ARTIFACT EVALUATED usenix ASSOCIATION AVAILABLE

ARTIFACT EVALUATED usenix ASSOCIATION FUNCTIONAL

ARTIFACT EVALUATED usenix ASSOCIATION REPRODUCED

SeaK: Rethinking the Design of a Secure Allocator for OS Kernel

Zicheng Wang, **Yicheng Guang**, Yueqi Chen

Zhenpeng Lin, Michael Le, Dang K Le

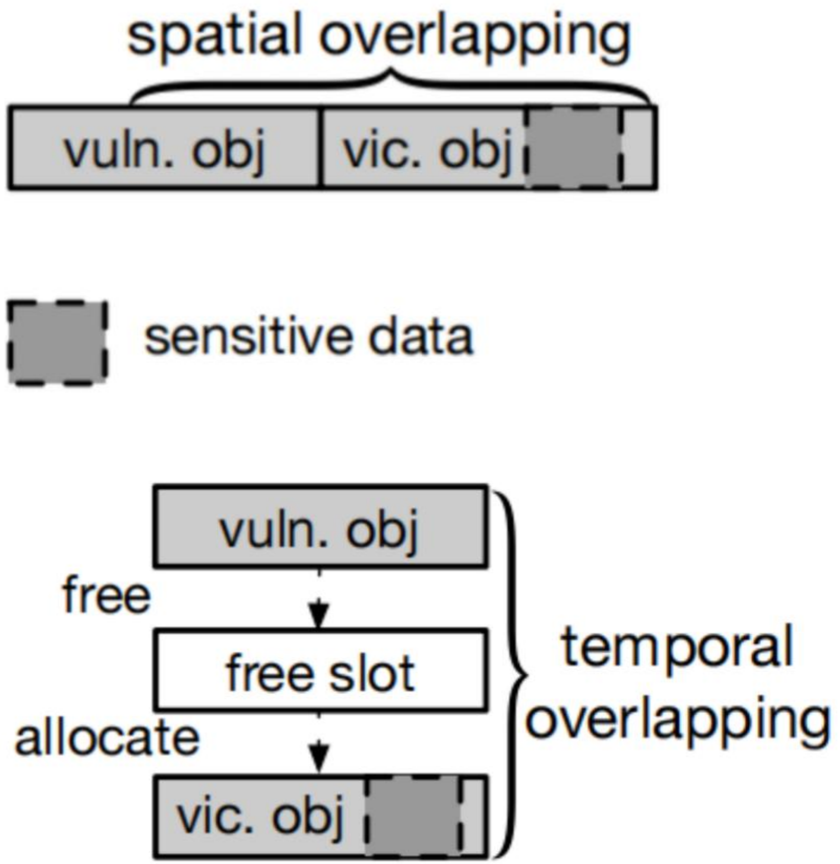
Dan Williams, Xinyu Xing, Zhongshu Gu, Hani Jamjoom

南京大學 NANJING UNIVERSITY

Northwestern University IBM VIRGINIA TECH

Summary of (Linux) Kernel Heap Exploits

- Taxonomy:
  - spatial/temporal overlapping
  - within/cross cache
- Essence: overlapping between corruptions introduced by vulnerable objects and sensitive data in victim objects



Existing Linux Kernel Hardenings

- By-default enabled (C1): freelist randomization, freelist obfuscation, and heap zeroing
- By-default disabled (C2): KFENCE, structure layout randomization
- Lightweight “debugging” (C3): slub_debug

Exploits	C1	C2	C3
2021-4154 (exp1)	○	○/◑	●
2021-22600 (exp3)	○	○/◑	●
2022-0185 (exp4)	○	○/◑	●
2022-27666 (exp6)	○	○/◑	●
2022-29582 (exp9)	○	○/◑	●
2022-1786 (exp13)	○	○/◑	●
2022-20409 (exp15)	○	○/◑	○

Hardenings in C1 are widely bypassed

In C2, KFENCE can isolate only 0.005% -0.35% target objects; Securely storing random seed is challenging in structure layout randomization

C3 can be bypassed by Dirtycred attack (exp15)

Existing Linux Kernel Hardenings (cont.)

In addition, C3 has significant performance overhead.

Used as a debugging feature by default.

