南方科技大学
SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

香港科技大學
THE HONG KONG UNIVERSITY OF SCIENCE AND TECHNOLOGY

COMPASS Lab
COMPuter And System Security Lab

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

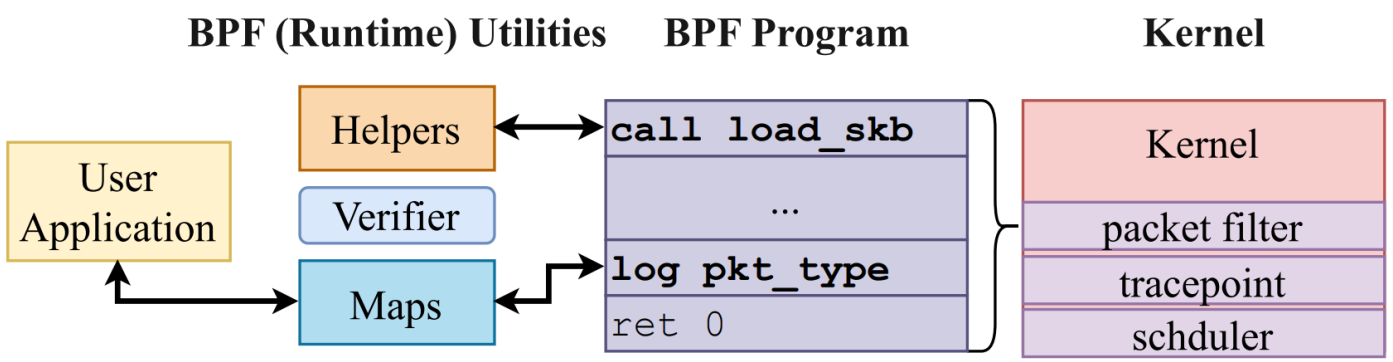
[2]Hong Kong University of Science and Technology

Background

What is (e)BPF?

Extended Berkeley Packet Filter:

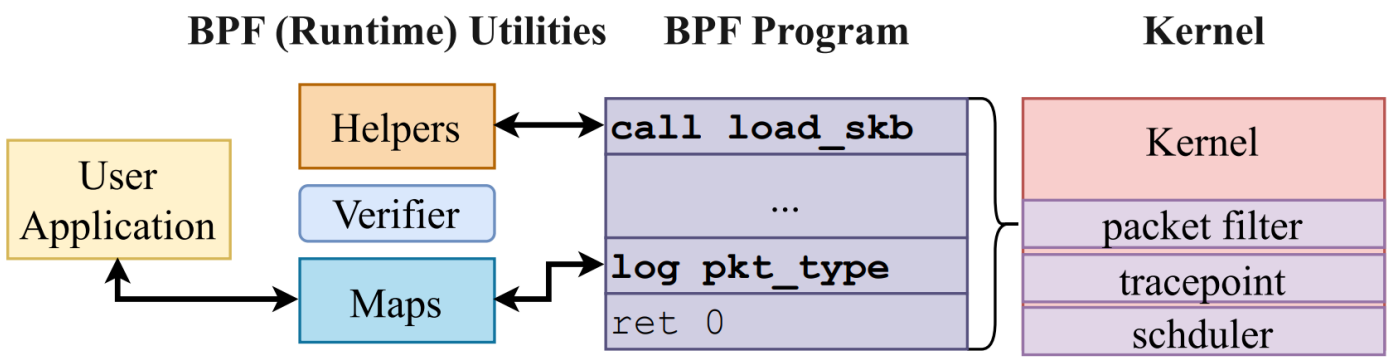
- Kernel Virtual Machine



What is (e)BPF?

Extended Berkeley Packet Filter:

- Kernel Virtual Machine



- Extended from classic BPF (cBPF), introduced to Linux in 2014.
- Packet Filter ⟶ Tracing/Network/Security…

Why (e)BPF instead of LKM?

Why (e)BPF instead of LKM?

- **Fast**: Run in JITed native code.

Why (e)BPF instead of LKM?

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

Why (e)BPF instead of LKM?

- **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*.

Why (e)BPF instead of LKM?

- **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*.

Verifier: Do not load it, or your kernel will go kaboom!

Sounds good, but?

BPF security is a concern.

(**26** arbitrary R/W CVEs).

Because…

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

Sounds good, but?

BPF memory safety is a concern.

Because…

- Static analysis is **hard**.

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

Sounds good, but?

BPF memory safety is a concern.

Because…

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

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

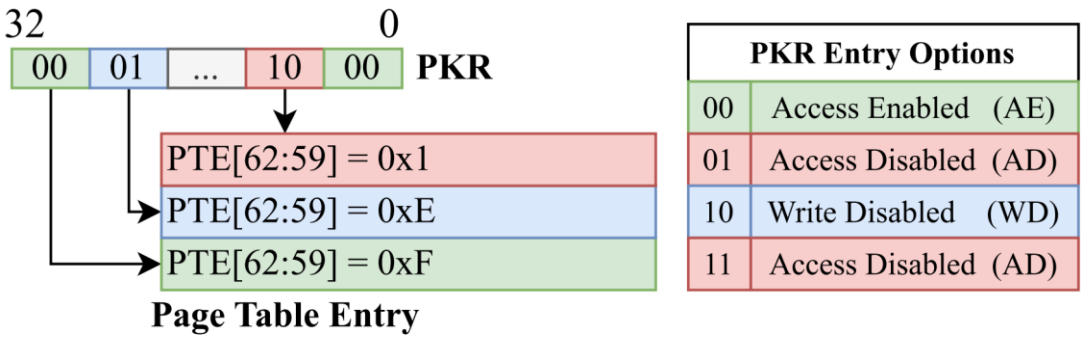
BPF CVEs

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!

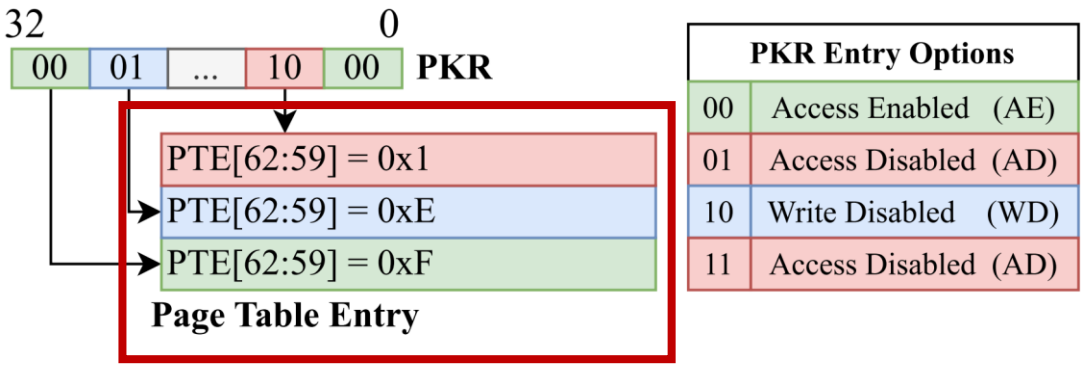
Wait..., what is Intel MPK?



