Linux System Security and Safety in extended Berkeley Packet Filter (eBPF)

Gbadamosi Bolaji, Tobias Pulls, Per Hurtig, Anna Brunstrom,

Toke Høiland-Jørgensen, Simo Sorce

What is eBPF

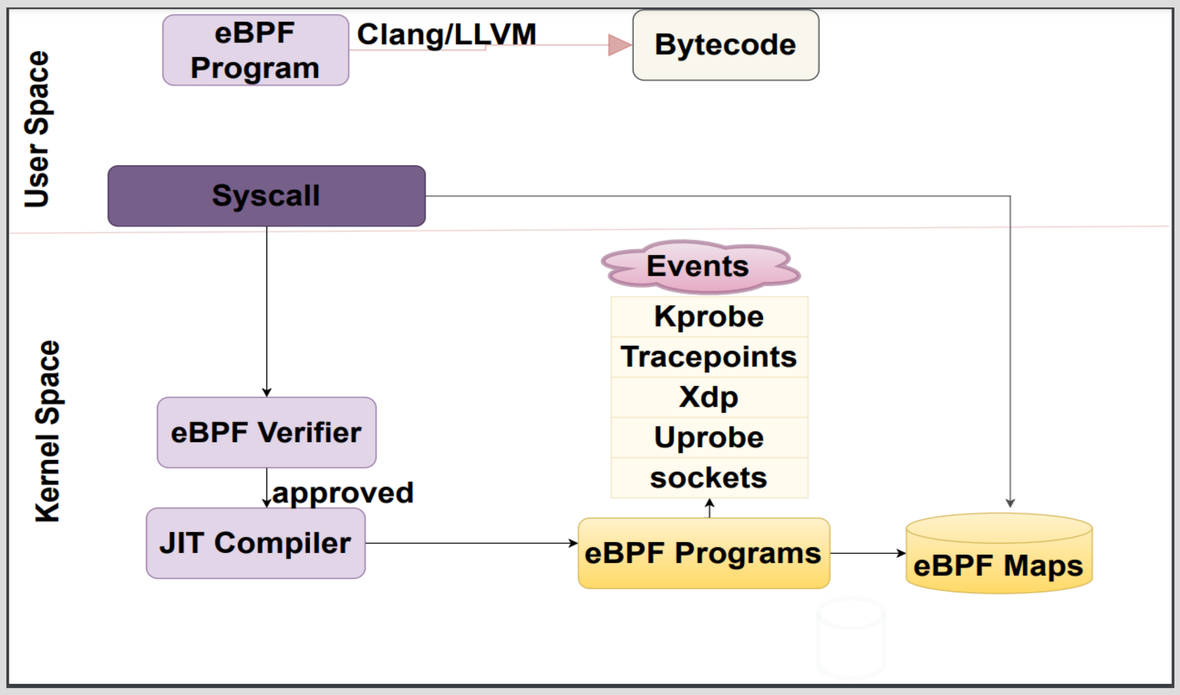
- The extended Berkeley Packet Filter (eBPF) is a virtual machine that allows the kernel to run custom programs in response to specific events such as network packets, system calls, and tracepoints [1].
- By using eBPF, code can be injected from application space into the kernel without the need to recompile or install additional kernel modules during runtime [1].
- eBPF can be used by executing a program that is compiled into instructions that are executed in kernel space [2].
- eBPF has a virtual CPU, which consists of a 512 byte memory for storing variables and 11 registers. The 11th register is the frame pointer [2].
- Code execution at the kernel level is a very powerful tool that can be abused if it falls into the hands of someone with malicious intentions.

References

[1] Cyril Renaud Cassagnes, Lucian Trestioreanu, Clément Joly, and Radu State. The rise of eBPF for non-intrusive performance monitoring. NOMS 2020.

[2] Cilium Authors. BPF and XDP Reference Guide. Revision abdf7ac6 2023 https://docs.cilium.io/en/latest/bpf/

- To ensure that eBPF programs are safe for the kernel, it uses a *verifier*.
- The eBPF verifier only allows a program to run if it can prove that it will not crash the kernel.



Adapted from eBPF Documentation. What is eBPF?, eBPF summit 2022 https://ebpf.io/what-is-ebpf/

The eBPF Verifier

The eBPF verifier does a static analysis to check the validity of each eBPF program before it is loaded into the kernel. This to ensure that it does not violate security requirements and thus compromise the kernel. It simulates execution of all parts of the program in the virtual machine.