LMbench	C1	C2	C3
Simple syscall	0.35%	1.06%	0.90%
Simple read	0.98%	3.73%	0.70%
Simple write	0.41%	1.71%	2.46%
Select on 100 fd's	-0.64%	1.21%	0.04%
Signal handler install	-1.35%	-1.88%	-1.17%
Signal handler overhead	0.75%	3.29%	169.16%
fork+exit	0.60%	1.76%	168.17%
fork+execve	2.42%	1.56%	177.22%
fork+/bin/sh -c	1.21%	2.32%	151.55%
UDP latency	3.91%	4.97%	144.34%
TCP/IP connection	-2.74%	5.25%	129.81%
AF_UNIX bandwidth	-0.20%	0.27%	52.16%
Pipe bandwidth	0.80%	1.16%	-1.98%

Phoronix	C1	C2	C3
Sockperf (Msgs/sec)	-0.27%	-0.61%	57.58%
OSBench (Ns/Event)	-0.08%	-1.00%	6.25%
7-Zip Compress (MIPS)	-0.34%	0.54%	-0.39%
FFmpeg Live (FPS)	-0.14%	0.28%	1.25%
OpenSSL SHA256 (B/s)	0.01%	0.04%	0.01%
Redis SET (Reqs/sec)	-0.37%	0.47%	0.55%
SQLite Speedtest (sec)	0.52%	1.34%	4.05%
Apache 100 (Reqs/sec)	-0.50%	-0.42%	46.29%

Our Insight

- Trade-off between overhead and effectiveness persists if we protect every kernel object
- Do we really need to protect every object?

What really matters is exploit-critical objects

Research on Exploit-critical Objects

- Vulnerable objects vary from bug to bug
- We keep finding new victim objects

Challenge: It's impossible to have an oracle set of all exploit-critical objects

GREBE: Unveiling Exploitation Potential for Linux Kernel Bugs

Zhenpeng Lin[††*], Yueqi Chen[††], Yuhang Wu[††], Dongliang Mu[†‖], Chensheng Yu[‡], Xinyu Xing[§‖], Kang li[¶]
[††]The Pennsylvania State University [†]School of Cyber Science and Engineering, HUST
[‡]George Washington University [§]Northwestern University [¶]Baidu USA
{zplin, ycx431, yuhang}@psu.edu, dzm91@hust.edu.cn,
i@shiki7.me, xinyu.xing@northwestern.edu, kangli01@baidu.com

SLAKE: Facilitating Slab Manipulation for Exploiting Vulnerabilities in the Linux Kernel

SCAVY: Automated Discovery of Memory Corruption Targets in Linux Kernel for Privilege Escalation

