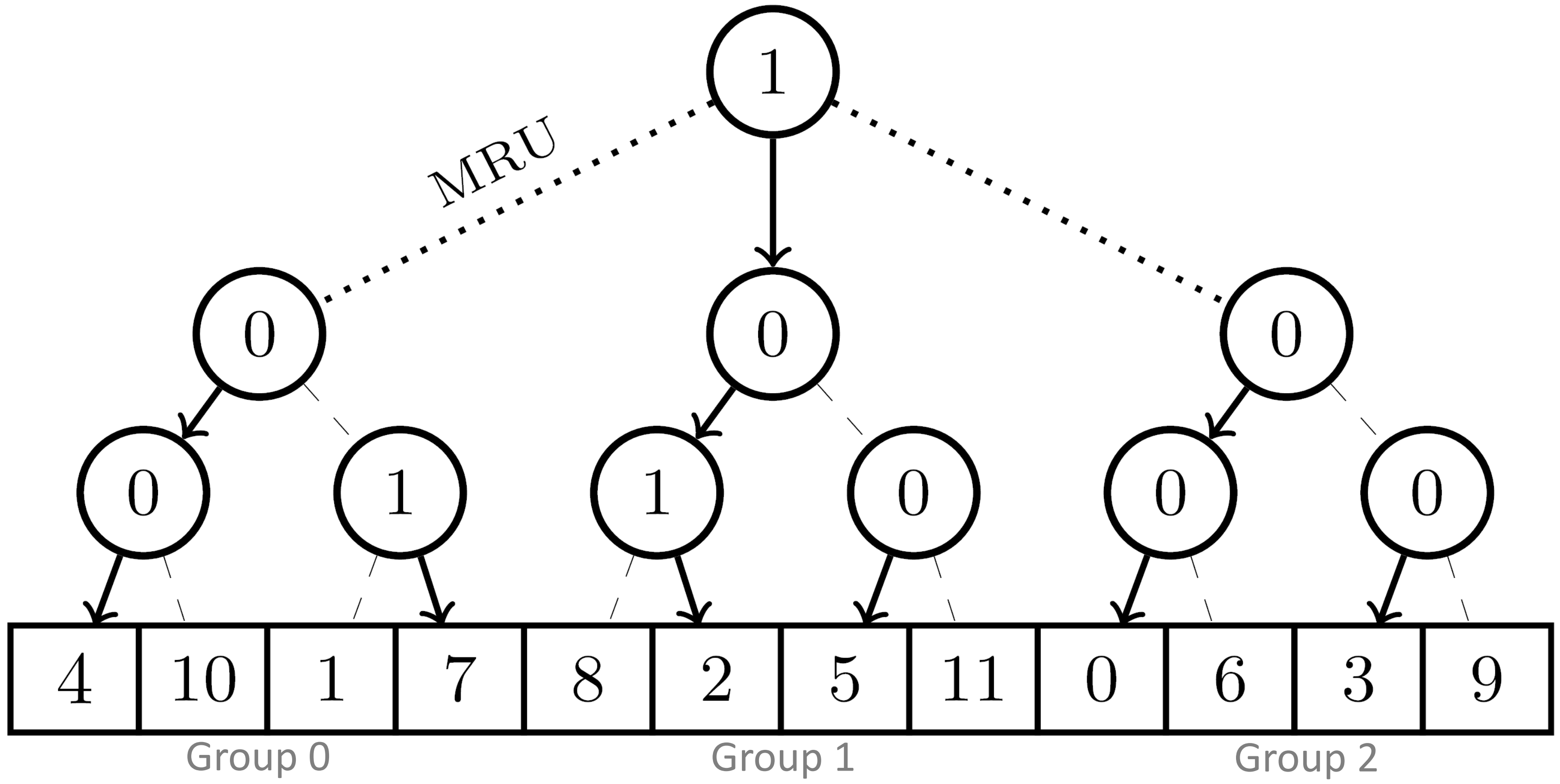
Example sequence: 1, 12



Reverse Engineering: $(\text{MRU}+1)_{\%3}\text{PLRU}_4$

$\Downarrow$

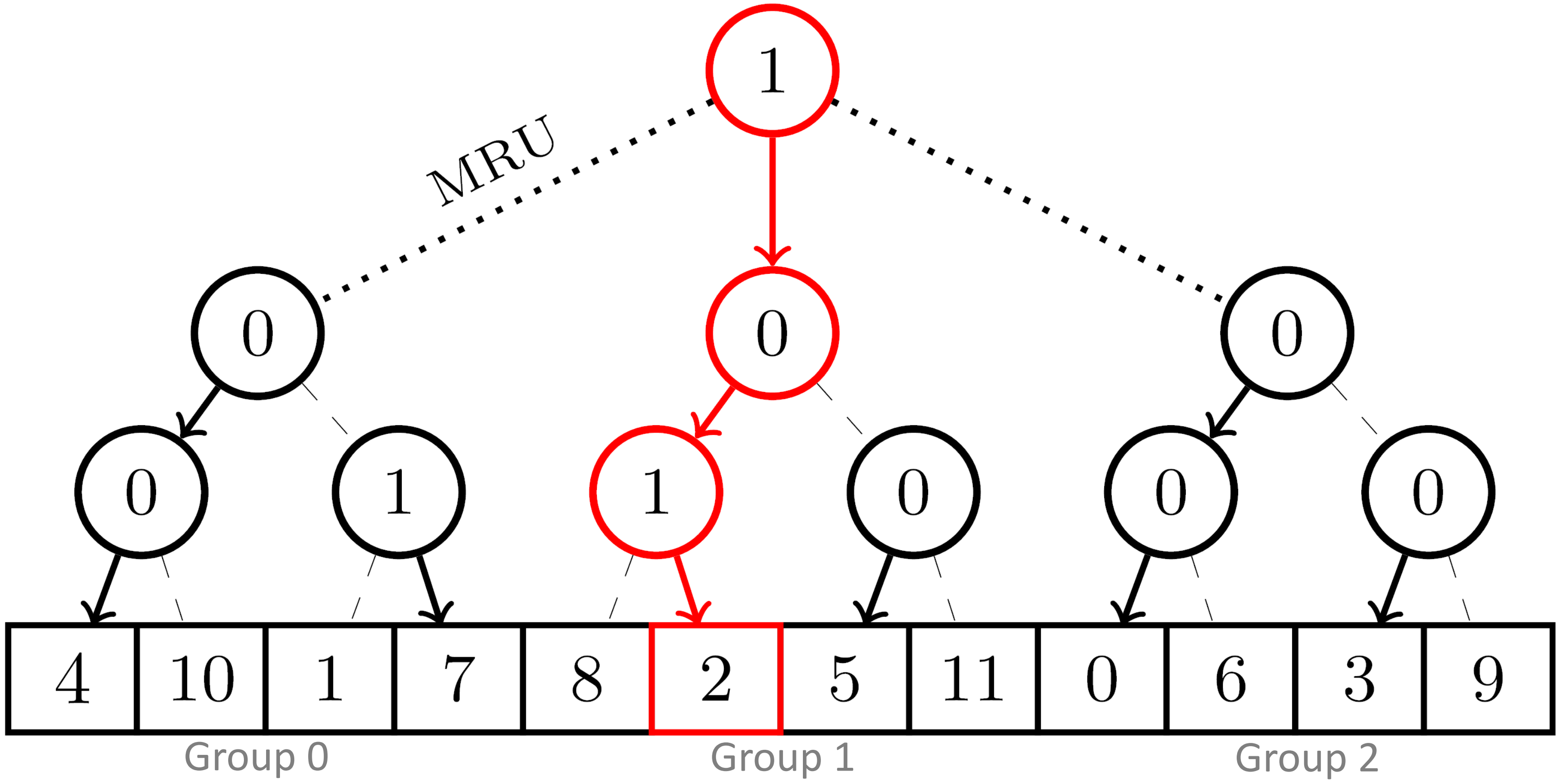
Example sequence: 1, 12



Reverse Engineering: $(\text{MRU}+1)_{\%3}\text{PLRU}_4$

$\Downarrow$

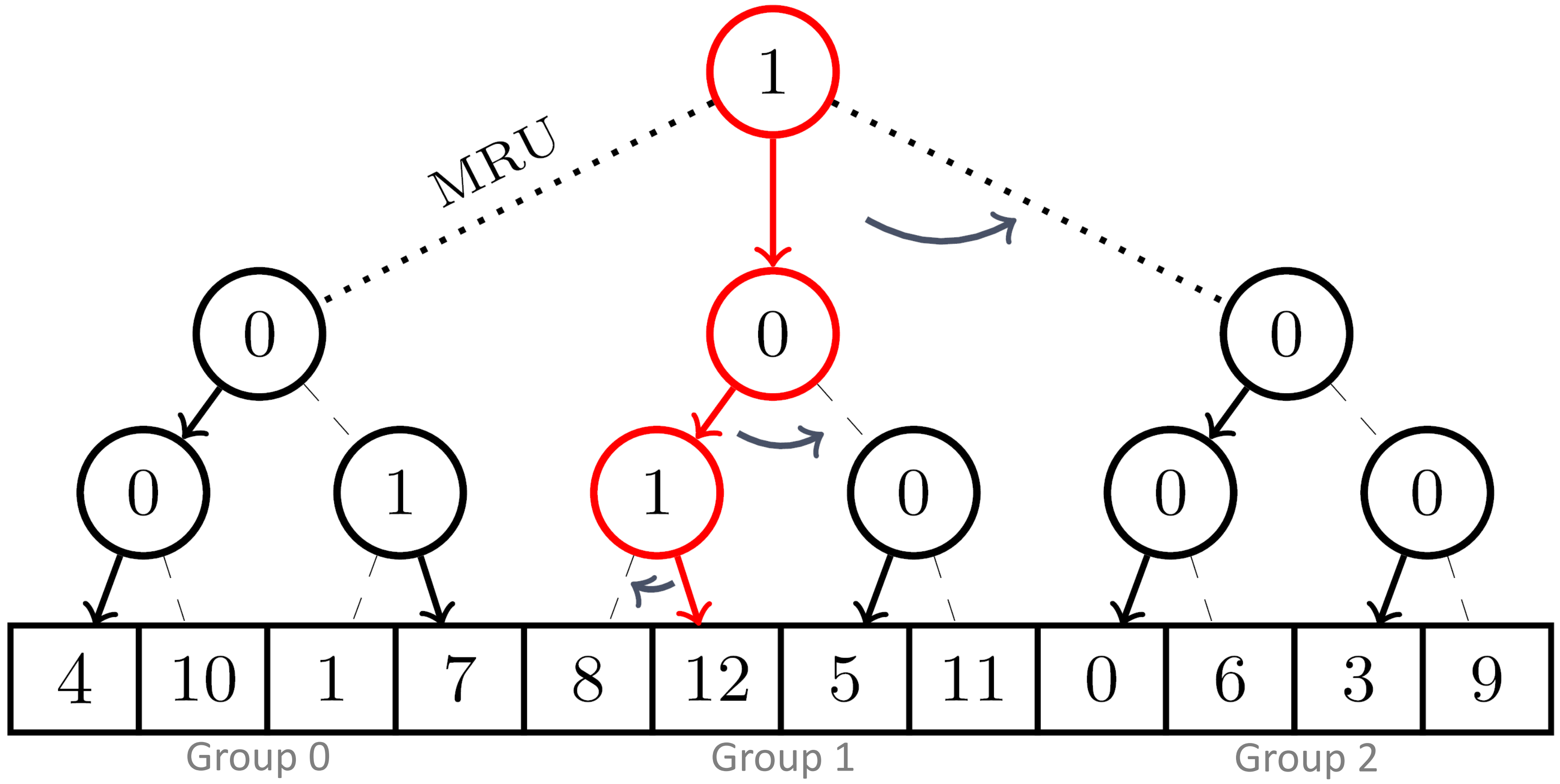
Example sequence: 1, 12



Reverse Engineering: $(MRU+1)_{\%3}PLRU_4$

$\Downarrow$

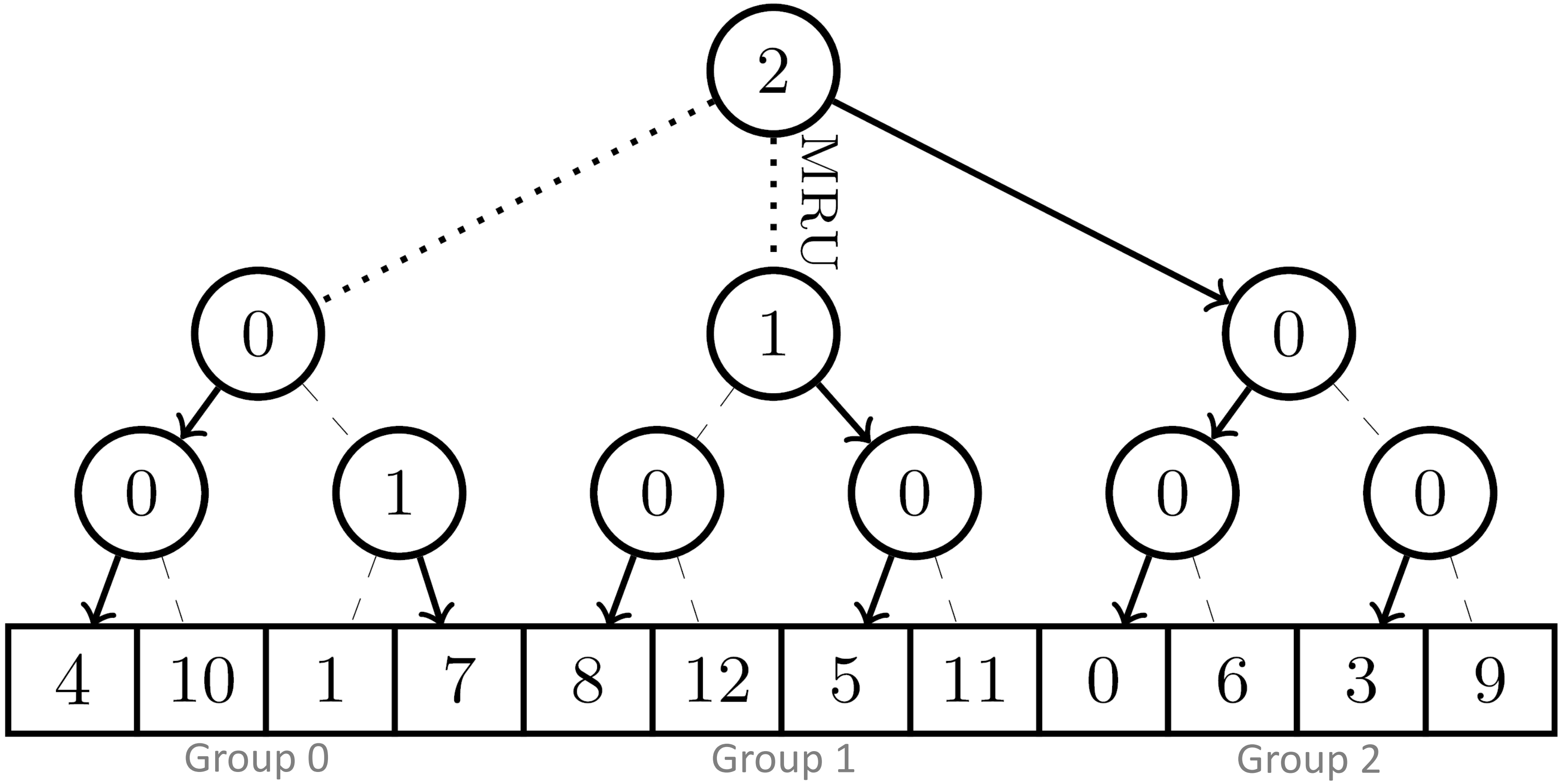
Example sequence: 1, 12