Hardware Isolation!

Wait..., what is Intel MPK?

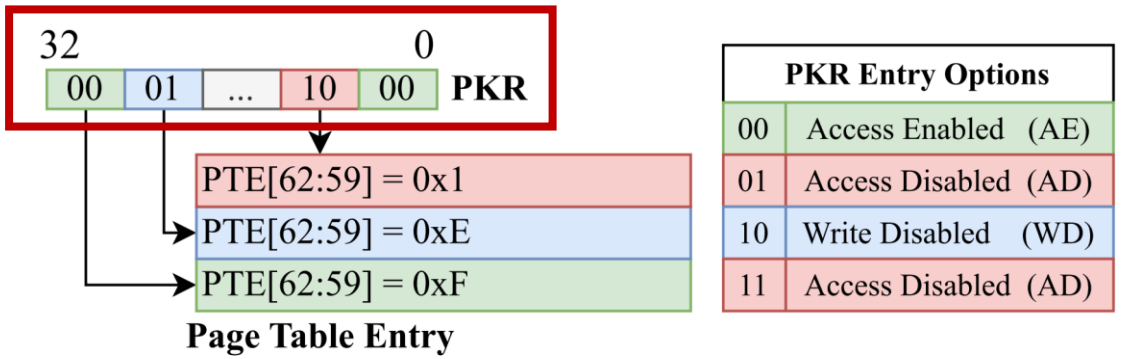
- Add a **4-bit tag** to PTEs (16 tags).



Hardware Isolation!

Wait..., what is Intel MPK?

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



Method

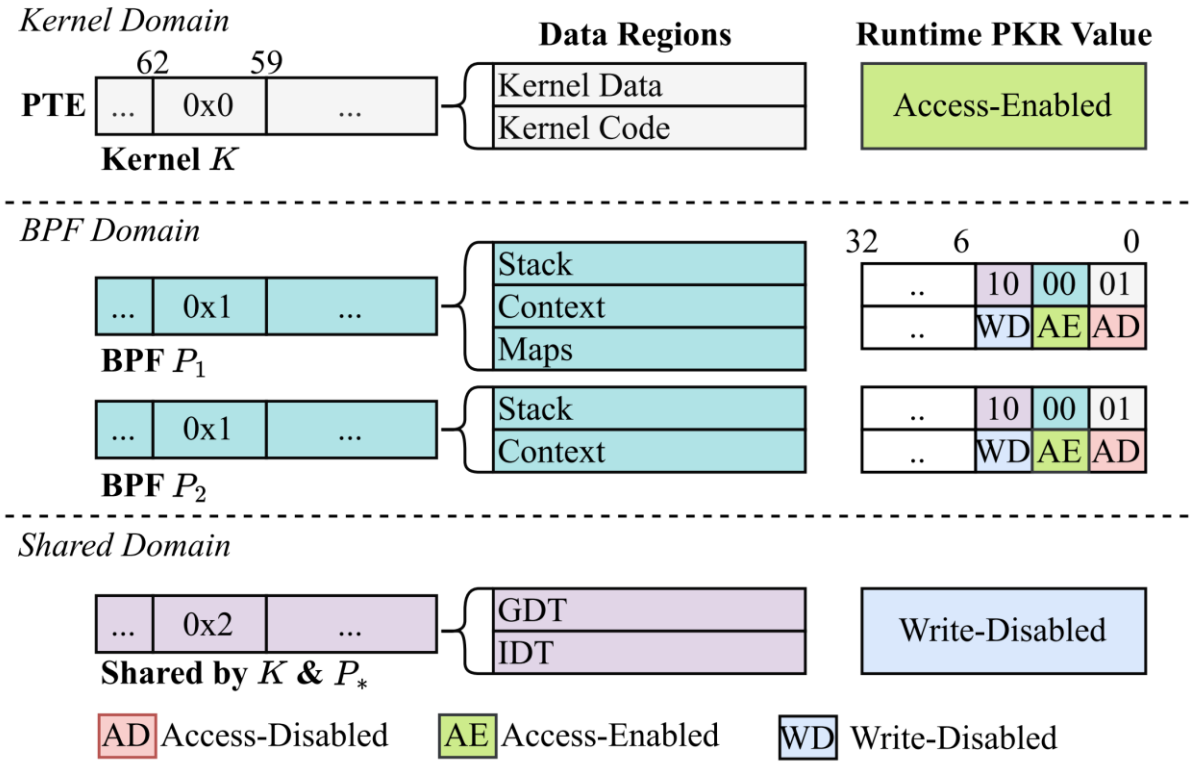
Limited MPK Tags

MPK is…

- Only 16 tags
- Lightweight

So… ***insufficient*** for multiple BPF programs.

But… ***abundant*** for isolating kernel/BPF.



