CS 315 Kernel Security

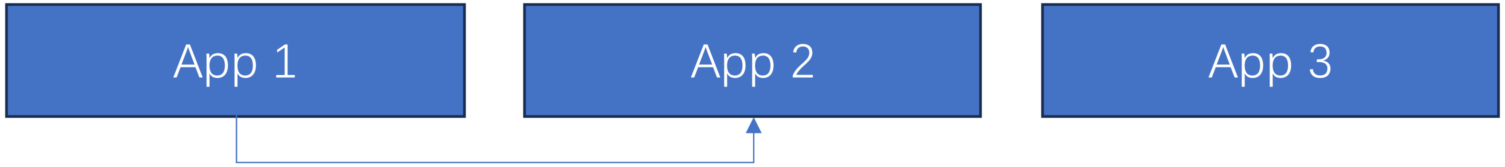
Fengwei Zhang and Hongyi Lu

Table of Contents

- **Introduction to Kernel**
- Kernel Security
- BPF --- A Hot Topic in Kernel

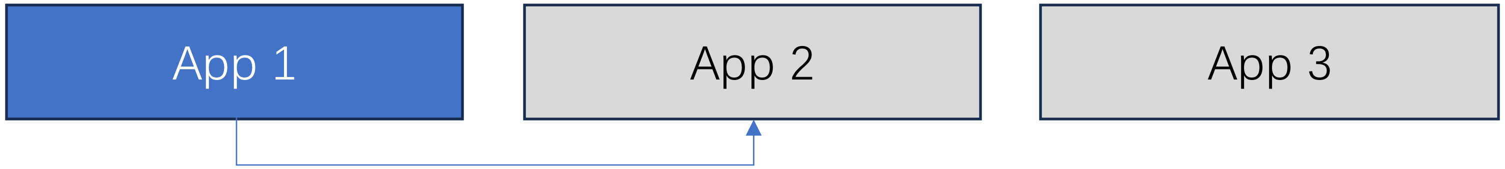
Kernel & Address Space.

- In the early days, there is only ONE address space.

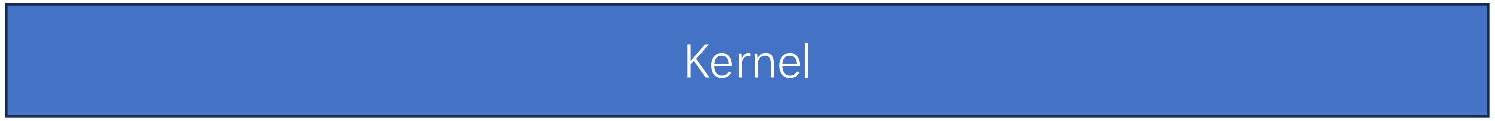


App 1: Let's see if App2 has something **juicy**

- Obviously, it's not good, so there is kernel and kernel space.

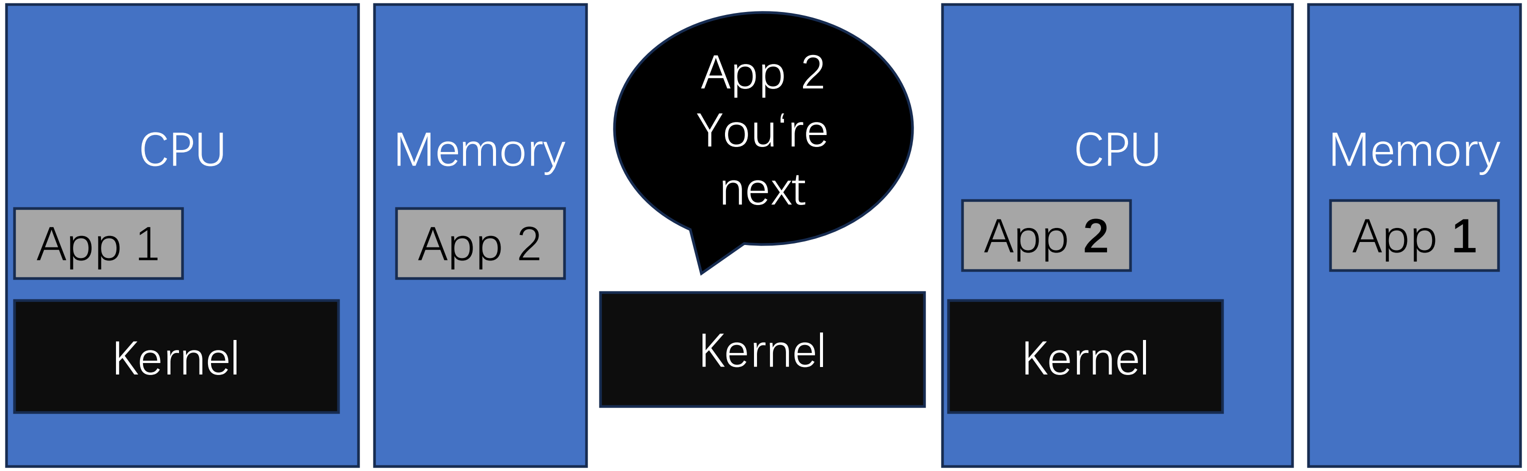


Kernel: Sorry, I have **removed** the App 2 from your address space.



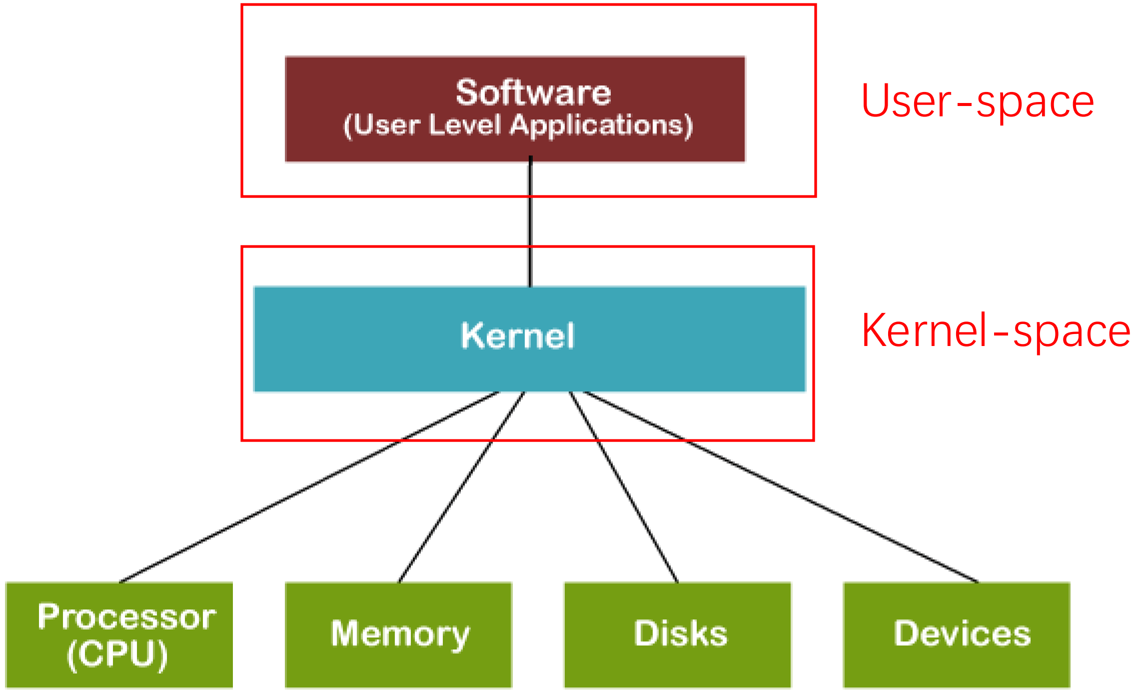
So, what else?

- Kernel schedules things and decides who uses the CPU.



