CROSS-X: Generalized and Stable Cross-Cache Attack on the Linux Kernel

ACM CCS'25

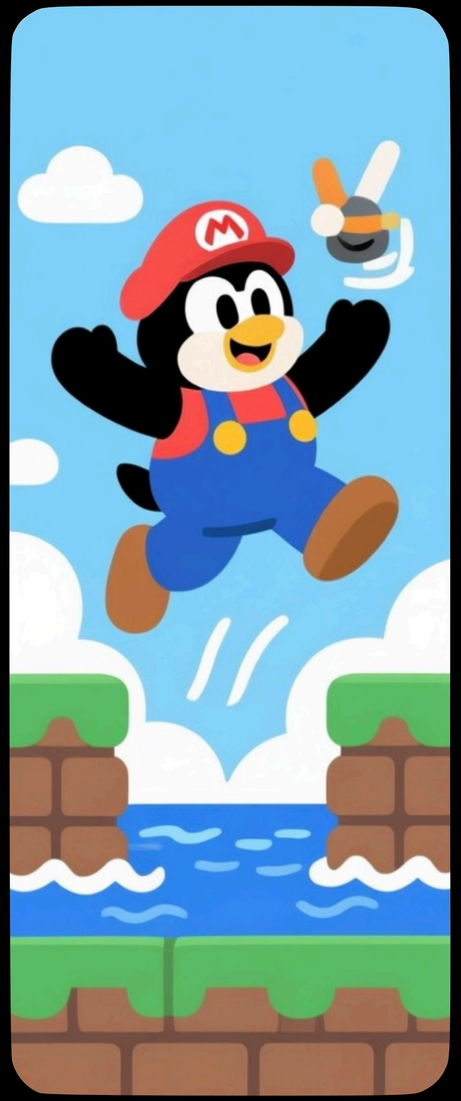
Dong-ok Kim[1]* Juhyun Song[2]* Insu Yun[2]

[1]ENKI Whitehat [2]KAIST

*Both authors contributed equally.

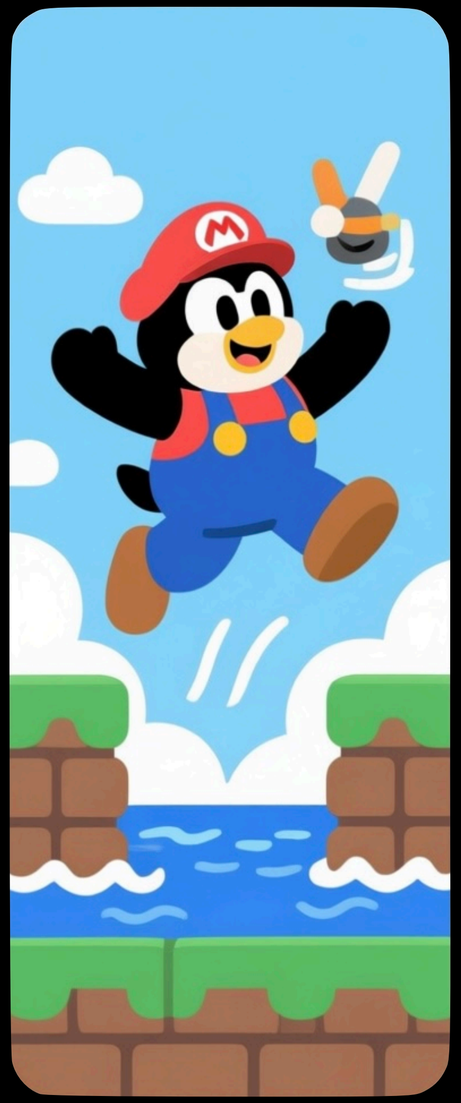
About CROSS-X

- **Cross-Cache Attack** underpins modern Linux kernel exploits.



About CROSS-X

- **Cross-Cache Attack** underpins modern Linux kernel exploits.
- There remains **uncertainty** around executing it effectively.

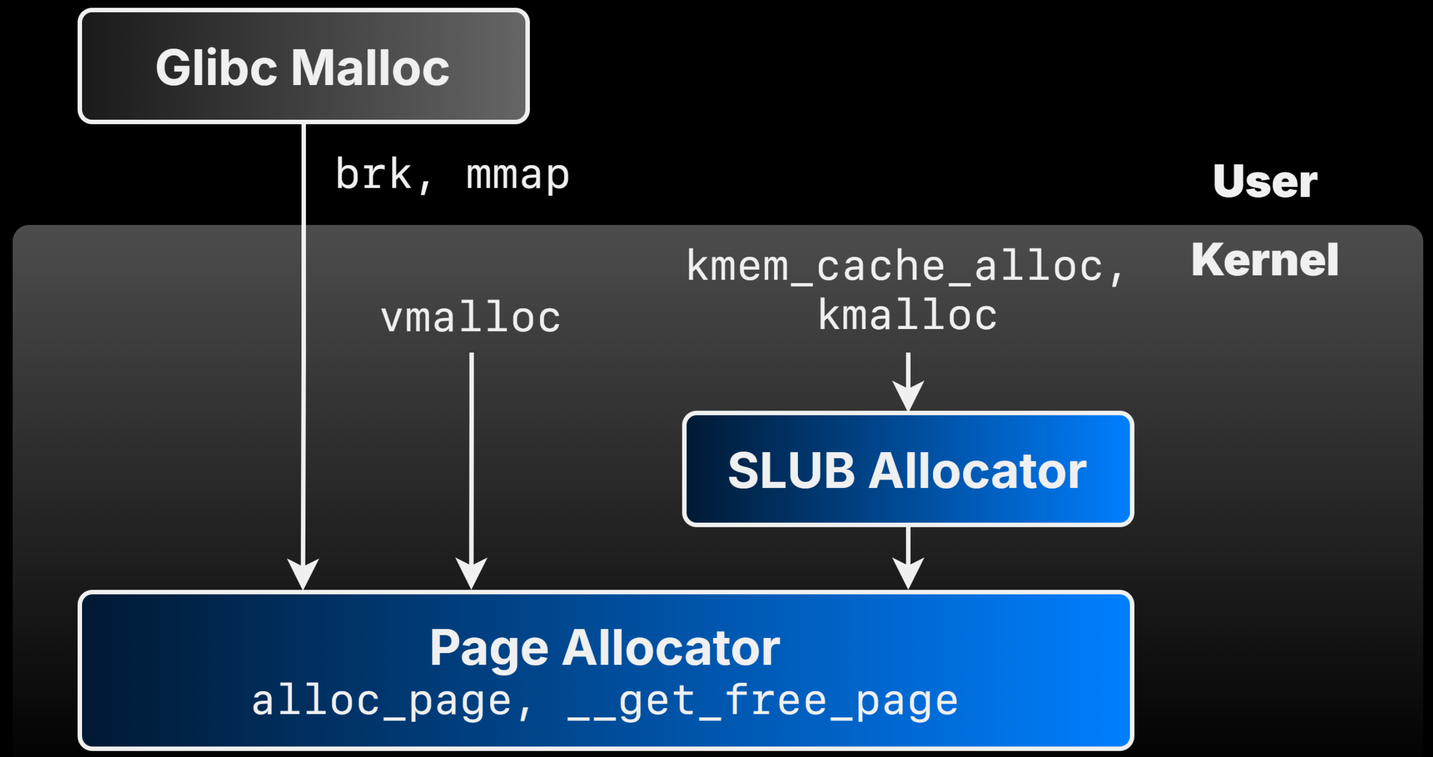


About CROSS-X

- **Cross-Cache Attack** underpins modern Linux kernel exploits.
- There remains **uncertainty** around executing it effectively.
- Our work focuses on:
  1. **Robust strategies**
  2. **Automated target object identification**
  3. **Real-world validation**

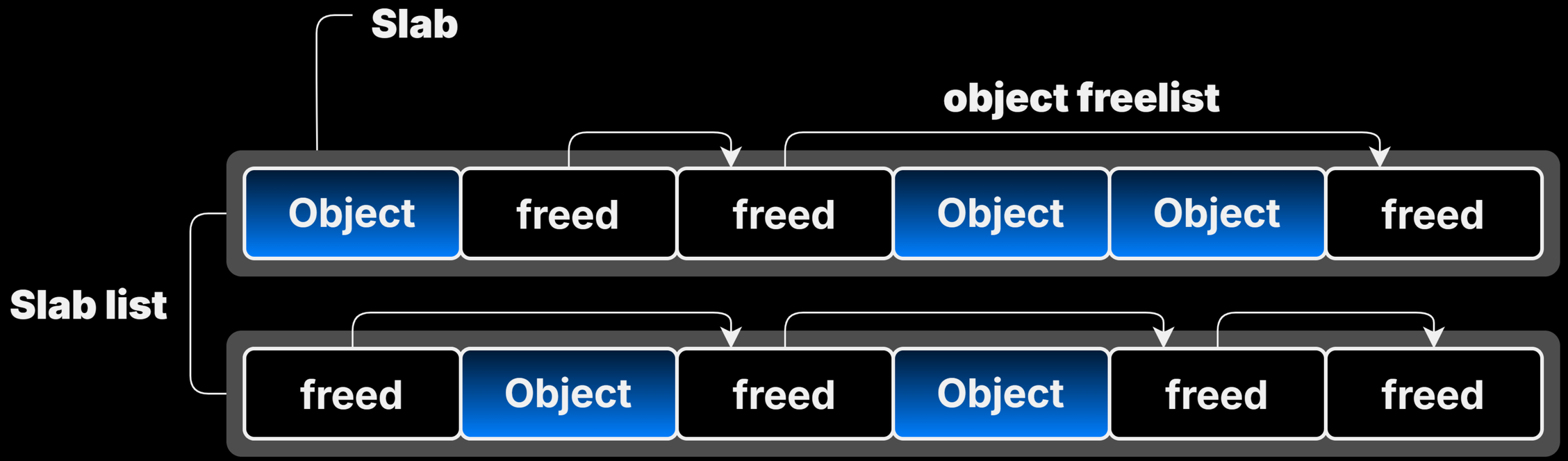
Linux Memory Management

- Linux kernel manages memory using the **page allocator** and the **SLUB allocator.**
  - **Page allocator** provides physically contiguous pages to system components.
  - **SLUB allocator** handles object-level allocation for small memory requests.