Reverse Engineering: $(MRU+1)_{\%3}PLRU_4$

$\Downarrow$

Example sequence: 1, 12



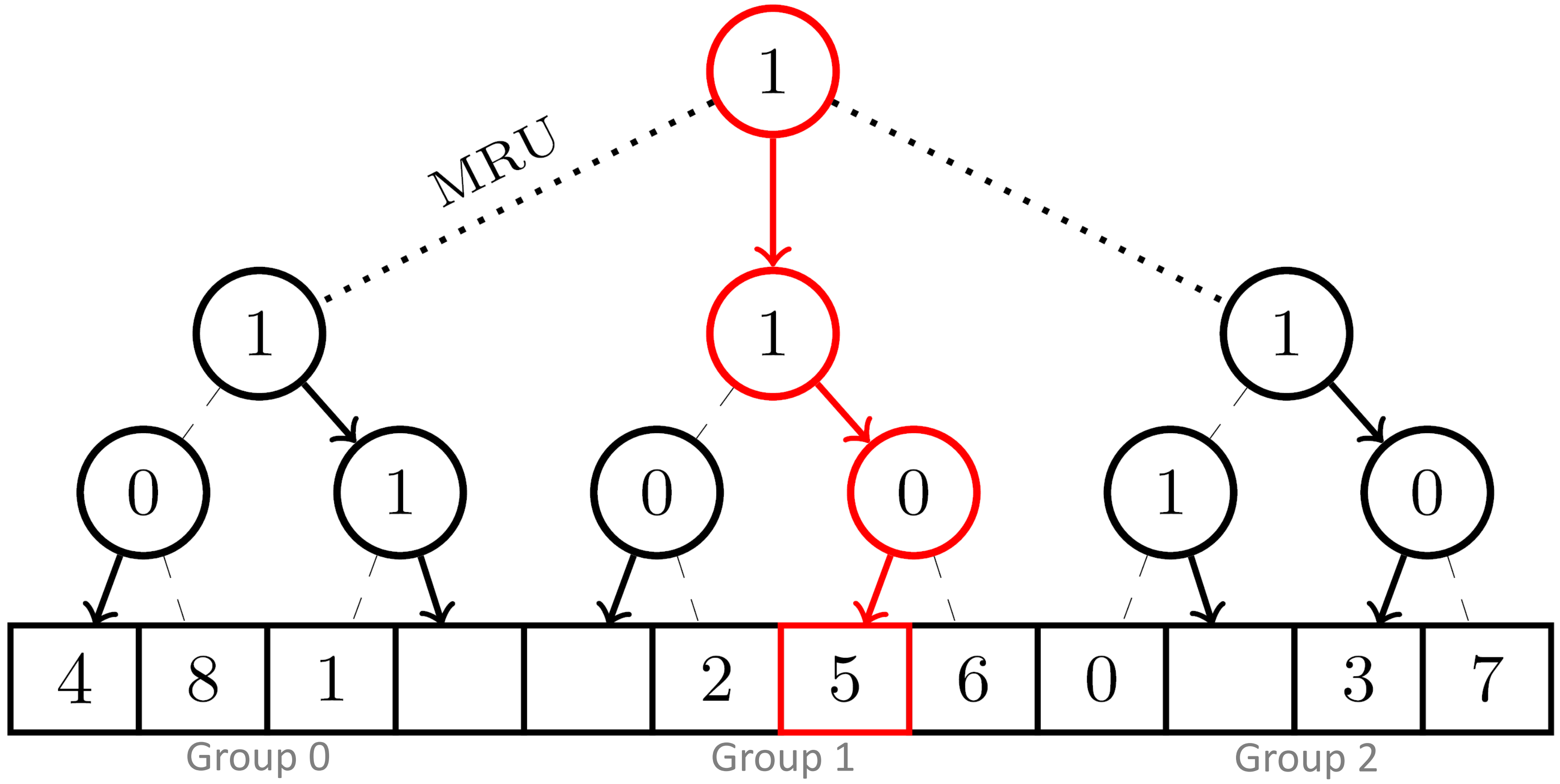
Improving TLB-based Attacks

- TLB-based attacks often need to evict one particular entry
  - TLBleed: to sample TLB, we need to evict translation of victim process to allow next sample
- The naive way: access *W* addresses to evict the entire set
- Knowledge of replacement policies allows for **optimized eviction**
- Self-synchronizing repeatable TLB access patterns