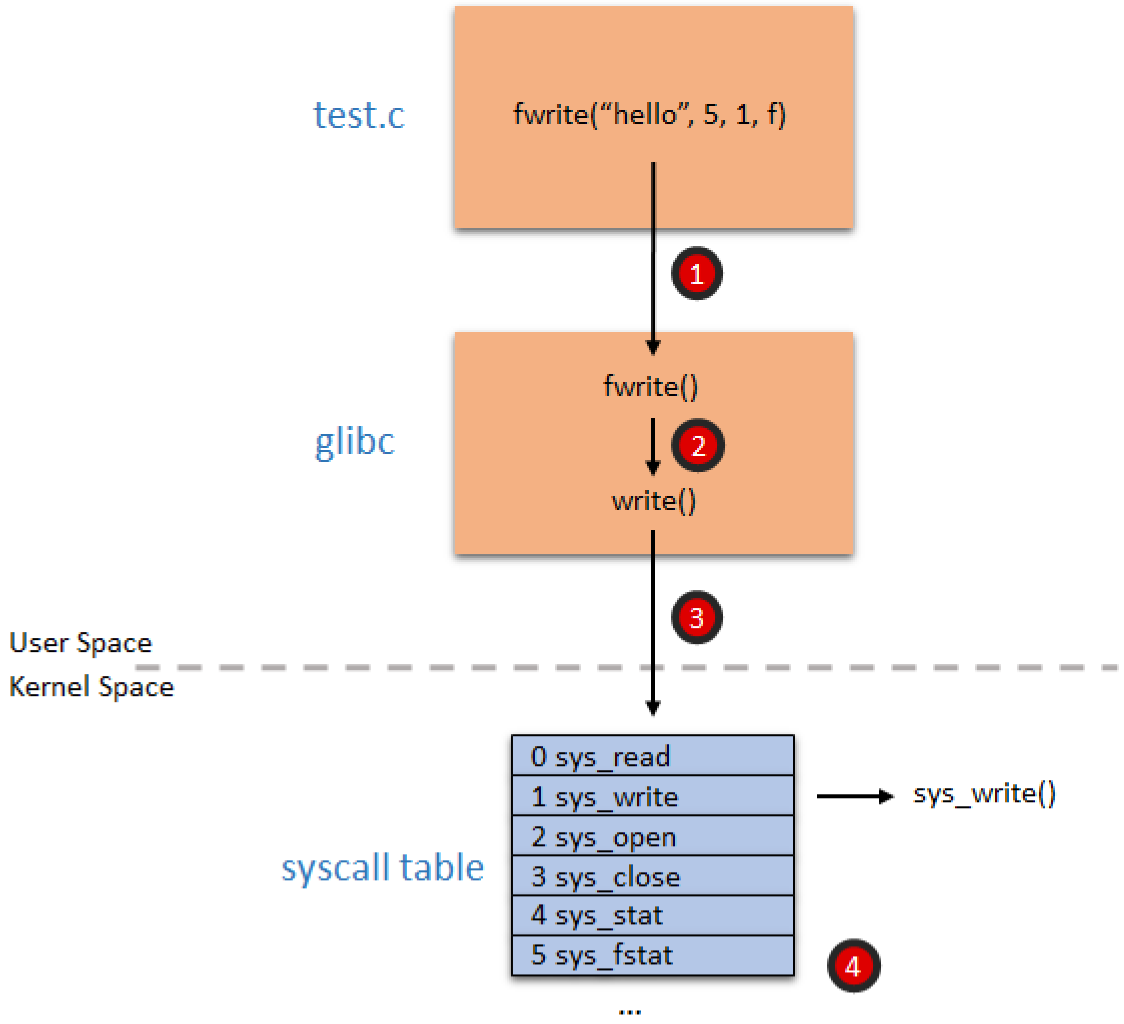
What's more?

- Peripheral management:
  - Network, Hard Disk, USB...
- Memory management:
  - malloc / free / ...
- Files, security, ...



System calls --- Bridging kernel and user

- Kernel defines a set of system calls.
- About ~300 of them.
- E.g., if you write `printf()` it is a combination of a series of systems calls like `brk/write/...`
- Use `strace` to explore more!



Syscall Security

- As the bridge between kernel and user, syscalls are often abused by attackers.

Though many defenses deployed, and kernel surely is not that naïve, but some -where there must be a vulnerability.

```
int SYS_handler(int buf[])
{
    int bof[10];
    strcpy(bof, buf);
}
```

Kernel Modules

The kernel is good, but there are **millions** of different devices.

How to support them?

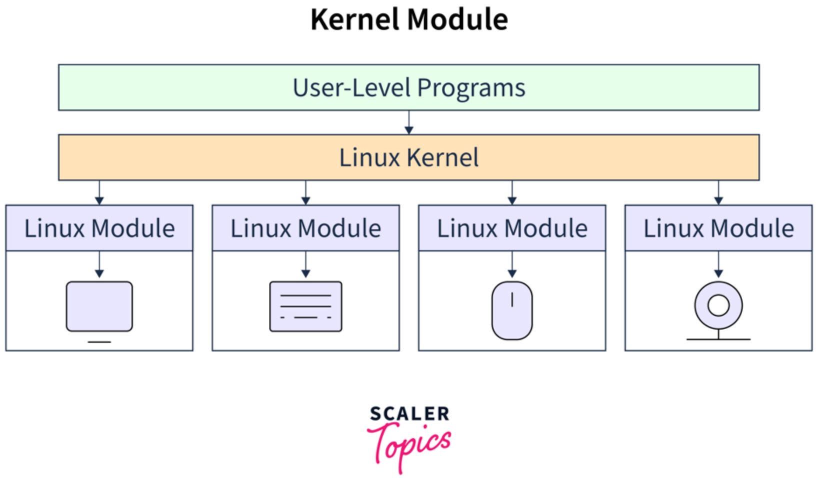
A. Including **all** drivers together -> The kernel will be as large as **Black Monkey Wukong.**

B. Design kernel for **every** single device -> The kernel will broken into pieces and nothing is compatible

Kernel Modules

What about maintaining a core part and everything else is just **plugins**?

- Developer can arbitrarily insert their code using kernel module and support their devices.



Kernel --- Try it yourself.

- https://sysprog21.github.io/lkmpg/
- Kernel Module Development Guide

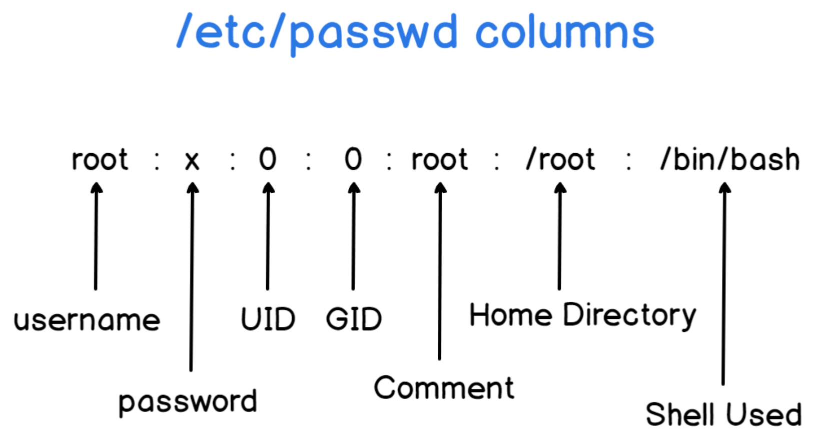
- https://github.com/rcore-os/rCore
- Writing your own kernel --- w. Rust

Table of Contents

- Introduction to Kernel
- **Kernel Security**
- BPF --- A Hot Topic in Kernel

Security --- User & Permission

- We want to separate different "users" on the same machine.
- Identify each user using UID/GID.
- Things that check your ID: File/Syscall/...



- You'll have to be 18 to have a driver's license
- You'll have to be root to use rm -rf

Security --- Alter permission?