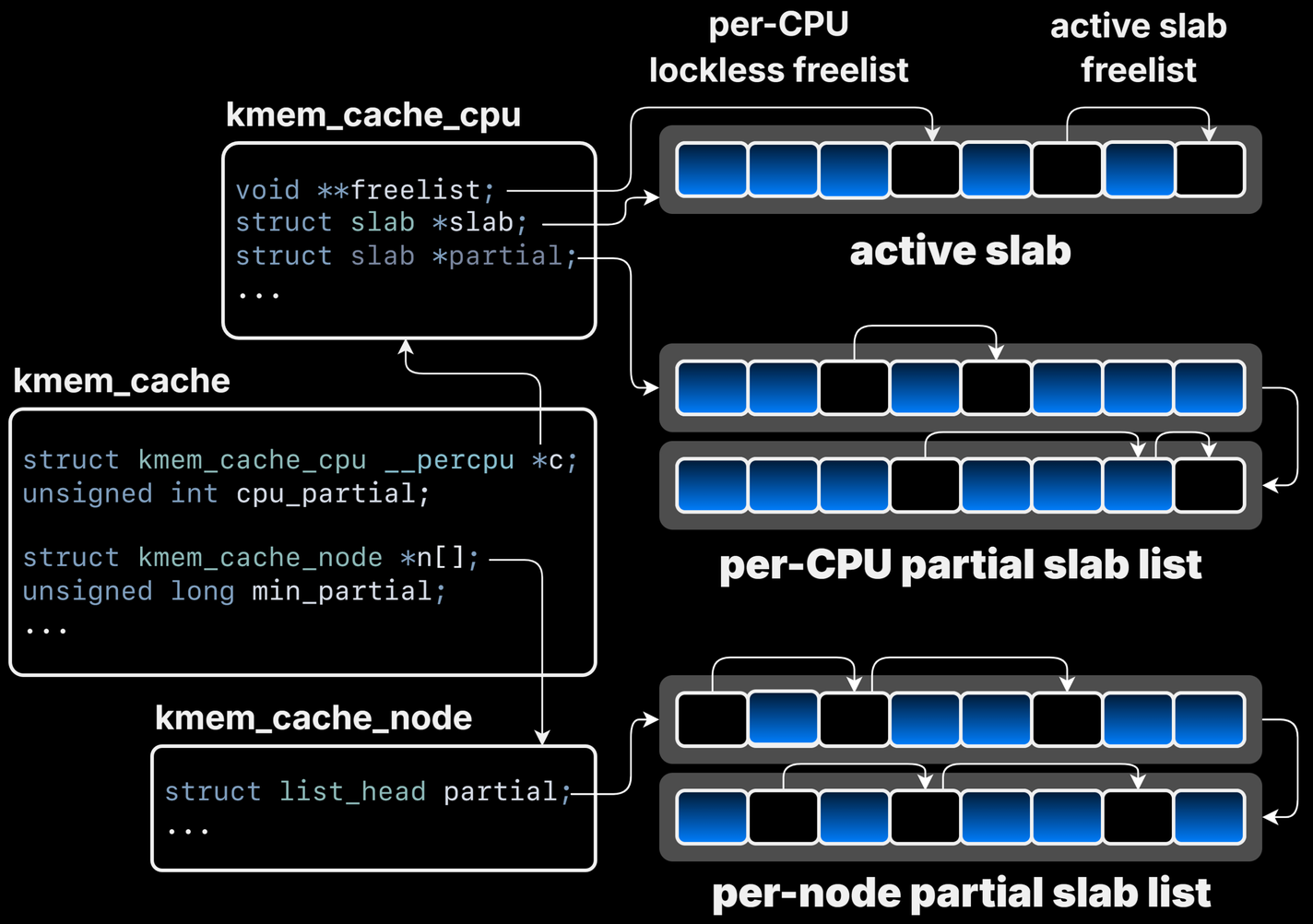


SLUB allocator

- SLUB allocator maintains a **pre-allocated cache** of objects.
- SLUB operates on a **slab** basis — block of pages divided into uniform-sized slots.
  - Slab forms an **object freelist** to chain freed objects inside it.
  - Slabs could be also chained into slab lists.

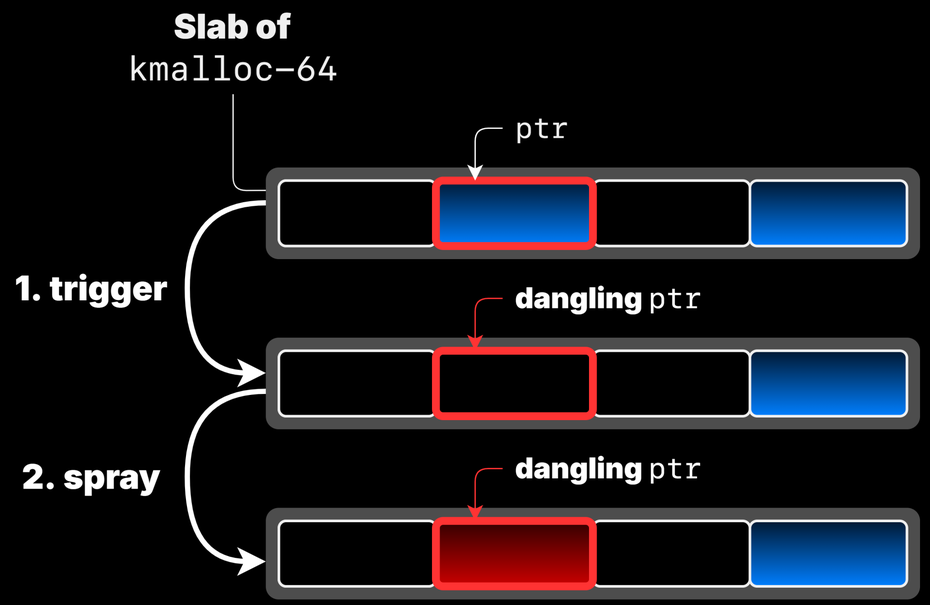


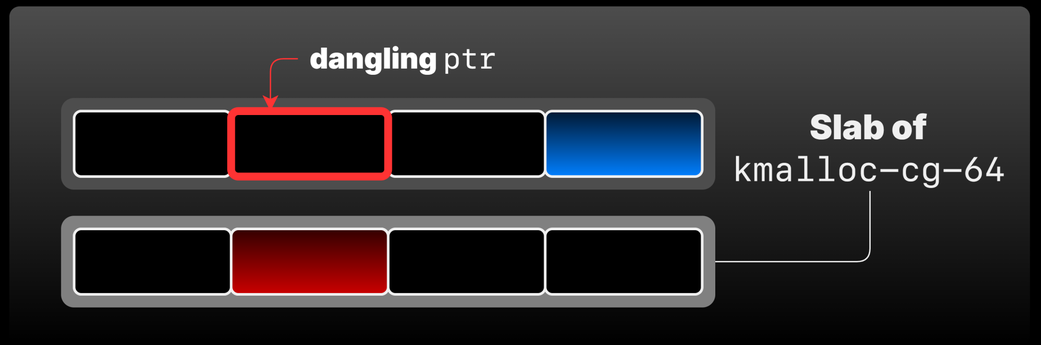
SLUB Cache Structure



Cache Separation

- In 2016, the kernel introduced **cache separation**.
- Key objects were moved to separate caches, disrupting many common exploits.
  - Made targeted object spraying significantly harder for attackers.



