MOAT: Towards Safe BPF Kernel Extention

Hongyi Lu[1,2], Shuai Wang[2], Yechang Wu[1], Wanning He[1], Fengwei Zhang[1,*]

[1]Southern University of Science and Technology

[2]Hong Kong University of Science and Technology

Background

What is (e)BPF?

Extended Berkeley Packet Filter:

- Kernel Virtual Machine
- Introduced in Linux 3.15 (2014)

- Extended from classic BPF (cBPF), which dates back to FreeBSD (1992)
- Packet Filter → Tracing/Network/Security…

Why eBPF?

- **Fast**: Run in JITed native code.
- **Portable**: Stable kernel API (named helpers).

- **Robust**: Does NOT crash your kernel; eBPF is statically checked by a *verifier*.

Sounds good, but?

BPF Security is a concern.

BPF verifier alone is NOT enough to ensure BPF's security.

And…

- Static analysis is **hard**.
- BPF is **rapidly** developed.
- Kernel is **critical**.

CVE ID
2016-2383, 2017-16995, 2017-16996, 2017-17852, 2017-17853, 2017-17854, 2017-17855, 2017-17856, 2017-17857, 2017-17862, 2017-17863, 2017-17864, 2018-18445, 2020-8835, 2020-27194, 2021-34866, 2021-3489, 2021-3490, 2021-20268, 2021-3444,2021-33200, 2021-45402, 2022-2785, 2022-23222, 2023-39191, 2023-2163

BPF CVEs

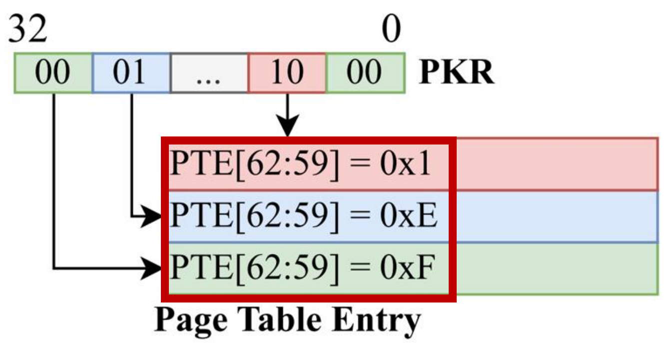
Hardware Isolation!

We therefore propose MOAT.
MOAT uses **hardware features** (e.g., MPK) to isolate BPF programs.
And… resolves a set of challenges, like **limited MPK and BPF API security**.

Hardware Isolation!

Wait..., what is Intel MPK?

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

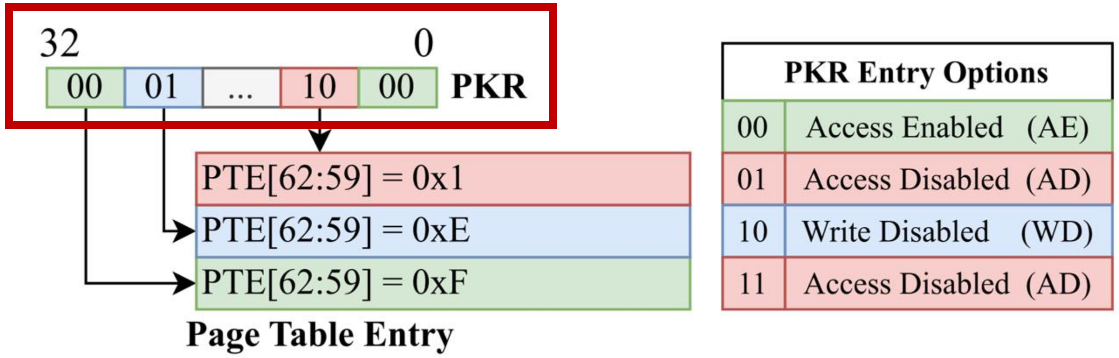


PKR Entry Options	
00	Access Enabled (AE)
01	Access Disabled (AD)
10	Write Disabled (WD)
11	Access Disabled (AD)

Hardware Isolation!

Wait..., what is Intel MPK?