A Systematic Study of Elastic Objects in Kernel Exploitation

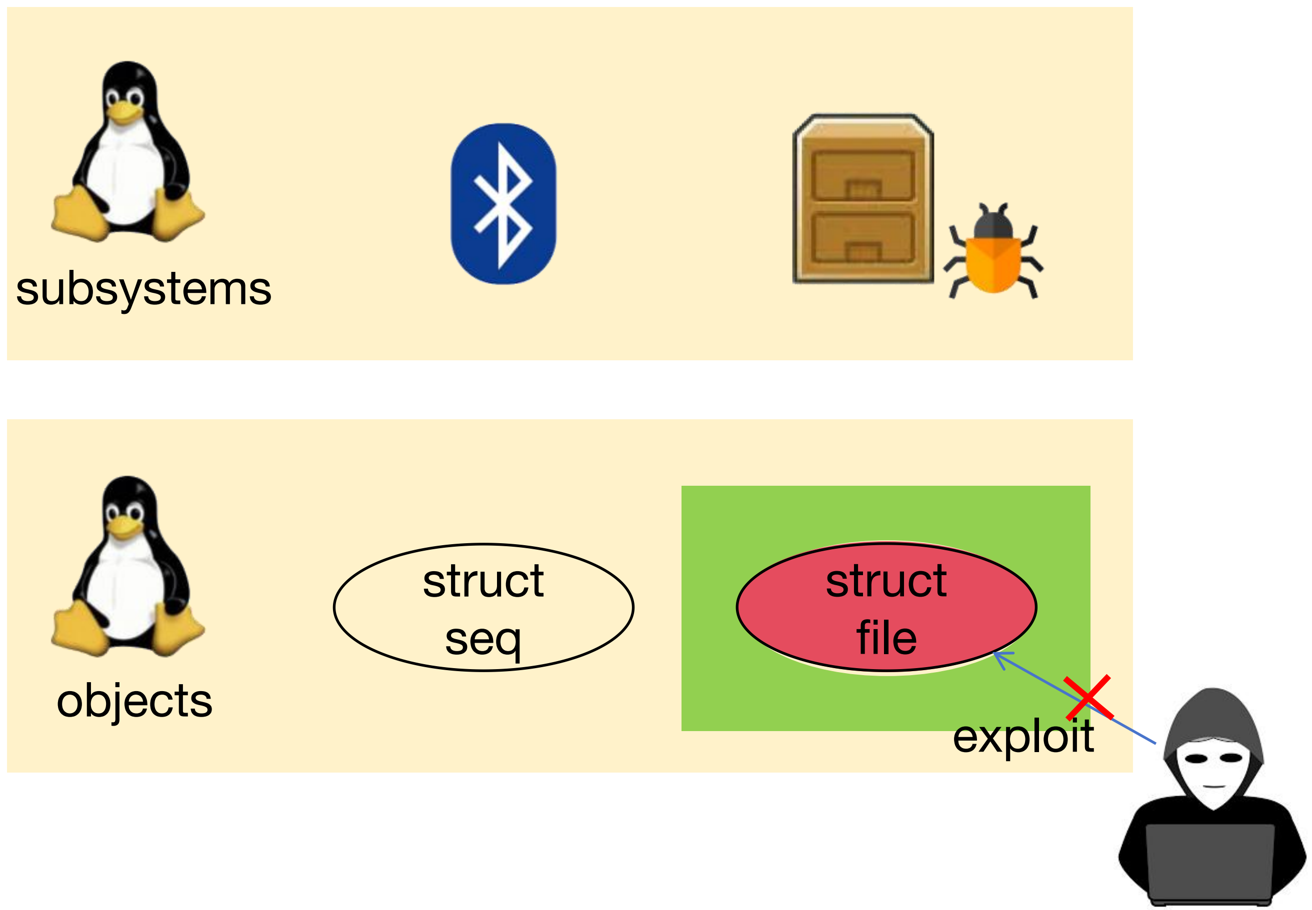
Yueqi Chen
ychen@ist.psu.edu
The Pennsylvania State University

