Revealing the Unstable Foundations of eBPF-Based Kernel Extensions

Shawn Zhong, Jing Liu, Andrea Arpaci-Dusseau, Remzi Arpaci-Dusseau

WISCONSIN UNIVERSITY OF WISCONSIN–MADISON Microsoft

depsurf.github.io

Introduction

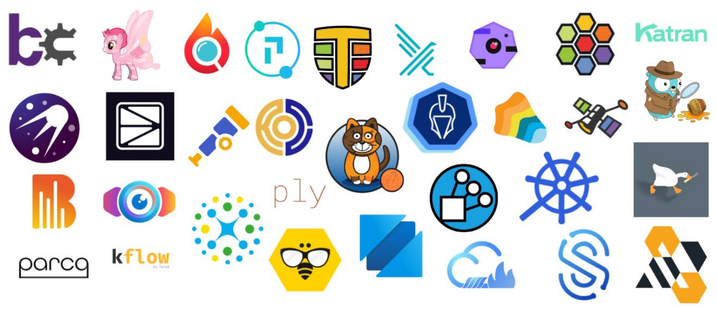
eBPF: Framework to extend Linux kernel functionality

- 🚀 Run custom programs in kernel
- 🪝 Triggered by hooks (e.g., on function entry point)
- 🔍 Read and traverse internal kernel data structures

Use cases

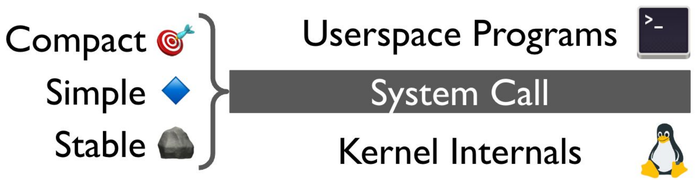
- 👀 Observability: Trace kernel functions
- 🔒 Security: Enforce security policies
- 🌐 Network: Process, filter, redirect packets

Popularity

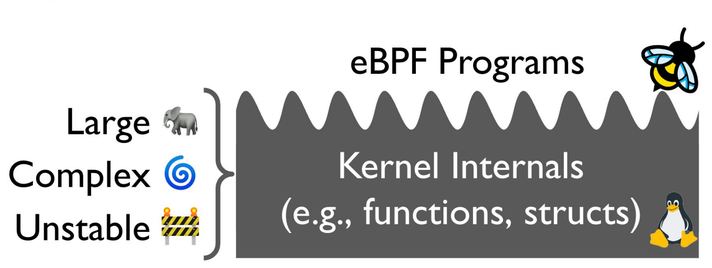


Portability Challenges

- Userspace programs depend on system call layer



- eBPF programs depend on kernel internals



Impact

- ⚓ eBPF programs are fundamentally unportable
- 💥 They frequently break on different kernels
- 🤔 People unclear if it will work on another kernel

Dependency Mismatch

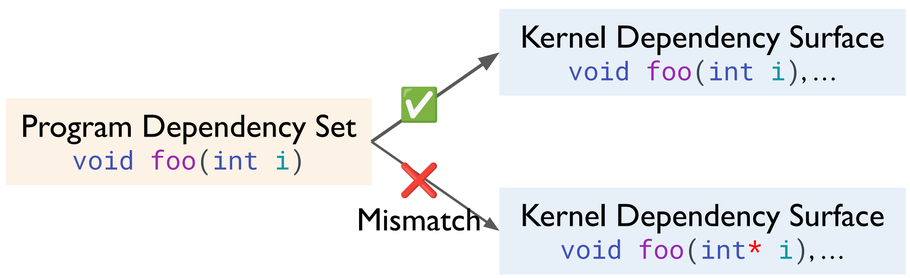
Program Dependency Set

A set of dependencies used by an eBPF program

Kernel Dependency Surface

All dependencies exposed by a kernel

Dependency Mismatch



Cause: Unstable kernel dependency surface

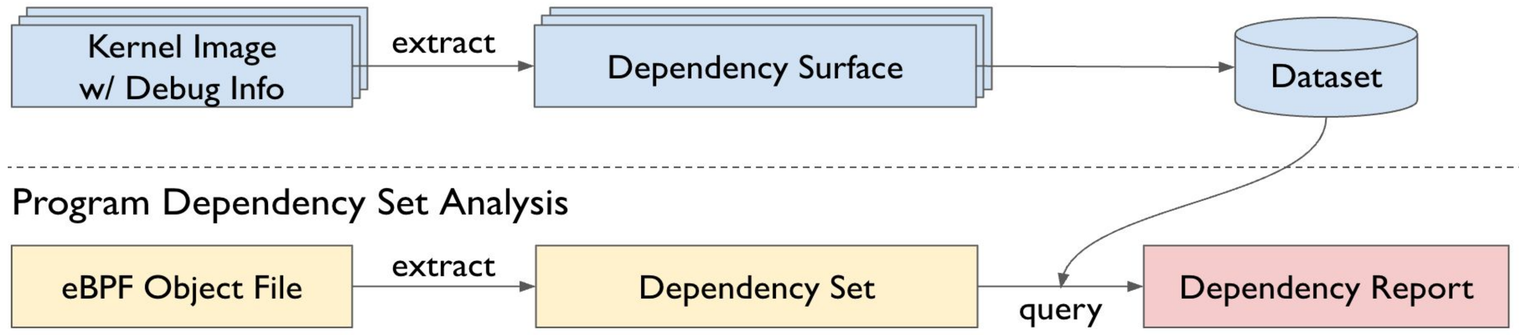
Consequences

- Fail to compile, load, or attach ⇒ Explicit error 🚨
- Stray read ⇒ Incorrect garbage results 🗑️
- Missing invocation ⇒ Incomplete results ⚠️