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



Method

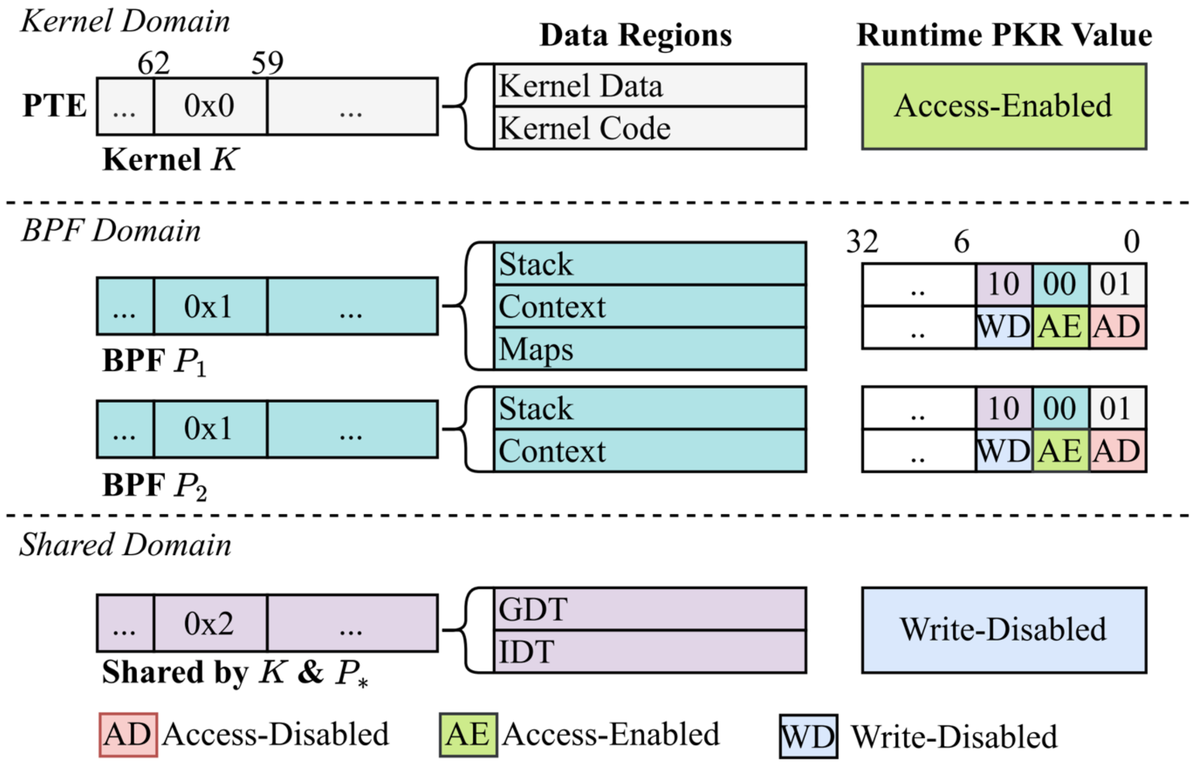
Limited MPK Tags

MPK is...

