Unleashing Unprivileged eBPF Potential with Dynamic Sandboxing

Soo Yee Lim (UBC), Xueyuan Han (WFU), Thomas Pasquier (UBC)

eBPF Improves Kernel Extensibility

Cilium
eBPF-based Networking, Security, and Observability

Falco
Cloud Native Runtime Security

Katran
High Performance Layer-4 Load balancer

Pixie
Scriptable observability for Kubernetes

These tools *cannot* be used by unprivileged users.

Content Overview

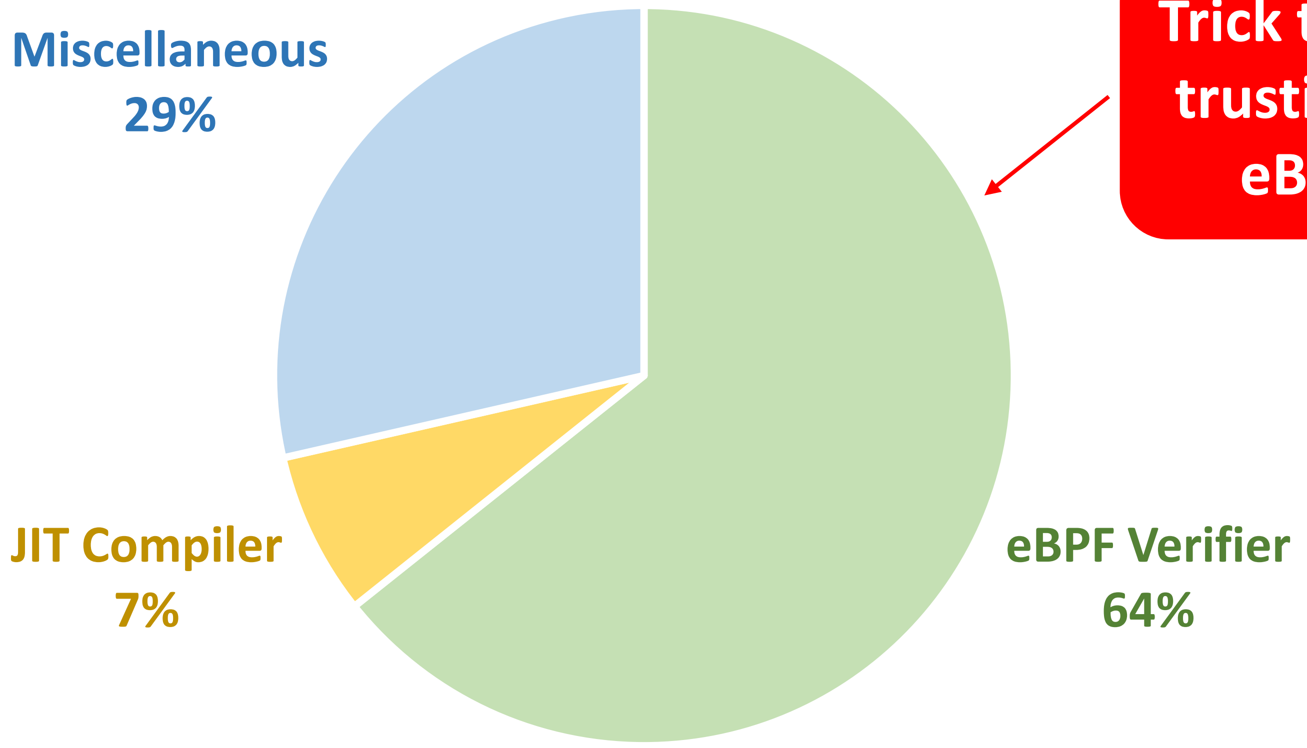
- Problem
- Related Work
- Our Solution
- Evaluation Results
- Future Work
- Conclusion

eBPF ensures safety via the eBPF Verifier

But…

Is the safety of eBPF programs always guaranteed at *runtime*?