DepSurf

DepSurf: A tool to detect, diagnose, and analyze dependency mismatches

Kernel Dependency Surface Analysis



Kernel Dependency Surface Analysis

Kernel Source Code 📝

- New features (e.g., folio)
- Depreciations (e.g., single-queue bio)
- Perf. optimization
- Bug fixes

Kernel Configuration ⚙️

- Arch-specific definitions (e.g., register/syscall)
- Features (e.g., NUMA)
- Parameters (e.g., timer)
- Set by OS distro

Kernel Compilation 📦

- Function Optimizations (e.g., inline)
- Driven by compiler
- Opaque to developers

Dataset: 25 kernel images, 17 versions, 8 years, 5 architectures, 5 flavors, 14 compiler versions

Kernel Source Code 📝: Every 2 years …

- Function: 24% added, 10% removed ⇒ Explicit error 🚨
  E.g., `mark_page_accessed` converted to `folio_mark_accessed`
- Function: 6% changed ⇒ Incorrect results 🗑️
  E.g., `vfs_rename(/* 6 params */)` became `vfs_rename(struct renamedata *)`
- Struct: 24% added, 4% removed, 18% changed ⇒ Explicit error / Incorrect results 🚨🗑️
  E.g., in `task_struct`, field `long state` changed to `unsigned int __state`
- Tracepoint: 39% added, 5% removed, 16% changed ⇒ Explicit error / Incorrect results 🚨🗑️
  E.g., `kmem_alloc` removed and `kmem_alloc_node` renamed to it

Kernel Configuration ⚙️: See paper

Kernel Compilation 📦: Within a kernel image …

- 36% full inline (function copied to all call sites and symbol disappeared) ⇒ Explicit error 🚨
- 11% selective inline (function inlined at some call sites but not all) ⇒ Incomplete results ⚠️

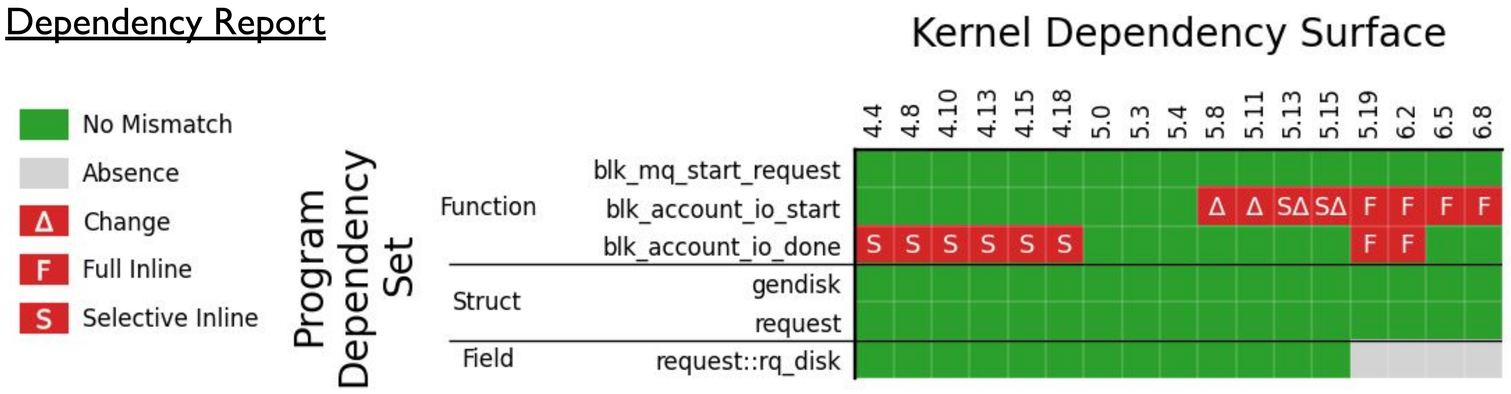
```
fs/sync.c:       int vfs_fsync() { /* logic */ }    // func definition
                long sys_fsync() { vfs_fsync(); }   // inlined          ❌ NOT traced
fs/aio.c: extern int vfs_fsync();                   // func declaration
          void aio_fsync_work() { vfs_fsync(); }    // not inlined      ✅ Traced
```

Conclusion: Kernel dependency surface is inherently unstable

More results in paper!

Program Dependency Set Analysis

Dependency Report



Results: 53 programs from BCC and Tracee ⇒ All depends on some unstable kernel internals

- 43 programs depend on structs: 22 affected by absence ⇒ Explicit error 🚨
- 25 programs depend on functions: 14 affected by selective inline ⇒ Incomplete results ⚠️
- 25 programs depend on tracepoints: 18 affected by change ⇒ 🚨🗑️

Conclusion: Dependency mismatches are widespread in eBPF programs