- Only 16 tags
- Lightweight

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

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



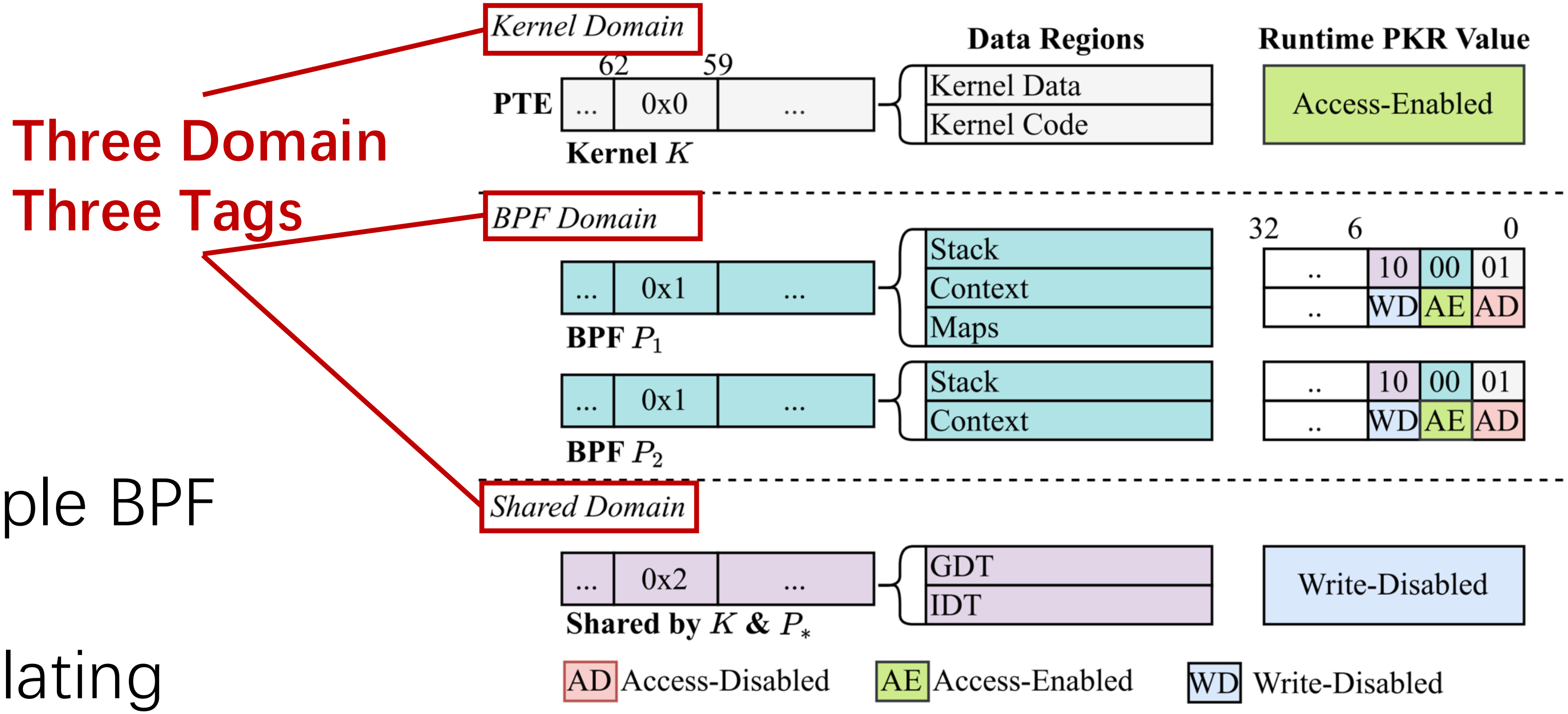
Limited MPK Tags

MPK is...

- Only 16 tags
- Lightweight

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

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



Limited MPK Tags

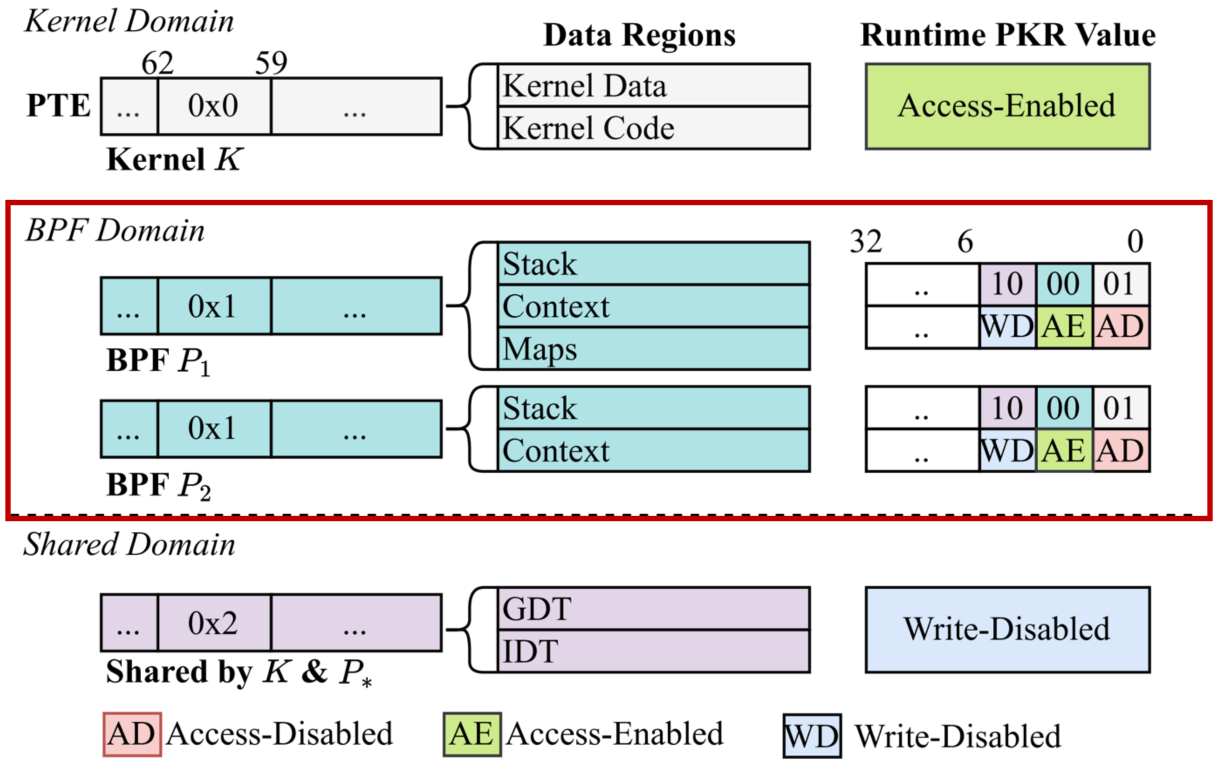
MPK is...

