A Practical, Lightweight, and Flexible Confinement Framework in eBPF

William P. Findlay

MCS Thesis Defence

August 30th, 2021

Co-Supervisors: **Anil Somayaji** and **David Barrera**

Carleton University

Why Are We Here?

To make process and container confinement **better**.

- What does **better** mean?
  - Simple policy, suited to ad-hoc use
  - Application- and container-specific
  - The need for a new kernel primitive
    - Must be **adoptable**
- How can we do this?
  - **eBPF** enables the development of such a kernel primitive
  - **Safe**, **flexible**, **kernel-agnostic**
  - Program the kernel at runtime

Defining Confinement

The Threat Model

- Consider a **remote adversary**:
  - Has already achieved **code execution** in some local process
  - This process may be **root-privileged** under Unix DAC (i.e. sysadmin)
- As **defenders**, we want to:
  - **Confine** the process such that its **set of allowed actions** is the **minimal subset required to function**
  - Minimize any potential damage to resources **outside** of our security boundary

Existing Mechanisms

How are **containers** **confined on Linux**?

- **Process confinement**
  - Virtual memory + protection bits
  - Unix DAC
  - POSIX capabilities
  - Seccomp-bpf†
  - LSMs + MAC (SELinux, AppArmor)
- **Container confinement**
  - Combination of **namespaces**, **cgroups**, and **seccomp-bpf**
  - Optionally, a **MAC LSM** like AppArmor or SELinux
  - **Fail-open** approach to security

†This "BPF" is **not** the same thing as eBPF.

Identifying the Problem

What is wrong with **the status quo**?

How can we **do better**?

- Existing confinement primitives are **ill-suited** to containers
    - Complexity + interdependence
    - Lack of container semantics
    - Container runtimes would rather "just work."
- We need **new kernel code** to fix this
    - Tackling the root of the problem
    - Trace the lifecycle of a container
    - Encode container semantics into policy enforcement

How eBPF Can Help

Safely extend the kernel using **programs** and **maps**.

- **Programs** run code on events
  - Verified for safety
  - JIT compiled for performance
- **Maps** keep track of state
  - Can be accessed from a eBPF program or from userspace
- How does this help us?
  - What if we used eBPF to **enforce policy**?
  - Introduce **new confinement primitives** into the kernel

Contributions

Contribution Highlights

1. A **new architecture** for enforcing confinement policy using eBPF
    - Instrument system state and enforce policy with eBPF
    - Fine-grained enforcement and the introduction of new confinement semantics
    - Implicit security and adoptability advantages provided by eBPF (safety + flexibility)
2. Two novel **confinement implementations**
    - **BPFBox** focuses on **application sandboxing**
    - **BPFContain** focuses on **container security**
3. **Performance evaluation** + **informal security analysis**
    - Comparable overhead to AppArmor (better in worst-case, slightly worse in average-case)
    - Potential to be as or more secure than traditional LSMs (smaller code base + eBPF verifier)

An Architecture for eBPF-Based Confinement

BPFBox and BPFContain follow the same basic architecture:

1. A **privileged daemon** parses and encodes policy into **eBPF maps**
2. Instrument events in **userspace** and **kernelspace** using **eBPF programs**
3. These **programs** store information about **system state** in **eBPF maps**
4. Enforce policy using **eBPF LSM programs**
    - **Programs** query **state** + **policy** from **maps** to arrive at **policy decisions**
5. The **privileged daemon** logs security events as they occur

Major Implementation Differences

Despite following a similar basic architecture, **implementation details** differ significantly between **BPFBox** and **BPFContain**.

- BPFBox is implemented in **Python3** and **bcc**†
  - Large dependency overhead, LLVM compiler toolchain required at runtime
- BPFContain is implemented in **Rust** using **libbpf-rs**† and **BPF CO-RE**
  - Embed eBPF bytecode **directly into the ELF binary**
  - **Load-time relocation logic** → A single BPFContain binary works on **any** supported kernel
- BPFContain instruments **more LSM hooks** and deals in **container semantics**
  - Namespace + container membership is considered when making a policy decision
  - Prohibit containers from switching namespaces at runtime

†I am a major contributor to both bcc and libbpf-rs.