Limited MPK Tags

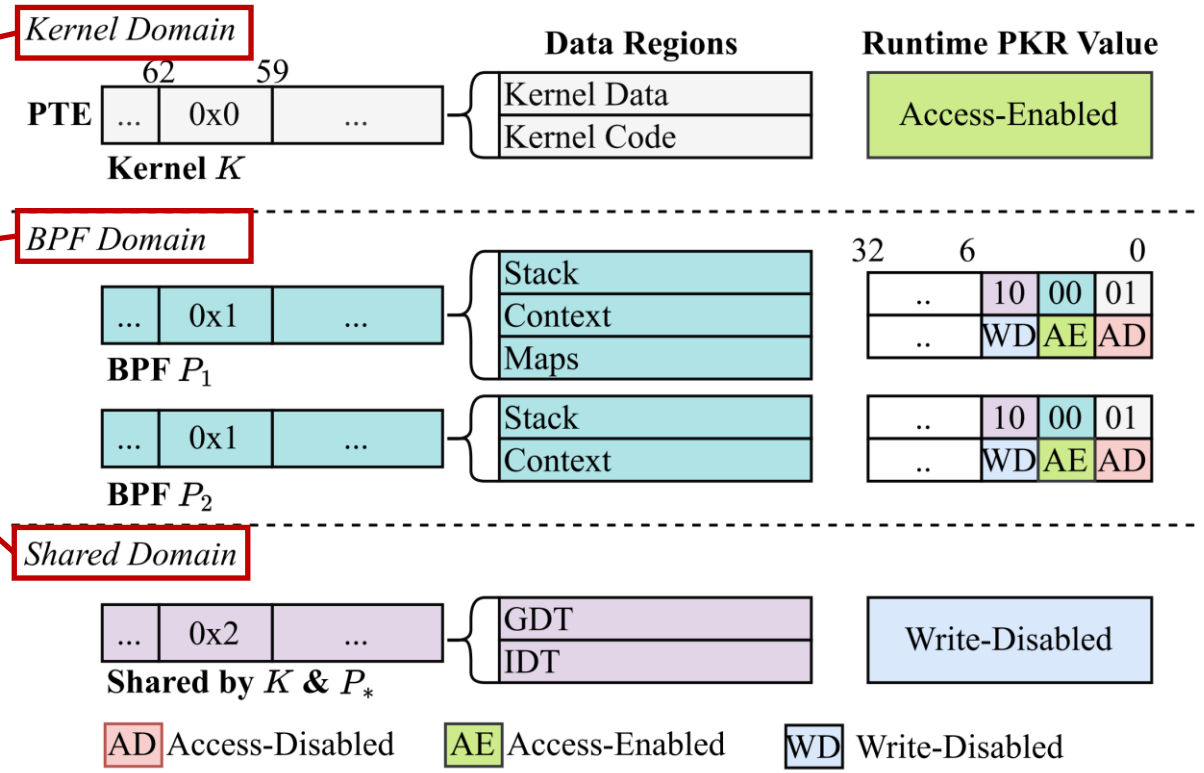
MPK is…

- Only 16 tags
- Lightweight

So… *bad* for multiple BPF programs.

But… *good* for isolating kernel/BPF.

Three Domain Three Tags

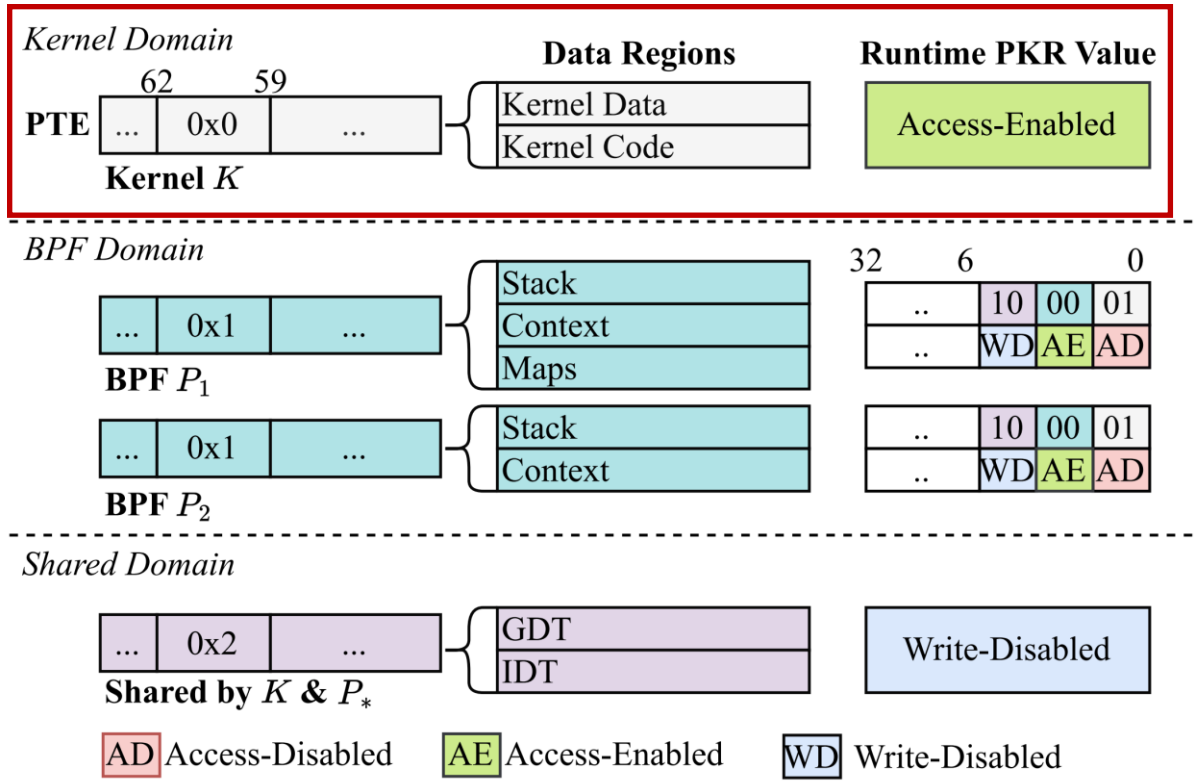


Limited MPK Tags

MPK is…

- Only 16 tags
- Lightweight

Kernel Stuff



So… *bad* for multiple BPF programs.

But… *good* for isolating kernel/BPF.