- Only 16 tags
- Lightweight

Constrain ALL BPF programs

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

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



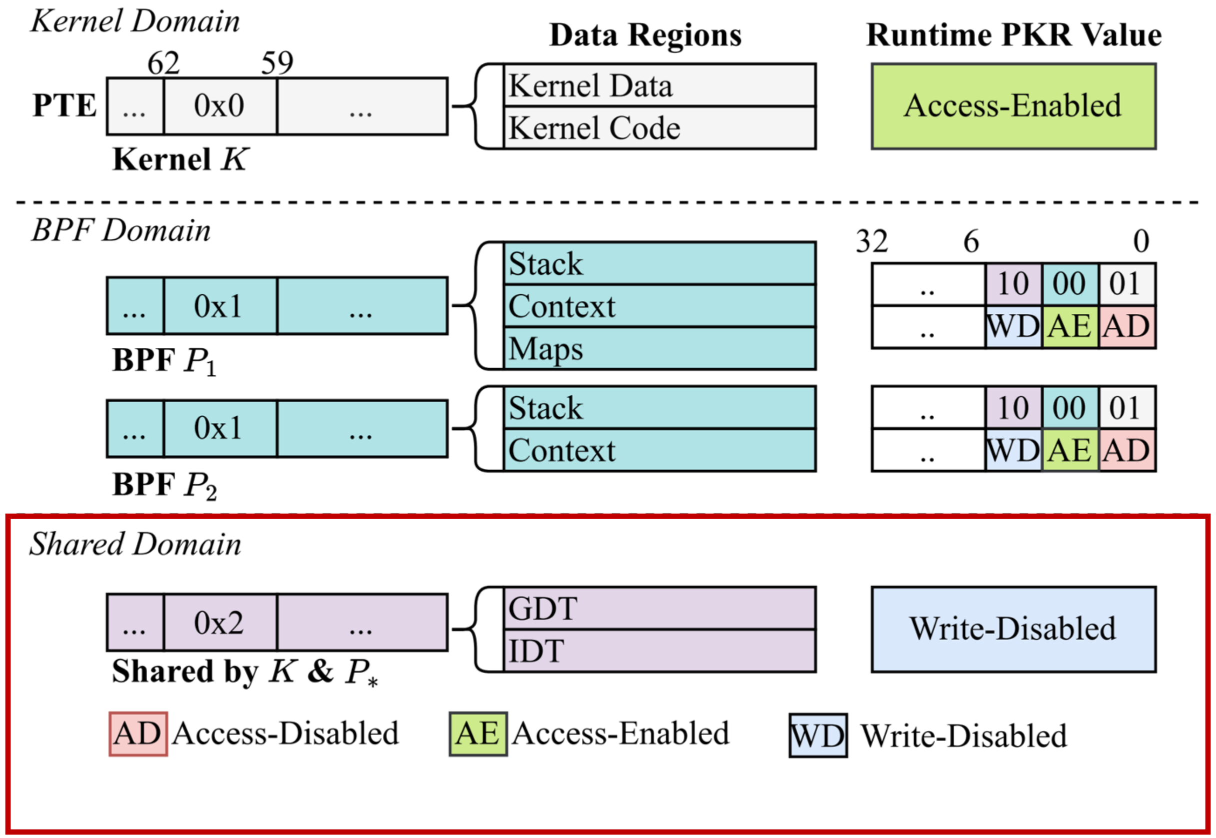
Limited MPK Tags

MPK is...

