A Comparative Analysis of Linux Mandatory Access Control Policy Enforcement Mechanisms

Brennon Brimhall
bbrimha1@jhu.edu
Johns Hopkins University
Baltimore, Maryland, USA

Justin Garrard
jgarrar1@jhu.edu
Johns Hopkins University
Baltimore, Maryland, USA

Christopher De La Garza
cdelaga1@jhu.edu
Johns Hopkins University
Baltimore, Maryland, USA

Joel Coffman[*]
joel.coffman@jhu.edu
Johns Hopkins University
Baltimore, Maryland, USA

ABSTRACT

Unix—and by extension, Linux—traditionally uses a discretionary access control (DAC) paradigm. DAC mechanisms are decentralized by design, which makes it difficult to audit the security of a computer system. Furthermore, Unix systems have the concept of a root user who can bypass any DAC policies in place. These issues led to the development of mandatory access control (MAC) mechanisms, such as AppArmor, Security-Enhanced Linux (SELinux), and eBPF.

We compare and contrast the performance differences between two popular MAC mechanisms for the Linux kernel: SELinux and Berkeley Packet Filter (BPF)/kernel runtime security implementation (KRSI). We demonstrate that BPF policies offer superior performance, have greater expressive power, and are easier to implement than comparable SELinux policies. Our results suggest that BPF/KRSI is the leading MAC mechanism for Linux systems.

CCS CONCEPTS

• **Security and privacy → Operating systems security**; **File system security**.

KEYWORDS

mandatory access control (MAC), Linux Security Modules (LSM), Security-Enhanced Linux (SELinux), extended Berkeley Packet Filter (eBPF), kernel runtime security implementation (KRSI)

ACM Reference Format:
Brennon Brimhall, Justin Garrard, Christopher De La Garza, and Joel Coffman. 2023. A Comparative Analysis of Linux Mandatory Access Control Policy Enforcement Mechanisms. In *European Workshop on System Security (EUROSEC '23), May 8, 2023, Rome, Italy.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3578357.3589454

[*]Also with United States Air Force Academy, Department of Computer and Cyber Sciences.

1 INTRODUCTION

The POSIX API, a standard for Unix-based operating systems, defines a DAC paradigm. Users own files and processes and can allow other users to interact with them. This model works well for multiuser systems where multiple users must coexist. However, these users must also understand the POSIX file system security API and the security implications of their actions. For example, an unprivileged user who owns a particular file may inadvertently change the file mode to allow other users to write and read it. The prevalence of incorrect file permissions modes is one reason OpenSSH checks the file mode of private key files; if other users can read a user's private key file, they could impersonate the user on the network.

MAC policies are designed by the system administrator and complement DAC. When a MAC policy and a DAC policy disagree, the MAC policy supersedes the DAC policy. This mechanism gives system administrators fine-grained control over numerous operating system facilities. For example, SELinux allows the configuration of MAC policies that prevent the use of particular system calls even for privileged system users. Properly configured MAC policies can harden a machine against attackers by restricting the use of key operating system facilities. Adopting fine-grained MAC policies for system hardening is crucial for cloud computing environments, as providers must limit vulnerabilities and the blast radius of incidents to protect their systems from untrusted customer workloads.

One downside of MAC is that a policy check must occur every time a MAC-enabled action is taken by a user. This policy check imposes a performance degradation. SELinux policies can introduce an overhead of up to 33% with typical overheads of 10–20% for microbenchmarks [13]. This performance overhead is unacceptable for many computing applications, which led to the creation of the KRSI patch set. KRSI uses the Linux kernel's eBPF facilities to provide MAC mechanisms for system administrators [3].

Our work provides the following contributions:

- an overview of MAC implementations in Linux;
- a set of microbenchmarks to compare the performance of MAC implementations; and
- a qualitative comparison of SELinux and BPF/KRSI.

To our knowledge, no comparative analysis of these MAC mechanisms currently exists.

The remainder of this paper is organized as follows. Section 2 provides background information about DAC and MAC. Section 3 outlines our experimental methods, and Section 4 provides a performance and qualitative comparison of SELinux and BPF/KRSI. Section 5 highlights related work. Finally, we conclude in Section 6.

2 BACKGROUND

Linux offers DAC for users to manage access to files and MAC for system administrators to harden systems. This section summarizes both approaches including how the MAC API is provided by Linux Security Modules (LSM) [12]. We also discuss how MAC mechanisms like SELinux and BPF/KRSI use LSM.

2.1 Discretionary Access Control

Time-sharing operating systems were introduced in the late 1960s. As a multi-user utility, these systems required mechanisms to isolate users from each other. In Unix, the operating system represents all system resources as files. These files have permission modes set on them to control read, write, and execute privileges [16]. Unix file modes are an example of DAC. Under DAC, each file system object has an owner that can edit the permissions of the object. A special root privilege enables bypassing these permissions. Many popular operating system kernels are derived from Unix, such as Linux, FreeBSD, and Mach [1]. Each of these operating systems has the same basic permissions architecture. Unfortunately, DAC mechanisms have fundamental scalability issues:

- Because owners of file system objects have the ability to edit permissions, they can arbitrarily widen (i.e., grant) access at will. System administrators are unable to prevent such occurrences without changing the file system object owner to be an unprivileged system user. However, changing ownership mandates direct administrator involvement if there are any permissions issues on the system.
- Because root (i.e., superuser) privilege bypasses normal ownership checks, a root user has unfettered access to an operating system. There is no middle ground between an unprivileged user and a root user.