Limited MPK Tags

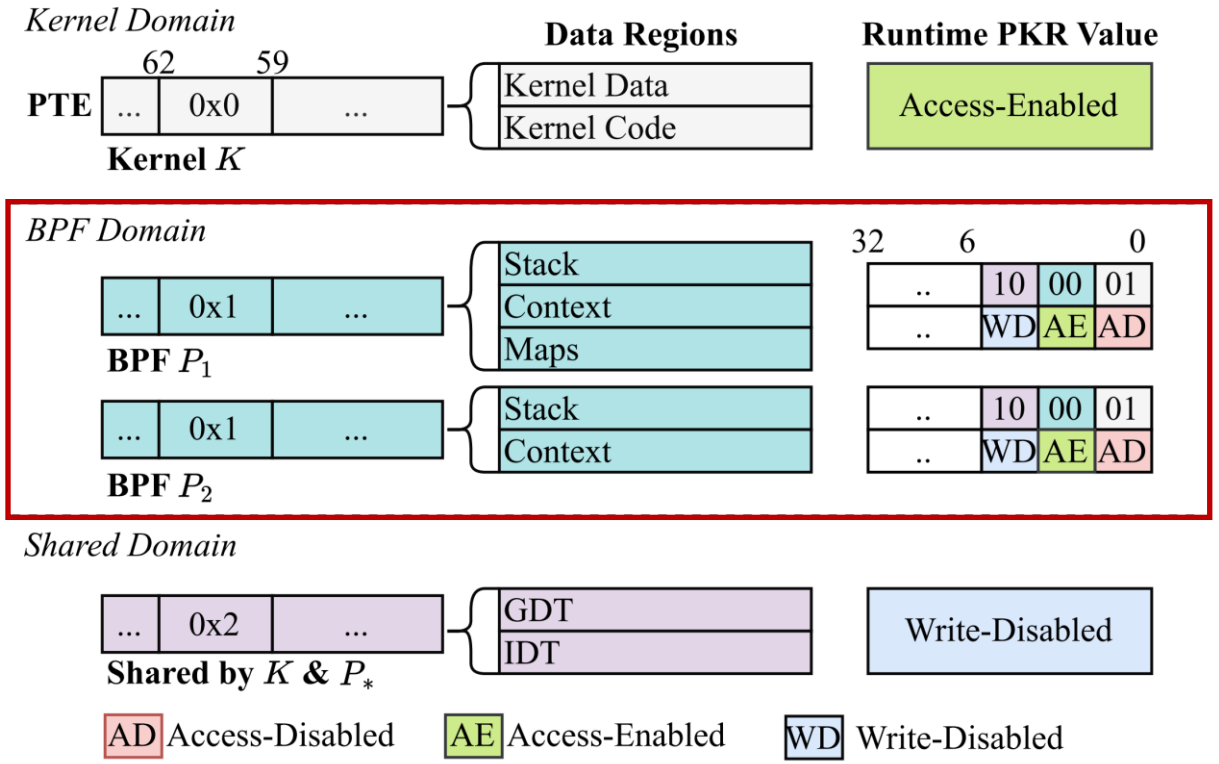
MPK is…

- Only 16 tags
- Lightweight

Constrain ALL BPF programs

So… *bad* for multiple BPF programs.

But… *good* for isolating kernel/BPF.



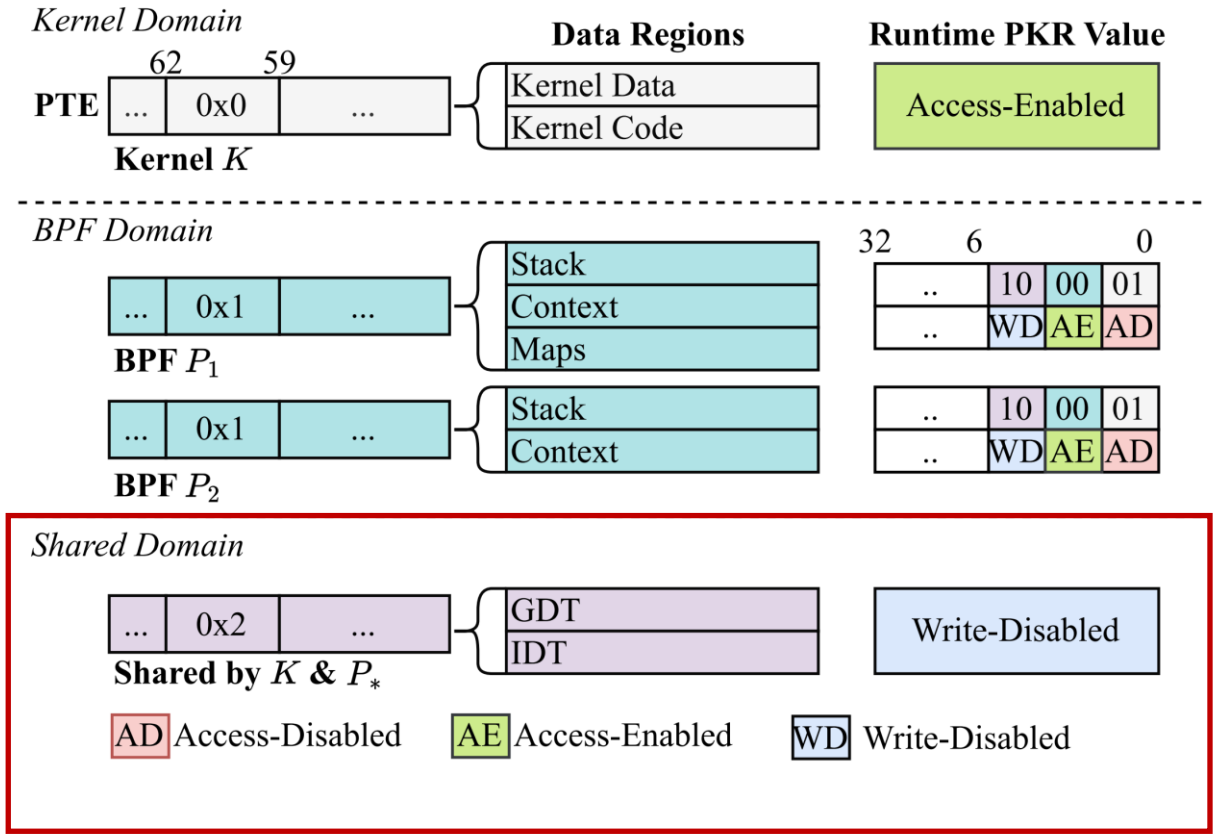
Limited MPK Tags

MPK is…

- Only 16 tags
- Lightweight

So… *bad* for multiple BPF programs.

But… *good* for isolating kernel/BPF.

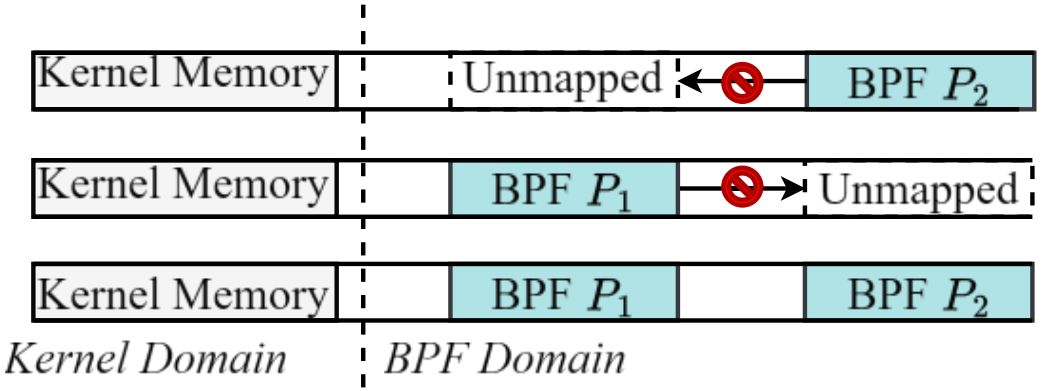


Things both BPF & Kernel need

Intra-BPF exploitation

Problem:

Bad BPFs attack the good ones.