Fine-Grained Policies in BPFBox

- Policies are written in a custom domain-specific language
    - **Rule blocks** and **decorators**
- “Process tainting” to **simplify policies** and **improve security**
    - Processes spawn untainted
    - Matching a “taint rule” taints the process
    - Focus on enforcing policy after the application’s **setup phase**
- Fine-grained policy context using **kprobes** and **uprobes**
    - Instrument on function calls in **kernelspace** and **userspace**
    - Augment policy with information about control flow
    - No existing confinement implementation can do this

Container-Specific Policies in BPFContain

- Policies are written in existing **data serialization languages**
  - YAML, TOML, and JSON are currently supported
  - Modular design means adding support for new languages is trivial
- Keep the same notion of “tainting” from BPFBox
  - But apply it to the **whole container** rather than each process
- Container-specific policy defaults
  - Trace container execution using **programs** and **maps**
  - Use this information to define a **security boundary around the container**
  - **Grant access** to resources **within** the container, **deny access** to global resources
  - **Deny access** to privileged operations that impact **global system state**

How is This Work Novel?

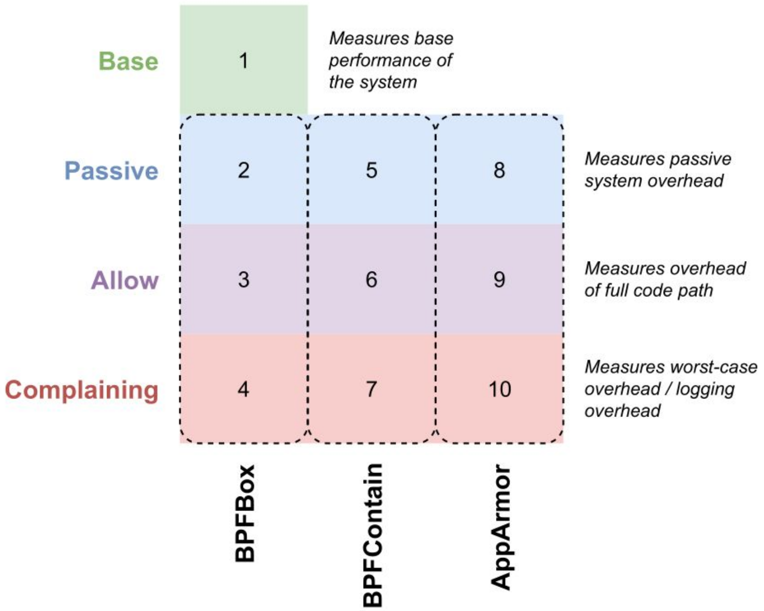
- Existing eBPF-based security focuses on **network policy** + **observability**
  - BPFBox and BPFContain enforce **local confinement policy**
  - Policy at the **application and container level**
- Container security solutions recombine existing primitives in new ways
  - Policy generation, higher-level policy languages, etc.
  - This ignores the root of the problem: **insufficient confinement primitives**
  - BPFContain introduces **new container policy semantics** in the kernel
- BPFBox and BPFContain are **not traditional LSMs**
  - Fully stackable, can be dynamically loaded into the kernel
  - Do not require recompiling the kernel or even rebooting the system

Evaluation

Performance Evaluation Methodology

Measured performance overhead of BPFBox and BPFContain, compared with AppArmor.

- **OSBench**
  - Micro-benchmarks, measuring overhead of Linux system calls (file I/O, process creation, etc.)
- **Kernel Compilation**
  - Measures the time it takes to compile the Linux kernel (heavy I/O and CPU load)
- **Apache Web Server**
  - Measures requests handled per second by the Apache web server



Performance Highlights

- **BPFBox** and **BPFContain** win out in the **Complaining** case
  - 6.7% and 8.7% overhead respectively
  - 20.5% for AppArmor
  - eBPF ring buffer is more efficient than the kernel's audit framework
- **AppArmor** wins in the **Passive** and **Allow** cases
  - 3.7% and 4.6% vs 1.3% in the **Passive** case
  - 4.7% and 7.6% vs 2.4% in the **Allow** case
  - These are research prototypes (room for future optimization)
  - eBPF LSM programs could also be getting faster in the future (K.P. Singh, 2020)
- **AppArmor** had an unfair advantage in the **Apache Web Server** tests
  - In Linux 5.X kernels, AppArmor network policy is broken
  - Plans to redo benchmarks with a patched kernel before final thesis

Geometric Means of All Results (HIB)