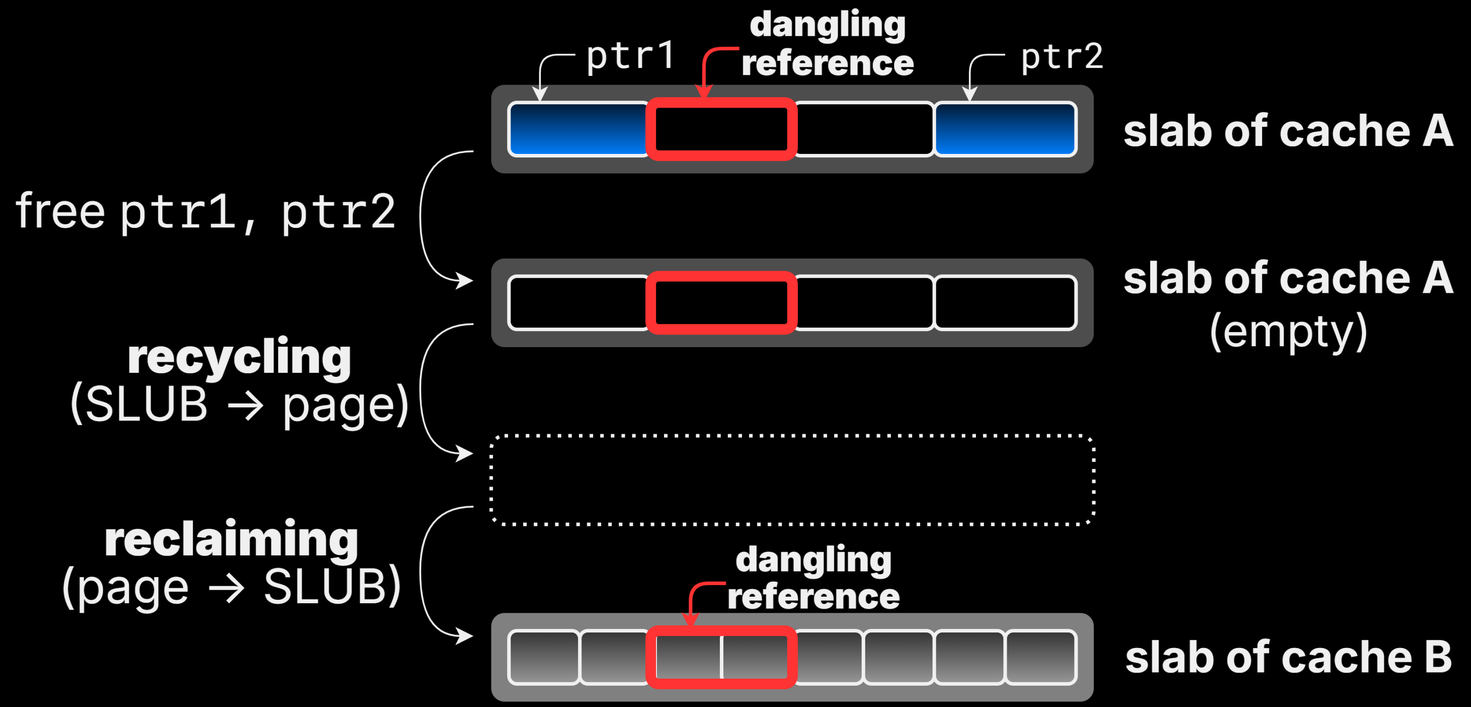


UAF condition

with Cache Separation

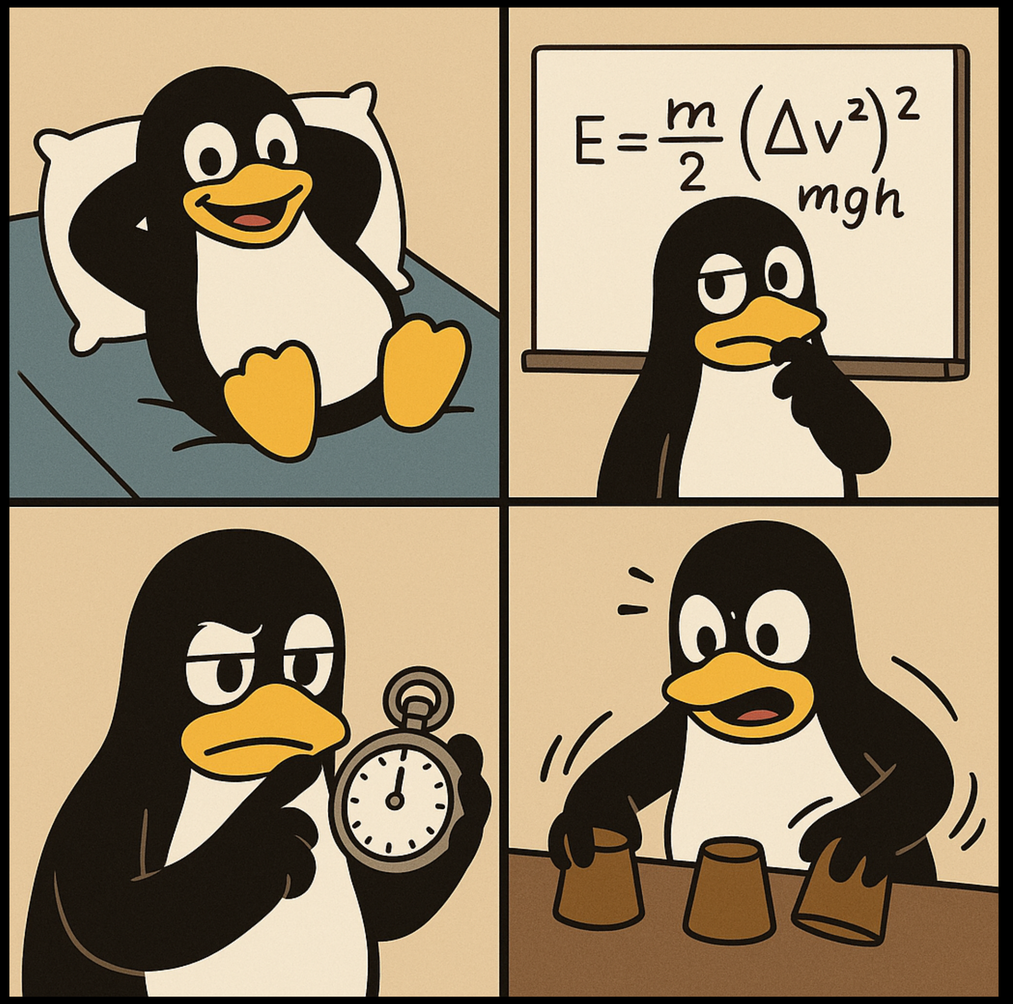
Cross-Cache Attack

- SLUB discards slabs and allocates new ones from the page allocator.
- **Cross-Cache Attack** exploits this reuse to bypass cache separation.
  - **Recycling** phase forces SLUB to drop a vulnerable slab.
  - **Reclaiming** phase reuses that slab by spraying objects from a different cache.



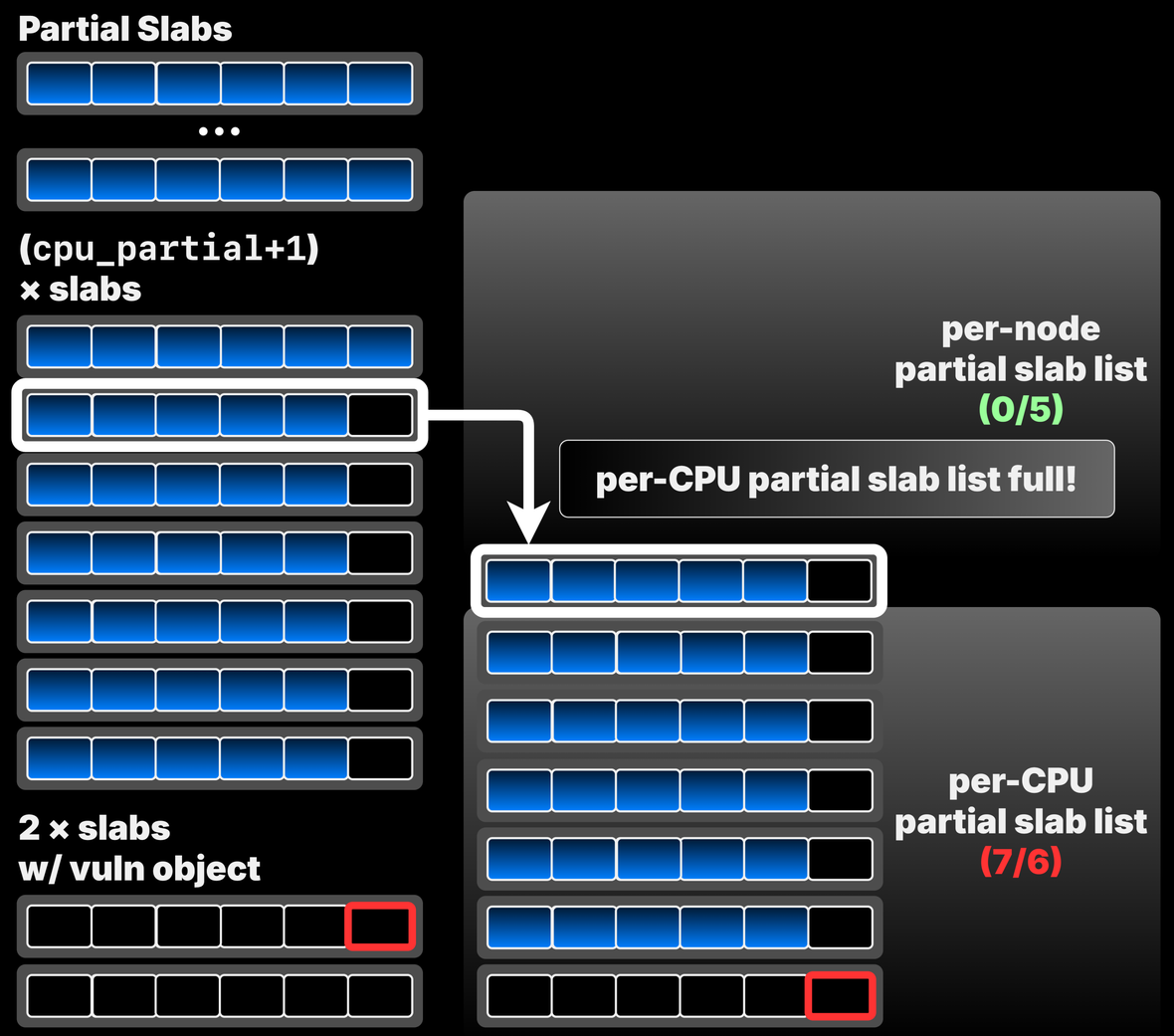
Public Exploits Review

- We classified public strategies for recycling into:
  - **Naive**
  - **Partial Free**
  - **Timing Side-Channel (of SLUBStick)**
  - **Interleaving Cores**
- Due to time constraints, we will focus exclusively on Partial Free.



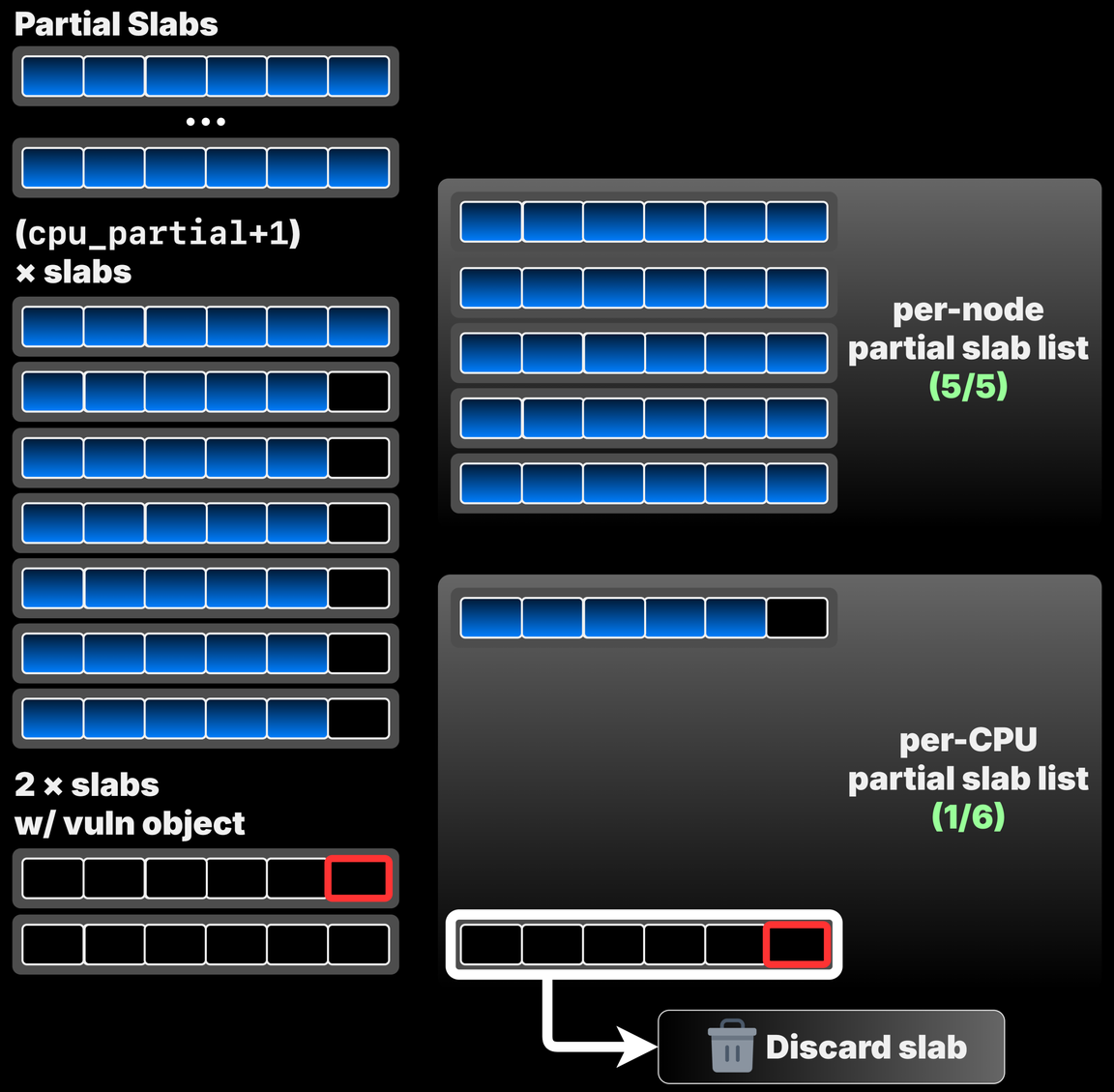
Partial Free

- First generalizable technique by Jann Horn.
- **Strategy workflow:**
  1. Defragmentation
  2. Allocate (`cpu_partial+1`) slabs
  3. Allocate 2 slabs with vulnerable object
  4. Free both slabs from step 3
  5. **Free one object per slab from step 4**