Improving TLB-based Attacks: $(MRU+1)_{\%3}PLRU_4$

<u>Optimized</u>

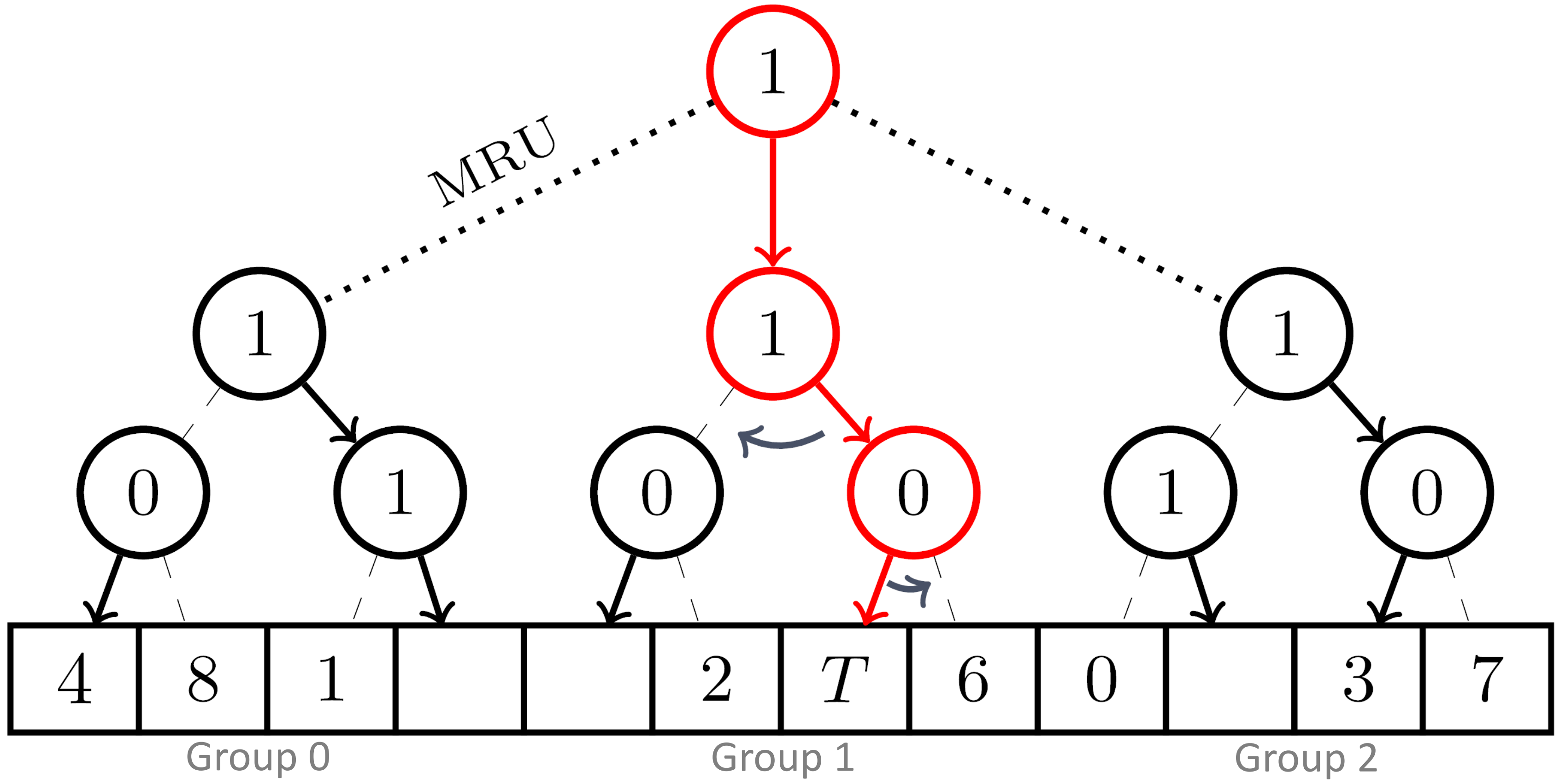
Insert T



Improving TLB-based Attacks: $(\text{MRU+1})_{\%3}\text{PLRU}_4$

Optimized

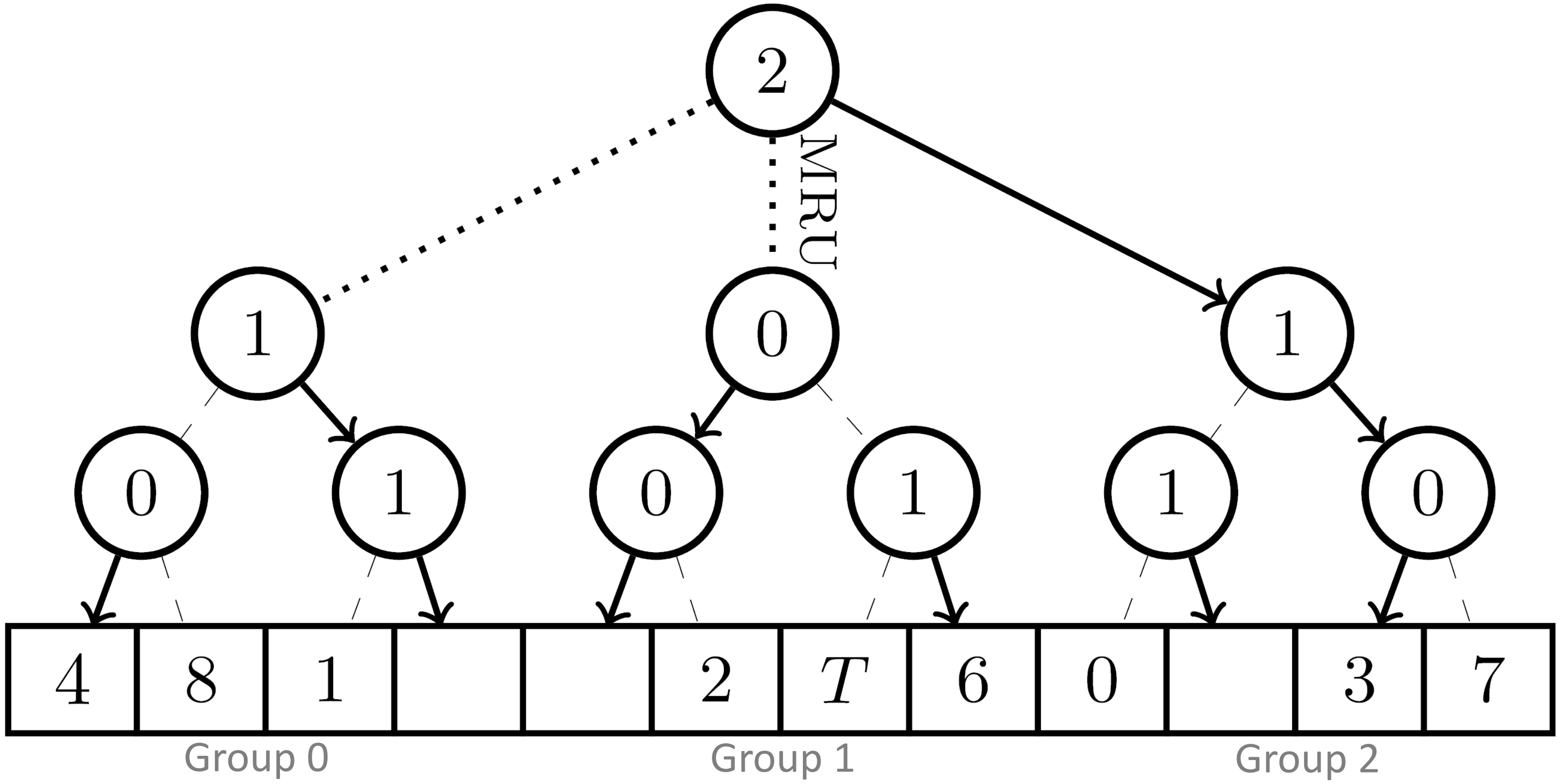
Insert T



Improving TLB-based Attacks: $(\text{MRU}+1)_{\%3}\text{PLRU}_4$

Optimized

Insert T

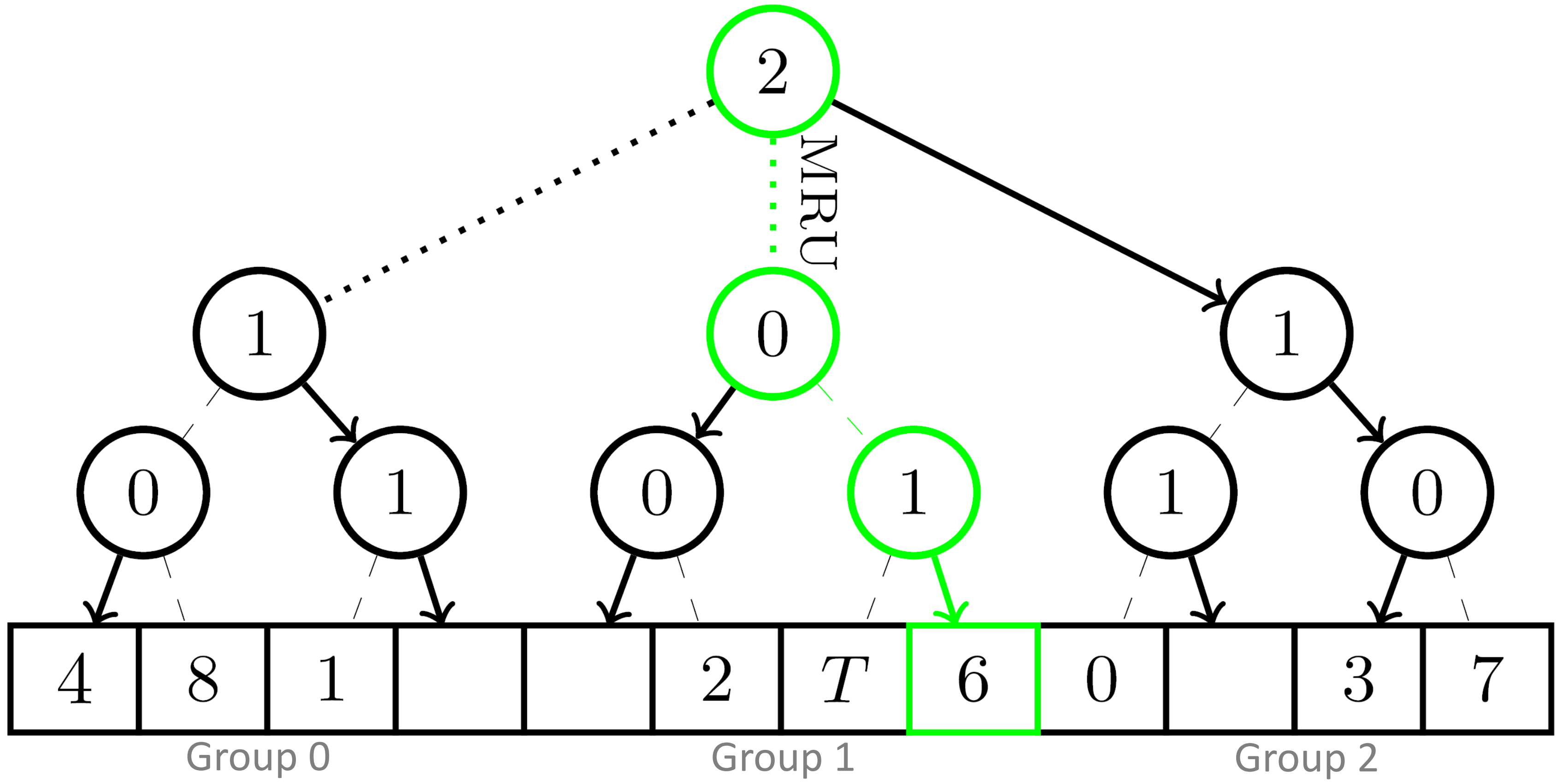


Improving TLB-based Attacks: $(MRU+1)_{\%3}PLRU_4$

Optimized

⇓

6, 2, 1, 5

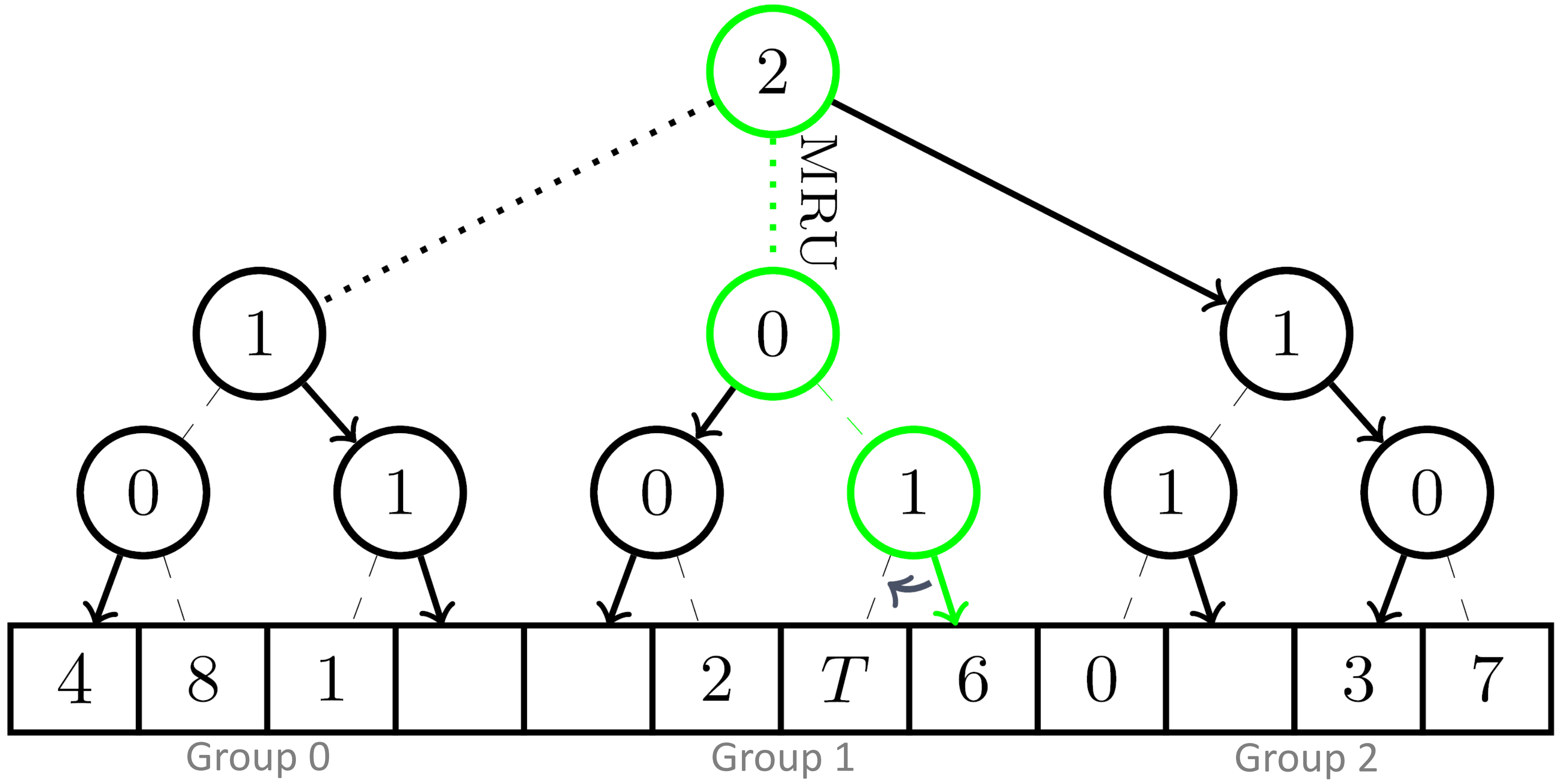