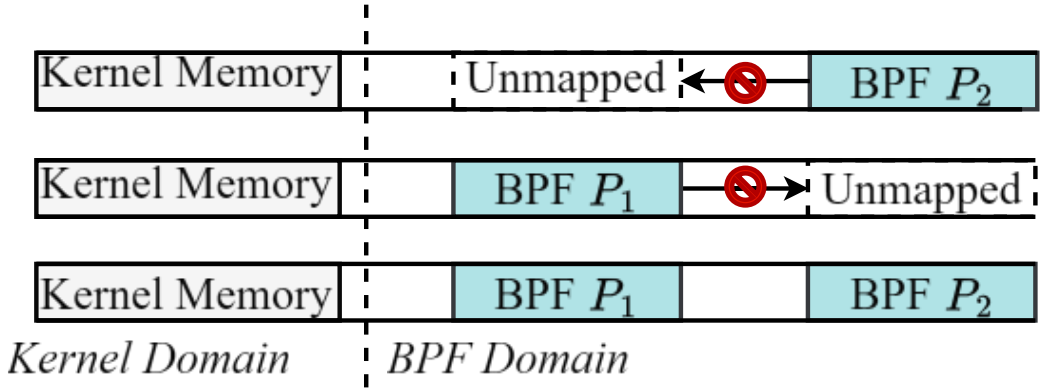


Intra-BPF exploitation

Problem:

Bad BPFs attack the good ones.

Solution: MOAT isolates them by address spaces.



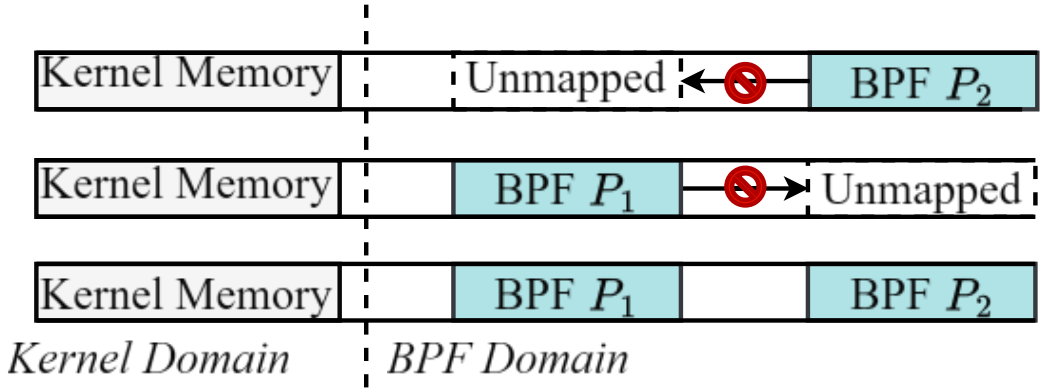
Intra-BPF exploitation

Problem:

Bad BPFs attack the good ones.

Solution: MOAT isolates them by address spaces.

Issue: Slow TLB flushes

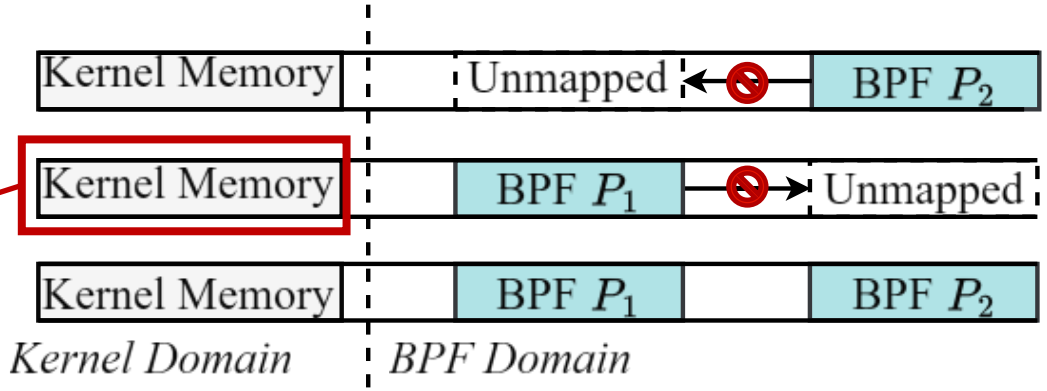


Intra-BPF exploitation

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

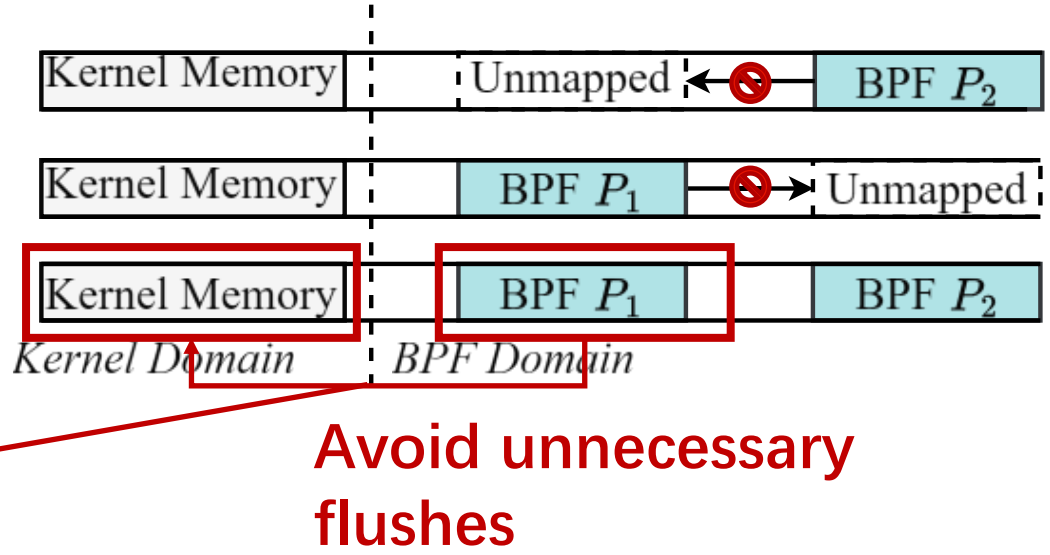
Problem:

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.



Kernel API Security

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

MOAT does...

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.

Kernel API Security

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

MOAT does...