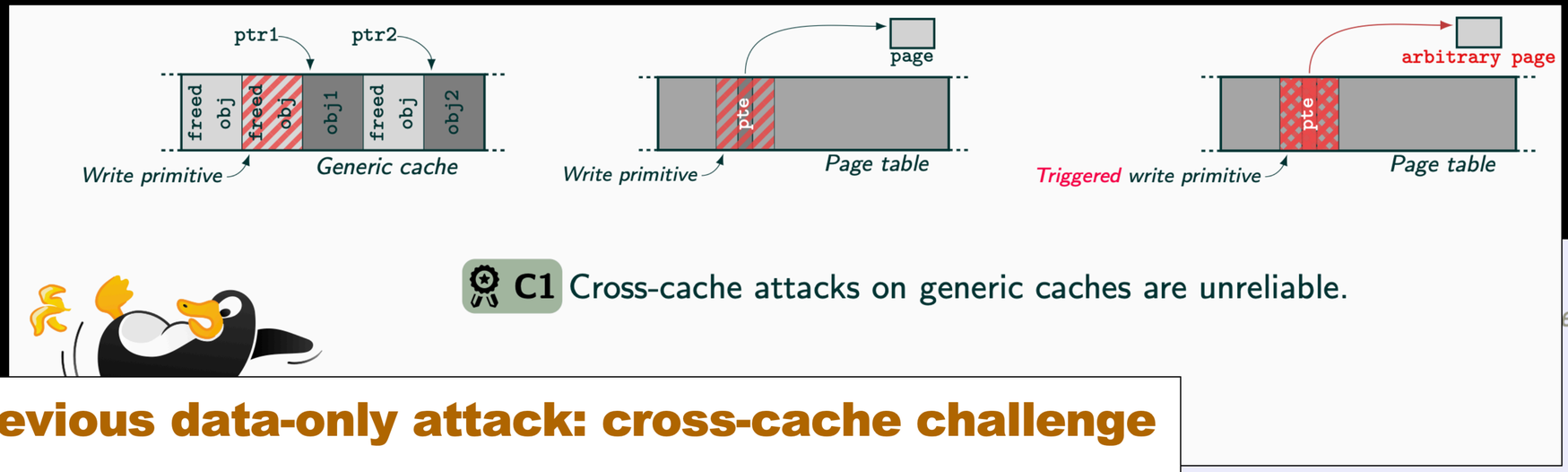
Partial Free

- First generalizable technique by Jann Horn.
- **Strategy workflow:**
  1. Defragmentation
  2. Allocate (`cpu_partial+1`) slabs
  3. Allocate 2 slabs with vulnerable object
  4. Free both slabs from step 3
  5. **Free one object per slab from step 4**



Partial Free Failure

- However, exploit writers are still blaming the **unreliability** of Cross-Cache Attacks.
  - Some introduced concepts like “overflow factors”, which we found to be misleading.
- So why would Partial Free have stopped working?




```
ely allocated enough
 we free an object from each
```

```
ount of objects to spray
= (OBJS_PER_SLAB * (CPU_PARTIAL + 1)) * OVERFLOW_FACTOR;
mt;
```

Previous data-only attack: cross-cache challenge

- Cross-cache attack techniques vary by vulnerability type, e.g.,
- OOB: less reliable
- UAF: more reliable but not future-proof
- Cross-cache still a significant hurdle for exploits

Partial Free Failure

- We identified the root case as the **reversal** of the `cpu_partial` and `min_partial` size relationship.
- In this case, Partial Free **deterministically fails** with a 0% success rate.

Table 1: Comparison of `cpu_partial` and `min_partial` values in object caches for Linux kernel versions 5.15 and 6.1. Caches where `cpu_partial` fell below `min_partial` in v6.1 are marked †, and those below in both versions are marked ★.

Object Cache	`cpu_partial` vs. `min_partial` (v5.15 → v6.1)
kmalloc-16 †	(30 > 5) → **(1 < 5)**
kmalloc-32 †	(30 > 5) → **(2 < 5)**
kmalloc-64 †	(30 > 5) → **(4 < 5)**
kmalloc-96	(30 > 5) → (6 > 5)
kmalloc-128	(30 > 5) → (8 > 5)
kmalloc-192	(30 > 5) → (12 > 5)
kmalloc-256	(13 > 5) → (7 > 5)
kmalloc-512	(13 > 5) → (7 > 5)
kmalloc-1024 †	(6 > 5) → **(3 < 5)**
kmalloc-2048 †	(6 > 5) → **(3 < 5)**
kmalloc-4096 ★	**(2 < 6)** → **(2 < 6)**

Partial Free Failure

- We identified the root case as the **reversal** of the `cpu_partial` and `min_partial` size relationship.
- In this case, Partial Free **deterministically fails** with

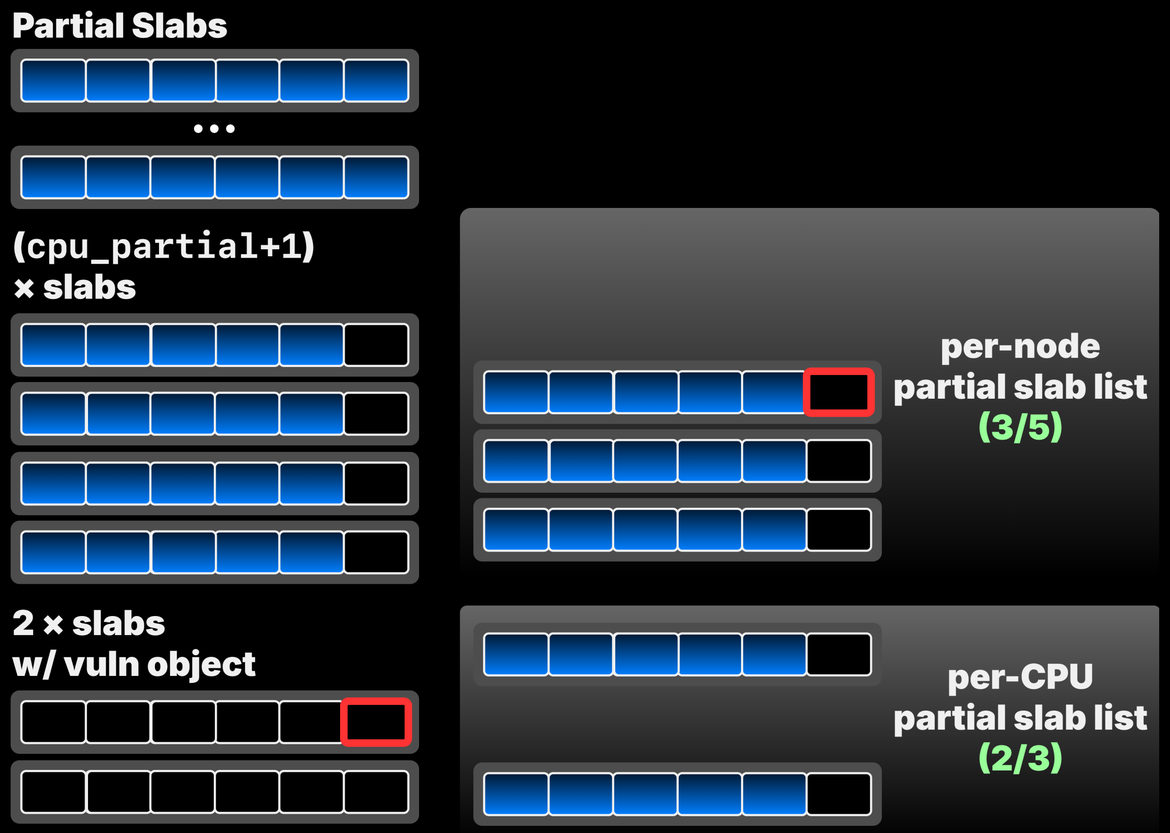
Consider the example where (`cpu_partial`, `min_partial`)=(3, 5).

Table 1: Comparison of `cpu_partial` and `min_partial` values in object caches for Linux kernel versions 5.15 and 6.1. Caches where `cpu_partial` fell below `min_partial` in v6.1 are marked †, and those below in both versions are marked ⋆.

Object Cache	`cpu_partial` vs. `min_partial` (v5.15 → v6.1)
kmalloc-128	$(30 > 5) \rightarrow (8 > 5)$
kmalloc-192	$(30 > 5) \rightarrow (12 > 5)$
kmalloc-256	$(13 > 5) \rightarrow (7 > 5)$
kmalloc-512	$(13 > 5) \rightarrow (7 > 5)$
kmalloc-1024 †	$(6 > 5) \rightarrow \mathbf{(3 < 5)}$
kmalloc-2048 †	$(6 > 5) \rightarrow \mathbf{(3 < 5)}$
kmalloc-4096 ⋆	$\mathbf{(2 < 6)} \rightarrow \mathbf{(2 < 6)}$

Partial Free Failure

- First generalizable technique by Jann Horn.
- **Strategy workflow:**
  1. Defragmentation
  2. Allocate (`cpu_partial+1`) slabs
  3. Allocate 2 slabs with vulnerable object
  4. Free both slabs from step 3
  5. **Free one object per slab from step 4**
- **Slab discard failed!**



Complete Free

- We propose an improved strategy, **Complete Free**, which:
  - Excludes defragmentation, heuristics, and SLUB misconceptions.
  - Considers per-node partial slab lists, effectively handling reversal scenarios.
- We also introduce a hybrid strategy, **Timing+Complete Free**:
  - Slab allocation is guided by **SLUBStick**'s timing side-channel, enhancing reliability.