Improving TLB-based Attacks: $(MRU+1)_{\%3}PLRU_4$

Optimized

$\Downarrow$

6, 2, 1, 5

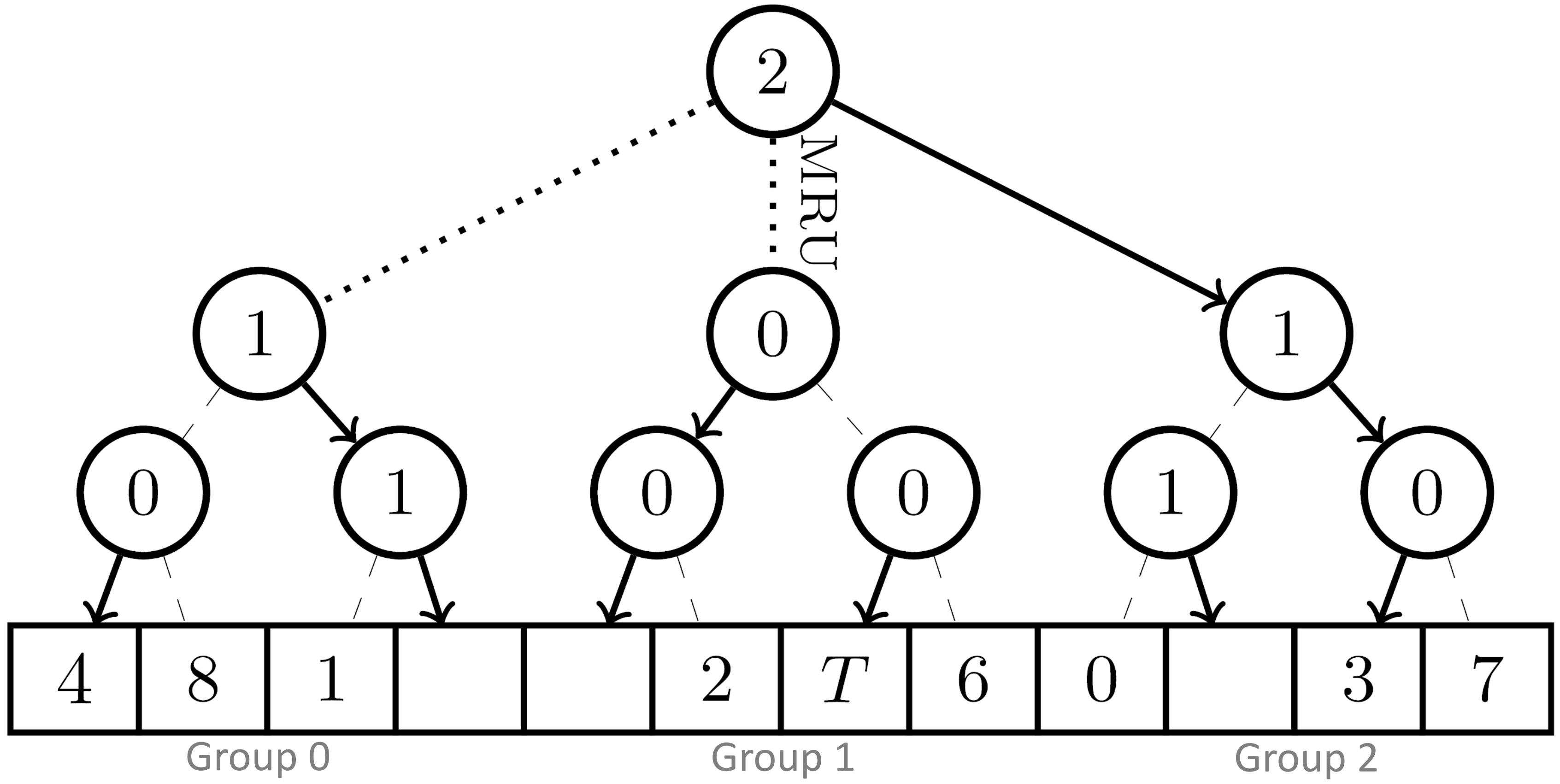


Improving TLB-based Attacks: $(MRU+1)_{\%3}PLRU_4$

Optimized

$\Downarrow$

6, 2, 1, 5

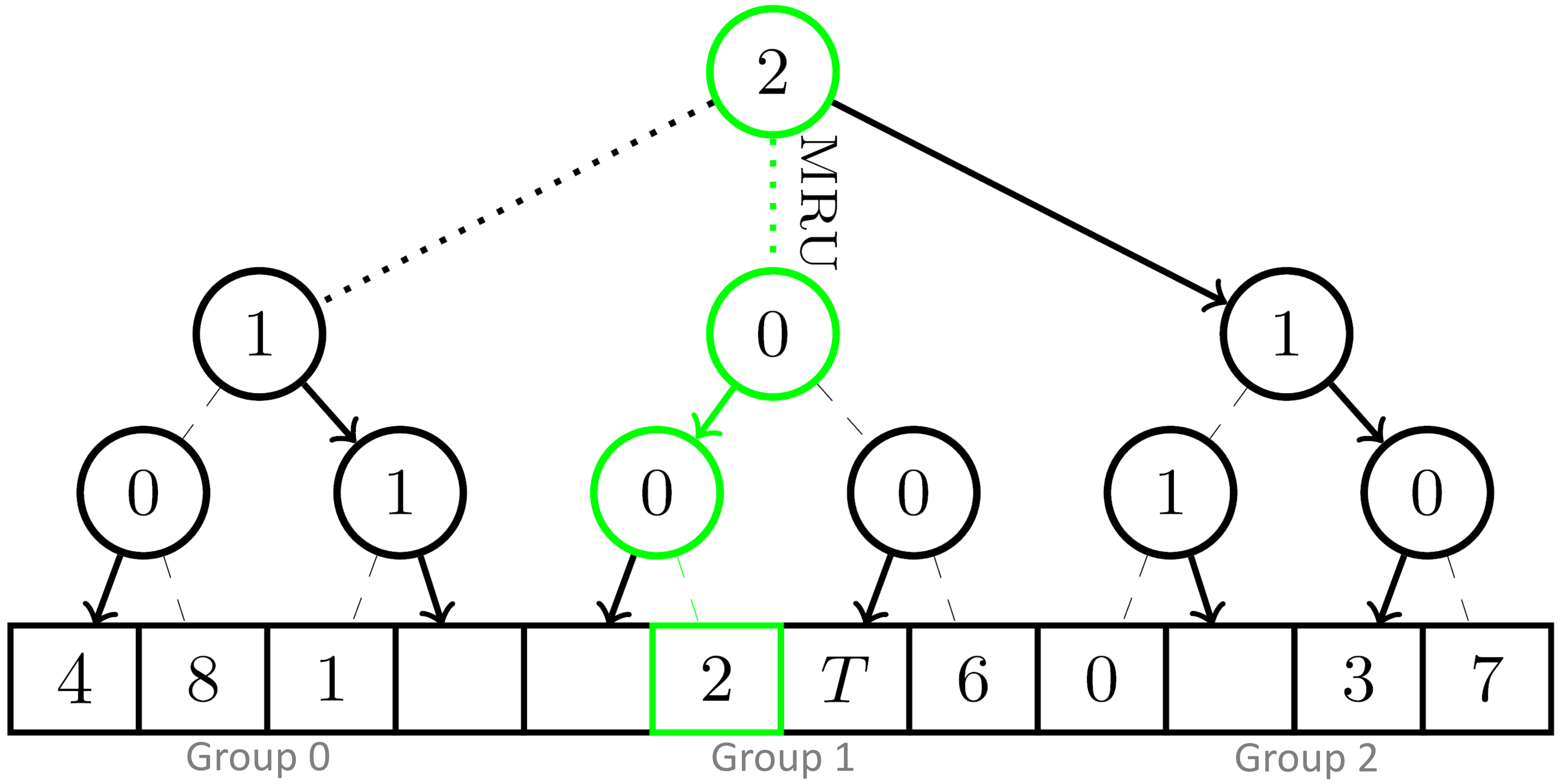


Improving TLB-based Attacks: $(\mathrm{MRU}+1)_{\%3}\mathrm{PLRU}_4$

Optimized

$\Downarrow$

6, 2, 1, 5

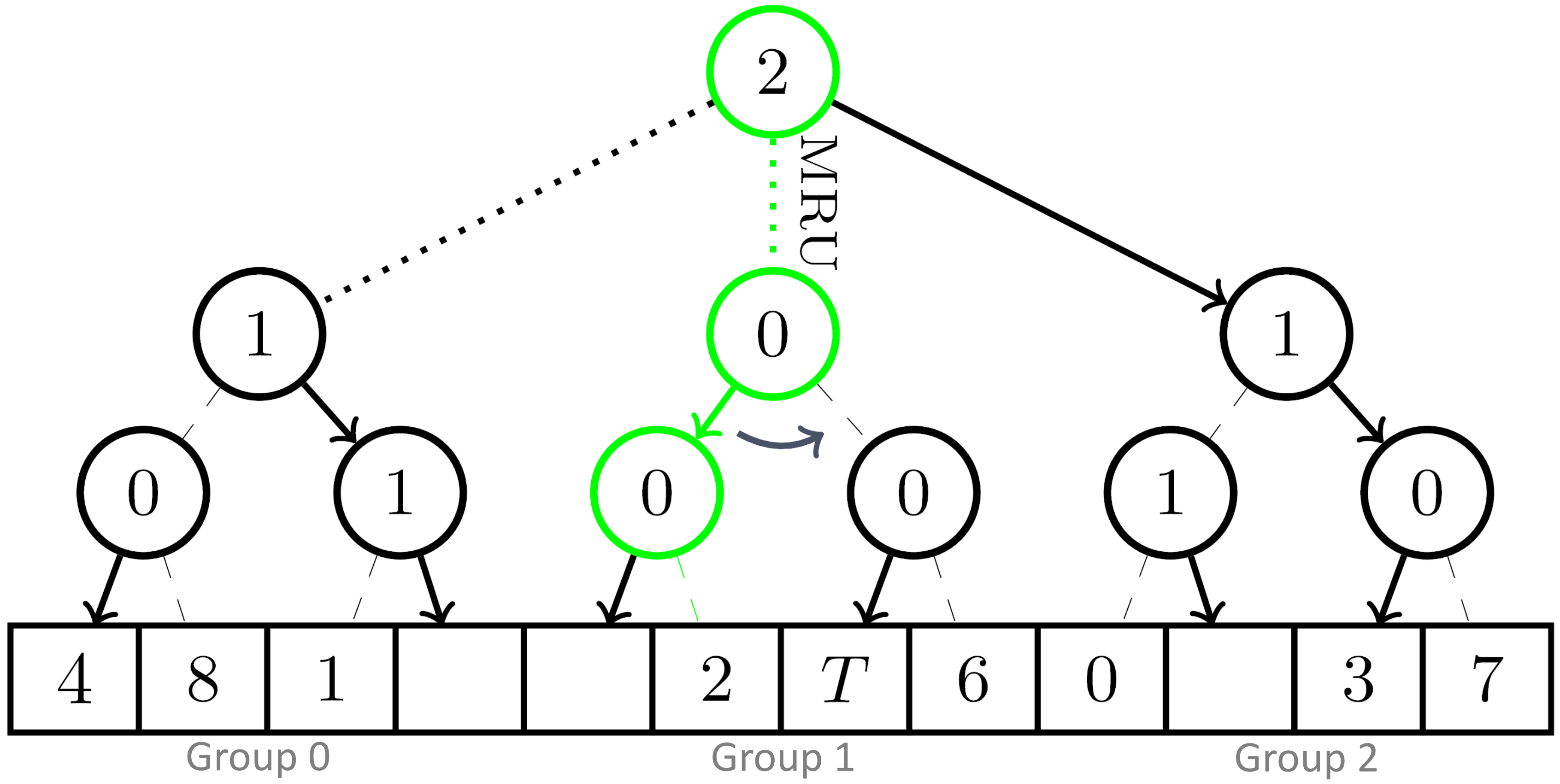


Improving TLB-based Attacks: $(\text{MRU}+1)_{\%3}\text{PLRU}_4$

Optimized