- Stack overflow attack:
  - Kernel is calling func_X, but the user diverts RA via stack overflow to func_A.
  - KASLR should defend this.

kernel address space layout randomization

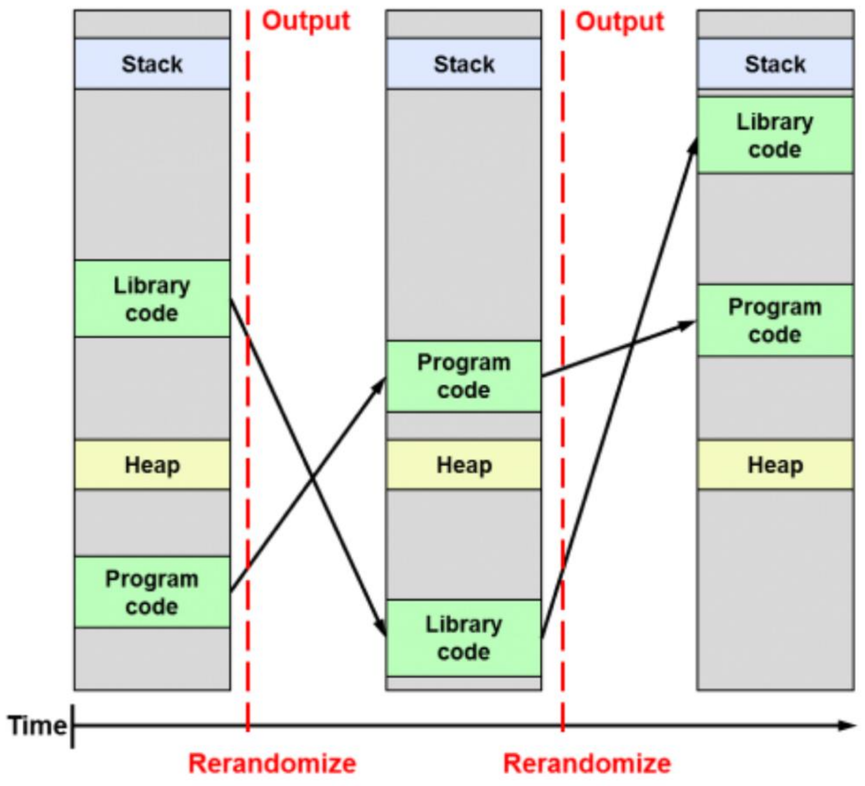
- Ret2user:
  - User hijacks kernel's flow to user_func.
  - SMEP/SMAP

- And many more:

- Alter function pointer.
- Race condition.
- ...

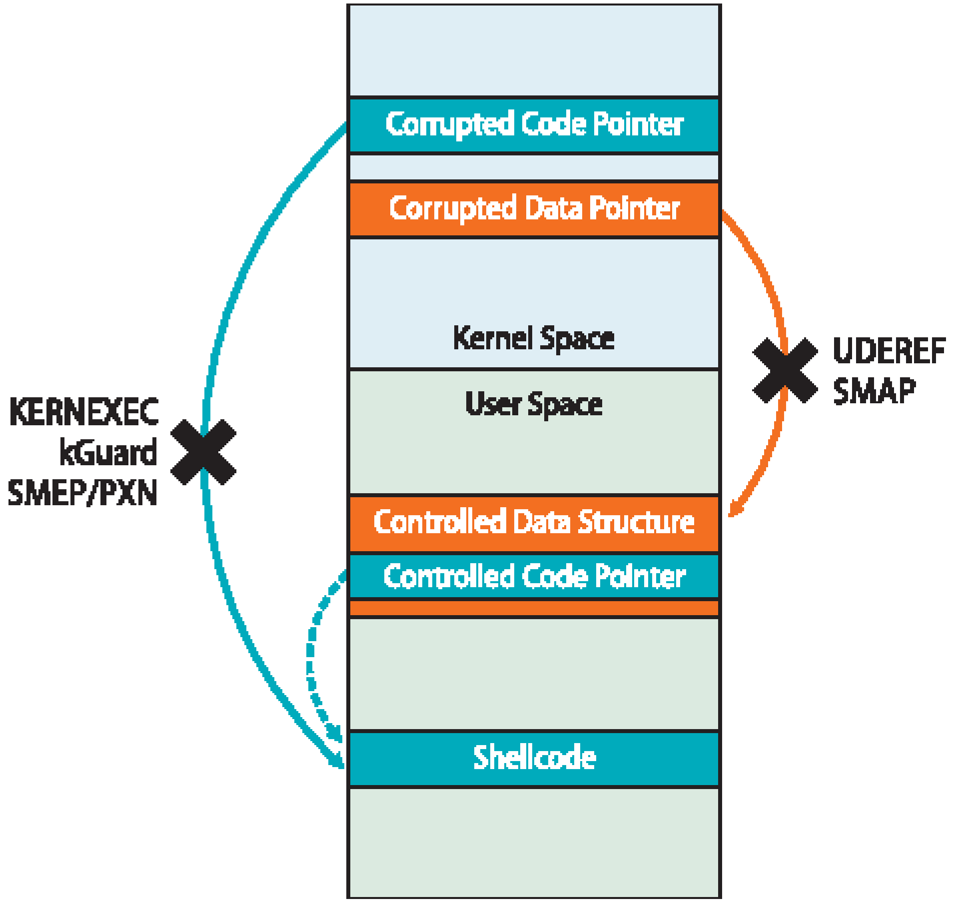
Kernel Security --- KASLR

- Kernel version of ASLR
- Randomize the location of codes.
- Make attacker's lives hard as they don't know what RA they should inject.



Kernel Security --- SMEP/SMAP

- Attacker forces the **kernel** to jump to **user memory**.
- **Because user memory is easier to manipulate.**
- SMEP / SMAP prevents access from kernel to userspace



Kernel Security --- KASAN / MTE / ...

- What if kernel memory corrupts?
- Use-after-free, double-free, ...
- KASAN uses 1 byte to track the **state** (free/allocated/...) of 8 bytes.


```
#define KASAN_FREE_PAGE          0xFF  /* page was freed */
#define KASAN_PAGE_REDZONE       0xFE  /* redzone for kmalloc_large allocations */
#define KASAN_KMALLOC_REDZONE    0xFC  /* redzone inside slub object */
#define KASAN_KMALLOC_FREE       0xFB  /* object was freed (kmem_cache_free/kfree) */
#define KASAN_GLOBAL_REDZONE     0xFA  /* redzone for global variable */
```

Kernel Security Materials

- https://docs.kernel.org/security/self-protection.html

Kernel Documentation --- Current Defenses

- https://github.com/search?q=CVE+PoC+Linux&type=repositories

Search PoC of Linux Vulnerabilities

- https://pwn.college/

Online Training Field

Table of Contents

- Introduction to Kernel
- Kernel Security
- **BPF --- A Hot Topic in Kernel**

Why BPF?

Imagine you need to monitor kernel's `write()` event and get notified whenever a user writes to a file.

A. Modify the kernel source?
   Works, but what if you need things like `read/close/open`, modify all of them?
B. Use `gdb` to track every program on the system.
   TOO SLOW…
C. Using a kernel **module**?
   Well, if you have a **null pointer**, this breaks your entire system.

What if you can "inject code" into the kernel

Berkeley Packet Filter

What is BPF?

- A bunch of kernel hook points.
- User can "attach" their codes to these hook points.
- Kernel executes user-supplied code whenever an "event" happens.
- Verifier makes sure they are **safe** to run in the kernel.

Tell the kernel you want `prog.bpf.c` to run whenever a write event occurs!