Test Case	System	Geom. Mean	Overhead (%)
Base		6.238	—
Passive	BPFBox	6.007	3.70%
	BPFContain	5.951	4.60%
	AppArmor	6.158	1.28%
Allow	BPFBox	5.944	4.71%
	BPFContain	5.763	7.61%
	AppArmor	6.086	2.35%
Complaining	BPFBox	5.823	6.65%
	BPFContain	5.693	8.74%
	AppArmor	4.962	20.46%

Security Highlights

- **Complete mediation** over security events
  - LSM hooks + some kernel probes
  - Complete mediation relies on **LSM hook placement**
  - All existing LSMs (SELinux, AppArmor, etc.) already rely on LSM to be secure
- **Tamper resistance** of the enforcement engine
  - BPFBox and BPFContain **protect themselves** (recall our threat model)
  - Forbid the bpf(2) system call and any other operations which could load kernel code
- **Correctness** of the implementation
  - Needs to be verified (there could be bugs!)
  - Likely a mix of testing + formal methods

Security Highlights (Continued)

BPFContain has the potential to be **as if not more secure** than **existing LSMs**.

- Two main reasons for this:
    1. **The eBPF verifier** (less potential for bugs/security vulnerabilities)
    2. **A significantly smaller code base**† than existing LSMs
        - BPFContain: 1,719 KSLOC
        - AppArmor: 12,608 KSLOC
        - SELinux: 20,876 KSLOC

- Compared with existing LSMs:
    - Default-deny on most privileged operations
    - Focus on simple confinement, rather than complex access control schemes (RBAC, etc.)
    - Simple policies (emergent from the other two properties)

†Statistics generated using SLOCCOUNT by David A. Wheeler.

Wrapping Up

Contribution Highlights

Revisited

1. A **new architecture** for enforcing confinement policy using eBPF
2. Two novel **confinement implementations** using this architecture
3. **Performance evaluation** + **informal security analysis**

Limitations

BPFBox and BPFContain are **neither perfect nor complete**.

- BPFBox never achieved a full implementation
  - BPFContain **implements a superset** of BPFBox
  - Think of BPFBox as an **early iteration of BPFContain**
- Semantic issues in policy expression
  - Pathname to inode translation
  - Device to major and minor number translation
- Fixed-size policy maps
  - Dynamically sized map support is coming thanks to sleepable eBPF

Limitations

BPFBox and BPFContain are **neither perfect nor complete**.

- Room for performance optimization
  - BPFContain is still a research prototype, with complex code paths
  - No optimization effort so far
- Security guarantees must be formally established
  - Formal verification should be realistic due to BPFContain's small code base

Future Work

Where do we go from here?

- Perform a **user study**
  - How does BPFContain's policy model match user expectations?
- Fine-grained **network policy**
  - Filter network traffic by IP address and port
- Full **Docker** and **OCI integration**
  - Confine Docker containers automatically, according to their manifest
- **Policy generation**
  - Generate policy automatically, using eBPF to capture events

Conclusion

My thesis uses eBPF to implement **novel confinement primitives** in the Linux kernel.

This work has applications in **application sandboxing** and **container security**.

Future iterations on BPFContain could improve its **performance** and **security**.
Integration with container runtimes could streamline policy enforcement.

Thank you — Please ask questions!

Backup Slides

eBPF 101

What is eBPF[1]?

- A relatively young technology
  - Alexei Starovoitov & Daniel Borkmann, circa 2014
- Make the kernel **programmable** from **userspace**
  - Attach minimal RISC programs to system events
- Run custom, **event-based** kernel code **in production**
  - Verified for **safety**
  - JIT-compiled for **performance**

[1]Also just called BPF these days.

eBPF 101

eBPF Programs

- **BPF Programs** are expressed in a minimal RISC instruction set
- Loaded into the kernel by a privileged userspace process
- Once loaded, a program can be attached to an event
- When the event fires, the JIT engine translates and runs the program in the native instruction set

eBPF 101

eBPF Programs

- There are many different **program types**

- Each type serves a specific purpose
  - Over 33 program types in Linux 5.13

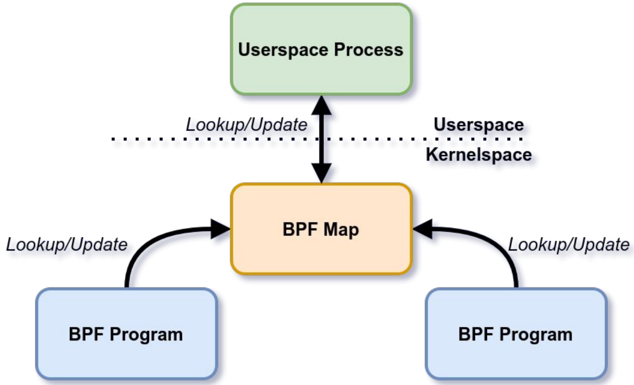
- Each program type has access to a specific set of helpers