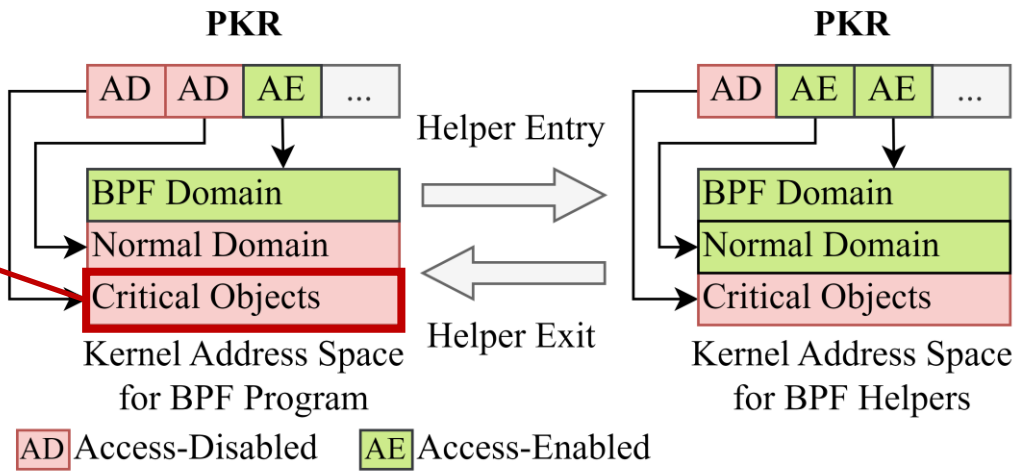
- Isolate **easy-to-exploit** structures from helpers.
- Check parameters against **verified bounds**.

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.

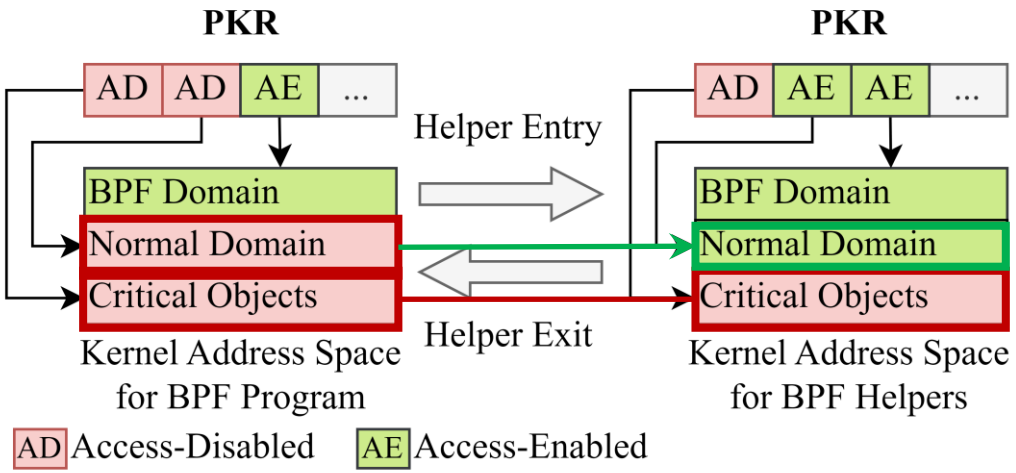


Critical Object Protection

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

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

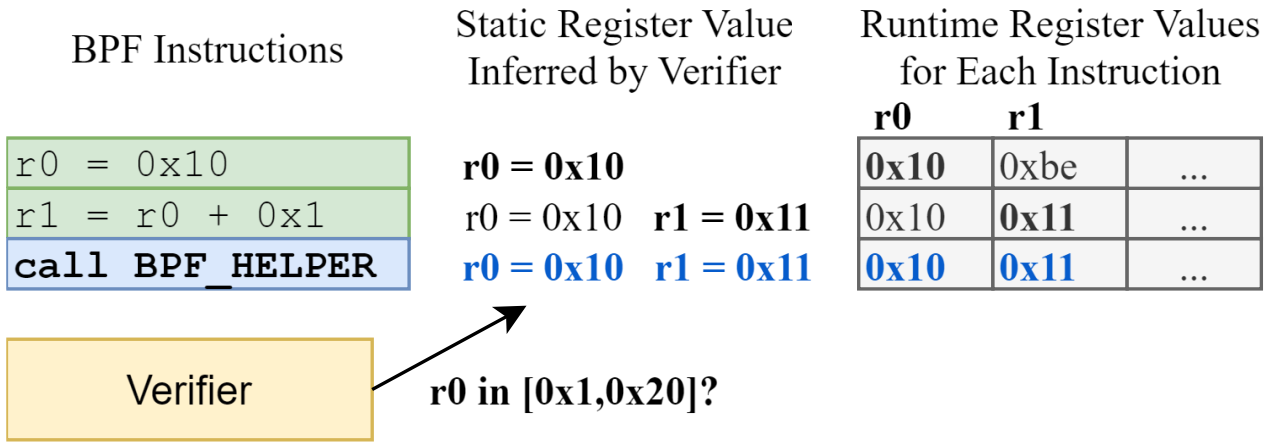
MOAT protects them with an extra MPK tag.



Dynamic Parameter Auditing

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

Why verifier is **trustworthy** now?

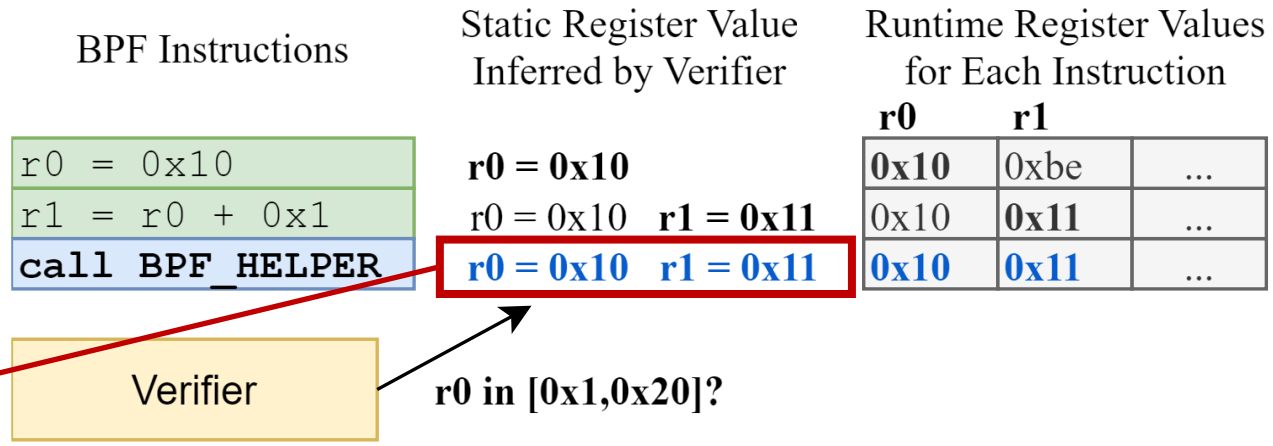


Dynamic Parameter Auditing

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

Why verifier is **trustworthy** now?