An Example BPF Program.

```
#include <linux/bpf.h>
#include <bpf/bpf_helpers.h>
char LICENSE[] SEC("license") = "Dual BSD/GPL";

int my_pid = 0;

SEC("tp/syscalls/sys_enter_write")
int handle_tp(void *ctx)
{
    int pid = bpf_get_current_pid_tgid() >> 32;
    if (pid != my_pid)
        return 0;
    bpf_printk("BPF triggered from PID %d.\n", pid);
    return 0;
}
```

Not-so-useful
but needed code

When this BPF prog.
should be “triggered”

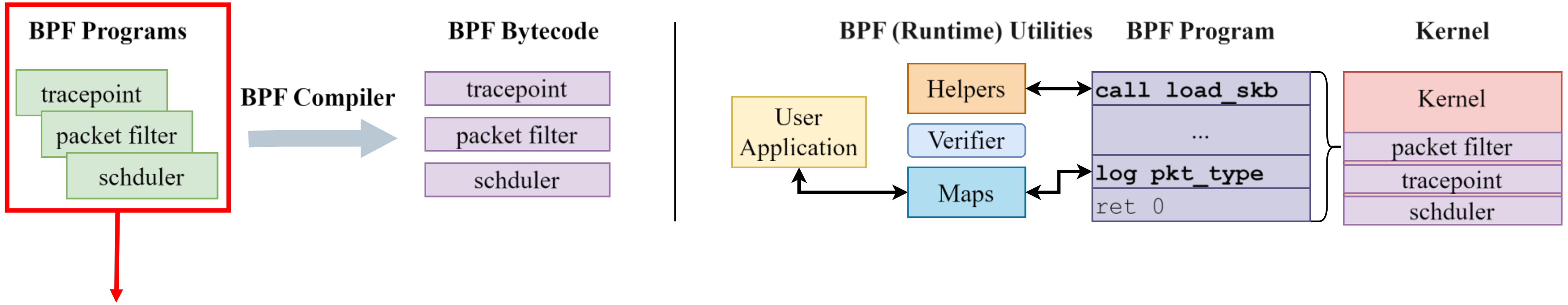
Helper function getting
pid of the triggering process

Write log to
/sys/kernel/debug/tracing/trace
(NOT your favorite stdout, sry)

Many other functionalities

- Network packet filtering
- Performance profiling
- System call filtering

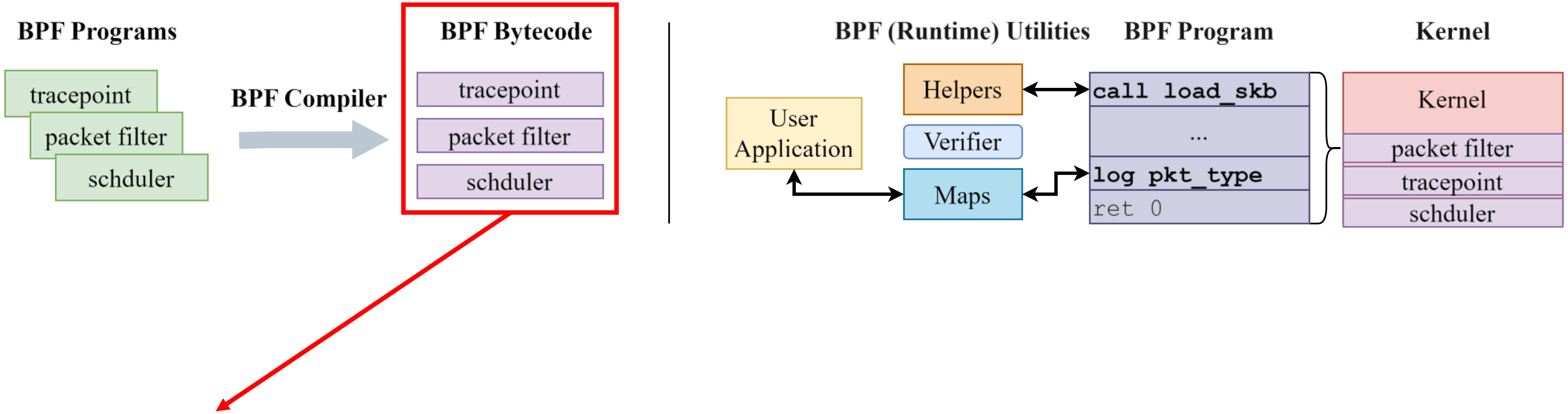
eBPF Internals



eBPF Programs： normally written in C, use "vmlinux.h" to invoke kernel-provided API

- `user_bpf.c`: user-space program, responsible for loading the BPF program into the kernel.
- `kern_bpf.c`: kernel-space BPF program

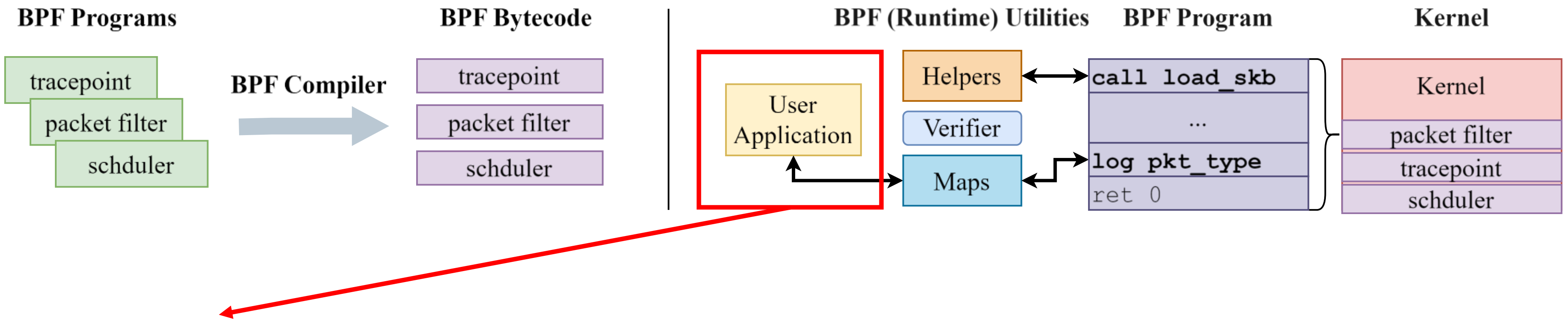
eBPF Internals



eBPF Bytecode: a ISA designed specifically for eBPF.

- **Security**: easier to verify.
- **Compatiblity**: can be translated into arm/x86/risc-v.
- **Expressibility**: can satisfy the need of kernel extension.

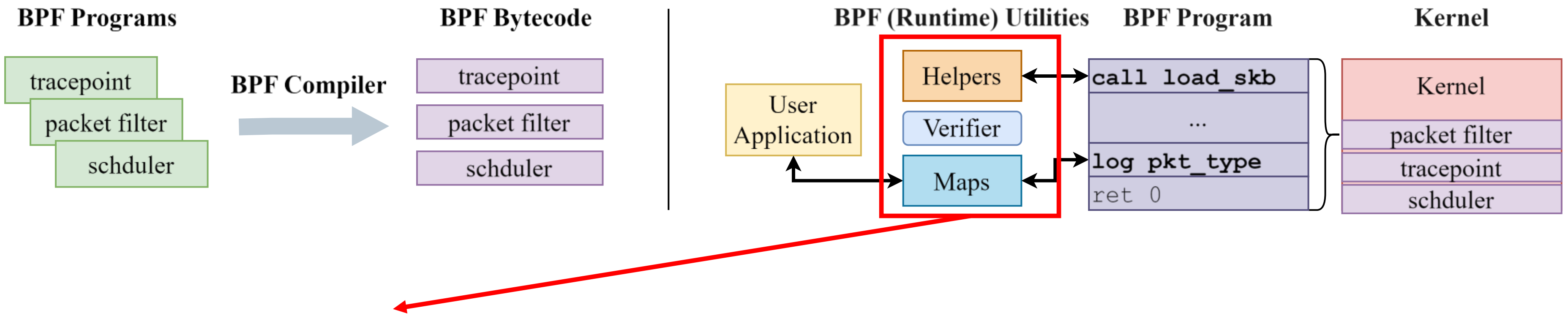
eBPF Internals



eBPF User-space program： responsible for calling “bpf(…)” system calls， notify the kernel to do the BPF’s loading/verifying/attaching.