- Only 16 tags
- Lightweight

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

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



Things both BPF & Kernel need

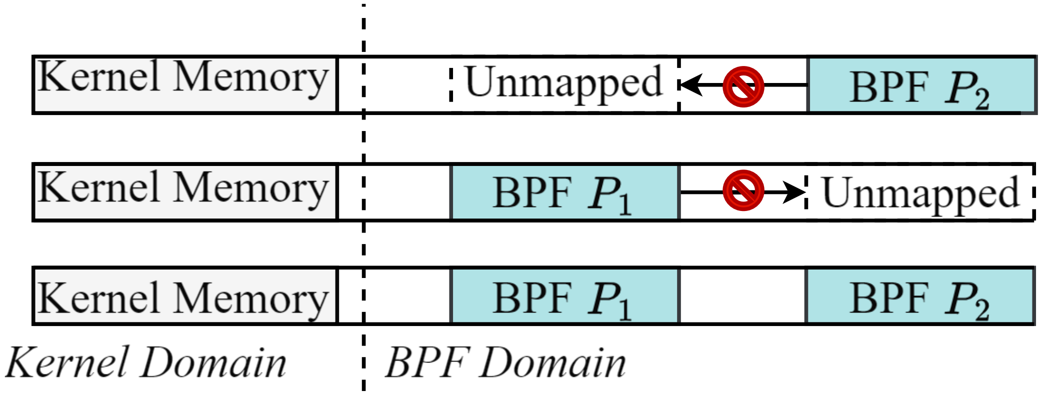
Intra-BPF exploitation

Problem:

Bad BPFs attack the good ones.

MOAT isolates them by address spaces.

TLB flush is slow?



Intra-BPF exploitation

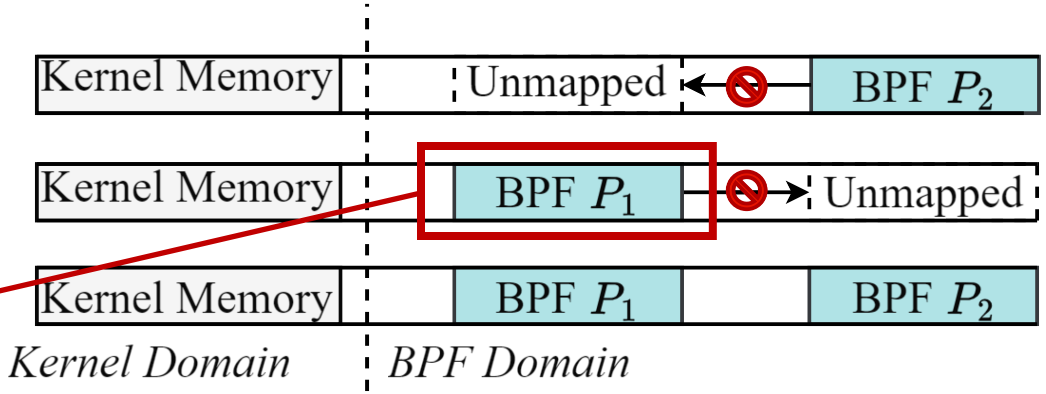
Problem:

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.



Intra-BPF exploitation

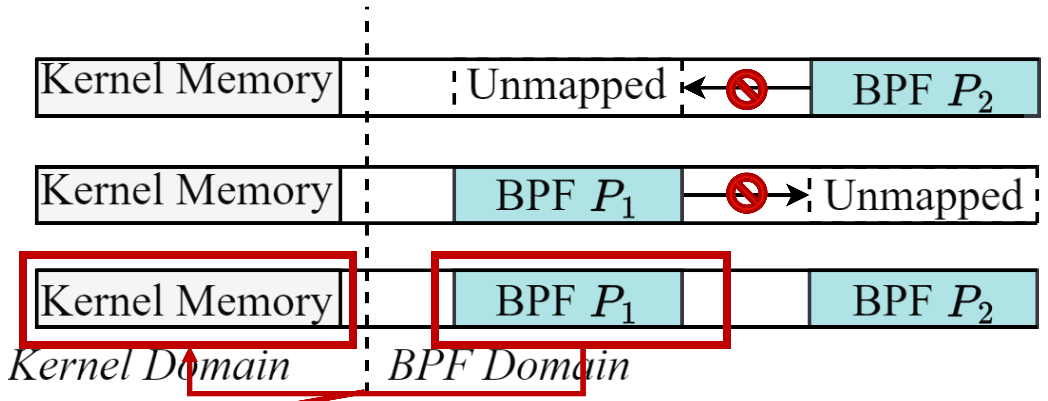
Problem:

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…

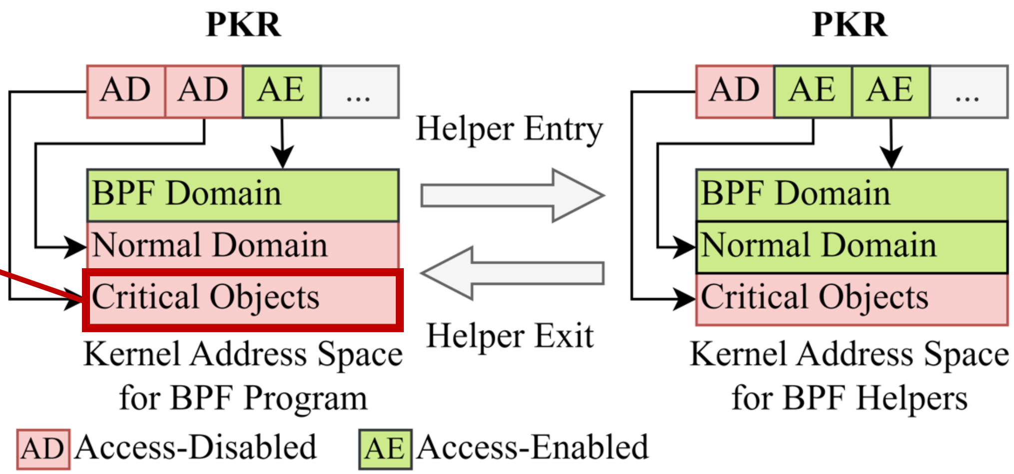
- Isolate **easy-to-exploit** structures from helpers.
- Check parameters against **verified bounds**.

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.

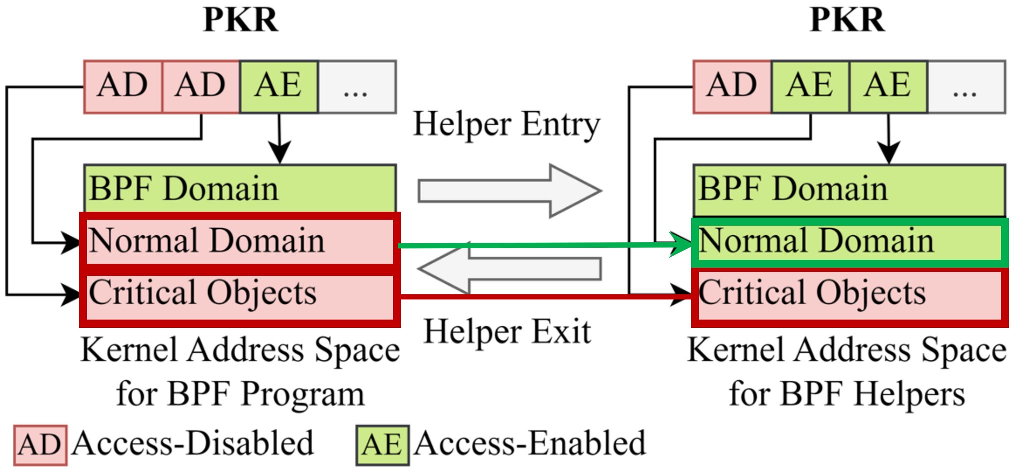


Critical Object Protection

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.

BPF Instructions	Static Register Value Inferred by Verifier	Runtime Register Values for Each Instruction: r0	r1	
`r0 = 0x10`	**r0 = 0x10**	**0x10**	0xbe	...
`r1 = r0 + 0x1`	r0 = 0x10 **r1 = 0x11**	0x10	**0x11**	...
`call BPF_HELPER`	**r0 = 0x10 r1 = 0x11**	**0x10**	**0x11**	...

Why verifier is trustworthy now?

- *Bad* deduced values $\boldsymbol{D}$.
- *Good* bounds $\boldsymbol{E}$ for helpers.

- $\boldsymbol{E}$ never deviates from ground truth $\boldsymbol{T}$ in practice.

	R	D	E	T	**State**
1	0x10	0x10	[0,0x20]	[0,0x20]	✓
2	**0xba**	0xba	[0,0x20]	[0,0x20]	$✓_V$
3	**0xba**	**0x10**	[0,0x20]	[0,0x20]	$✓_M$
4	**0xba**	**0xba**	**[0,0xba]**	[0,0x20]	✗

Dynamic Parameter Auditing

MOAT uses the verifier's bounds to double-check the helper's arguments.