- ***Bad*** deduced values.

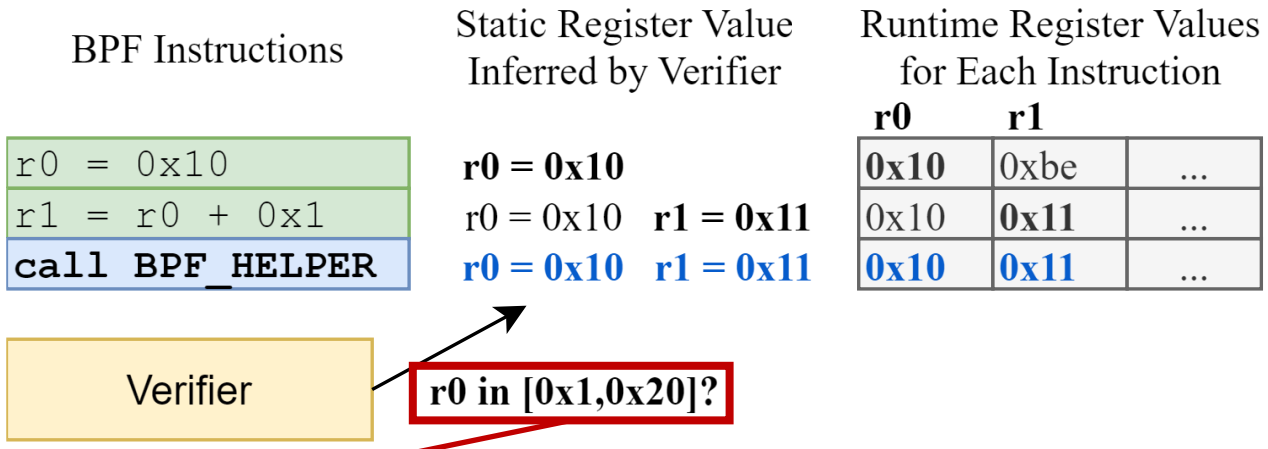


Dynamic Parameter Auditing

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

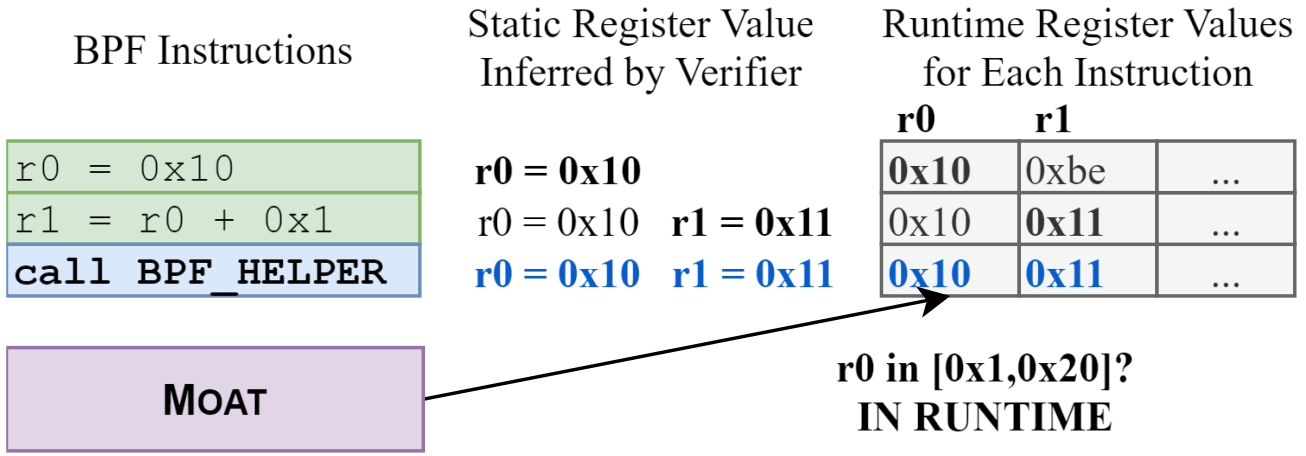
Why verifier is **trustworthy** now?

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



Dynamic Parameter Auditing

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



Evaluation

Security Evaluation

We verified that MOAT mitigates all **26** BPF CVEs within MOAT's scope

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, 2021-23866, 2021-3489, 2021-3490, 2021-20268, 2021-3444,2021-33200, 2021-45402, 2022-2785, 2022-23222, 2023-39191, 2023-2163

Security Evaluation

Now, let's go through one in detail.

- 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
```

R: Runtime Value V: Verifier Deduced Value

Security Evaluation

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

- L5: `OR32` performed a wrong truncation
- `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
```

R: Runtime Value **V: Verifier Deduced Value**

Security Evaluation

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

- MOAT saves the day!

```
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
```