$\Downarrow$

6, 2, 1, 5

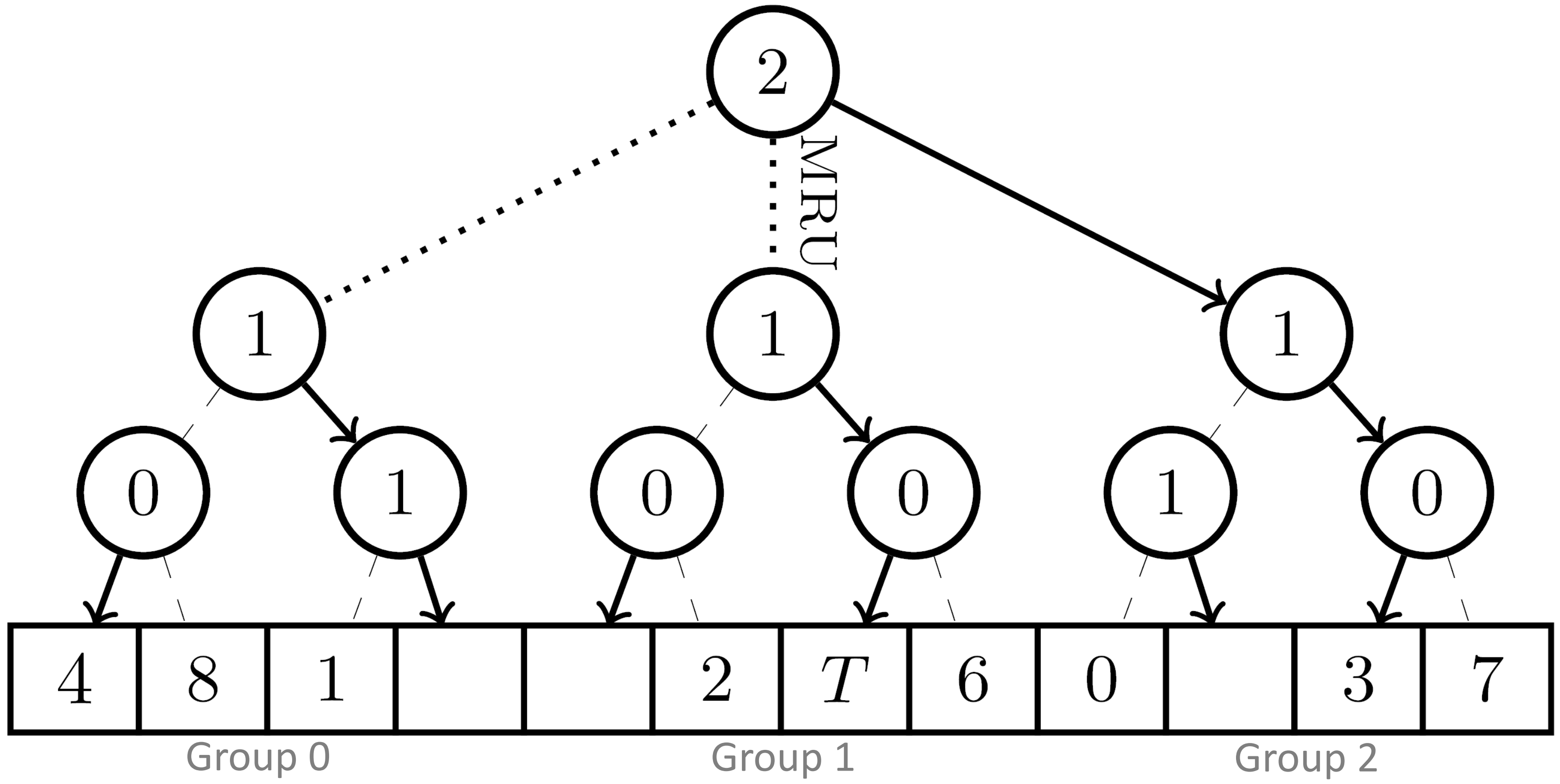


Improving TLB-based Attacks: $(\text{MRU}+1)_{\%3}\text{PLRU}_4$

Optimized

$\Downarrow$

6, 2, 1, 5

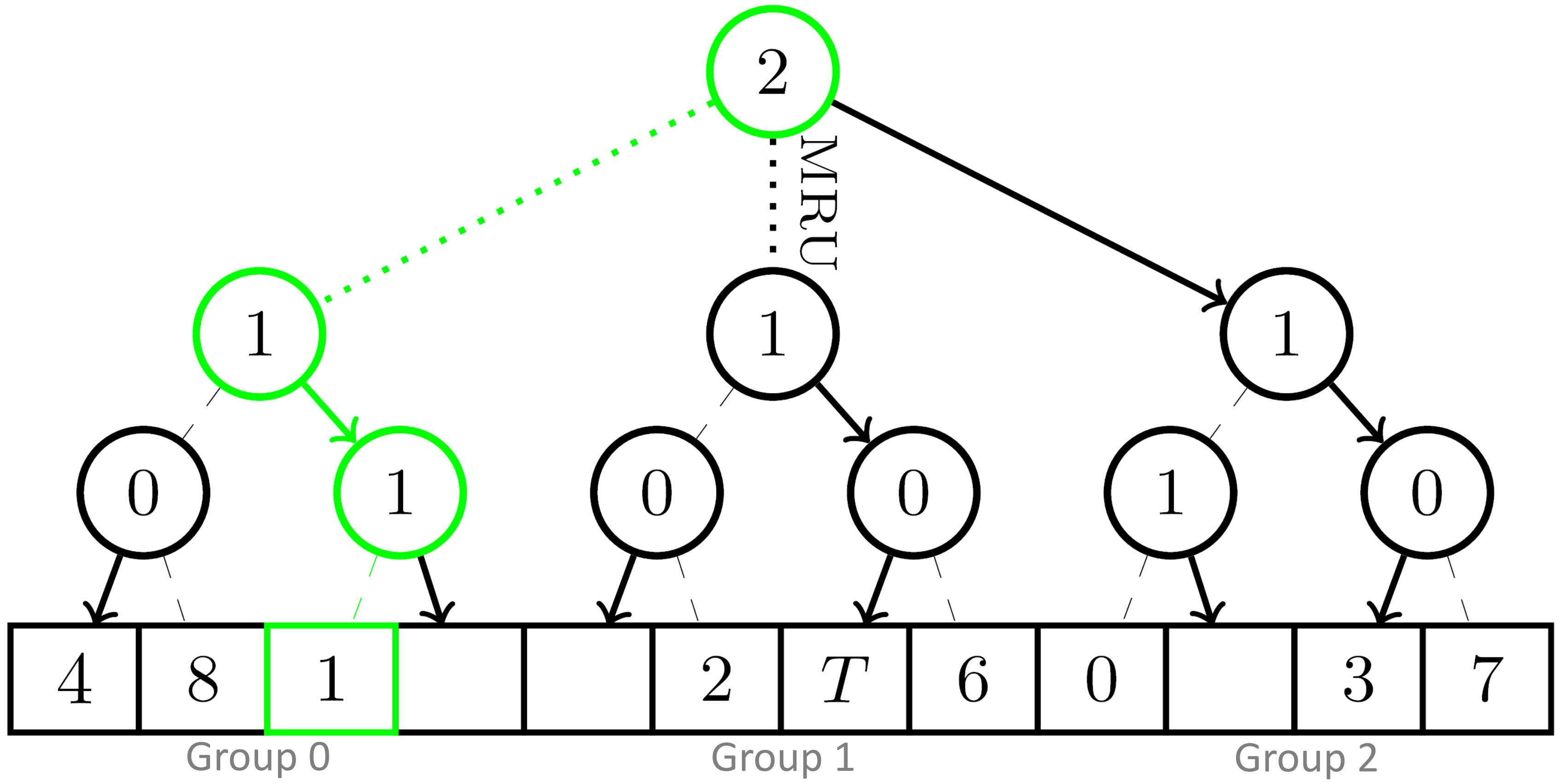


Improving TLB-based Attacks: $(\text{MRU}+1)_{\%3}\text{PLRU}_4$

Optimized

$\Downarrow$

6, 2, 1, 5

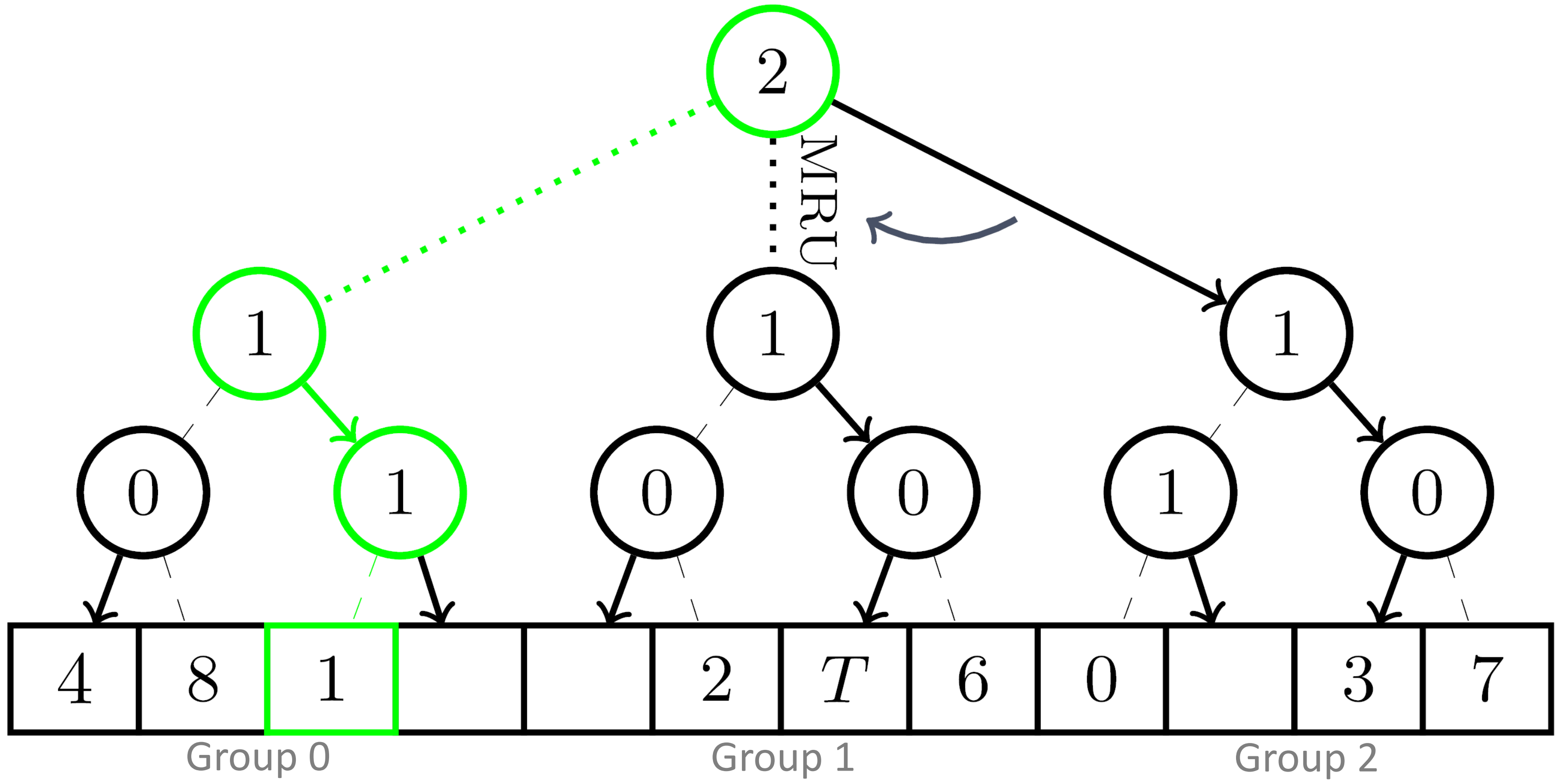


Improving TLB-based Attacks: $(\text{MRU+1})_{\%3}\text{PLRU}_4$

Optimized

$\Downarrow$

6, 2, 1, 5

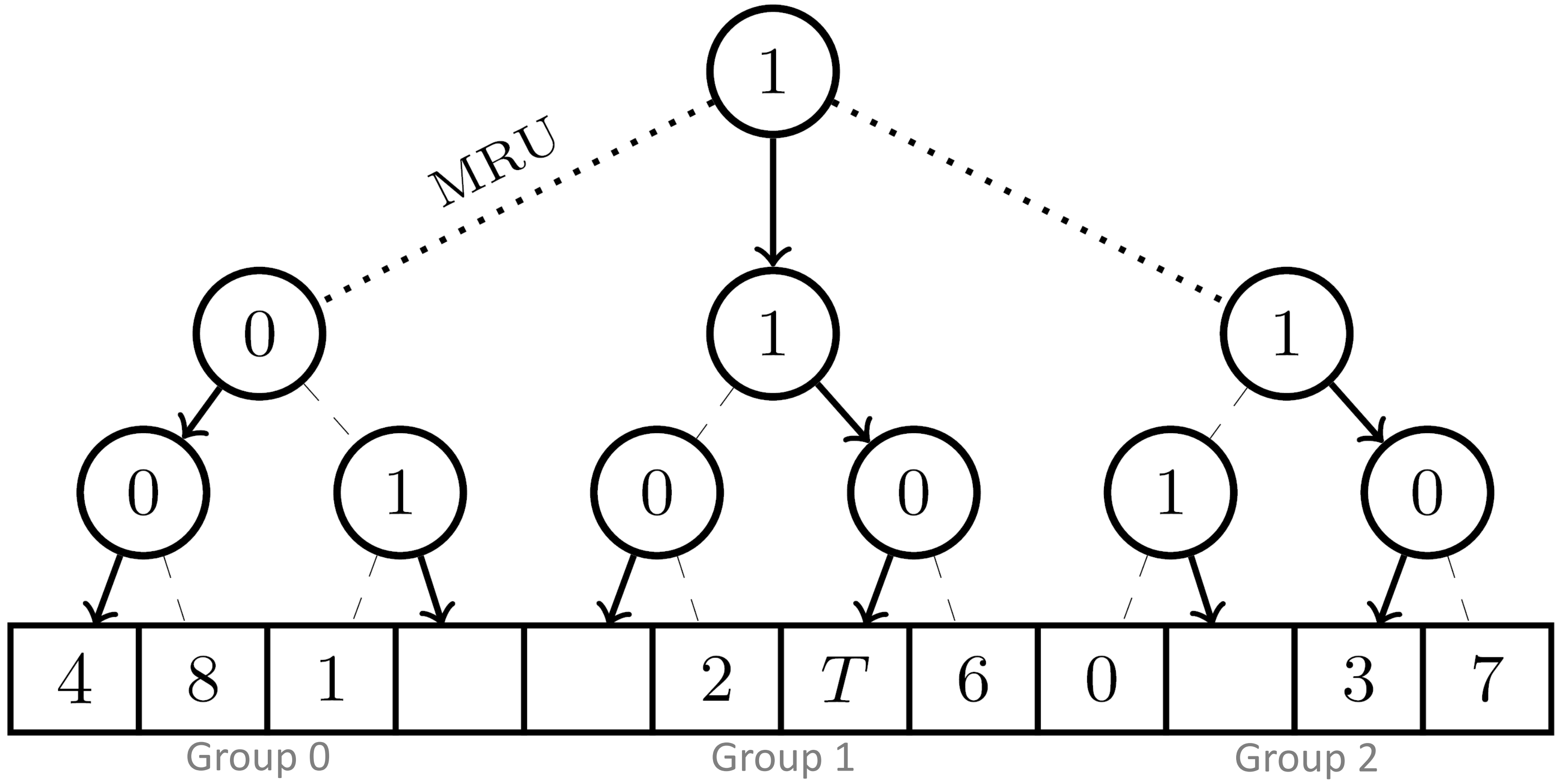