Zhenpeng Lin
zplin@psu.edu
The Pennsylvania State University

Xinyu Xing
xxing@ist.psu.edu
The Pennsylvania State University

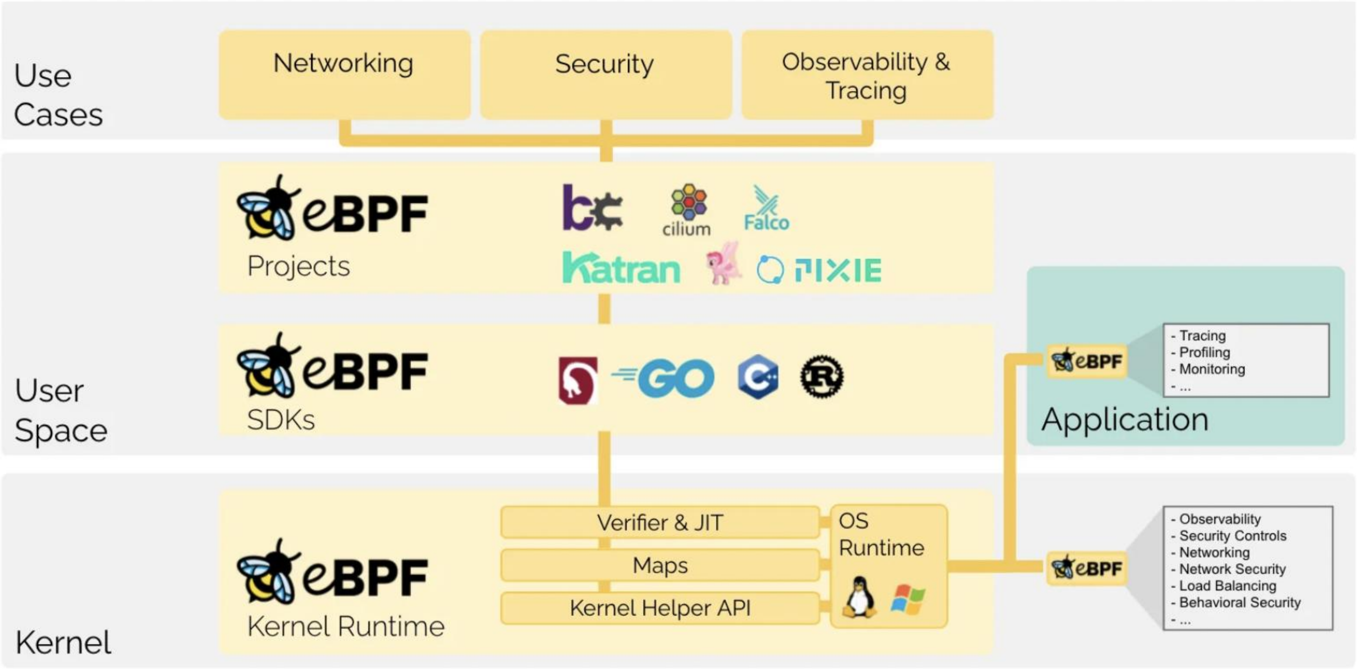
Key Idea: An On-demand Secure Allocator

- Protection on demand
- Type granularity, named atomic alleviation
- Dynamic enforcement



Technical Background: eBPF

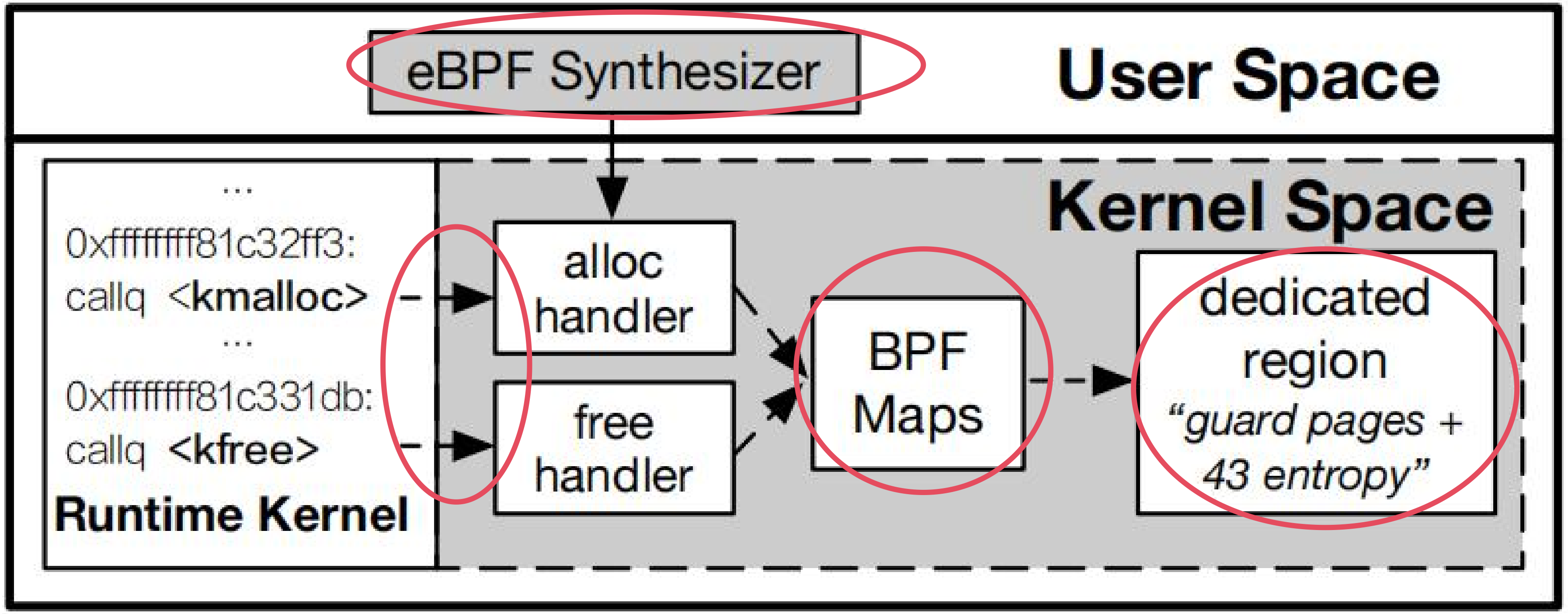
- In-kernel virtual machine which can safely and efficiently execute C programs from user space
- eBPF programs can be attached to any kernel instructions