BPF Instructions
`r0 = 0x10`
`r1 = r0 + 0x1`
`call BPF_HELPER`

BPF Instructions

r0 = 0x10	
r0 = 0x10	**r1 = 0x11**
r0 = 0x10	**r1 = 0x11**

Static Register Value Inferred by Verifier

r0	r1	
0x10	0xbe	...
0x10	**0x11**	...
0x10	**0x11**	...

Runtime Register Values for Each Instruction

Why verifier is trustworthy now?

- *Bad* deduced values $\boldsymbol{D}$.
- *Good* bounds $\boldsymbol{E}$ for helpers.

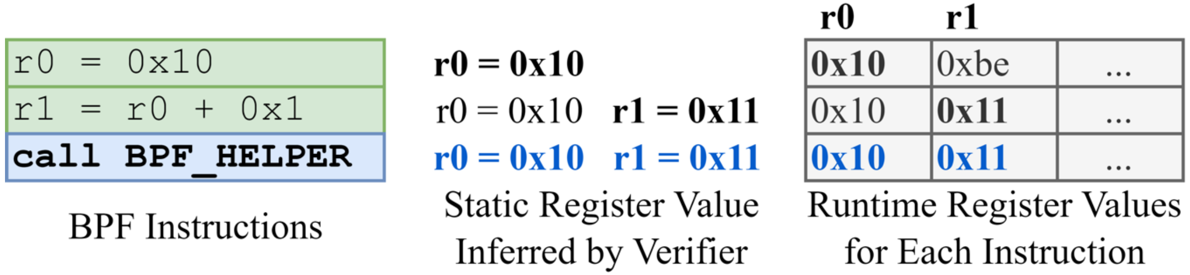
- $\boldsymbol{E}$ never deviates from ground truth $\boldsymbol{T}$ in practice.

Runtime Value

	R	D	E	T	**State**
1	`0x10`	`0x10`	`[0,0x20]`	`[0,0x20]`	✓
2	**`0xba`**	`0xba`	`[0,0x20]`	`[0,0x20]`	$✓_V$
3	**`0xba`**	**`0x10`**	`[0,0x20]`	`[0,0x20]`	$✓_M$
4	**`0xba`**	**`0xba`**	**`[0,0xba]`**	`[0,0x20]`	✗

Dynamic Parameter Auditing

MOAT uses the verifier's bounds to double-check the helper's arguments.



Why verifier is trustworthy now?

- *Bad* deduced values $\boldsymbol{D}$.
- *Good* bounds $\boldsymbol{E}$ for helpers.

Deduced Value

	R	D	E	T	State
1	0x10	0x10	[0,0x20]	[0,0x20]	✓
2	**0xba**	0xba	[0,0x20]	[0,0x20]	$✓_V$
3	**0xba**	**0x10**	[0,0x20]	[0,0x20]	$✓_M$
4	**0xba**	**0xba**	**[0,0xba]**	[0,0x20]	✗

- $\boldsymbol{E}$ never deviates from ground truth $\boldsymbol{T}$ in practice.

Dynamic Parameter Auditing

MOAT uses the verifier's bounds to double-check the helper's arguments.

```
r0 = 0x10
r1 = r0 + 0x1
call BPF_HELPER
```

BPF Instructions

r0 = 0x10
r0 = 0x10 **r1 = 0x11**
r0 = 0x10 r1 = 0x11

Static Register Value Inferred by Verifier

r0	r1	
0x10	0xbe	...
0x10	**0x11**	...
0x10	**0x11**	...

Runtime Register Values for Each Instruction

Why verifier is trustworthy now?

- *Bad* deduced values D.
- *Good* bounds E for helpers.

- E never deviates from ground truth T in practice.

Expected Safe Value

	R	D	E	T	State
1	0x10	0x10	[0,0x20]	[0,0x20]	✓
2	**0xba**	0xba	[0,0x20]	[0,0x20]	$✓_V$
3	**0xba**	**0x10**	[0,0x20]	[0,0x20]	$✓_M$
4	**0xba**	**0xba**	**[0,0xba]**	[0,0x20]	✗

Dynamic Parameter Auditing

MOAT uses the verifier's bounds to double-check the helper's arguments.

```
r0 = 0x10
r1 = r0 + 0x1
call BPF_HELPER
```

BPF Instructions

r0 = 0x10
r0 = 0x10 **r1 = 0x11**
r0 = 0x10 **r1 = 0x11**

Static Register Value Inferred by Verifier

r0	r1	
0x10	0xbe	...
0x10	**0x11**	...
0x10	**0x11**	...

Runtime Register Values for Each Instruction

Why verifier is trustworthy now?