Improving TLB-based Attacks: $(\text{MRU}+1)_{\%3}\text{PLRU}_4$

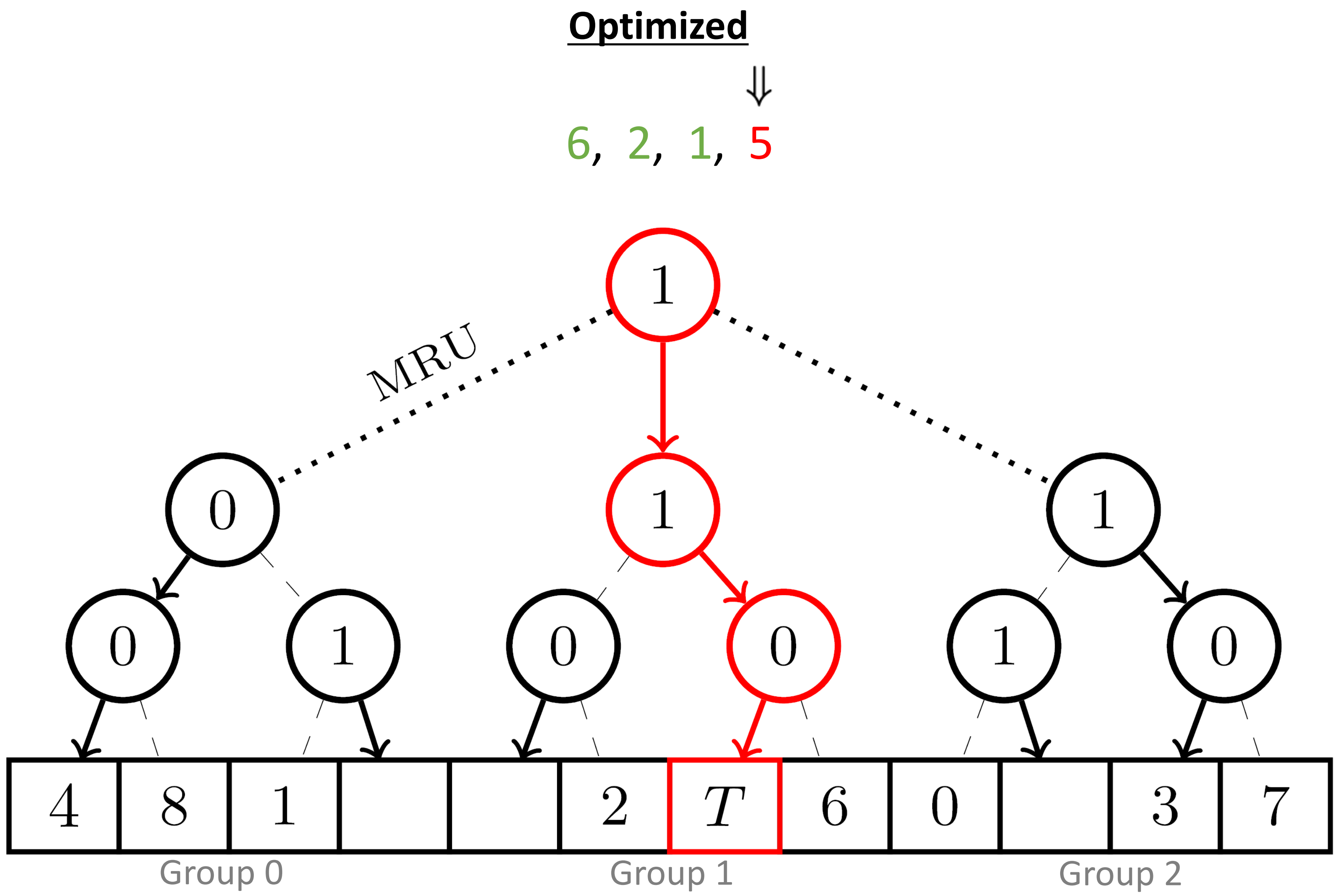
Optimized

$\Downarrow$

6, 2, 1, 5



Improving TLB-based Attacks: $(\mathrm{MRU}+1)_{\%3}\mathrm{PLRU}_4$

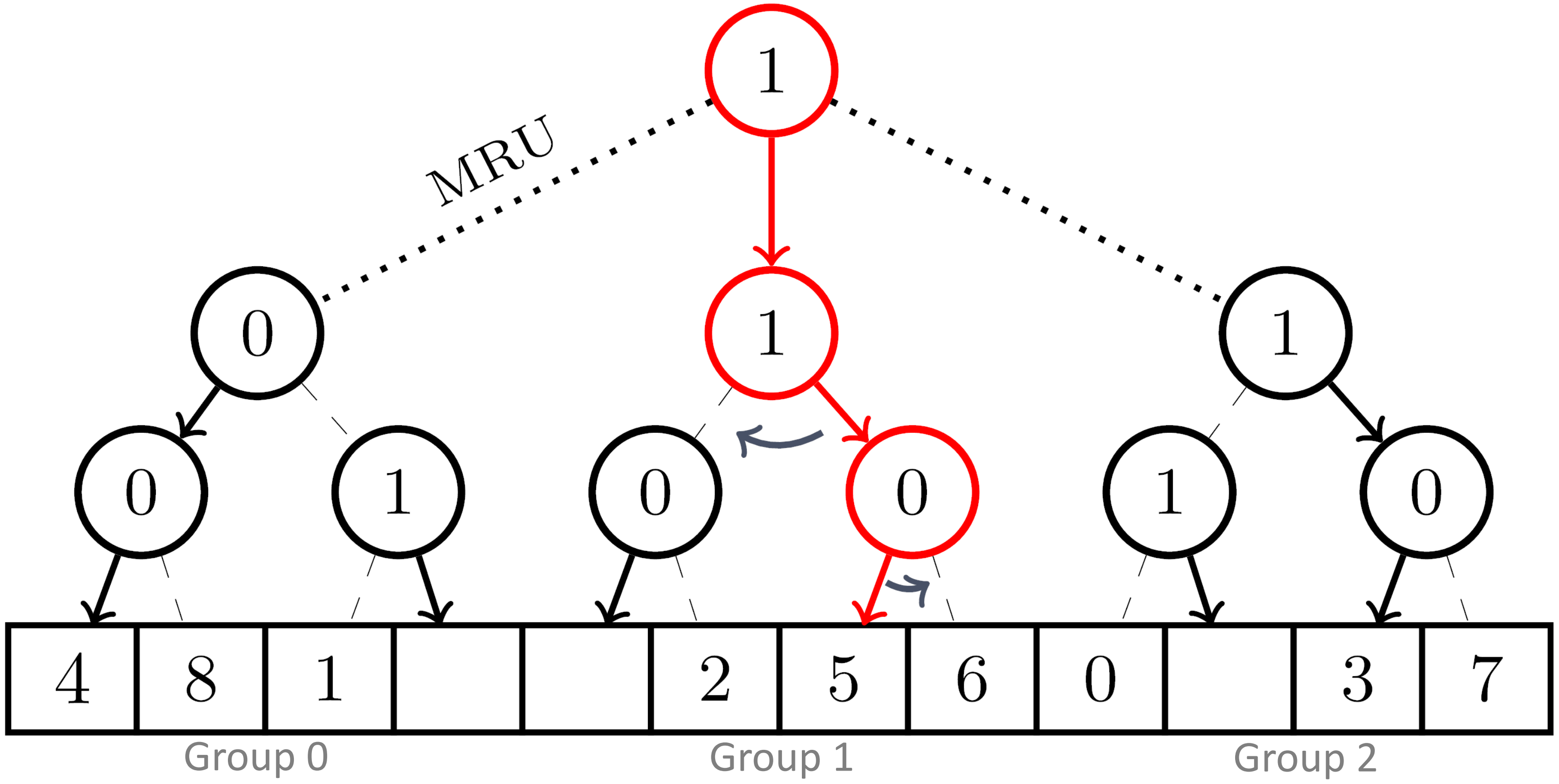


Improving TLB-based Attacks: $(\text{MRU+1})_{\%3}\text{PLRU}_4$

Optimized

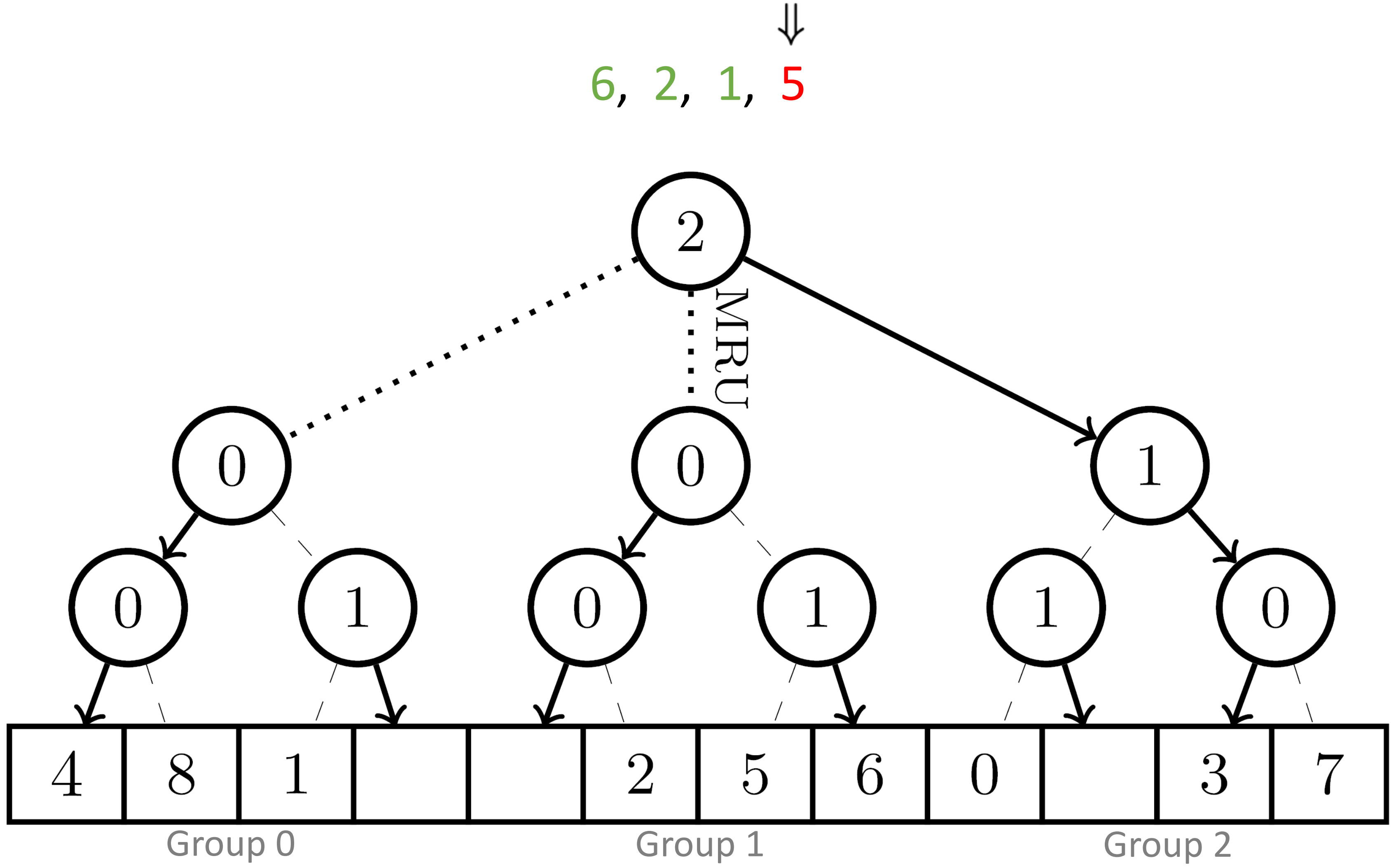
$\Downarrow$

6, 2, 1, 5



Improving TLB-based Attacks: $(\text{MRU}+1)_{\%3}\text{PLRU}_4$