- Some common examples:
  - **Kprobes** hook kernel functions
  - **Uprobes** hook user functions
  - **Tracepoints** hook stable tracing interfaces

eBPF 101

eBPF Maps

- Since eBPF programs are **event-based**, they are inherently **stateless**
- **BPF Maps** provide a means of adding **state** to a set of programs



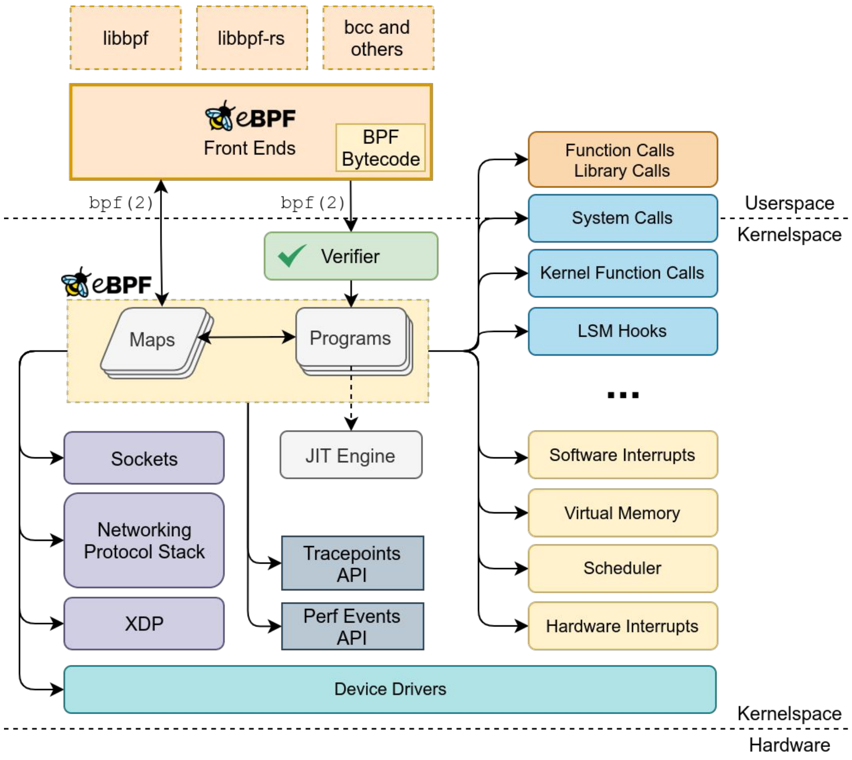
eBPF 101

The eBPF Verifier

- eBPF programs are **limited** in what they can do
  - No unbounded loops
  - No write access to kernel memory
  - No execution of arbitrary kernel code
  - No out-of-bounds memory access
- The goal is to make programs **verifiably safe** to run
- An in-kernel verifier checks programs at load time
  - When program safety can't be proved, the program is rejected

eBPF 101

The Big Picture



eBPF 101

eBPF vs Kernel Modules

- eBPF is **kernel-native**
  - No need to recompile a custom kernel
  - No need to audit third-party kernel code
  - CO-RE enables **one** binary to be distributed and used across all systems
- eBPF is **verified for safety**
  - Not 100% fool-proof, but far less likely to crash a system or introduce a security vulnerability
- eBPF is **flexible**
  - Trace across userspace and kernelspace
  - Aggregate data in the kernel and pass control back to userspace when needed

eBPF 101

eBPF for Security

- eBPF is already being used for security in industry
  - Cilium, Tracee, and Falco
  - Custom LSM programs at Google
  - Misc. programs deployed at Facebook, Netflix, and even Apple
- But the focus is almost entirely on **observability** or **network policy**
- In this thesis, we examine how eBPF can also be used for **local policy enforcement**

Confining Containers

Existing confinement solutions are unsuitable for containers.