A Summary of eBPF CVEs (2010-2023)

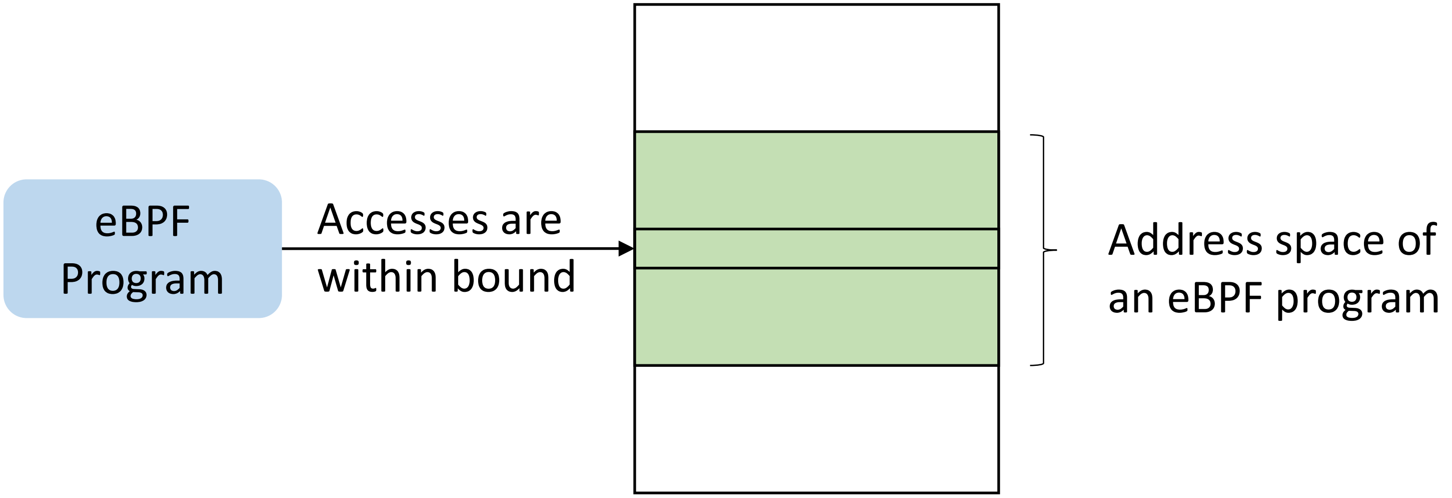


Trick the verifier into trusting a malicious eBPF program.

Implication of eBPF Vulnerabilities

- CVE-2021-3490: The eBPF verifier's ALU32 bounds tracking for bitwise ops (AND, OR and XOR) did not properly update 32-bit bounds.

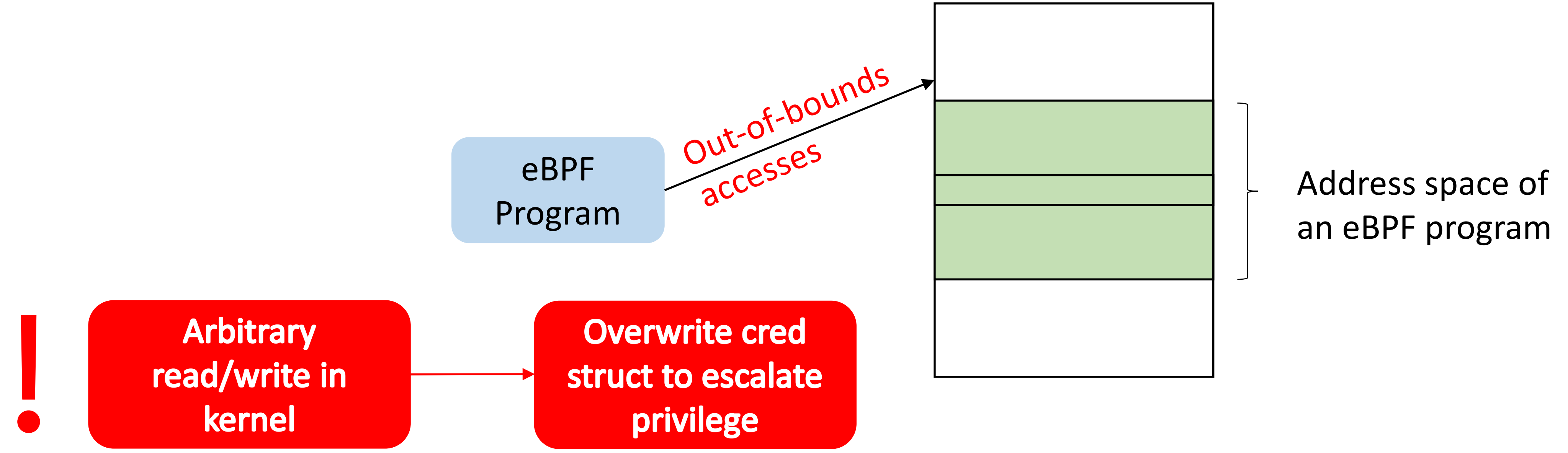
What the *static* verifier believes the program is doing:



Implication of eBPF Vulnerabilities

- CVE-2021-3490: The eBPF verifier’s ALU32 bounds tracking for bitwise ops (AND, OR and XOR) did not properly update 32-bit bounds.