Source: https://ebpf.io/what-is-ebpf/

Design Overview

synthesize an eBPF program to
instrument the kernel



isolate objects
with guard pages
and random offset

replace kmalloc and
kfree with our strategy

manage the metadata of dedicated regions
and isolated objects

eBPF Synthesis in Detail

function+offset: alloc site/free site (where to instrument the eBPF programs)

```
SEC("kprobe/__alloc_file+0x101")
int probe_alloc_file(struct pt_regs *ctx, int kpi_type){
    u64 ip = 0;
```

```
SEC("kprobe/file_free_rcu+0x50")
int probe_free_file(struct pt_regs* ctx, int kpi_type){
    u64 ip = 0;
```

the type of alloc/free function in kernel: different kpis have different prototypes

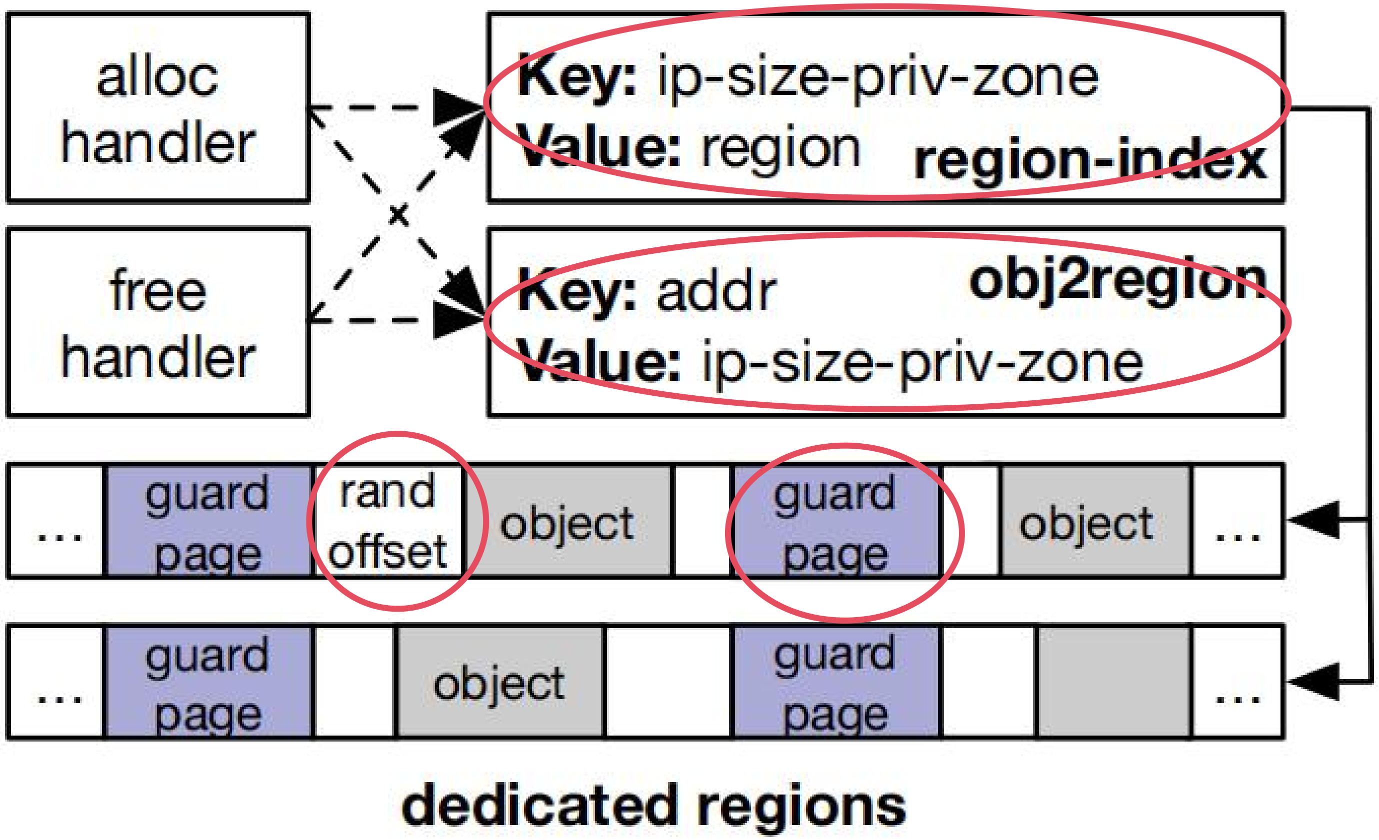
object size is the first parameter

```
void *kmalloc(size_t size, gfp_t gfp);
*kmem_cache_alloc(struct kmem_cache *cachep, int flags)
```