- Containers runtimes rely on **existing** Linux **confinement primitives**
- These primitives pre-date the invention of containers
- They were designed to solve **different problems**
    - System-wide MAC policy
    - No notion of container semantics

Confining Containers

Existing confinement solutions are unsuitable for containers.

Image source: https://konkolcustom.com

Confining Containers

What we **have**:

- Complex entanglements of policy, using multiple confinement primitives
- Security mechanisms which were not designed to lock down containers
- Overly-generalized, fail-open policies, designed to "just work" instead of provide real security

Confining Containers

What we **want**:

- Define a **clear protection boundary** around the container
    - Security mechanism should be designed with containers in mind
- Ensure that no access is granted over this boundary
- Provide a **simple policy language** for defining **exceptions to this boundary**

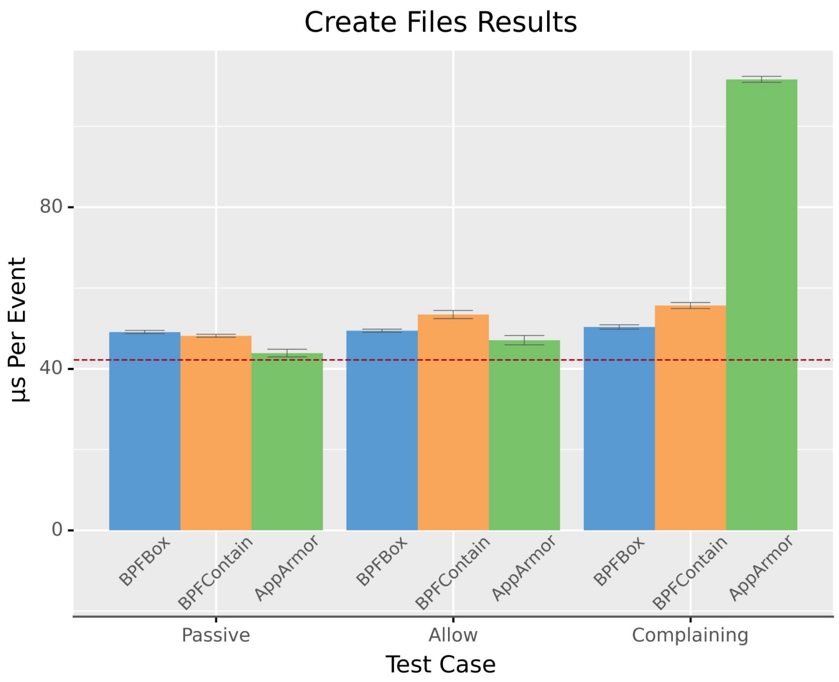
Performance Evaluation

Improving Test Accuracy

- Disable SMT hyperthreading
- Disable CPU turbo boost
- Set CPU frequency scaling governor to “performance”
- Disable ASLR
- Run each test 11 times and discard the first run