What actually happens during *runtime*:



Research Problem

The eBPF verifier alone
does not
guarantee runtime safety.

Content Overview

- Problem
- Related Work
- Our Solution
- Evaluation Results
- Future Work
- Conclusion

Disable Unprivileged eBPF

- Many Linux distributions (e.g., Ubuntu, SUSE) *disable unprivileged eBPF by default* to prevent unprivileged users from exploiting eBPF vulnerabilities.

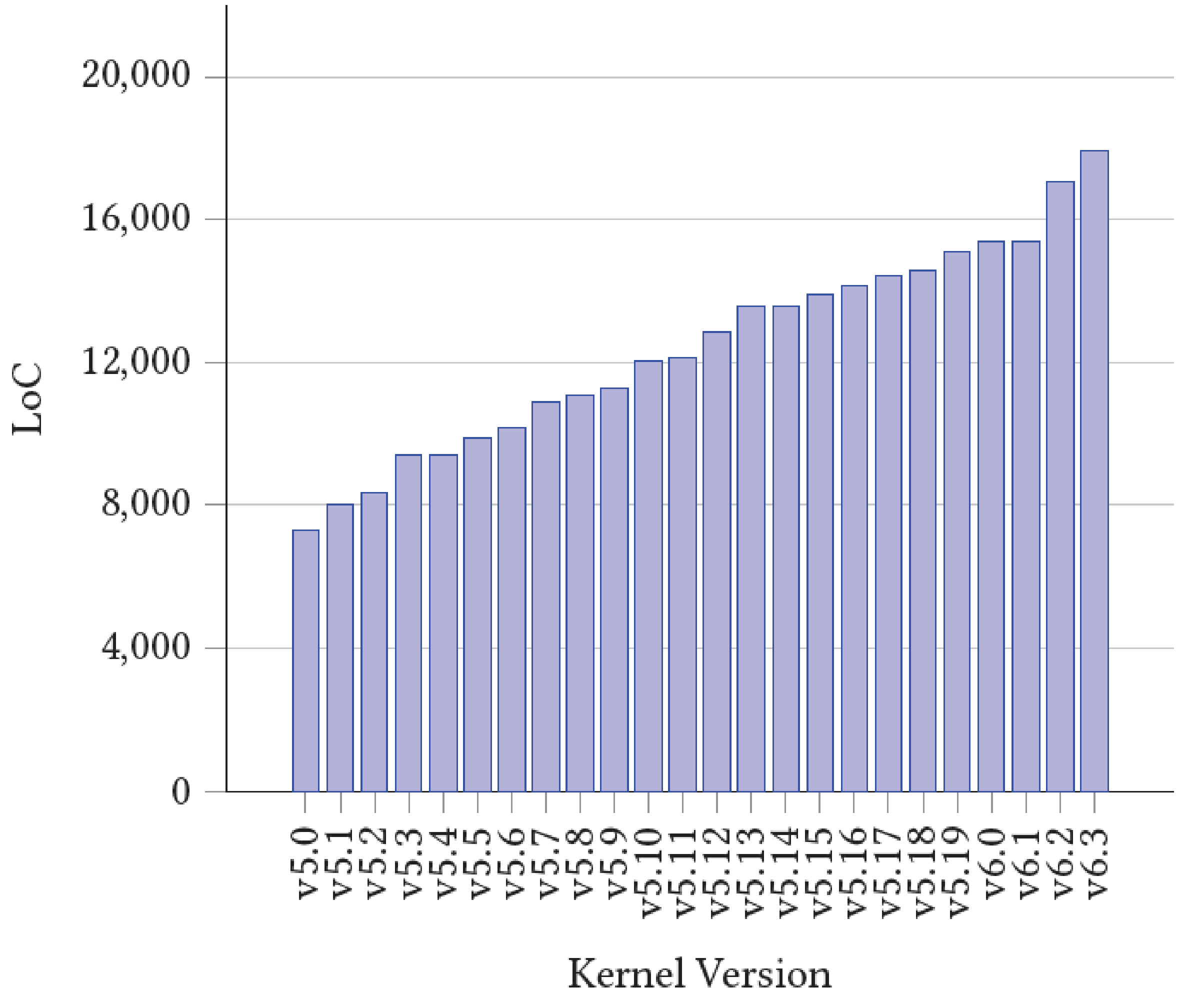
Unprivileged users cannot use eBPF to customize policies for a particular application or container.