object size is a field of kmem_cache

we apply different methods to read object size considering different kpis

Run-time Separation in Detail

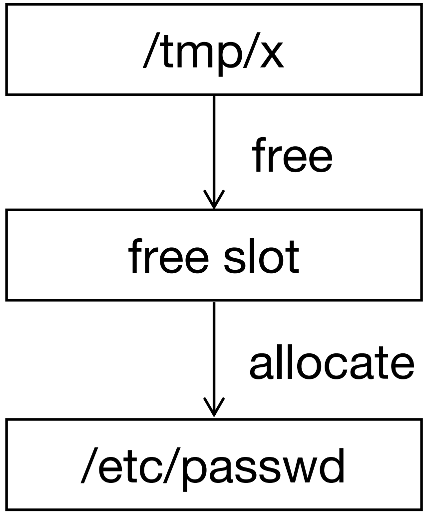


look up the dedicated region to allocate objects with ip-size-priv-zone as keys

look up metadata according to object address

set guard pages to prevent spatial corruption intro dedicated region
set random offset to prevent temporal corruption intro dedicated region

Example: CVE-2021-4154 (DirtyCred)



struct file representing a writable file

a dangling pointer generated

struct file representing a readable and privileged file

temporal overlapping between objects with different privileges

the allocation site of file object

```
dumping location of allocating file
__alloc_file fs/file_table.c:101  <-- real allocation site!
Possible Caller for __alloc_file
alloc_empty_file
alloc_empty_file_noaccount
file_free_rcu fs/file_table.c:50 <-- real release site!
Possible Caller for file_free_rcu
__alloc_file
```

the free site of file object

we use alloc/free site to generate the eBPF program to protect struct file

privilege

key: 0xffffffff80adbcd2-256-0-0

0xffffffff80adbcd2-256-1-0

Demo

- an Intel CPU with VT-X virtualization feature
- 64GB memory
- 300GB disk space
- Ubuntu 22.04.4 LTS (Jammy Jellyfish)

Effectiveness Evaluation

Exploits	C1	C2	C3	SeaK
2021-4154 (exp1)	○	○/◑	●	●
2021-22600 (exp3)	○	○/◑	●	●
2022-0185 (exp4)	○	○/◑	●	●
2022-27666 (exp6)	○	○/◑	●	●
2022-29582 (exp9)	○	○/◑	●	●
2022-1786 (exp13)	○	○/◑	●	●
2022-20409 (exp15)	○	○/◑	○	●

Separating vulnerable objects

SYZ Title	C1	C2	C3	SeaK
GPF-delayed_uprobe_remove	○	○/◑	●	●
WARNING-call_rcu	○	○/◑	●	●
WARNING-ODEBUG bug-tcf_queue_work	○	○/◑	●	●
KASAN-uaf-read-route4_get	○	○/◑	●	●
UBSAN-shift-oob-dummy_hub_control	○	○/◑	●	●
KASAN-uaf-read-hci_send_acl	○	○/◑	●	●
BUG-corrupted list-kobject_add_internal	○	○/◑	●	●
KMSAN-uninit-value-geneve_xmit	○	○/◑	●	●
KASAN-slab-oob-write-decode_data	○	○/◑	●	●