Optimized



Improving TLB-based Attacks: $(MRU+1)_{\%3}PLRU_4$

$$\mathbf{3}\ \mathrm{HITS} + \mathbf{1}\ \mathrm{MISS}$$

$$\mathbf{vs.}$$

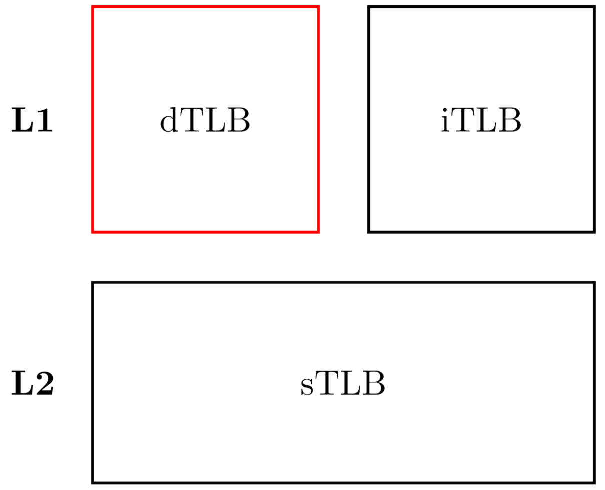
$$\mathbf{12}\ \mathrm{MISSES}$$

Also developed optimal eviction sets for Tree-PLRU

4 misses vs. 3 hits + 1 miss

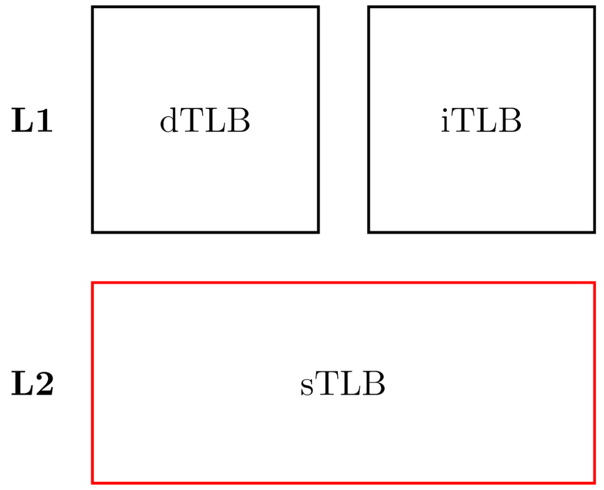
Improving TLB-based Attacks: Original TLBleed

- Original TLBleed attack leaks over dTLB
- Optimized eviction sets allow for **20% faster L1 dTLB sampling**!