Formally Verifying the eBPF Verifier

- Formally verifying that the eBPF verifier can ensure that it *correctly* implements the specification.

The *size* and *complexity* of the eBPF verifier makes it difficult to formally verify the verifier in its entirety.

The Evolution of the Verifier's Size



The size of the eBPF verifier has more than doubled in the last four years.

No existing work has managed to formally verify the verifier in its entirety.

Rust-based eBPF

- Rust-based eBPF *replaces the eBPF verifier* with the Rust tool-chain to perform static checks (e.g., memory safety).

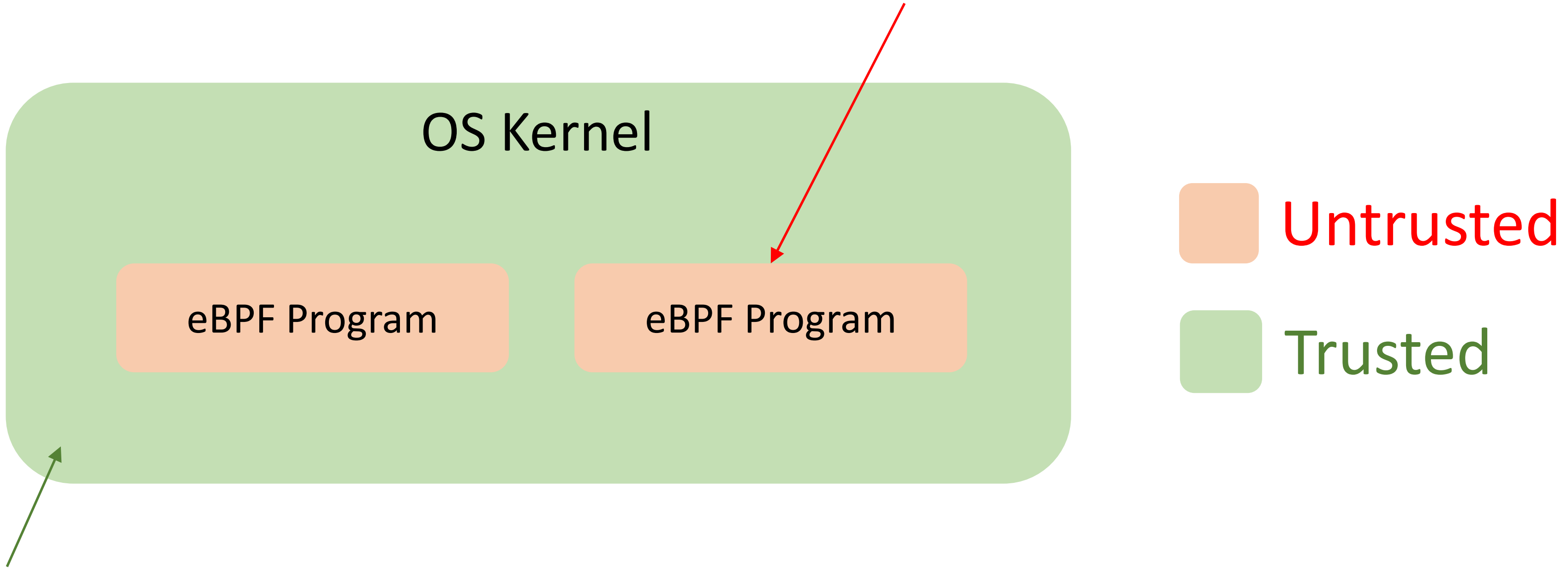
eBPF programs can exploit vulnerabilities
in the Rust verifier
to violate safety at runtime.

Content Overview

- Problem
- Related Work
- Our Solution
- Evaluation Results
- Future Work
- Conclusion

Threat Model

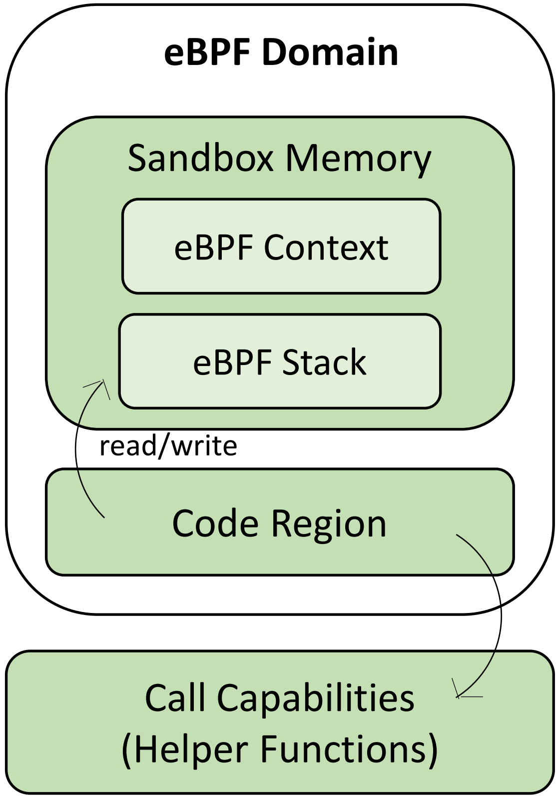
Unprivileged adversary capable of exploiting eBPF vulnerabilities to achieve *out-of-bounds access within kernel memory*.