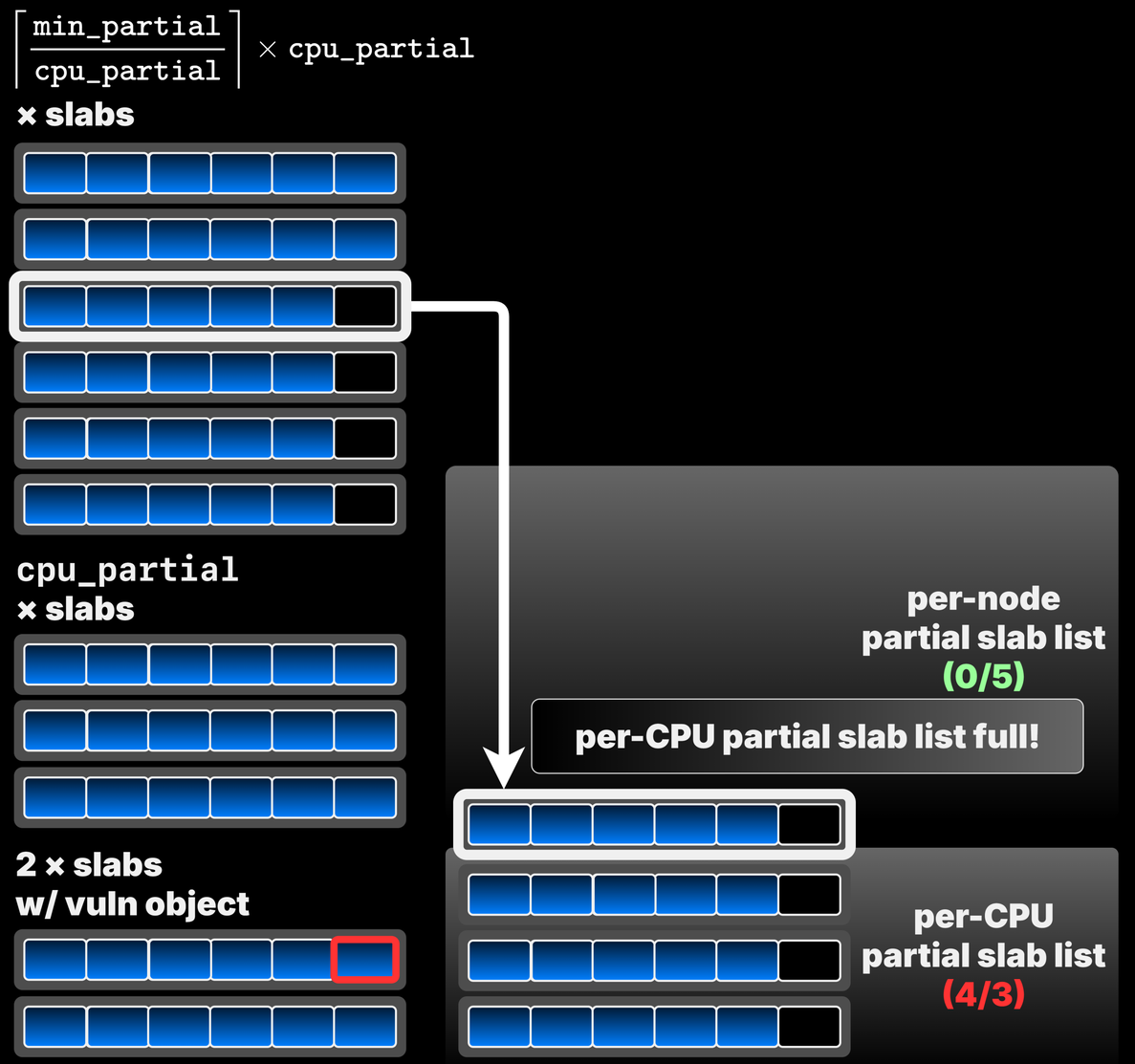
Complete Free

- We propose an improved strategy, **Complete Free**, which:
  - Excludes defragmentation, heuristics, and SLUB misconceptions.
  - Considers per-node partial slab lists, effectively handling reversal scenarios.

Complete Free **works** in the previous example, unlike Partial Free.

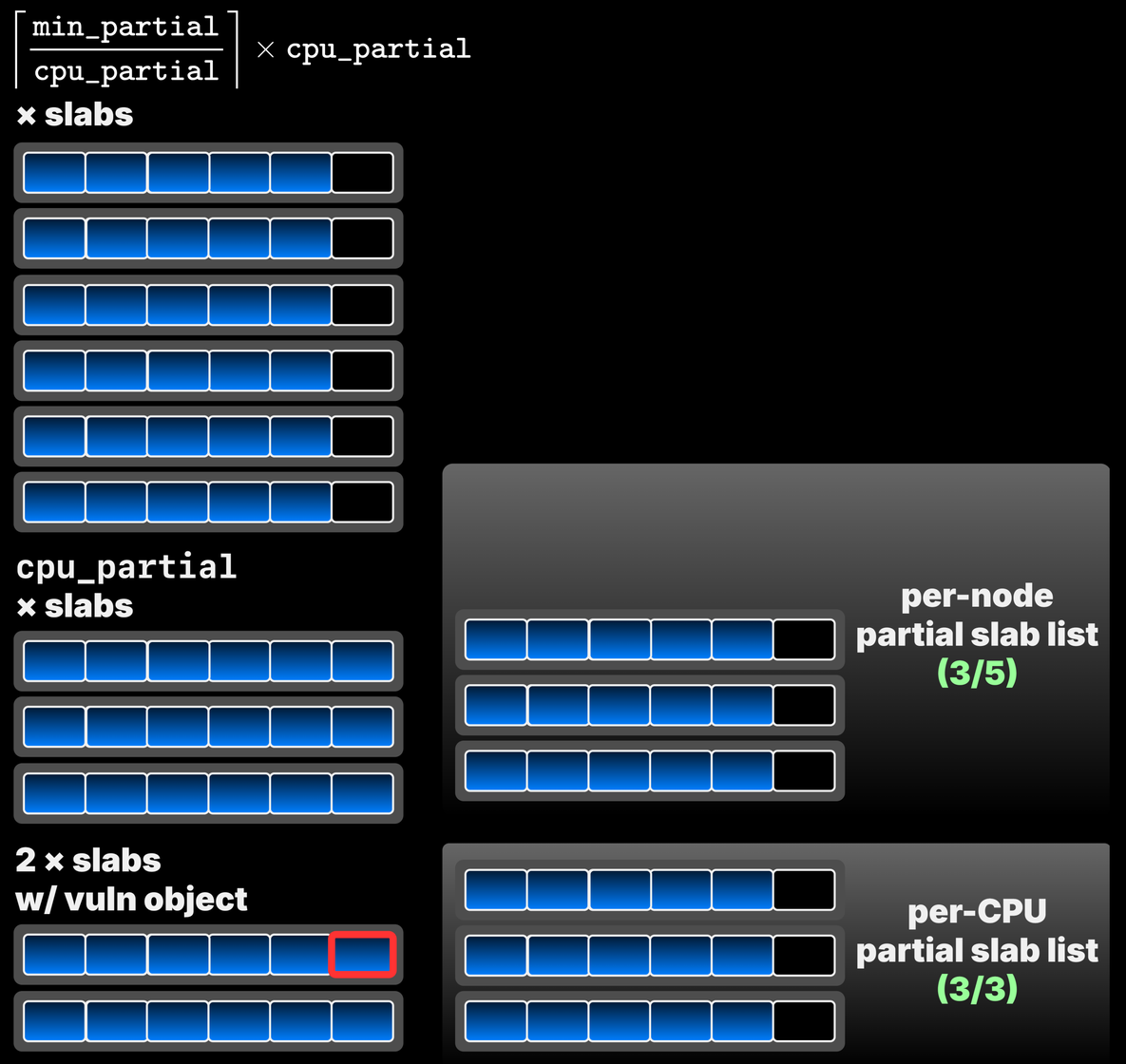
Complete Free

- **Strategy workflow:**
  1. Allocate $\left\lceil \frac{\texttt{min\_partial}}{\texttt{cpu\_partial}} \right\rceil \times \texttt{cpu\_partial}$ slabs
  2. Allocate (cpu_partial+1) slabs
  3. Allocate 2 slabs with vulnerable object
  4. **Free one object per slab from step 1**



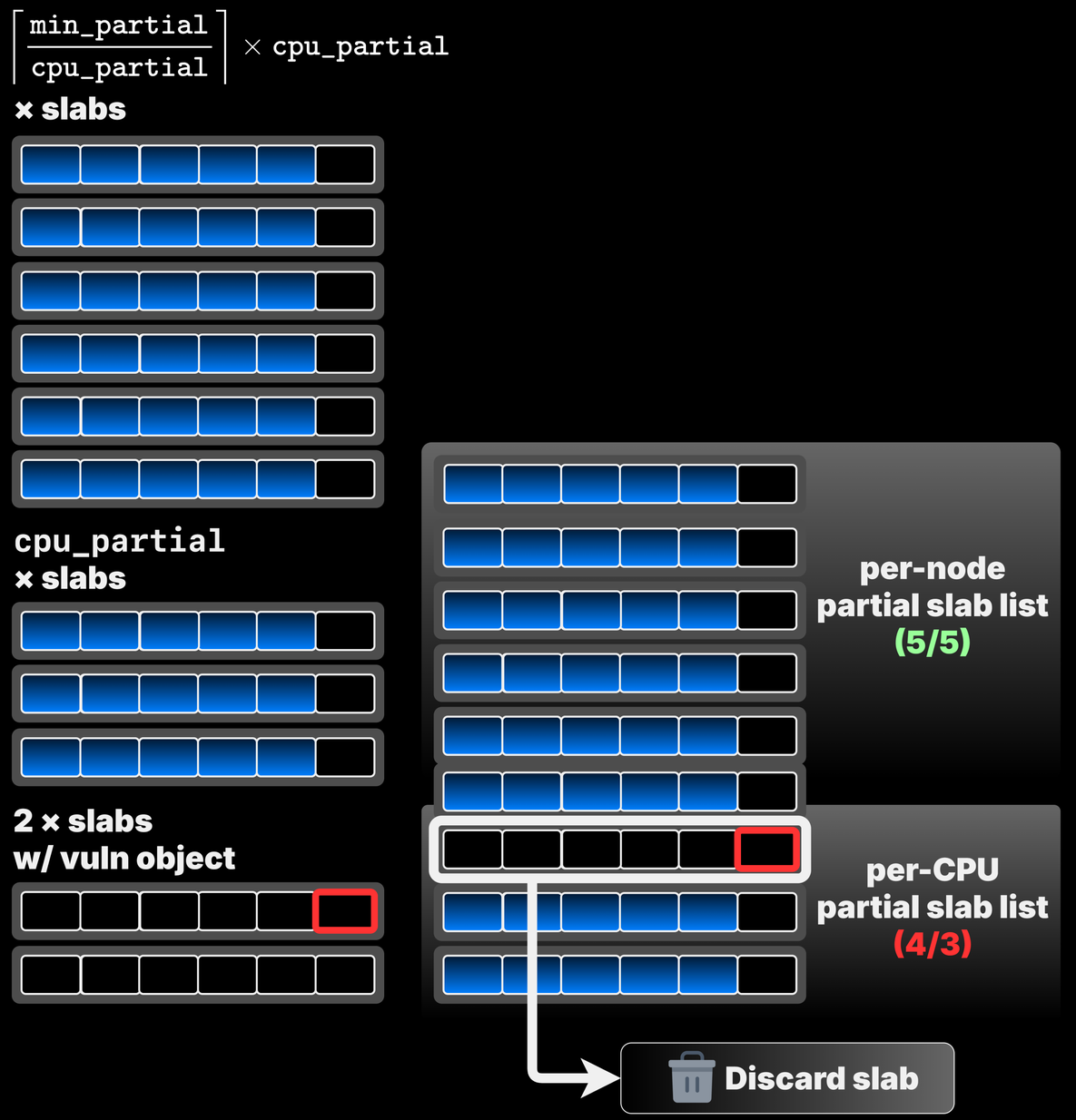
Complete Free

- **Strategy workflow:**
  1. Allocate $\left\lceil \frac{\texttt{min\_partial}}{\texttt{cpu\_partial}} \right\rceil \times \texttt{cpu\_partial}$ slabs
  2. Allocate (cpu_partial+1) slabs
  3. Allocate 2 slabs with vulnerable object
  4. **Free one object per slab from step 1**