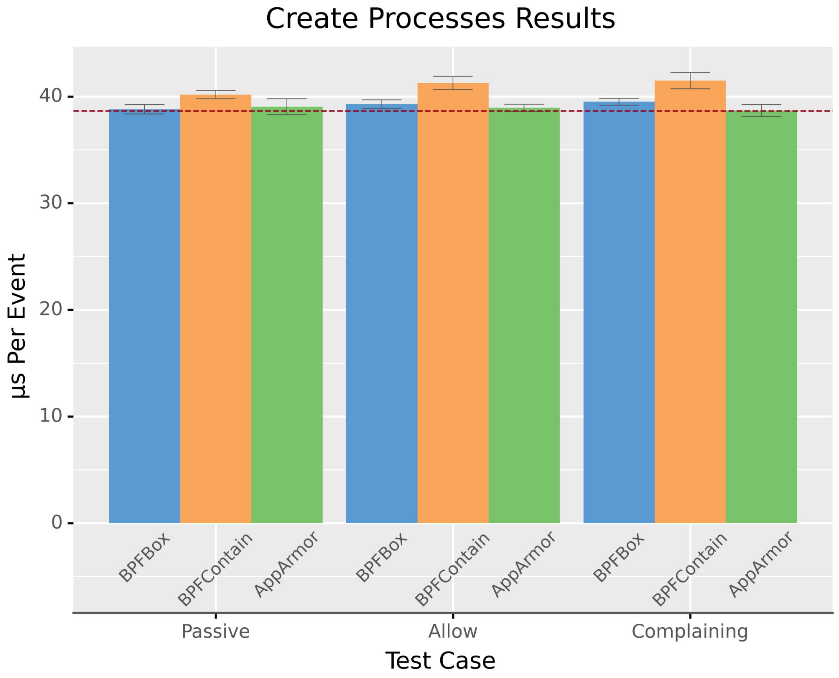
OSBench

File Creation + Deletion
Lower Times are Better



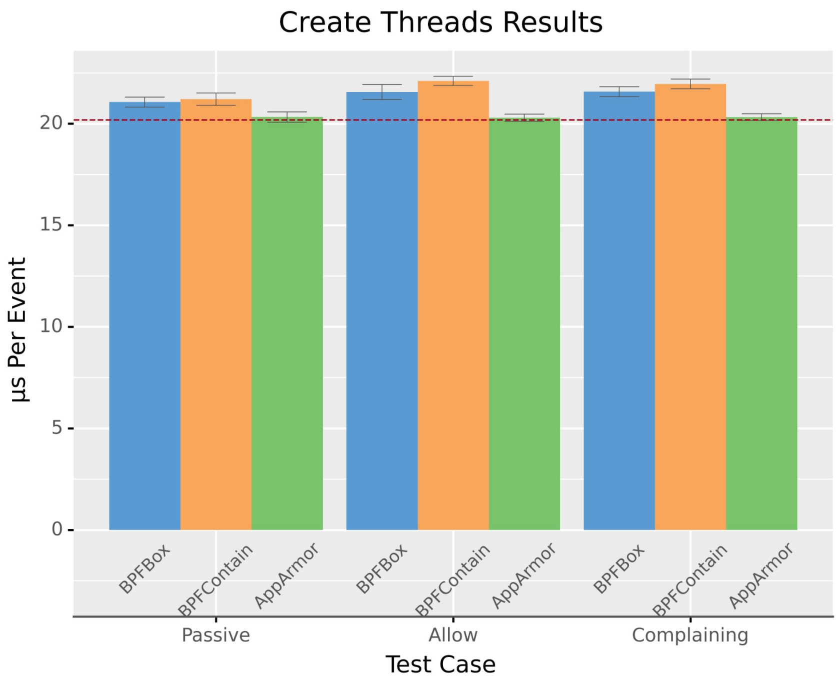
OSBench

Process Creation
Lower Times are Better



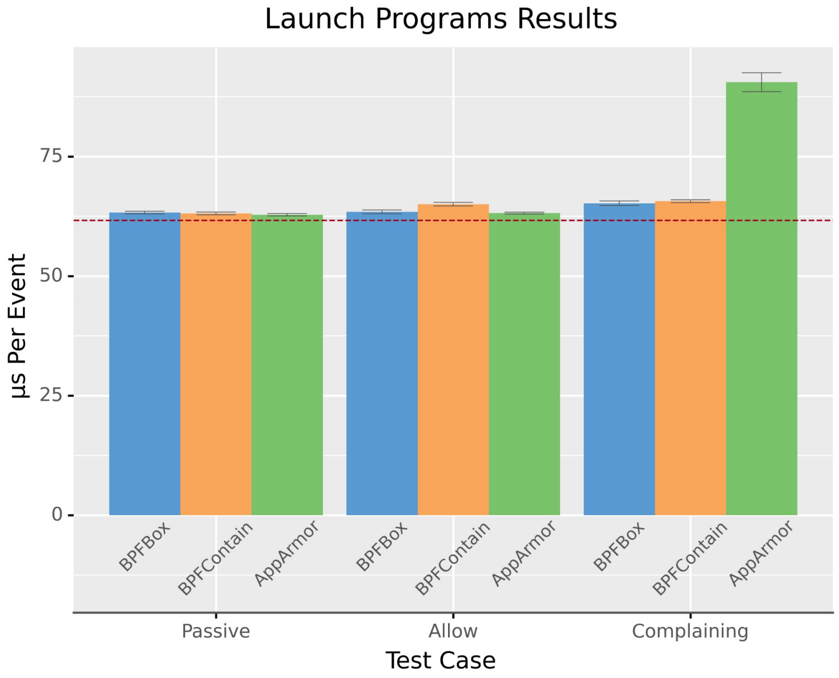
OSBench

Thread Creation
Lower Times are Better



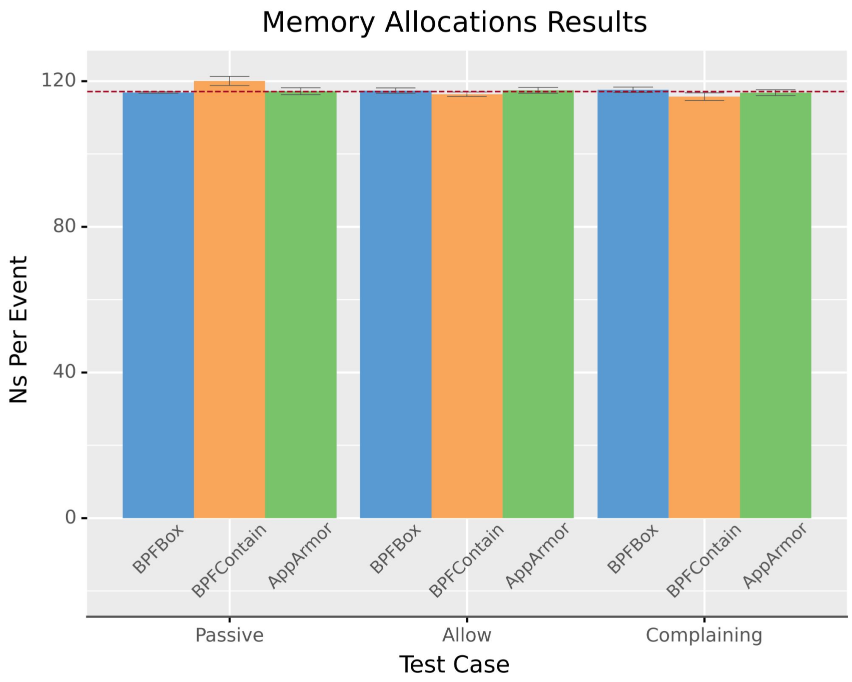