The kernel is assumed benign and side-channel attacks are considered orthogonal.

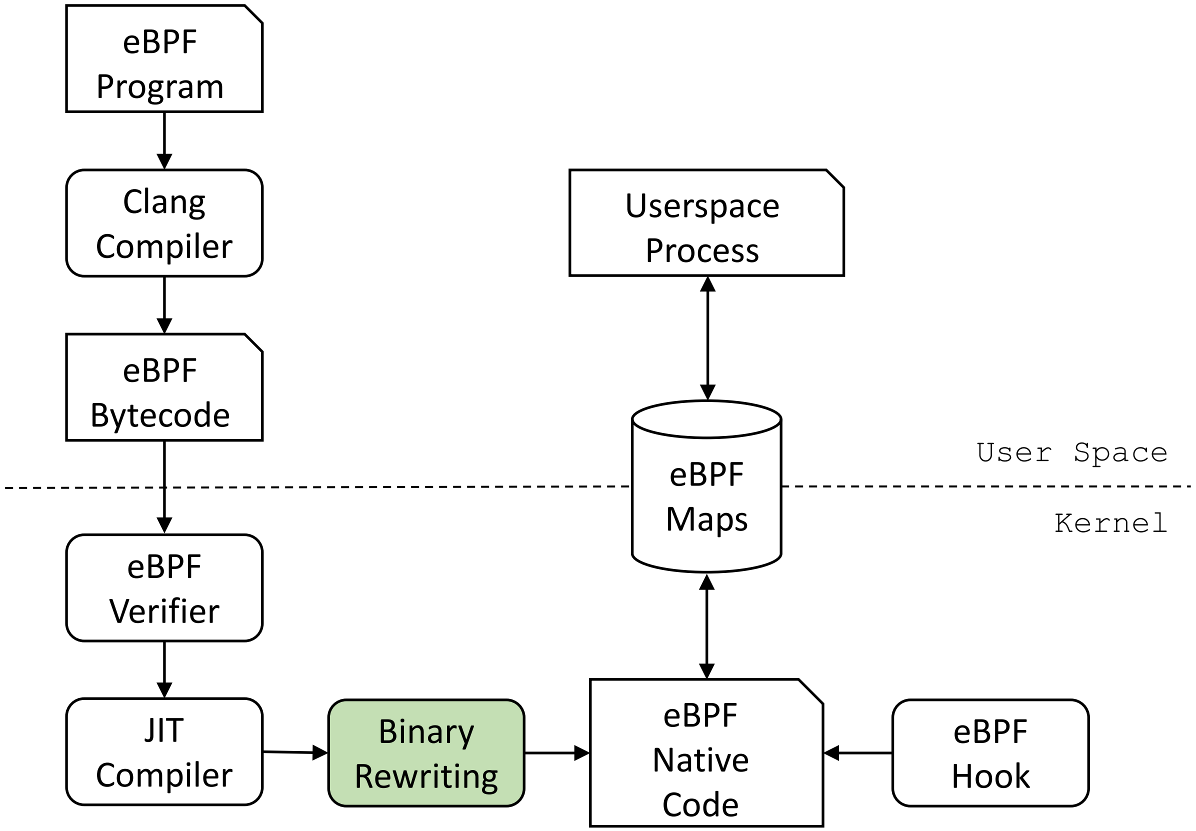
SandBPF: Dynamically Sandboxed eBPF

1 Address Masking (Memory Safety)



2 Redirect Calls to Trampoline (Control Flow Integrity)

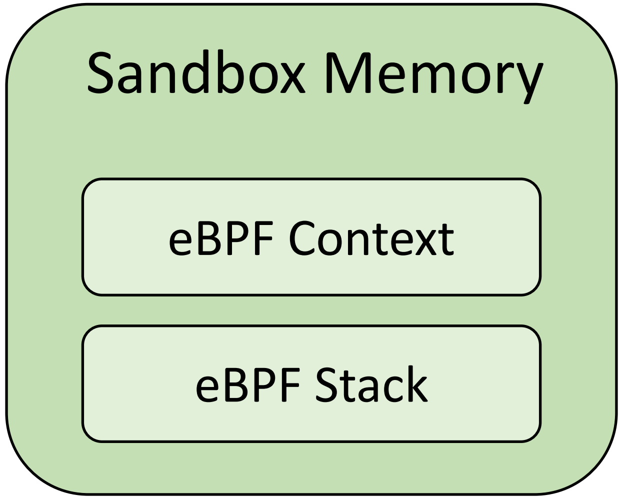
SandBPF Is Minimally Invasive



We reuse existing eBPF pipeline and extend only what is necessary.