For programs that monitor kernel status, user-space program also collects the data BPF sends from the kernel.

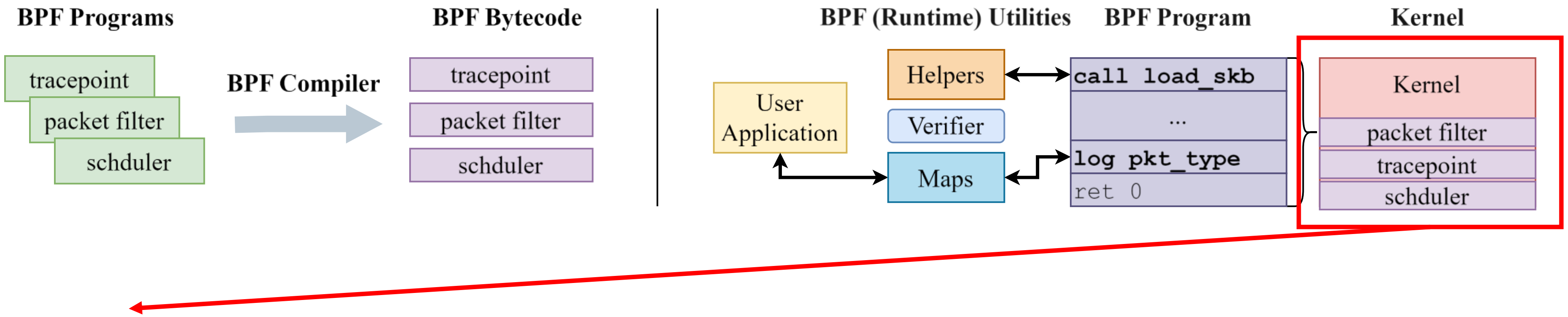
eBPF Internals



eBPF Kernel components：

- **Helper functions:** BPF kernel API, including 200 different functions for BPF to interact with kernel.
- **Verifier:** BPF verifier, preventing the BPF program from sabotage the kernel (e.g., deadloop, out-of-bound R/W).
- **Maps:** BPF data structures for bridging the kernel space and user-space

eBPF Internals



eBPF Kernel Hookpoints: Kernel has over 700 hookpoints for BPF programs. This allows user to run their own code whenever a kernel event occurs.

eBPF Examples --- Socket filter

```
#include <stddef.h>
#include <linux/bpf.h>
#include <bpf/bpf_helpers.h>
#include <bpf/bpf_endian.h>

char LICENSE[] SEC("license") = "Dual BSD/GPL";

SEC("socket")
int dropfilter(struct __sk_buff *skb)
{
    return 0;
}
```

Hook to a socket

Just drop it

```c
struct {
    __uint(type, BPF_MAP_TYPE_ARRAY);
    __type(key, uint32_t);
    __type(value, long);
    __uint(max_entries, 256);
} my_map SEC(".maps");
```

A BPF map for sending data back to the userspace.
Note: **user-space program also need code for receiving the data.**

```c
SEC("socket")
```

Same hookpoint

```c
int sockex3(struct __sk_buff *skb)
{
    int proto = load_byte(skb, ETH_HLEN + offsetof(struct iphdr, protocol));
    int size = ETH_HLEN + sizeof(struct iphdr);
    switch (proto) {
        case IPPROTO_TCP: size += sizeof(struct tcphdr); break;
        case IPPROTO_UDP: size += sizeof(struct udphdr); break;
        default: size = 0; break;
    }
    return size;
}
```

The return value means how many byte we want to **keep.**
So 0 means drop

```c
struct {
    __uint(type, BPF_MAP_TYPE_RINGBUF);
    __uint(max_entries, 512 * 1024);
} pf_rb SEC(".maps");
```

There can be other types of map.

```c
SEC("tracepoint/exceptions/page_fault_user")
int pfu(struct trace_event_page_fault *ctx)
{
  struct pf_event *evt;
  evt = bpf_ringbuf_reserve(&pf_rb, sizeof(*evt), 0);
  if (!evt)
    return 0;
  evt->address = ctx->fault_address;
  evt->error_code = ctx->error_code;
  evt->ip = ctx->ip;
  bpf_snprintf(evt->type, 16, "pfu", NULL, 0);
  bpf_ringbuf_submit(evt, 0);
  return 0;
}
```

There can also be other types of helper functions, hookpoints, and many things.

Go read the document!

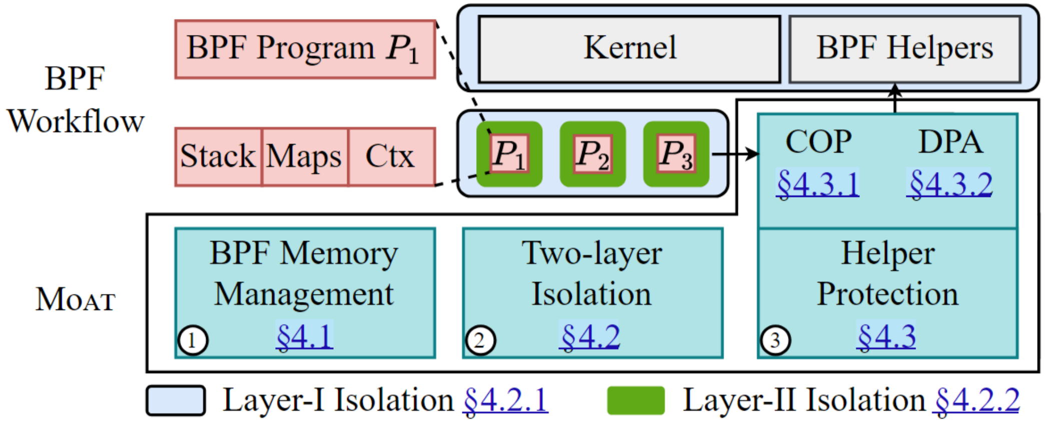
Secure? Except it is not.

Though BPF deploys a verifier to check if the program is **safe** for the kernel, the **over-bloated** verifier and the **challenging static analysis (soundness versus. completeness)** bring many security issues.

There are over **151** CVEs related to BPF; most of them are due to verifier bugs.

BPF Research --- its own security.

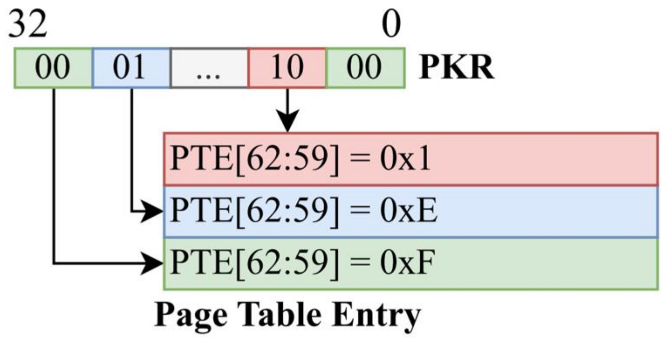
- Hardware-isolation for BPF programs.
- Use Intel MPK + PCID to protect kernel from malicious BPF programs.
- https://www.usenix.org/system/files/usenixsecurity24-lu-hongyi.pdf



Done by COMPASS, whose author is also making this PPT.

Hardware Isolation!

Wait..., what is Intel MPK?