OSBench

Program Execution
Lower Times are Better



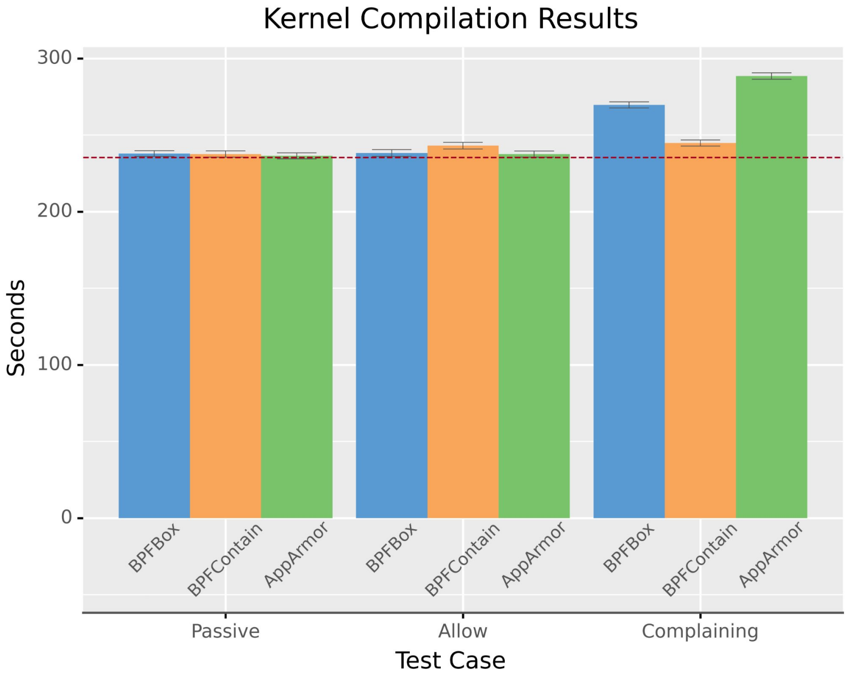
OSBench

Memory Allocation
Lower Times are Better



Kernel Compilation

Lower Times are Better



Apache Web Server

Higher Req/S is Better

*AppArmor does not correctly enforce network policy in 5.x kernels