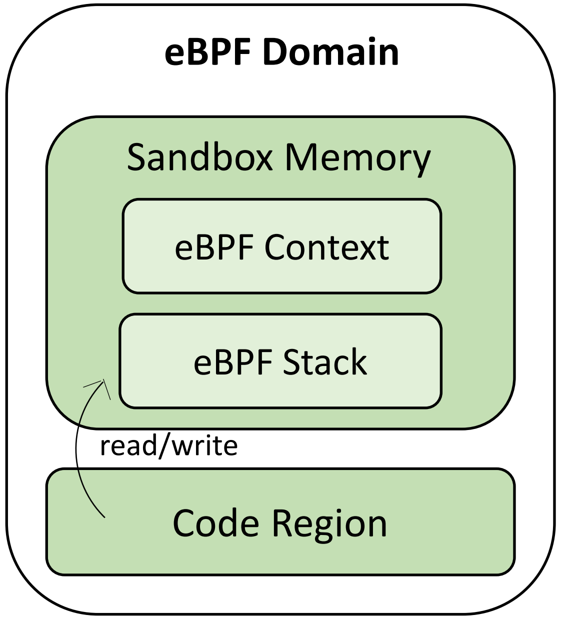
Memory Safety

Sandbox Management



A region of memory (the sandbox) is pre-allocated to store the data of an eBPF program.

Address Masking



addr = 0xA000
len = 4096 bytes

Consider an invalid memory access at address 0xB123

and_mask = 0xFFF; or_mask = 0xA000

0xB123

and 0x0FFF

0x0123

or 0xA000

0xA123

All memory accesses *always* fall within the bounds of an eBPF sandbox.