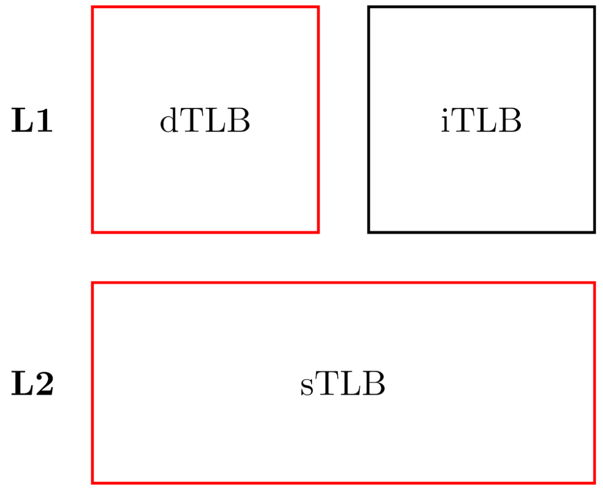
Improving TLB-based Attacks: sTLBleed

- We make it practical to leak over L2 sTLB
- Optimized eviction sets result in **2x faster L2 sTLB sampling**!
  - Close to the sampling rate on L1 dTLB (naive eviction)

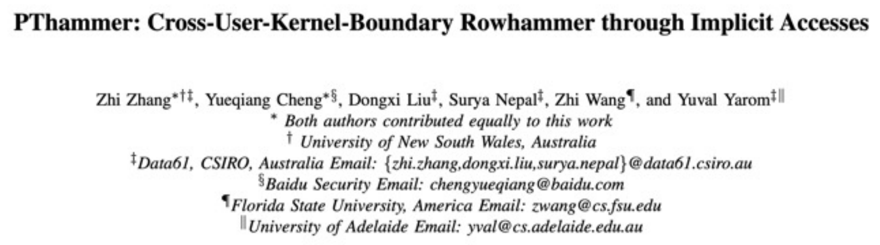


Improving TLB-based Attacks: set-pair TLBleed

- We introduce a variant that leaks from dTLB and sTLB simultaneously
- Optimized eviction sets result in almost **5x faster set-pair sampling**!

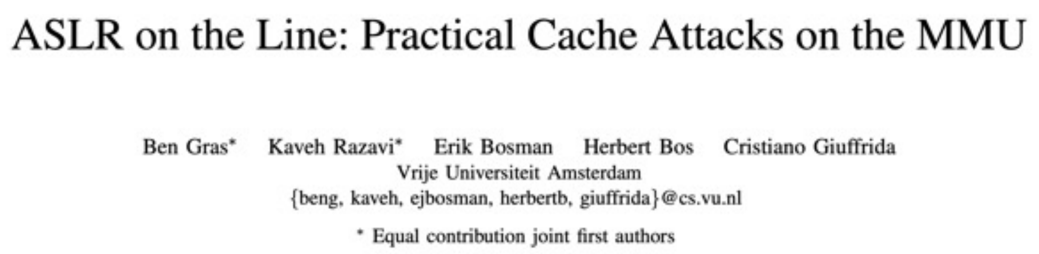


Improving TLB-based Attacks: Rowhammer & ASLR



PThammer: Cross-User-Kernel-Boundary Rowhammer through Implicit Accesses

Zhi Zhang*†‡, Yueqiang Cheng*§, Dongxi Liu‡, Surya Nepal‡, Zhi Wang¶, and Yuval Yarom‡‖
** Both authors contributed equally to this work*
† University of New South Wales, Australia
‡Data61, CSIRO, Australia Email: {zhi.zhang,dongxi.liu,surya.nepal}@data61.csiro.au
§Baidu Security Email: chengyueqiang@baidu.com
¶Florida State University, America Email: zwang@cs.fsu.edu
‖University of Adelaide Email: yval@cs.adelaide.edu.au



ASLR on the Line: Practical Cache Attacks on the MMU

Ben Gras* Kaveh Razavi* Erik Bosman Herbert Bos Cristiano Giuffrida
Vrije Universiteit Amsterdam
{beng, kaveh, ejbosman, herbertb, giuffrida}@cs.vu.nl
* Equal contribution joint first authors

- Other attacks indirectly using the TLB rely on continuous page walks
  - PTHammer: page walk can be used to hammer DRAM
  - ASLR on the Line: page walk side-effects break ASLR
- We use optimized eviction sets to cause TLB eviction
- PTHammer: **12% shorter** hammer time!
- ASLR on the Line: **20% less** time to break ASLR!

More interesting results in the paper…

- More details on Intel® TLBs
- Undocumented cache that keeps track of address spaces
- iTLB partitions dynamically based on workload
- Inclusivity and set sizes on AMD

Summary

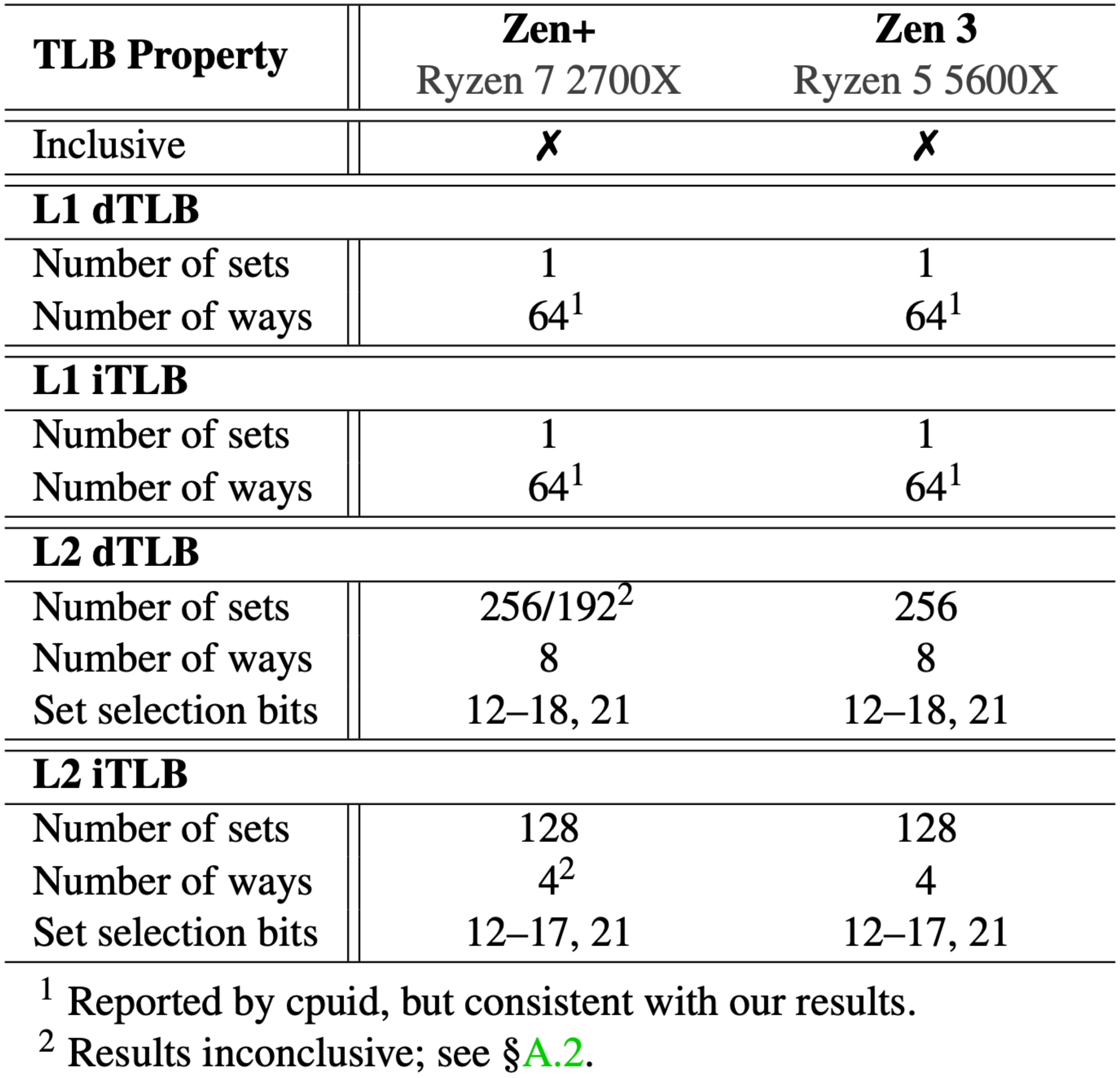
- Reverse engineered TLBs
  - TLB Desynchronization as a reverse engineering primitive
- Better understanding of TLB behavior
  - Novel TLB properties
- Speed up TLB eviction
  - Optimized eviction sets
- Improved attacks relying on TLB interaction
  - TLBleed, PTHammer, ASLR on the Line

Thank you!

Questions?

TLB Property	**Westmere-EP** E5645	**Ivy Bridge** i3-3220	**Haswell** i7-4790	**Skylake-SP** Silver 4110	**Kaby Lake** i7-7700K	**Coffee Lake(-S)** i7-8750H, i9-9900K
Inclusive	✗	✗	✗	✗	✗	✗
Exclusive	✗	✗	✗	✗	✗	✗
L2 is victim cache	✗	✗	✗	✗	✗	✗
L2 hit inserts into L1	✓	✓	✓	✓	✓	✓
L1 hit inserts into L2	✗	✗	✗	✗	✗	✗
L1 dTLB						
Number of sets	16	16	16	16	16	16
Number of ways	4	4	4	4	4	4
Hash function	linear	linear	linear	linear	linear	linear
Replacement policy	tree-PLRU$_4$	tree-PLRU$_4$	tree-PLRU$_4$	tree-PLRU$_4$	tree-PLRU$_4$	tree-PLRU$_4$
Max PCIDs	N/A	N/A	N/A	N/A	N/A	N/A
L1 iTLB						
Number of sets	32	16 / 32[1]	8 / 16[1]	8 / 16[1]	8 / 16[1]	8 / 16[1]
Number of ways	4	4	8	8	8	8
Hash function	linear	linear	linear	linear	linear	linear
Replacement policy	tree-PLRU$_4$	LRU$_4$	tree-PLRU$_8$[2]	tree-PLRU$_8$[2]	tree-PLRU$_8$[2]	tree-PLRU$_8$[2]
Max PCIDs	1 / 4[3]	1 / 4[3]	1 / 4[3]	1 / 4[3]	1 / 4[3]	1 / 4[3]
L2 sTLB						
Number of sets	128	128	128	128	128	128
Number of ways	4	4	8	12	12	12
Hash function	linear	linear	linear	XOR	XOR	XOR
Replacement policy	tree-PLRU$_4$	tree-PLRU$_4$	tree-PLRU$_8$	(MRU+1)$_{\%3}$PLRU$_4$	(MRU+1)$_{\%3}$PLRU$_4$	(MRU+1)$_{\%3}$PLRU$_4$
Max PCIDs	4	4	4	4	4	4

[1] Depending on the activity of the co-resident hyperthread; see §4.2.
[2] Model closest to our observations, but very high error rate; see §4.4.2.
[3] Depending on whether the *NOFLUSH* bit is set when switching PCIDs; see §4.5.



TLB Property	**Zen+** Ryzen 7 2700X	**Zen 3** Ryzen 5 5600X
Inclusive	✗	✗
L1 dTLB		
Number of sets	1	1
Number of ways	64[1]	64[1]
L1 iTLB		
Number of sets	1	1
Number of ways	64[1]	64[1]
L2 dTLB		
Number of sets	256/192[2]	256
Number of ways	8	8
Set selection bits	12–18, 21	12–18, 21
L2 iTLB		
Number of sets	128	128
Number of ways	4[2]	4
Set selection bits	12–17, 21	12–17, 21

[1] Reported by cpuid, but consistent with our results.
[2] Results inconclusive; see §A.2.