Separating victim objects

Peformance Overhead

LMbench (ms)	Vanilla	Cold	Hot	Durable			File
Simple syscall	0.1942	-1.68%	-0.67%	0.06%	0.08%	-0.29%	-0.94%
Simple read	0.2946	0.20%	-0.58%	0.49%	-0.48%	0.03%	-0.45%
Simple write	0.2502	-2.67%	-2.42%	0.51%	0.15%	0.56%	-0.18%
Select on 100 fd's	1.0718	0.26%	0.20%	-0.16%	-0.49%	-0.10%	-0.01%
Signal handler install	0.2538	-1.28%	-1.32%	0.17%	-0.33%	0.11%	0.02%
Signal handler overhead	0.8815	-0.90%	-1.53%	0.12%	1.54%	0.35%	-0.33%
fork+exit	99.6357	0.83%	2.49%	-0.49%	-2.82%	-3.44%	-2.43%
fork+execve	283.2725	1.51%	0.23%	2.32%	1.82%	-1.76%	3.34%
fork+/bin/sh -c	678.1250	2.93%	2.70%	2.35%	0.23%	-1.16%	2.28%
UDP latency	5.8852	1.25%	-1.10%	0.07%	-0.73%	-1.37%	-0.32%
TCP/IP connection	10.1259	0.13%	0.78%	0.51%	-0.01%	2.04%	1.62%
AF_UNIX bandwidth	9460.5067	0.67%	-0.56%	0.71%	0.92%	-1.85%	-1.26%
Pipe bandwidth	4569.4767	0.87%	-1.37%	-1.03%	1.94%	0.56%	-3.02%
Phoronix	**Vanilla**	**Cold**	**Hot**	**Durable**			**File**
Sockperf (Msgs/sec)	739608	-0.04%	-1.73%	-1.30%	0.75%	0.63%	0.93%
OSBench (Ns/Event)	78.28	-0.92%	-0.30%	-0.23%	-1.18%	-0.15%	-2.23%
7-Zip Compress (MIPS)	29521	-1.31%	-0.95%	1.07%	0.60%	1.62%	0.97%
FFmpeg Live (FPS)	178.08	0.45%	-1.29%	1.63%	1.57%	0.86%	0.68%
OpenSSL SHA256 (B/s)	1225189783	0.28%	-0.31%	-0.05%	0.02%	0.23%	-0.08%
Redis SET (Reqs/sec)	1932771	1.49%	-1.21%	-3.36%	-0.28%	0.30%	1.03%
SQLite Speedtest (sec)	62.63	0.57%	1.64%	-1.41%	-0.88%	1.44%	-0.41%
Apache 100 (Reqs/sec)	48216	-0.63%	-0.95%	-0.40%	0.49%	0.68%	0.18%

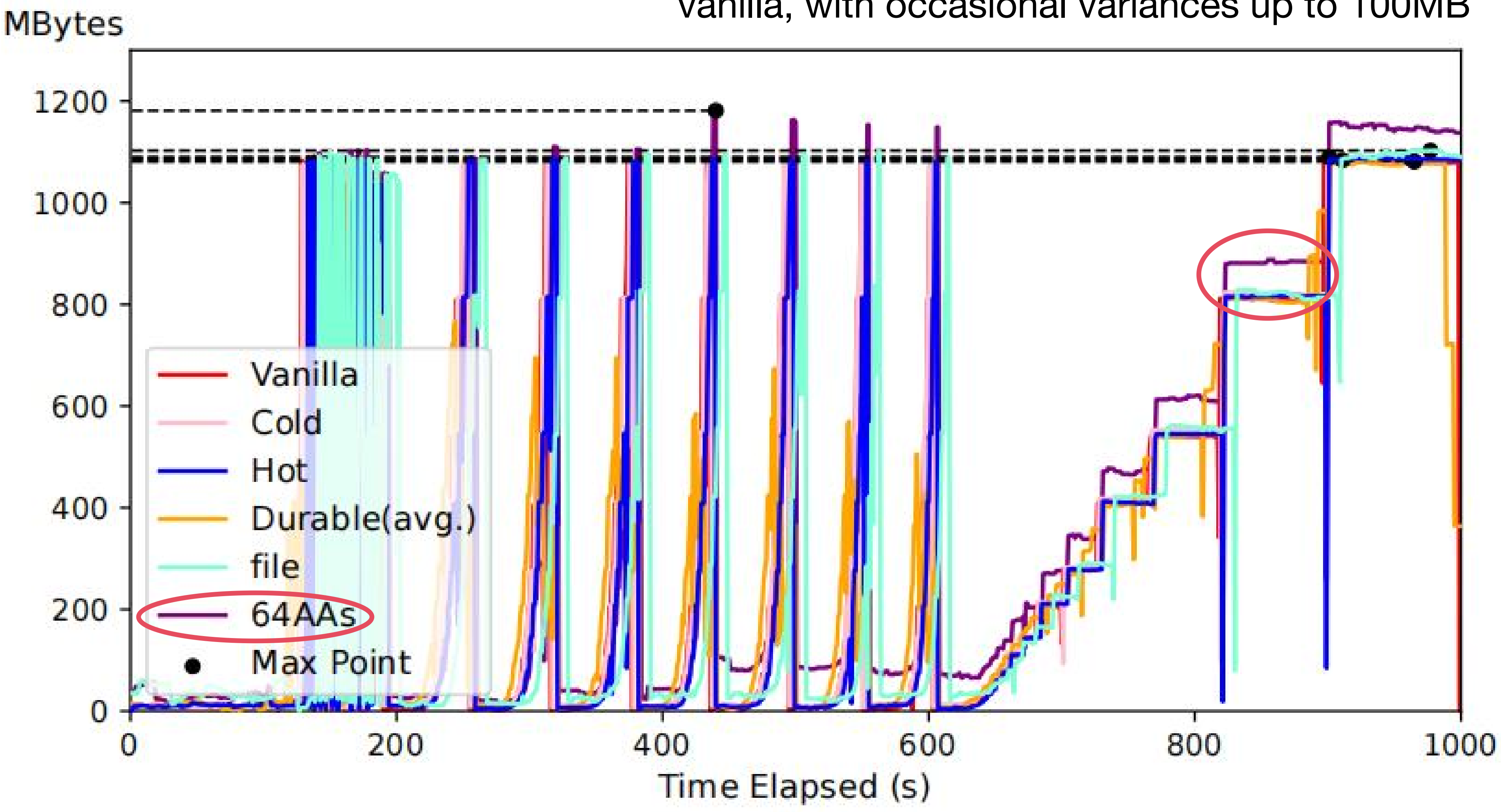
Cold: scarcely allocated objects
Hot: frequently allocated objects
Durable: objects with long lifespan
File: struct file