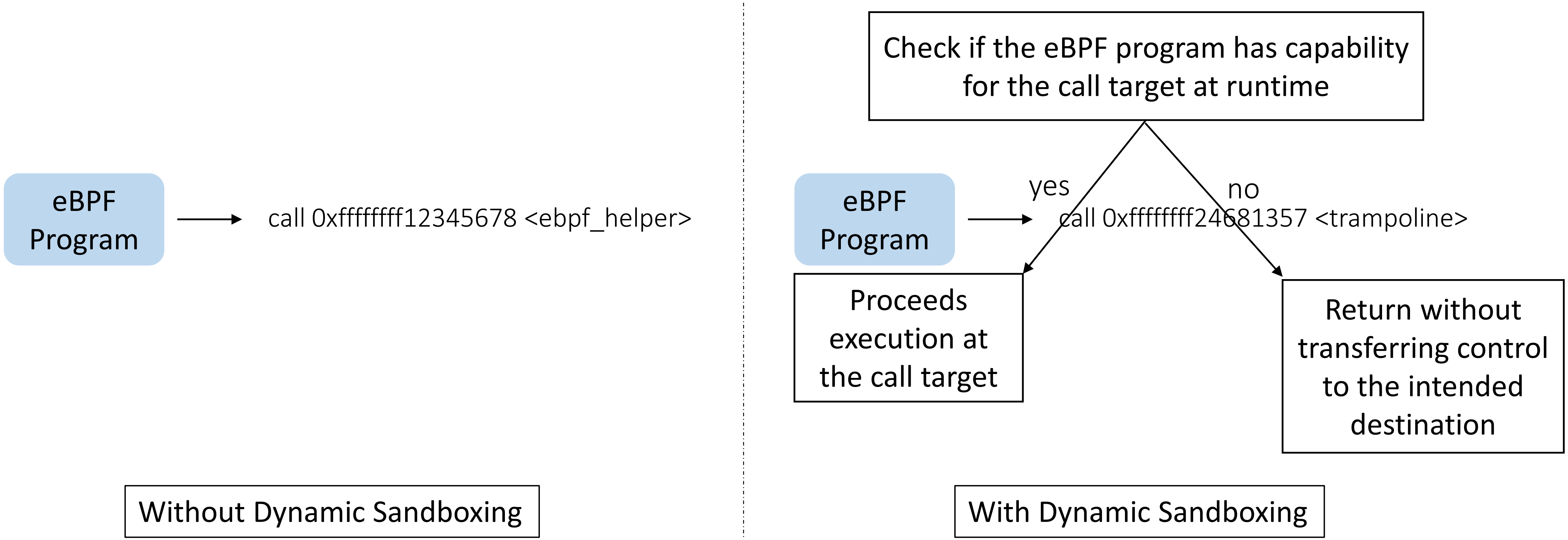
Control Flow Integrity

Call Capabilities

- At load time, we associate each eBPF program type with a set of capabilities corresponding to the helper functions it is allowed to call.

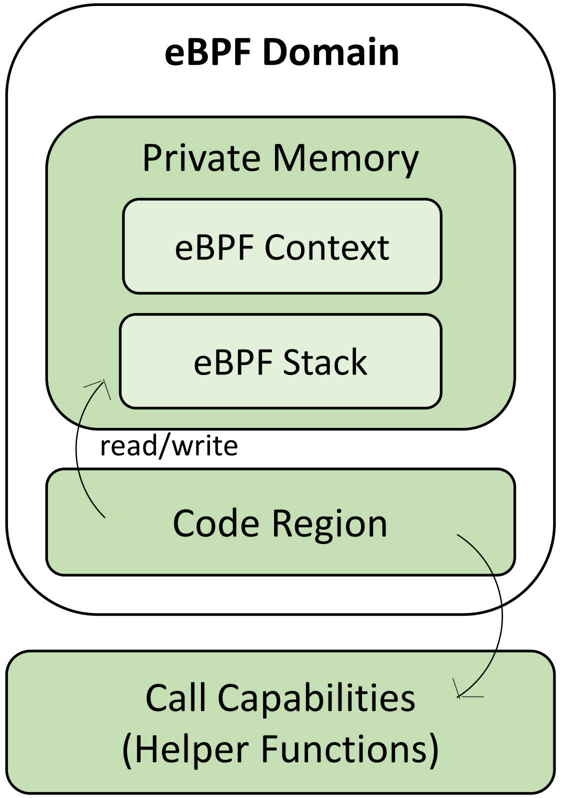
- The capabilities are stored in a hash table to provide O(1) search time.

Redirect Control Transfers to Trampoline



Recap: SandBPF

① Address Masking (Memory Safety)



② Redirect Calls to Trampoline (Control Flow Integrity)

Content Overview

- Problem
- Related Work
- Our Solution
- Evaluation Results
- Future Work
- Conclusion

Breakdown of SandBPF Overhead

Table 1: The number of checks inserted and executed by SandBPF in our example programs.

Program	Injected		Executed	
	Address Masking	Trampoline	Address Masking	Trampoline
XDP	2	1	2	1
Socket Filter	12	10	12	10
Katran	641	42	35-37	1-2