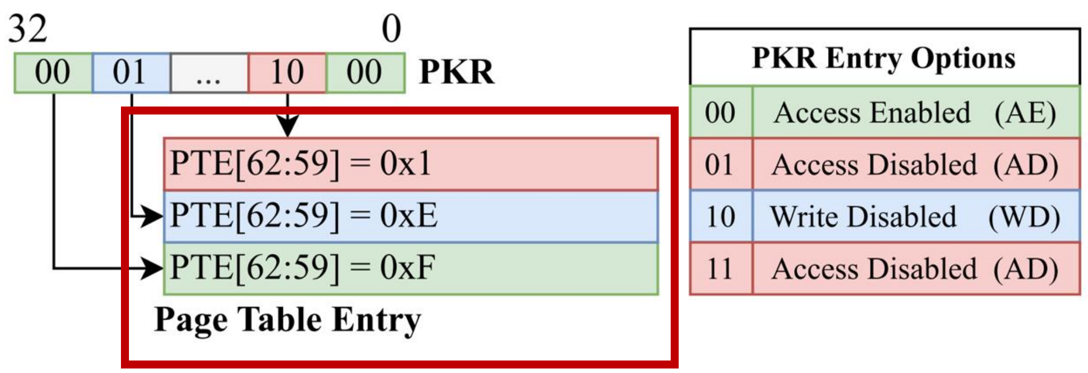


PKR Entry Options	
00	Access Enabled (AE)
01	Access Disabled (AD)
10	Write Disabled (WD)
11	Access Disabled (AD)

Hardware Isolation!

Wait…, what is Intel MPK?

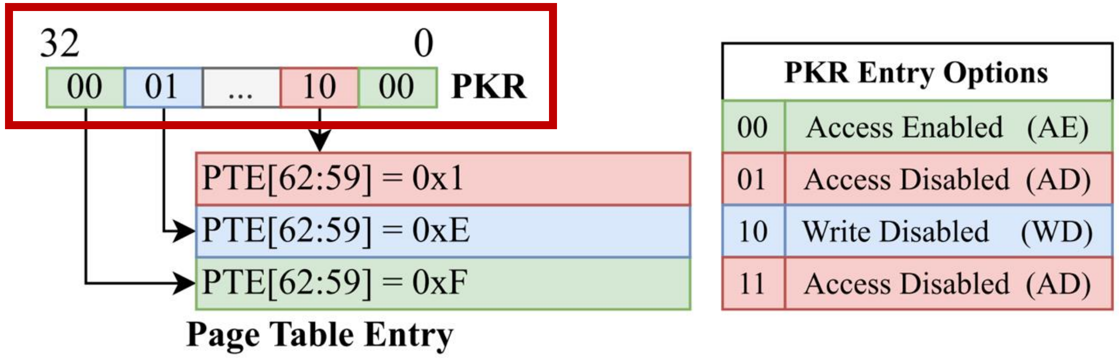
- Add a **4-bit tag** to PTEs (16 tags).



Hardware Isolation!

Wait..., what is Intel MPK?

- Add a 4-bit tag to PTEs (16 tags).
- **Toggle PTEs** with the same tag.



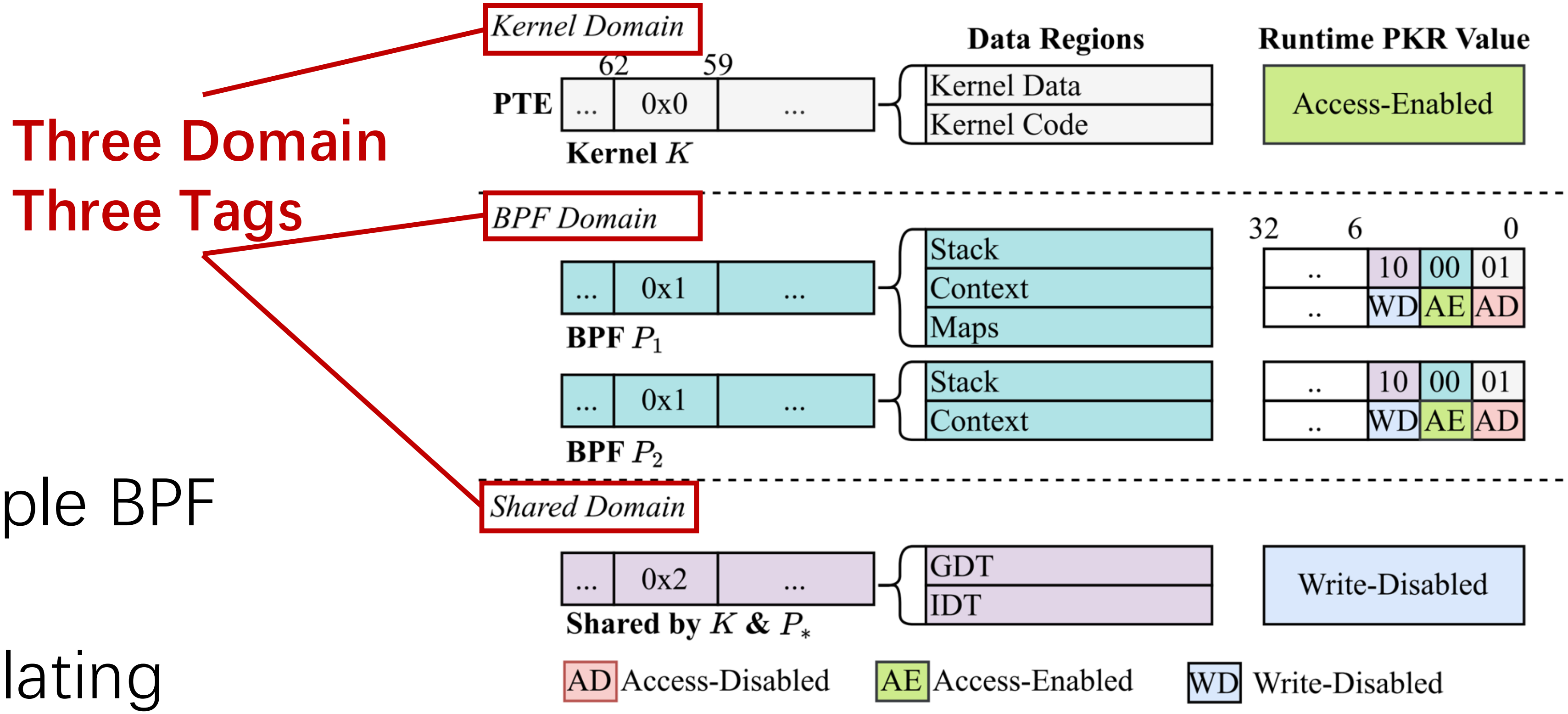
Limited MPK Tags

MPK is...

- Only 16 tags
- Lightweight

So... *bad* for multiple BPF programs.

But... *good* for isolating kernel/BPF.