Complete Free

- **Strategy workflow:**
  1. Allocate $\left\lceil \frac{\texttt{min\_partial}}{\texttt{cpu\_partial}} \right\rceil \times \texttt{cpu\_partial}$ slabs
  2. Allocate (cpu_partial+1) slabs
  3. Allocate 2 slabs with vulnerable object
  4. Free one object per slab from step 1
  5. Free both slabs from step 3
  6. **Free one object per slab from step 2**
- **Successfully discarded slab!**



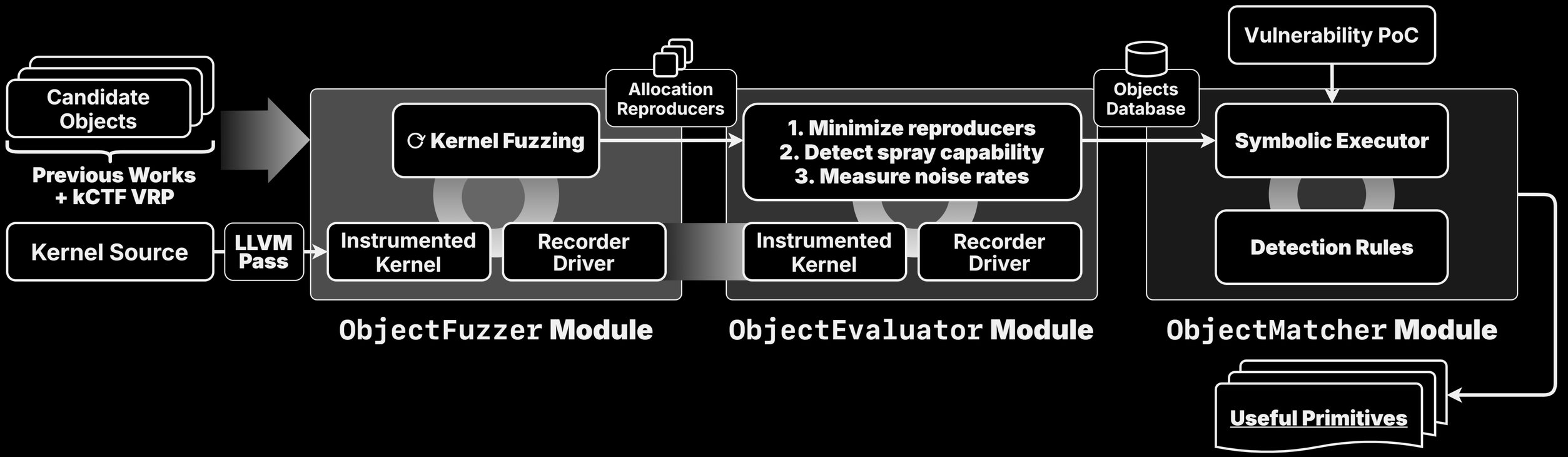
Target Object Properties

- Not all objects identified in previous works are suitable for Cross-Cache Attacks.
- What then qualifies as a versatile target objects?

Target Object Properties

- Not all objects identified in previous works are suitable for Cross-Cache Attacks.
- What then qualifies as a versatile target objects?
  1. **Spray Capability**
  2. **Minimal Interferering Allocations**
     - **Interfering allocation:** Allocation of unrelated objects.
     - **Noise Rates:** Number of interfering allocation during single object allocation.
  3. **Useful Primitives**

CROSS-X System



Evaluation

- To evaluate our findings, we performed three experiments:
  1. **Stability experiment** using a synthetic vulnerability
  2. **Object identification** from a set of 346 candidate objects
  3. **Real-world exploitability experiment** extending the stability test across 9 CVEs

Table 3: Performance of recycling strategies across various kernel versions and system workloads. For each object size and workload, the highest success rates are both bolded and underlined. The bottom row, marked with 🏆, indicates the number of winning cases where each strategy outperforms the others. The highest count is underlined.

(a) v5.15 in Idle State

Size	Naïve	Partial Free	Complete Free	SLUBStick + C.F.
16	0.13%	**100.00%**	**100.00%**	**100.00%**
32	80.03%	**100.00%**	93.33%	0.00%
64	40.07%	**100.00%**	**100.00%**	**100.00%**
96	0.60%	95.97%	**96.67%**	46.63%
128	10.10%	93.33%	93.33%	**100.00%**
192	0.43%	80.00%	86.67%	**99.87%**
256	3.33%	86.67%	96.67%	**100.00%**
512	3.33%	73.33%	93.33%	**99.97%**
1024	30.40%	73.33%	86.67%	**90.03%**
2048	17.03%	**100.00%**	76.67%	43.40%
4096	0.00%	0.00%	60.00%	**100.00%**
🏆	0/11	4/11	3/11	**8/11**

(b) v5.15 in Busy State

Size	Naïve	Partial Free	Complete Free	SLUBStick + C.F.
16	0.00%	84.63%	**91.13%**	0.00%
32	0.00%	**83.67%**	67.17%	0.27%
64	7.77%	93.20%	93.03%	**97.73%**
96	38.93%	96.37%	**97.90%**	39.57%
128	41.10%	96.10%	**96.93%**	96.57%
192	38.00%	79.60%	92.77%	**99.67%**
256	22.77%	72.27%	49.70%	**99.80%**
512	24.77%	75.90%	92.03%	**99.07%**
1024	38.50%	**59.97%**	43.73%	29.23%
2048	11.67%	**84.53%**	79.47%	14.77%
4096	0.00%	0.00%	85.87%	**98.27%**
🏆	0/11	3/11	3/11	**5/11**

(c) v6.1 in Idle State

Size	Naïve	Partial Free	Complete Free	SLUBStick + C.F.
16	0.00%	0.00%	99.93%	**100.00%**
32	0.07%	0.00%	99.97%	**100.00%**
64	83.80%	0.00%	**100.00%**	**100.00%**
96	7.17%	99.70%	**100.00%**	**100.00%**
128	93.27%	86.67%	96.67%	**100.00%**
192	99.40%	96.67%	**100.00%**	**100.00%**
256	72.60%	96.67%	**100.00%**	**100.00%**
512	56.53%	96.67%	93.33%	**99.97%**
1024	81.30%	0.00%	96.67%	**100.00%**
2048	99.03%	0.00%	**100.00%**	**100.00%**
4096	0.00%	0.00%	96.67%	**99.97%**
🏆	0/11	0/11	8/11	**11/11**

(d) v6.1 in Busy State

Size	Naïve	Partial Free	Complete Free	SLUBStick + C.F.
16	0.00%	0.00%	**85.83%**	0.00%
32	3.80%	1.37%	**33.20%**	6.27%
64	9.93%	0.03%	96.10%	**97.50%**
96	9.87%	80.30%	**97.53%**	97.20%
128	35.40%	88.83%	92.03%	**98.37%**
192	40.57%	**99.07%**	96.20%	98.67%
256	34.17%	92.73%	90.47%	**99.37%**
512	49.90%	84.90%	89.73%	**98.77%**
1024	62.50%	0.60%	**91.60%**	58.07%
2048	78.37%	0.00%	92.60%	**97.47%**
4096	0.00%	0.00%	89.63%	**98.60%**
🏆	0/11	0/11	4/11	**6/11**

(e) v6.6 in Idle State

Size	Naïve	Partial Free	Complete Free	SLUBStick + C.F.
16	0.00%	0.00%	96.53%	**99.60%**
32	0.00%	0.00%	**100.00%**	99.70%
64	**99.97%**	0.00%	**99.97%**	98.87%
96	5.27%	93.30%	90.00%	**97.97%**
128	98.07%	86.67%	83.33%	**98.50%**
192	**98.33%**	83.33%	80.00%	95.60%
256	**99.90%**	86.67%	93.33%	96.30%
512	99.90%	93.33%	**100.00%**	96.10%
1024	96.87%	0.00%	86.67%	**97.37%**
2048	**99.10%**	0.00%	80.00%	98.33%
4096	0.00%	0.00%	86.67%	**97.70%**
🏆	4/11	0/11	3/11	**5/11**

(f) v6.6 in Busy State

Size	Naïve	Partial Free	Complete Free	SLUBStick + C.F.
16	0.00%	0.00%	52.27%	**79.40%**
32	66.60%	0.00%	81.10%	**85.27%**
64	66.93%	0.00%	**90.53%**	39.30%
96	54.53%	85.20%	**89.57%**	77.87%
128	16.10%	9.50%	**93.70%**	77.70%
192	33.33%	**80.53%**	75.27%	56.03%
256	31.83%	**87.57%**	15.87%	5.37%
512	0.73%	92.10%	**92.40%**	88.17%
1024	17.60%	0.00%	88.87%	**93.17%**
2048	0.17%	0.37%	75.13%	**94.00%**
4096	0.00%	0.00%	69.00%	**94.50%**
🏆	0/11	2/11	4/11	**5/11**

Table 3: Performance of recycling strategies across various kernel versions and system workloads. For each object size and workload, the highest success rates are both bolded and underlined. The bottom row, marked with 🏆, indicates the number of winning cases where each strategy outperforms the others. The highest count is underlined.

(a) v5.15 in Idle State

Size	Naïve	Partial Free	Complete Free	SLUBStick + C.F.
16	0.13%	**100.00%**	**100.00%**	**100.00%**
32	80.03%	**100.00%**	93.33%	0.00%
64	40.07%	**100.00%**	**100.00%**	**100.00%**
96	0.60%	95.97%	**96.67%**	46.63%
128	10.10%	93.33%	93.33%	**100.00%**
192	0.43%	80.00%	86.67%	**99.87%**
256	3.33%	86.67%	96.67%	**100.00%**

(c) v6.1 in Idle State

Size	Naïve	Partial Free	Complete Free	SLUBStick + C.F.
16	0.00%	0.00%	99.93%	**100.00%**
32	0.07%	0.00%	99.97%	**100.00%**
64	83.80%	0.00%	**100.00%**	**100.00%**
96	7.17%	99.70%	**100.00%**	**100.00%**
128	93.27%	86.67%	96.67%	**100.00%**
192	99.40%	96.67%	**100.00%**	**100.00%**
256	72.60%	96.67%	**100.00%**	**100.00%**

(e) v6.6 in Idle State

Size	Naïve	Partial Free	Complete Free	SLUBStick + C.F.
16	0.00%	0.00%	96.53%	**99.60%**
32	0.00%	0.00%	**100.00%**	99.70%
64	**99.97%**	0.00%	**99.97%**	98.87%
96	5.27%	93.30%	90.00%	**97.97%**
128	98.07%	86.67%	83.33%	**98.50%**
192	**98.33%**	83.33%	80.00%	95.60%
256	**99.90%**	86.67%	93.33%	96.30%

We achieved **over 99%** (idle) and **85%** (busy) success across most caches.