R: Runtime Value V: Verifier Deduced Value

Performance Evaluation

In sum…

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

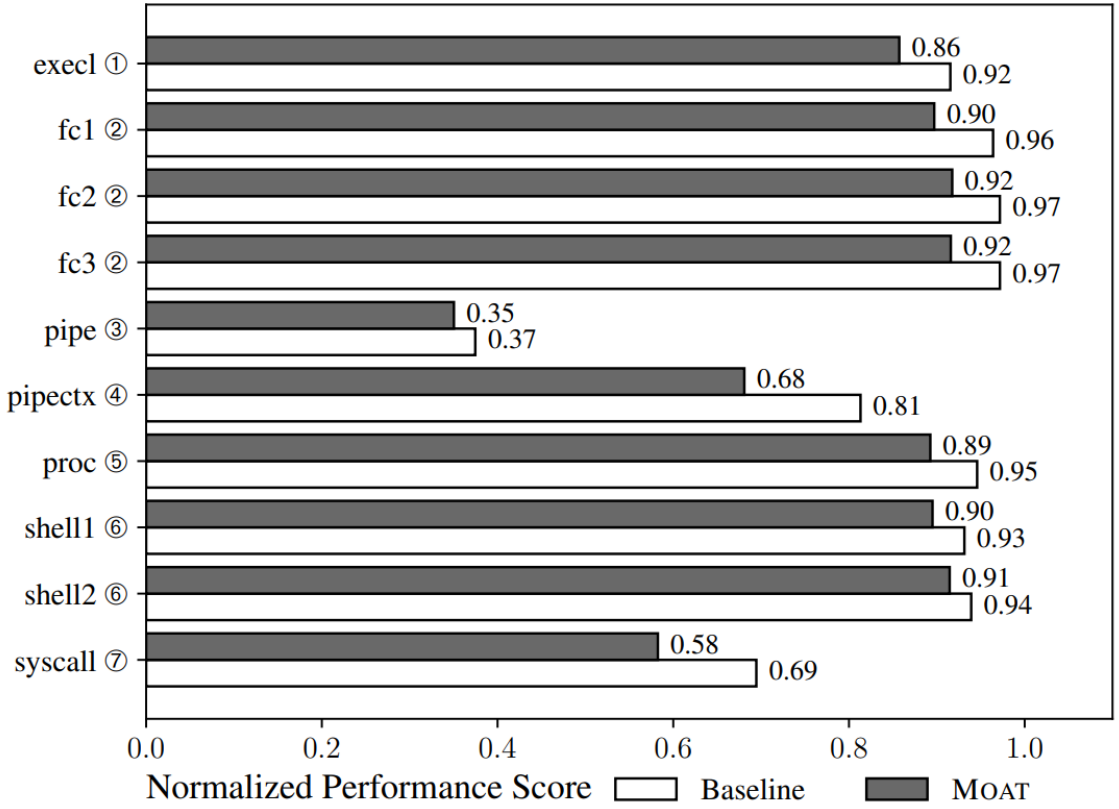
Throughput (TPPS)	drop	byte	pkt	trim	flow	all
Baseline	594.39 (99.70%)	594.67 (99.73%)	594.26 (99.66%)	594.74 (99.73%)	594.39 (99.68%)	587.22 (98.47%)
MOAT	593.10 (99.46%)	594.31 (99.66%)	594.43 (99.68%)	594.69 (99.73%)	593.10 (99.46%)	575.33 (96.48%)

Throughput (TPPS)	xdp1	xdp2	adj	rxq1	rxq2
Baseline	560.58 (99.84%)	557.78 (99.34%)	531.11 (99.66%)	528.36 (99.15%)	530.52 (99.55%)
MOAT	560.15 (99.76%)	557.76 (99.33%)	530.65 (99.58%)	527.57 (99.00%)	527.66 (99.05%)

Performance Evaluation

In sum…

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



Performance Evaluation

In sum…

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

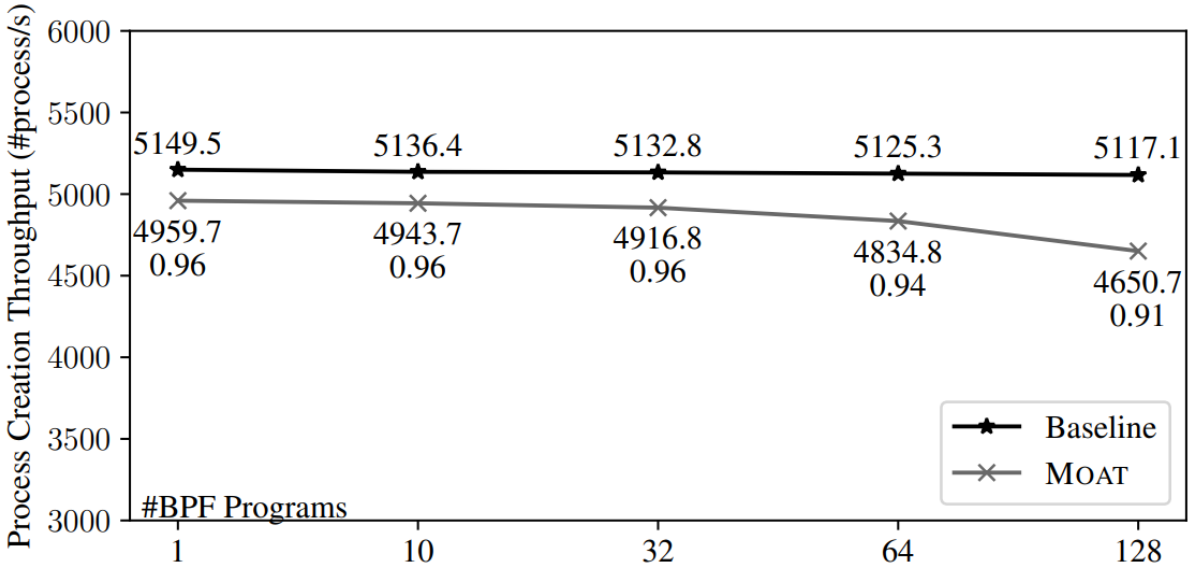
Throughout (Treq/s)	1 worker	2 worker	3 worker
Vanilla (no `seccomp-BPF`)	148.1 (100%) ±12.81	179.5 (100%) ± 8.35	165.2 (100%) ±4.72
Baseline	147.2 (99.4%) ±9.56	171.3 (95.4%) ±8.08	160.5 (97.2%) ±5.28
MOAT	142.3 (96.1%) ±8.77	166.3 (92.6%) ±6.70	158.0 (95.6%) ±4.48

Performance Evaluation

Comparing with SFI-based SandBPF

Test #Conn (req./s)	XDP				Socket Filter			
	Base	MOAT	Rel.	Ref.	Base	MOAT	Rel.	Ref.
Apache 20	34,303	33,689	2%	**0%**	40,666	40,286	**1%**	4%
Apache 100	31,929	30,726	**4%**	8%	37,998	36,546	**4%**	4%
Apache 200	27,751	26,657	**4%**	5%	32,652	31,344	4%	**3%**
Apache 500	24,786	24,439	**1%**	7%	30,262	29,423	**3%**	7%
Apache 1000	24,597	24,470	**1%**	6%	29,545	28,961	**2%**	7%
Nginx 20	22,688	21,892	**3%**	7%	23,359	23,530	**0%**	10%
Nginx 100	21,492	20,689	**4%**	7%	22,870	22,482	**2%**	8%
Nginx 200	19,972	19,216	**4%**	6%	21,562	20,984	**3%**	8%
Nginx 500	18,470	17,814	**4%**	6%	19,421	18,713	**4%**	7%
Nginx 1000	17,024	16,735	**2%**	3%	17,392	17,098	**2%**	6%

1->128 BPF programs at the same time



Takeaways.

- BPF is powerful but its **security** is a concern.
- BPF security can benefit from **hardware features**.
- MOAT protection is **multi-folded**.
  (Software + Hardware & Memory + API)

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

Thank You!

My Homepage

Email Me

Project Site