Limited MPK Tags

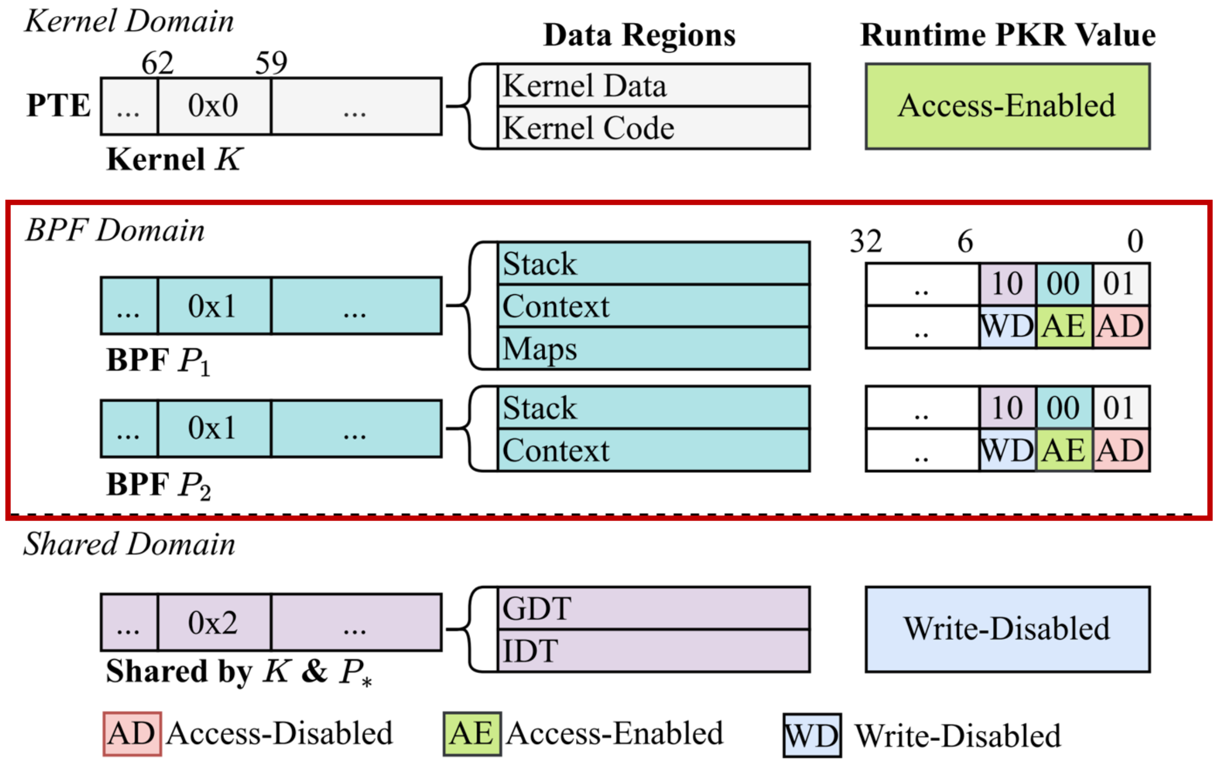
MPK is…

- Only 16 tags
- Lightweight

Constrain ALL BPF programs

So… *bad* for multiple BPF programs.

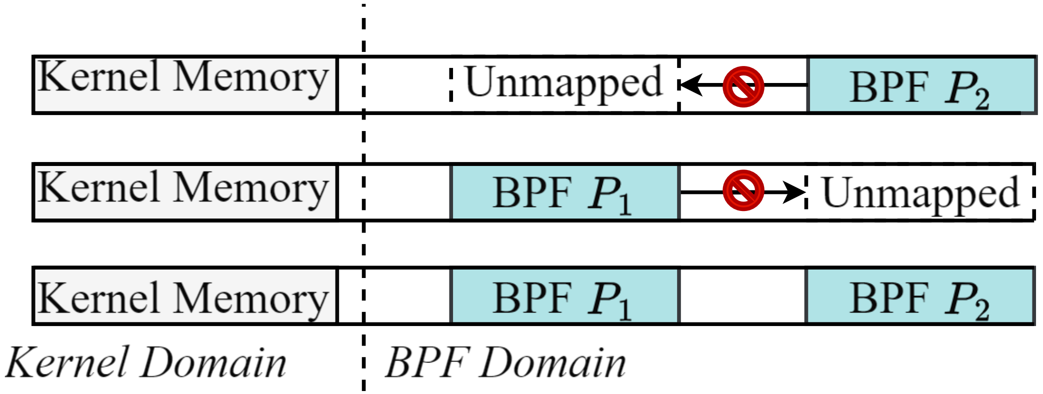
But… *good* for isolating kernel/BPF.



Intra-BPF exploitation

Problem:

Bad BPFs attack the good ones.



Solution: MOAT isolates them by address spaces.

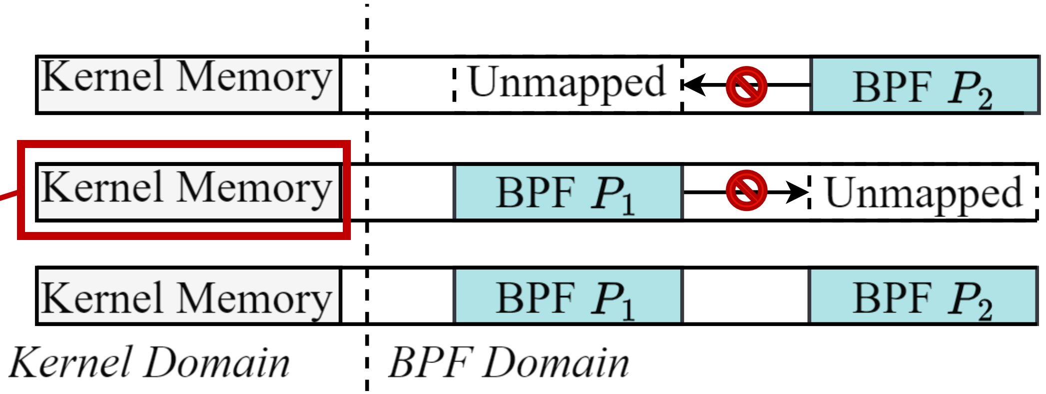
Issue: Slow TLB flushes

Intra-BPF exploitation

Problem:

Bad BPFs attack the good ones.

Solution: MOAT isolates them by address spaces.



TLB flush is slow?

- **Constant kernel** mapping
- We use PCID to minimize #flushes.

Intra-BPF exploitation

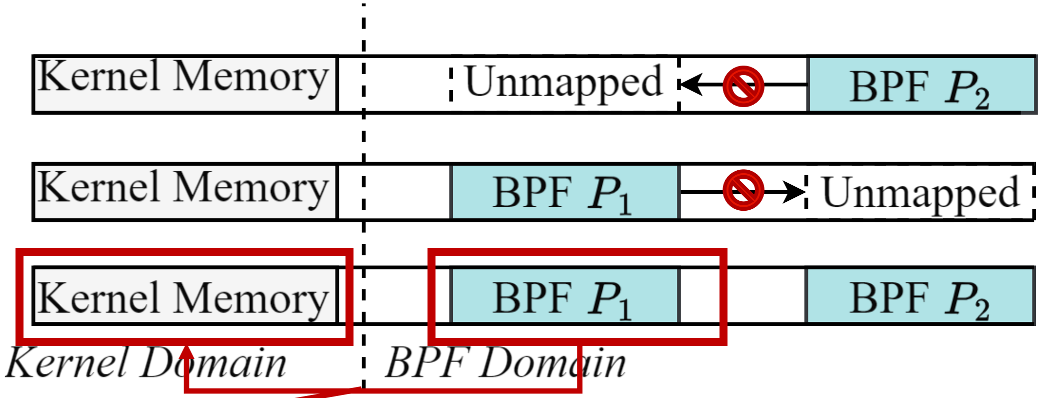
Problem:

Bad BPFs attack the good ones.

MOAT isolates them by address spaces.

TLB flush is slow?

- BPF has **small** memory footprints.
- We use **PCID** to minimize #flushes.



Avoid unnecessary flushes

Kernel API Security

BPF is isolated, but it might still access kernel via its API (BPF Helpers)

MOAT does…

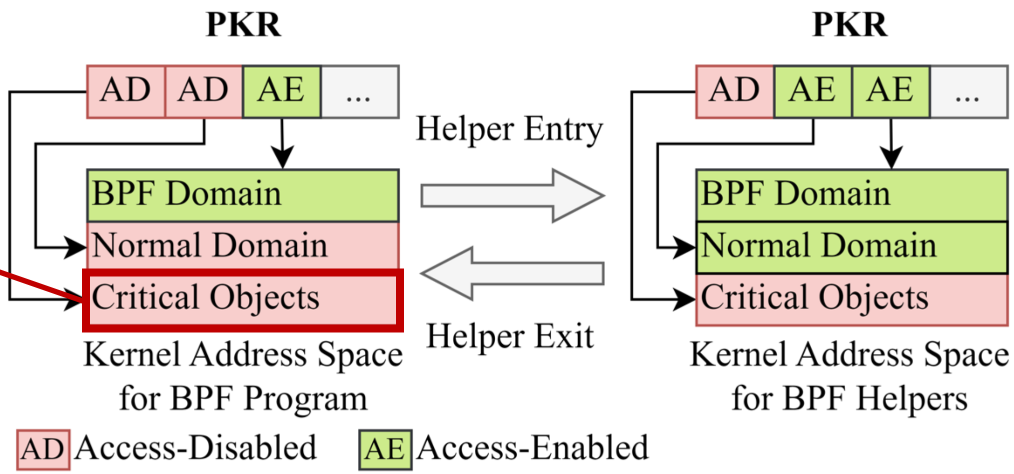
- Isolate **easy-to-exploit** structures from helpers.
- Check parameters against **verified bounds**.