(b) v5.15 in Busy State

Size	Naïve	Partial Free	Complete Free	SLUBStick + C.F.
16	0.00%	84.63%	**91.13%**	0.00%
32	0.00%	**83.67%**	67.17%	0.27%
64	7.77%	93.20%	93.03%	**97.73%**
96	38.93%	96.37%	**97.90%**	39.57%
128	41.10%	96.10%	**96.93%**	96.57%
192	38.00%	79.60%	92.77%	**99.67%**
256	22.77%	72.27%	49.70%	**99.80%**
512	24.77%	75.90%	92.03%	**99.07%**
1024	38.50%	**59.97%**	43.73%	29.23%
2048	11.67%	**84.53%**	79.47%	14.77%
4096	0.00%	0.00%	85.87%	**98.27%**
🏆	0/11	3/11	3/11	**5/11**

(d) v6.1 in Busy State

Size	Naïve	Partial Free	Complete Free	SLUBStick + C.F.
16	0.00%	0.00%	**85.83%**	0.00%
32	3.80%	1.37%	**33.20%**	6.27%
64	9.93%	0.03%	96.10%	**97.50%**
96	9.87%	80.30%	**97.53%**	97.20%
128	35.40%	88.83%	92.03%	**98.37%**
192	40.57%	**99.07%**	96.20%	98.67%
256	34.17%	92.73%	90.47%	**99.37%**
512	49.90%	84.90%	89.73%	**98.77%**
1024	62.50%	0.60%	**91.60%**	58.07%
2048	78.37%	0.00%	92.60%	**97.47%**
4096	0.00%	0.00%	89.63%	**98.60%**
🏆	0/11	0/11	4/11	**6/11**

(f) v6.6 in Busy State

Size	Naïve	Partial Free	Complete Free	SLUBStick + C.F.
16	0.00%	0.00%	52.27%	**79.40%**
32	66.60%	0.00%	81.10%	**85.27%**
64	66.93%	0.00%	**90.53%**	39.30%
96	54.53%	85.20%	**89.57%**	77.87%
128	16.10%	9.50%	**93.70%**	77.70%
192	33.33%	**80.53%**	75.27%	56.03%
256	31.83%	**87.57%**	15.87%	5.37%
512	0.73%	92.10%	**92.40%**	88.17%
1024	17.60%	0.00%	88.87%	**93.17%**
2048	0.17%	0.37%	75.13%	**94.00%**
4096	0.00%	0.00%	69.00%	**94.50%**
🏆	0/11	2/11	4/11	**5/11**

Table 4: Summary of Object Identification Results. Each row builds on previous findings. For example, objects with high success rates are verified as sprayable.

Description	Number of Objects
Total Candidate Objects	346
↪ Present in the Target Kernel	248
↪ Allocation Reproducers Generated	53
↪ Confirmed Spray Capability	**11**
↪ Exhibited High Success Rates	**7**

Table 5: Results of cross-cache experiments for sprayable objects. NR_{SC}, NR_{SO}, and NR_{DO} represent noise rates for objects in the same cache, same-order cache, and different-order cache, respectively. Objects shaded in gray exhibit low NR_{SC} and NR_{SO} values. (NR_{SC}= 0, NR_{SO}< 3.5)

Object	NR_{SC}	NR_{SO}	NR_{DO}	Success Rate
sock_fprog_kern	1	52.3	366.52	0.0%
sk_security_struct	0	15.31	102.55	0.0%
snd_info_buffer	1	9.35	7.61	28.7%
kernfs_open_file	0	1.64	0.71	**99.7%**
timerfd_ctx	0	1.85	1.82	**99.8%**
io_ring_ctx	0	1.66	0.00	10.9%
msg_msg	0	0.15	0.35	**99.7%**
pipe_inode_info	0	3.42	9.98	**94.2%**
pipe_buffer	0	0.00	0.68	**99.9%**
fsnotify_group	0	2.20	3.16	**99.9%**
shmid_kernel	0	0.89	3.02	**99.8%**

Table 4: Summary of Object Identification Results. Each row builds on previous findings. For example, objects with high success rates are verified as sprayable.

↪ Present in the Target Kernel	248
↪ Allocation Reproducers Generated	53
↪ **Confirmed Spray Capability**	**11**
↪ **Exhibited High Success Rates**	**7**

CROSS-X identified **7 low-noise objects** under same-order allocations.

Table 5: Results of cross-cache experiments for sprayable objects. NR_{SC}, NR_{SO}, and NR_{DO} represent noise rates for objects in the same cache, same-order cache, and different-order cache, respectively. Objects shaded in gray exhibit low NR_{SC} and NR_{SO} values. (NR_{SC}= 0, NR_{SO}< 3.5)

Object	NR_{SC}	NR_{SO}	NR_{DO}	Success Rate
kernfs_open_file	0	1.64	0.71	**99.7%**
timerfd_ctx	0	1.85	1.82	**99.8%**
io_ring_ctx	0	1.66	0.00	10.9%
msg_msg	0	0.15	0.35	**99.7%**
pipe_inode_info	0	3.42	9.98	**94.2%**
pipe_buffer	0	0.00	0.68	**99.9%**
fsnotify_group	0	2.20	3.16	**99.9%**
shmid_kernel	0	0.89	3.02	**99.8%**

Table 6: Results of real-world experiments. Gray-highlighted CVEs had limitations (e.g., no measurement, deferred free) in applying SLUBStick + Complete Free.

(a) In Idle State

CVE	Naïve	Partial Free	Complete Free	SLUBStick + C.F.
CVE-2023-20938	0.00%	36.67%	**38.00%**	19.33%
CVE-2023-3609	0.00%	94.67%	95.33%	**99.33%**
CVE-2023-5345	0.00%	85.33%	**86.67%**	83.00%
CVE-2020-29660	0.00%	75.67%	**77.00%**	N/A
CVE-2022-2588	28.67%	**92.67%**	91.00%	N/A
CVE-2022-32250	1.67%	**99.33%**	98.33%	N/A
CVE-2022-2585	99.67%	98.67%	**100.00%**	N/A
CVE-2023-3910	0.00%	**100.00%**	99.67%	N/A
CVE-2023-2235	99.67%	0.00%	**100.00%**	N/A

(b) In Busy State

CVE	Naïve	Partial Free	Complete Free	SLUBStick + C.F.
CVE-2023-20938	0.00%	32.00%	**40.67%**	25.33%
CVE-2023-3609	0.00%	**98.00%**	97.67%	95.00%
CVE-2023-5345	0.00%	82.00%	**83.00%**	78.33%
CVE-2020-29660	0.00%	73.67%	**75.33%**	N/A
CVE-2022-2588	2.00%	21.33%	**34.67%**	N/A
CVE-2022-32250	19.00%	**63.67%**	60.33%	N/A
CVE-2022-2585	**23.00%**	10.33%	12.00%	N/A
CVE-2023-3910	1.33%	3.33%	**3.67%**	N/A
CVE-2023-2235	31.00%	0.00%	**98.67%**	N/A

Table 6: Results of real-world experiments. Gray-highlighted CVEs had limitations (e.g., no measurement, deferred free) in applying SLUBStick + Complete Free.

(a) In Idle State

CVE	Naïve	Partial Free	Complete Free	SLUBStick + C.F.
CVE-2023-20938	0.00%	36.67%	**38.00%**	19.33%
CVE-2023-3609	0.00%	94.67%	95.33%	**99.33%**
CVE-2023-5345	0.00%	85.33%	**86.67%**	83.00%
CVE-2020-29660	0.00%	75.67%	**77.00%**	N/A
CVE-2022-2588	28.67%	**92.67%**	91.00%	N/A

Complete Free performed **reliably** and **often ranked best** across 9 real-world CVEs.

(b) In Busy State

CVE	Naïve	Partial Free	Complete Free	SLUBStick + C.F.
CVE-2023-20938	0.00%	32.00%	**40.67%**	25.33%
CVE-2023-3609	0.00%	**98.00%**	97.67%	95.00%
CVE-2023-5345	0.00%	82.00%	**83.00%**	78.33%
CVE-2020-29660	0.00%	73.67%	**75.33%**	N/A
CVE-2022-2588	2.00%	21.33%	**34.67%**	N/A
CVE-2022-32250	19.00%	**63.67%**	60.33%	N/A
CVE-2022-2585	**23.00%**	10.33%	12.00%	N/A
CVE-2023-3910	1.33%	3.33%	**3.67%**	N/A
CVE-2023-2235	31.00%	0.00%	**98.67%**	N/A

Conclusion

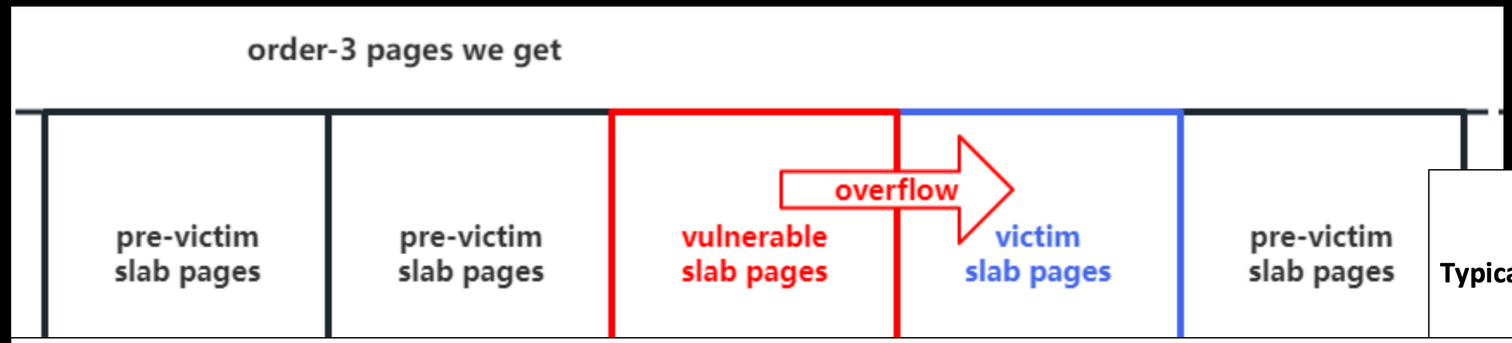
- Cross-Cache Attack reliability **has been limited** by incomplete understanding.
- We present new insights and propose **two optimized strategies** along with an **CROSS-X automated system.**
- We identified **7 suitable objects** and achieved **high stability** across various environments.

thank you.

juhyun167@kaist.ac.kr

Discussion

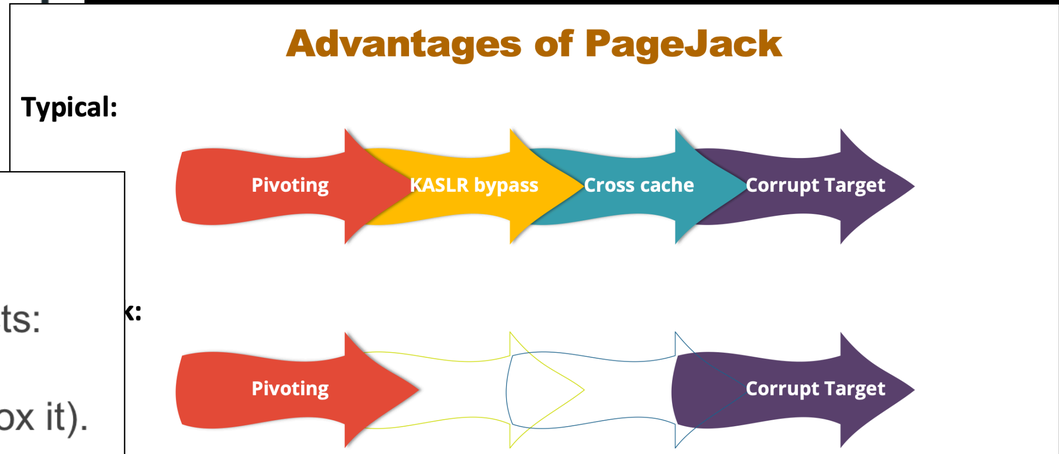
- **Mitigations**
  - Slab freelist defenses and cache randomization **do not impact** Cross-Cache Attacks.
  - `SLAB_VIRTUAL` mitigates Cross-Cache Attacks but is not widely adopted.
- **Cross-Cache Overflow** is also feasible but demands a clever **page fengshui** strategy.
- Other page reusing techniques exist but are more easily countered by targeted defenses.