Even for the hot type, the highest overhead is only 2.49%

Negative number is caused by the influctation of LMbench

Memory Usage

The memory overhead of 64 AAs is on par with vanilla, with occasional variances up to 100MB



Scalability

LMBench	2 cases	4 cases	8 cases	16 cases	32 cases	64 cases
Simple syscall	0.70%	-0.01%	-1.52%	-1.20%	0.28%	1.43%
Simple read	0.06%	0.16%	-0.35%	0.16%	0.78%	0.05%
Simple write	0.55%	-2.28%	-2.58%	-2.28%	-0.21%	2.44%
Select on 100 fd's	-0.11%	-0.04%	0.11%	0.00%	-0.36%	0.01%
Signal handler install	-0.77%	-1.21%	-1.55%	-1.21%	-1.01%	-0.39%
Signal handler overhead	0.26%	-0.34%	-1.14%	-0.58%	1.55%	3.29%
fork+exit	-2.68%	3.26%	0.06%	3.26%	-2.04%	-3.30%
...	...	...	...	...	...	...
Pipe bandwidth	-1.45%	1.00%	-0.16%	1.89%	0.13%	0.04%
Avg.	-0.32%	0.05%	-0.55%	0.01%	0.20%	0.04%

Phoronix	2 cases	4 cases	8 cases	16 cases	32 cases	64 cases
Sockperf (Msgs/sec)	0.48%	-1.33%	-1.65%	-1.30%	4.20%	3.75%
OSBench (Ns/Event)	-0.24%	-0.16%	-0.19%	-0.23%	1.45%	0.45%
7-Zip Compress (MIPS)	-1.88%	-1.22%	-0.50%	1.07%	-0.29%	0.41%
FFmpeg Live (FPS)	0.48%	-0.83%	-0.34%	1.63%	1.97%	0.87%
OpenSSL SHA256 (B/s)	-0.10%	-0.16%	-0.09%	-0.05%	-0.07%	0.04%
Redis SET (Reqs/sec)	0.94%	-3.30%	-3.06%	-3.36%	-1.06%	-2.99%
SQLite Speedtest (sec)	0.37%	-0.31%	0.57%	1.41%	0.00%	0.15%
Apache 100 (Reqs/sec)	-0.30%	-0.52%	-0.71%	-0.40%	-0.55%	-0.85%
Avg.	-0.28%	-0.74%	-0.33%	-0.31%	0.71%	0.22%

Even for 64 cases, the average overhead is 0.04%

Contribution

- SeaK is a secure kernel allocator, protecting exploit-critical objects
  - Insights of inherent obstacles of designing a secure allocator
  - A new and practical strategy to secure kernel heap
  - Open-source design and implementation
  - Negligible overhead and high scalability

Github repo: https://github.com/a8stract-lab/SeaK

Email:yicheng.guang@colorado.edu