Critical Object Protection

We studied kernel objects that were **previously exploited** via BPF.

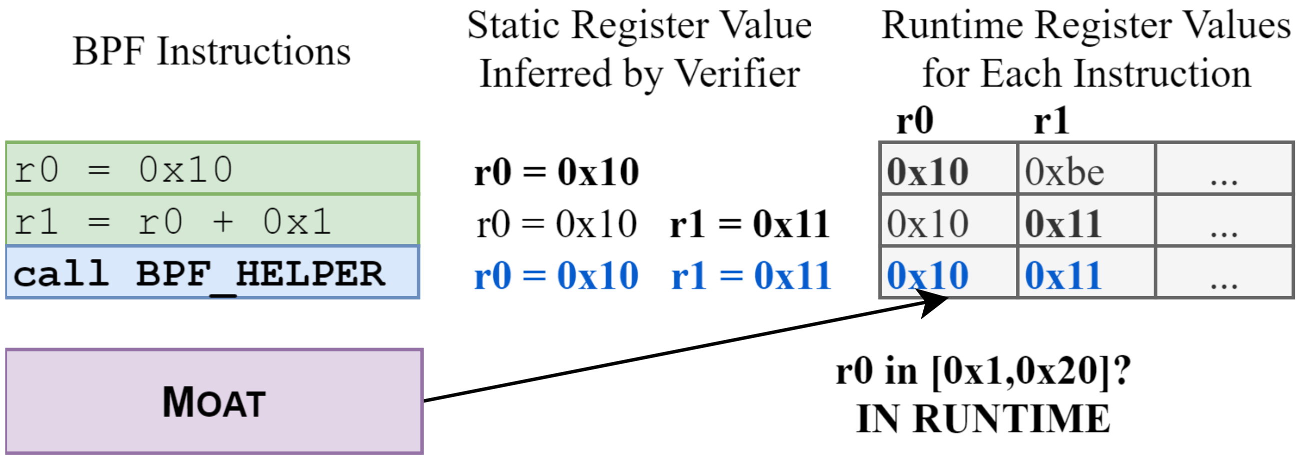
In sum, **44** of these are identified;

MOAT protects them with an extra MPK tag.



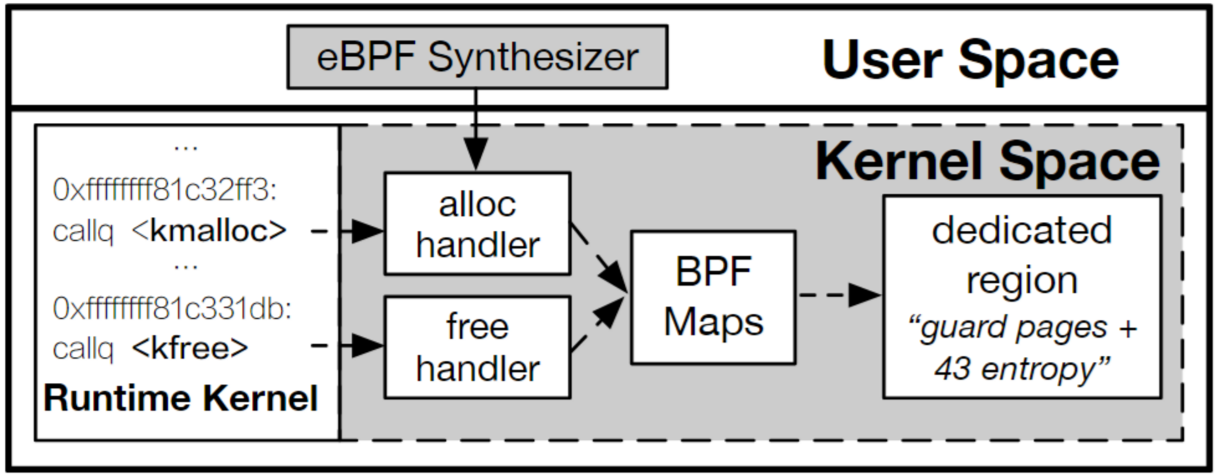
Dynamic Parameter Auditing

MOAT uses the verifier’s bounds to double-check the helper’s arguments **in runtime**.



BPF Research.

- Use BPF for secure allocator
- Hook `kmalloc` and `kfree`
- Isolate sensitive objects from common allocations.
- https://www.usenix.org/system/files/sec24fall-prepub-1504-wang-zicheng.pdf

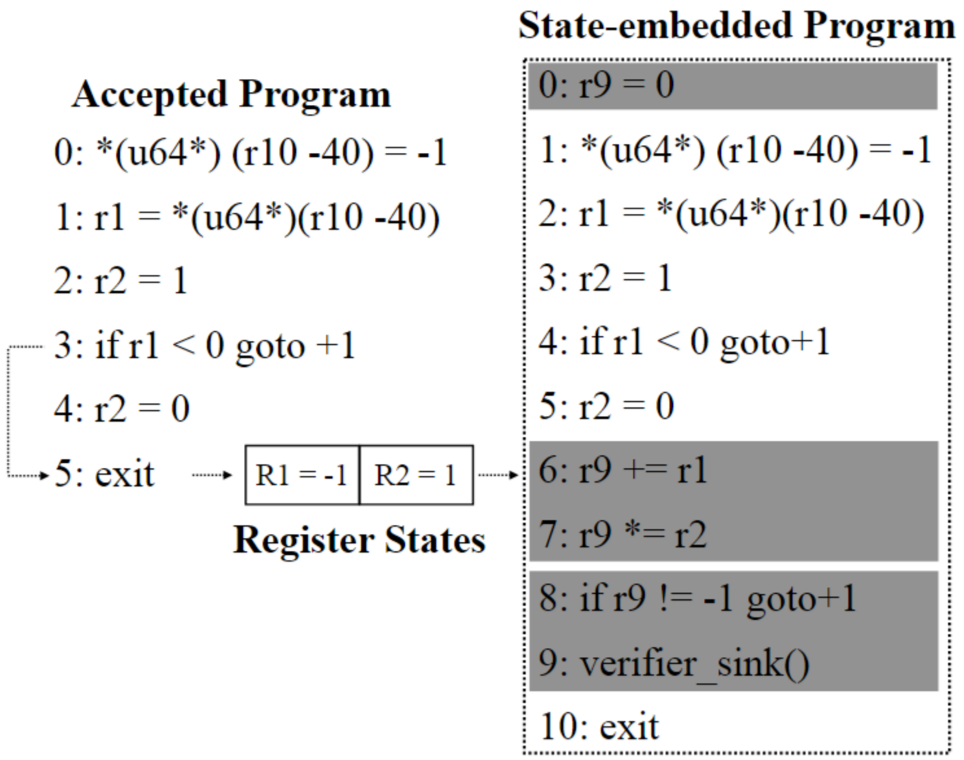


BPF Research.

- Testing BPF verifier.

- Mutate *valid* BPF programs into invalid ones.
- See if verifier can spot them out.

- https://www.usenix.org/system/files/osdi24-sun-hao.pdf



Also many others.

- HIVE; USENIX Security 2024
  - BPF Isolation on Arm
- BeeBox; USENIX Security 2024
  - BPF Isolation against *spectre*

- Three ways of viewing a thing (finding new ideas!):
  - What can we do **using** it? --- BPF-based secure allocator.
  - Can we **find something wrong** with it? --- Testing BPF components.
  - How can we **fix** it if it's wrong? --- BPF isolation.