Breakdown of SandBPF Overhead

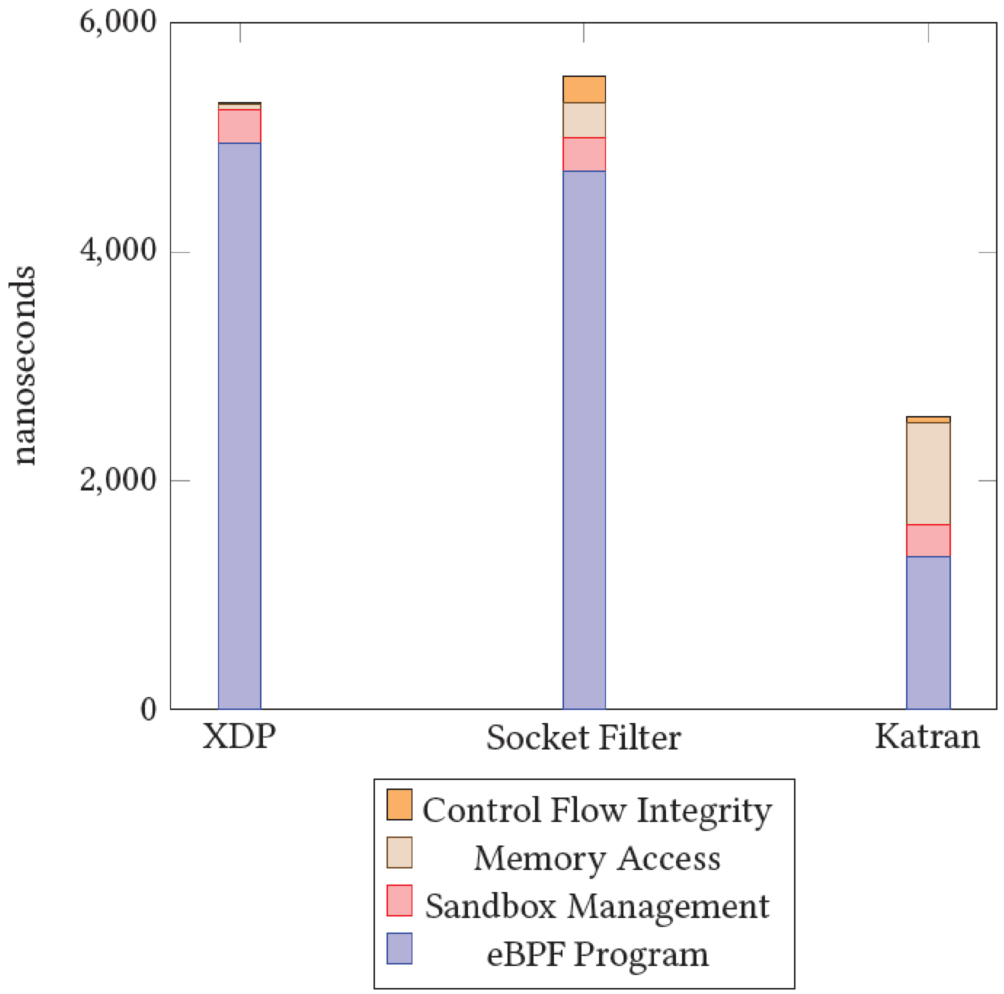


Figure 5: Breakdown of the overhead introduced by SandBPF

Sandbox Management
Constant overhead upon each eBPF invocation.

Memory Access & Control Flow Integrity
Overhead scales with the complexity of eBPF programs.

Macro-benchmark

Table 3: Macrobenchmark measuring web server performance of 20-1000 concurrent connections.

Test	XDP Program Vanilla	XDP Program SandBPF	Socket Filter Program Vanilla	Socket Filter Program SandBPF
Throughput (request/s)				
Apache 20	64,591	64,526 (**0%**)	62,269	60,089 (**4%**)
Apache 100	86,190	79,638 (**8%**)	87,576	83,751 (**4%**)
Apache 200	85,614	81,749 (**5%**)	85,671	83,381 (**3%**)
Apache 500	68,329	63,691 (**7%**)	72,399	67,177 (**7%**)
Apache 1000	66,472	62,508 (**6%**)	71,453	66,171 (**7%**)
Nginx 20	49,170	45,731 (**7%**)	50,095	45,331 (**10%**)
Nginx 100	58,613	54,494 (**7%**)	58,797	54,029 (**8%**)
Nginx 200	56,581	53,051 (**6%**)	58,447	53,869 (**8%**)
Nginx 500	50,495	47,699 (**6%**)	54,537	50,822 (**7%**)
Nginx 1000	46,302	44,977 (**3%**)	50,651	47,734 (**6%**)

SandBPF incurs no more than 10% overhead in terms of network throughput.

Security Evaluation

- We tested SandBPF against CVE-2021-3490 and CVE-2021-4204:
  - Both results in arbitrary read/write in the kernel.
  - Both can be exploited to escalate privileges.

SandBPF successfully prevents both vulnerabilities.

Content Overview

- Problem
- Related Work
- Our Solution
- Evaluation Results
- Future Work
- Conclusion

Future Work

- Optimizing Performance
  - We see ≤10% overhead on network throughput.
  - This is without any optimization to SandBPF (e.g., asynchrony).
  - We see this as a reasonable baseline for future work to improve performance.
- Simplify the eBPF verifier
  - Remove some constraints on eBPF program expressiveness.

Content Overview

- Problem
- Related Work
- Our Solution
- Evaluation Results
- Future Work
- Conclusion

Unprivilged eBPF for Better Kernel Extensibility

- Dynamic sandboxing is a viable approach to enforce security properties in eBPF programs, *complementary* to the current static mechanism employed by the eBPF verifier.

- SandBPF enhances runtime safety of the kernel to justify the (currently dismissed) support of unprivileged eBPF programs.

Thank you! Any Questions?

Soo Yee Lim (sooyee@cs.ubc.ca)

Joint work with Xueyuan Han and Thomas Pasquier