Limiting io_uring

To protect our users, we decided to limit the usage of io_uring in Google products:

- **ChromeOS:** We disabled io_uring (while we explore new ways to sandbox it).



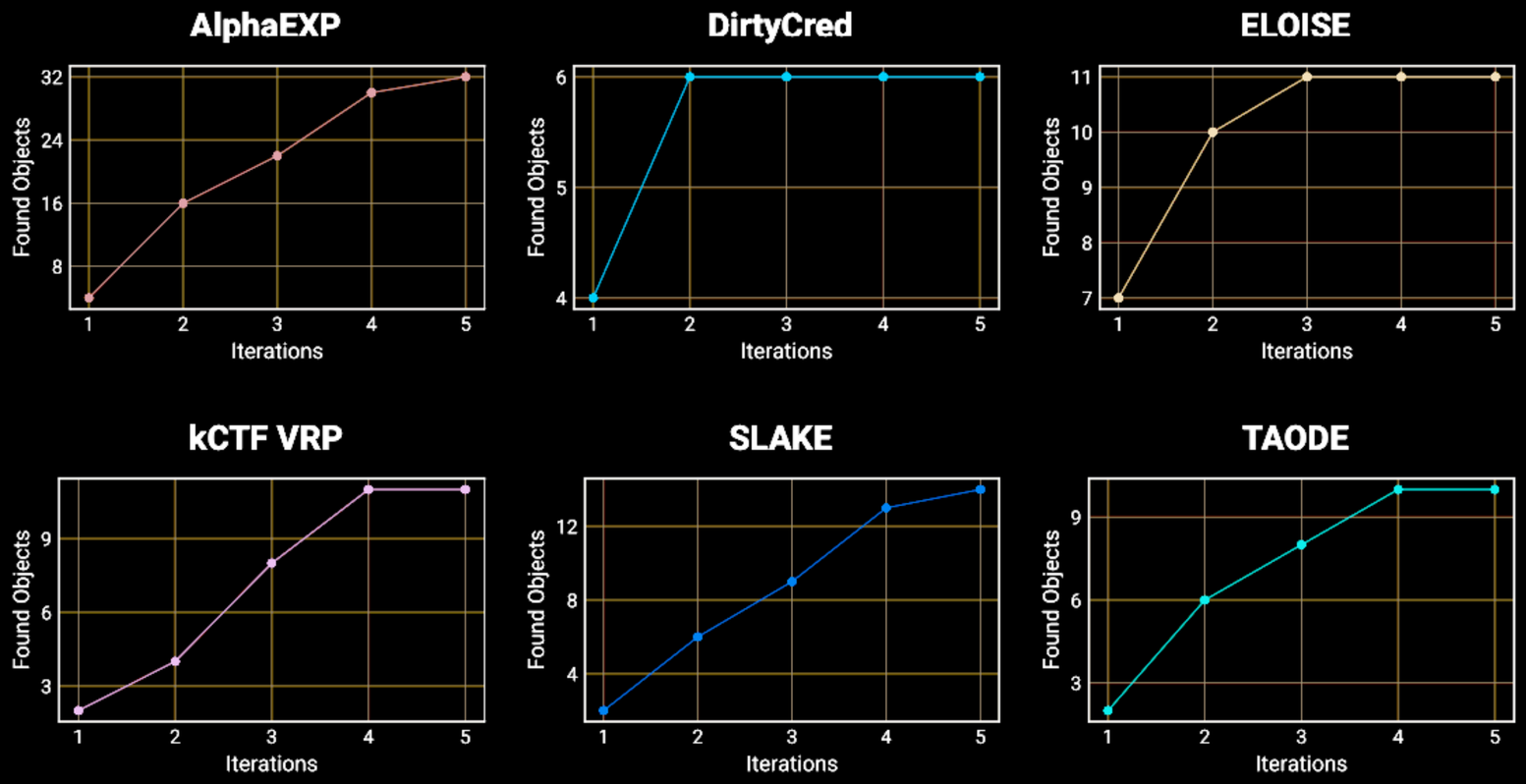


Figure 9: Accumulated number of discovered objects during preliminary testing of the `ObjectFuzzer` module. The module was run for 0.5 hours per iteration, repeated 5 times. Convergence was observed in 4 out of 6 object candidate sets (DirtyCred, ELOISE, kCTF VRP, TAODE), and near convergence in the remaining 2 (AlphaEXP, SLAKE).

Table 1: Success rate of triggering the recycling and reclamation process for generic caches.

Generic Cache	#Pages	Success Rate		
		Idle	No CPU pinning	External noise
		%	%	%
kmalloc-8	1	99.9 ± 0.1	99.9 ± 0.1	99.6 ± 0.7
kmalloc-16	1	99.4 ± 0.6	98.9 ± 1.2	99.9 ± 0.4
kmalloc-32	1	99.4 ± 0.9	99.7 ± 0.5	99.9 ± 0.3
kmalloc-64	1	99.2 ± 1.3	99.2 ± 0.9	81.0 ± 6.4
kmalloc-96	1	99.9 ± 0.4	99.9 ± 0.1	99.8 ± 0.6
kmalloc-128	1	99.9 ± 0.4	99.8 ± 0.5	99.9 ± 0.3
kmalloc-192	1	99.9 ± 0.4	99.8 ± 0.4	99.3 ± 1.2
kmalloc-256	1	99.9 ± 0.3	99.9 ± 0.3	99.7 ± 0.7
kmalloc-512	2	90.2 ± 5.4	87.2 ± 3.1	65.2 ± 2.8
kmalloc-1024	4	88.1 ± 7.2	79.5 ± 3.3	70.3 ± 8.1
kmalloc-2048	8	83.1 ± 9.2	70.5 ± 16	57.8 ± 5.7
kmalloc-4096	8	82.1 ± 3.4	73.3 ± 19	53.8 ± 10

Table 7: Results of stability experiments with SLUBStick.

(a) In Idle State

Size	SLUBStick	Complete Free
16	86.40%	**100.00%**
32	93.03%	**100.00%**
64	96.03%	**99.13%**
96	97.40%	**100.00%**
128	**97.67%**	96.67%
192	**98.73%**	96.67%
256	**97.70%**	93.33%

(b) In Busy State

Size	SLUBStick	Complete Free
16	38.10%	**96.67%**
32	46.47%	**87.23%**
64	69.23%	**93.33%**
96	77.23%	**96.60%**
128	85.27%	**96.57%**
192	88.90%	**89.90%**
256	**90.40%**	87.10%

Table 8: Detailed information on objects with confirmed spray capabilities from experiments. Objects with a grey background exhibited low NR_{SC} and NR_{SO}, while bolded objects represent the final confirmed suitable selections.

Object	Slab Cache	Allocation Site	Allocation System Call
sock_fprog_kern	kmalloc-16	bpf_prog_store_orig_filter in net/core/filter.c	setsockopt
sk_security_struct	kmalloc-32	selinux_sk_alloc_security in security/selinux/hooks.c	io_uring_setup
snd_info_buffer	kmalloc-32	snd_info_text_entry_open in sound/core/info.c	openat
kernfs_open_file	kmalloc-192	kernfs_fop_open in fs/kernfs/file.c	openat
timerfd_ctx	kmalloc-256	sys_timerfd_create in fs/timerfd.c	timerfd_create
io_ring_ctx	kmalloc-2048	io_ring_ctx_alloc in io_uring/io_uring.c	io_uring_setup
msg_msg	kmlloc-cg-64[1]	alloc_msg in ipc/msgutil.c	msgsnd
pipe_inode_info	kmalloc-cg-192	alloc_pipe_info in fs/pipe.c	pipe
fsnotify_group	kmalloc-cg-256	inotify_new_group in fs/notify/inotify/inotify_user.c	inotify_init
shmid_kernel	kmalloc-cg-256	newseg in ipc/shm.c	shmget
pipe_buffer	kmalloc-cg-1024	alloc_pipe_info in fs/pipe.c	pipe

[1] msg_msg is indeed an elastic object that can grow up to 4,096 bytes in size. The above is merely based on ObjectFuzzer results.

Table 9: Detailed environmental setup for each CVE evaluated in the real-world experiment. The Objective column specifies the critical operation whose successful execution indicates an effective exploit, accomplished through cross-cache attacks with objects listed in the Target Object column.

CVE	Vulnerability Type	Vulnerable Cache	Deallocation	Target Object	Objective	SLUBStick+C.F. Availability
CVE-2023-20938	UAF	`kmalloc-128`	Direct	`fsnotify_group`	Kernel Heap Leak	✓
CVE-2023-3609	UAF	`kmalloc-128`	Direct	`msg_msg`	Control Flow Hijacking	✓
CVE-2023-5345	DF	`kmalloc-128`	Direct	`msg_msg`	Kernel Heap Leak	✓
CVE-2020-29660	UAF	`pid`[1]	Direct	page table entries	Page Table Corruption	✗[2]
CVE-2022-2588	DF	`kmalloc-192`	Worker	`msg_msg`	Payload Delivery	✗
CVE-2022-32250	UAF	`kmalloc-64`	Worker	`msg_msg`	Kernel Heap Leak	✗
CVE-2022-2585	UAF	`posix_timer_cache`	RCU	`msg_msg`	Kernel Heap Leak	✗
CVE-2023-3910	UAF	`filp`	Worker	`pipe` buffer[3]	Kernel Base Leak	✗
CVE-2023-2235	UAF	`perf_event`	RCU	`msg_msg`	Kernel Heap Leak	✗

[1] This cache is merged with the `seq_file` and `eventpoll_epi` caches, enabling the use of `seq_file` and `epitem` objects during exploitation.
[2] Despite not using deferred frees, we could not find a method to enable measurement primitives (immediate free within the same system call) for this exploit.
[3] For CVE-2023-3910, we used the page spraying technique employed in the original exploit.