The verifier performs the following checks:

1) **Ensures that the program terminates** by building a *control flow graph* to detect and reject programs with unbounded loops or backward jumps.
2) **Symbolic execution for each instruction**, recording the state changes to the registers and the stack. This is to ensure memory safety to prevent out of bounds memory access.
3) **Value tracking** by tracking the state of each value stored in the register and each bit in the binary number to prevent invalid memory usage.
4) **Verifier tracks spill/fill of registers** into the stack, which may otherwise result in too many states being inspected.
5) **Path pruning** is used to solve the problem of large numbers of states, resulting in a dramatic reduction in verification time and allowing very complicated programs with many branches that would not normally pass verification to pass.
6) **After successful verification,** the BPF instructions and registers are mapped to the actual architectural instructions and registers. The BPF assembly is converted to the actual machine code using a JIT compiler, which gives BPF programs their native execution speed with very low or no overhead.

Research Questions

The overarching research questions for this project are:

- When is eBPF safe or unsafe to use?
- How can we increase the trust in the eBPF verifier?

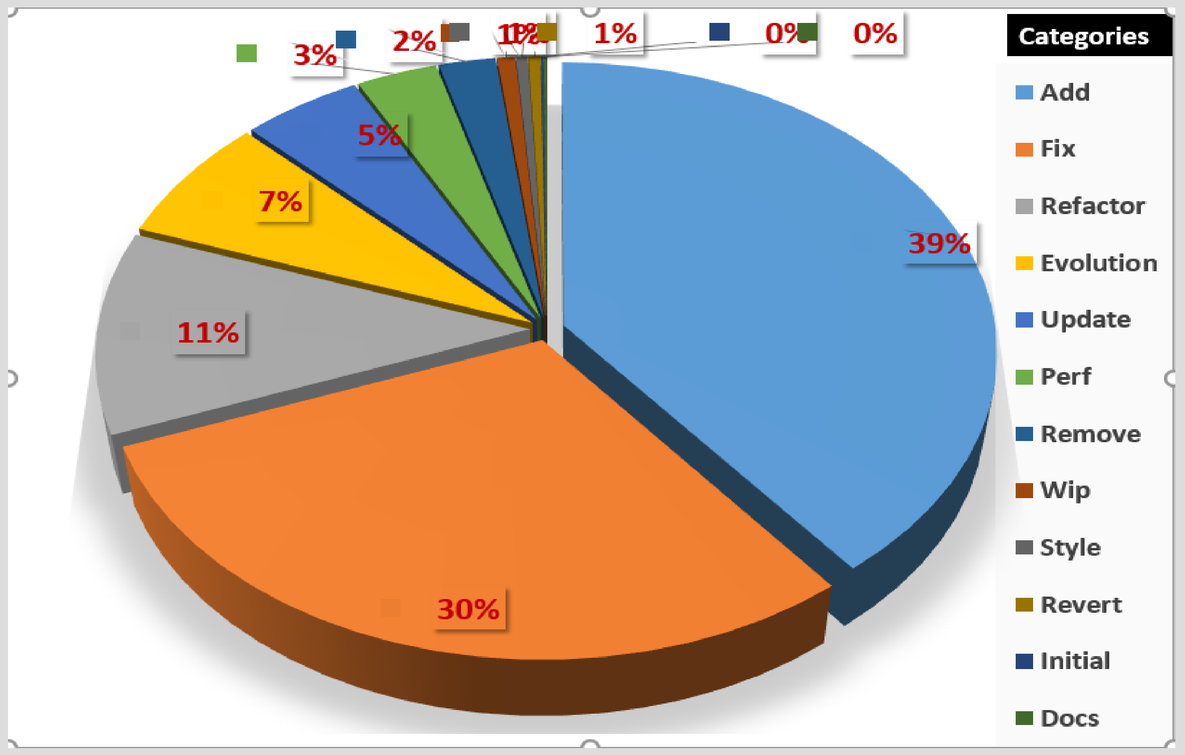
Ongoing Work

Currently collaborating with an external party on:

- How the eBPF runtime in the Linux kernel has evolved over time.
- How various components of the eBPF runtime works.
- Dependable safety guarantees of eBPF for users.
- Basic operation of the eBPF verifier to determine program safety.

A Moving Target

The percentage of categorized verifier commits from December 2014 in Linux 3.18 until November 2022 in Linux 6.0.6. In total 723 commits. Most commits are either added features or fixes accounting for 69.65% of the commits, the large number of fixes resulted in more lines of code being added to the verifier, which increased its complexity.



Contacts

- Gbadamosi Bolaji, bolaji.gbadamosi@kau.se
- Tobias Pulls, tobias.pulls@kau.se
- Per Hurtig, per.hurtig@kau.se
- Anna Brunstrom, anna.brunstrom@kau.se
- Toke Høiland-Jørgensen, toke@redhat.com
- Simo Sorce, simo@redhat.com

Acknowledgments

The project is a collaboration between Karlstad University and Red Hat Research. For more information and additional resources, see the project page.

Project Page