- *Bad* deduced values $\boldsymbol{D}$.
- *Good* bounds $\boldsymbol{E}$ for helpers.

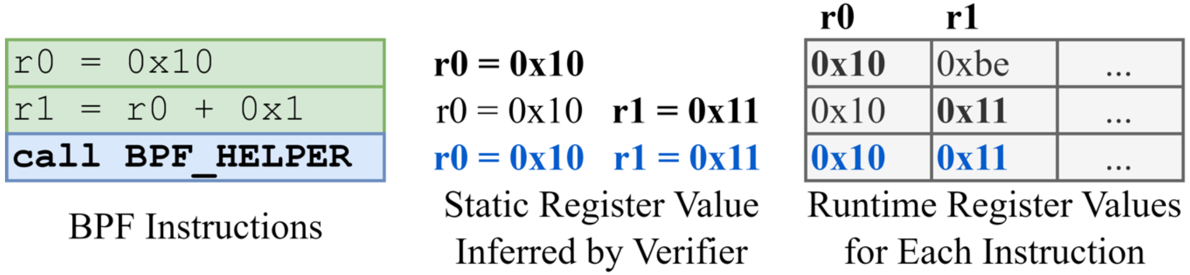
- $\boldsymbol{E}$ never deviates from ground truth $\boldsymbol{T}$ in practice.

Truly Safe Value

	R	D	E	T	State
1	0x10	0x10	[0,0x20]	[0,0x20]	✓
2	**0xba**	0xba	[0,0x20]	[0,0x20]	$✓_V$
3	**0xba**	**0x10**	[0,0x20]	[0,0x20]	$✓_M$
4	**0xba**	**0xba**	**[0,0xba]**	[0,0x20]	✗

Dynamic Parameter Auditing

MOAT uses the verifier's bounds to double-check the helper's arguments.



Why verifier is trustworthy now?

- *Bad* deduced values $\boldsymbol{D}$.
- *Good* bounds $\boldsymbol{E}$ for helpers.

	R	D	E	T	State
1	0x10	0x10	[0,0x20]	[0,0x20]	✓
2	**0xba**	0xba	[0,0x20]	[0,0x20]	$✓_{V}$
3	**0xba**	**0x10**	[0,0x20]	[0,0x20]	$✓_{M}$
4	**0xba**	**0xba**	**[0,0xba]**	[0,0x20]	✗

- $\boldsymbol{E}$ never deviates from ground truth $\boldsymbol{T}$ in practice.

Evaluation

Security Evaluation

We verified that MOAT mitigates all **26** memory-related BPF CVEs

- L3: verifier deduces `r5`

```
1 r5 = <bad addr>
2 r6 = 0x600000002
3 if (r5>=r6||r5<=0) // R&V:0x1<=r5<=0x600000001
4  exit(1)
5 r5 = r5 | 0       // R:r5=<bad addr> V: r5=0x1
6 *(ptr+r5)=0xbad   // PKS violation
```

Security Evaluation

We verified that MOAT mitigates all **26** memory-related BPF CVEs

- L5: `MOD32` *forgets* to track upper bits
- `r5` is mis-deduced to 0x1

```
1 r5 = <bad addr>
2 r6 = 0x600000002
3 if (r5>=r6||r5<=0) // R&V:0x1<=r5<=0x600000001
4   exit(1)
5 r5 = r5 | 0        // R:r5=<bad addr> V: r5=0x1
6 *(ptr+r5)=0xbad    // PKS violation
```

Security Evaluation

We verified that MOAT mitigates all **26** memory-related BPF CVEs

- MOAT saves the day!

```
r5 = <bad addr>
r6 = 0x600000002
if (r5>=r6||r5<=0) // R&V:0x1<=r5<=0x600000001
  exit(1)
r5 = r5 | 0        // R:r5=<bad addr> V: r5=0x1
*(ptr+r5)=0xbad    // PKS violation
```

Performance Evaluation

In sum…

- Network filtering: **<2%**.
- System profiling: **<13%**.
- Seccomp (cBPF): **<3%**

And many more…

- Numerous BPF programs…
- Comparison with SandBPF…
- Microbenchmark…

Takeaways.

- BPF is powerful but its **security** is a concern.

- BPF security can benefit from **hardware features**.

- Good protection is **multi-folded**.
  (Software + Hardware & Memory + API)

My Wife (Yuqi Qian) & Me (Hongyi Lu)

Thank You!

My Homepage

Email Me

Project Site