An early line of research attempted to address the second issue. FreeBSD jails [11] and Solaris zones [15] provide operating system virtualization. Operating system virtualization is distinct from conventional virtualization in that only a single kernel is running and hardware is not emulated by a hypervisor. These solutions scope root privilege, but do not restrict permissions in a fine-grained way.

2.2 Mandatory Access Control

The prior limitations motivate a second class of security mechanisms: MAC. Under MAC, security policies are set centrally and override any contradictory DAC policies. MAC policies even constrain users with root-level privileges. Thus, MAC lets a system administrator prevent certain actions from ever taking place.

Many MAC mechanisms have been implemented in Unix operating systems, such as the FreeBSD MAC framework [14] and SELinux [13]. There are a variety of MAC architectures; many rely on an administrator to apply labels to file system objects and system users, then provide a policy to allow or deny actions given those object and user labels. This framework allows for fine-grained decisions to be made. For example, SELinux can be configured to deny the ability for guest users to access networking resources.

In the last few years, a new MAC mechanism has emerged: eBPF. BPF was originally a kernel mechanism for filtering packets as they came through the TCP/IP stack. Over the years, BPF was extended to encompass more than packet filtering, as BPF can hook into all aspects of the Linux kernel and even userspace programs. While it is in its infancy, BPF is being used as another MAC mechanism in conjunction with the KRSI patch set, which allows BPF to enforce MAC security policies [3].

Unfortunately, implementing MAC introduces overhead regardless of MAC mechanism. A MAC decision must be computed for every action with a defined policy. Thus, a major consideration in deploying MAC is quantifying and identifying this overhead. Some performance critical workloads cannot tolerate even a 1% performance overhead, let alone the 10-20% of SELinux [13, 24].

LSM. Linux Security Modules (LSM) is a general security module framework that enables diverse security models without modifying the Linux kernel. It connects to the Linux kernel and permits the creation of access control policies in a modular way. This loosely-coupled design allows a system administrator to inject the necessary tools to implement MAC policies throughout the kernel. Furthermore, the kernel need not be modified each time the access control policy for certain actions and files is adjusted.

LSM has a list of actions that must be verified by the security modules loaded onto the system: if any of these modules return a deny decision, the kernel prevents the operation. LSM hooks allow additional security modules to stack enforcement functions at critical points in the kernel's execution. Security modules are not limited to controlling access to files, but can enforce access policies for operating system functions, tasks, processes, and inodes [17].

LSM provides a generic interface for inserting security module functions into the kernel and does not dictate which modules are loaded onto a system [12]. An administrator is responsible for defining the security modules that should be loaded at boot time. Consequently, LSM module development has proliferated to include projects like SELinux,[1] AppArmor,[2] and eBPF/KRSI. An administrator could use all these security modules without needing to concern themselves with applying patches to the underlying kernel due to the dynamism of the LSM interface [9].

2.2.1 *SELinux.*

SELinux was released as a patch to the Linux kernel by the National Security Agency (NSA) in 2001 [13]. SELinux extended the Linux security architecture by complementing the traditional DAC mechanism of user ownership and permissions with MAC policies defined by an administrator which explicitly grant access rights to objects on the system. Essentially, SELinux enforces security decisions by considering whether a subject may perform an action on an object [10].

Following the introduction of LSM to the kernel, SELinux was refactored from a kernel patch into a modular application comprised of enforcement functions, the Access Vector Cache (AVC), and a security server [22]. SELinux enforcement functions are attached to LSM hooks. These SELinux functions process kernel objects and user capabilities provided by the LSM hooks and pass that data to an in-kernel AVC, which caches recent security policy access

[1]https://selinuxproject.org/
[2]https://apparmor.net/

```
ALLOW apache_process apache_log:FILE READ;
```

Figure 1: Example SELinux policy that allows an Apache web server process to read its log file [19].

decisions. When a decision is not present in the AVC, the SELinux enforcement function makes a call to the SELinux security server, which supports Type Enforcement (TE) and role-based access control (RBAC) security models. Enforcement decisions are returned to the SELinux enforcement function by the AVC or security server and passed to the LSM hook function, which returns the result to the calling process.

The default SELinux security server consumes SELinux policy configurations (see Figure 1) based on the `User` to `Role` to `Domain` to `Type` transitions that SELinux uses to merge TE and RBAC into a cohesive enforcement engine [21]. `Domains` are TE labels SELinux uses to track processes and `Types` are TE labels SELinux uses to track system objects. For a process to interact with a system object, the `Domain` and `Type` labels must intersect. Processes are managed by users on the Linux host, so SELinux requires an abstraction between process `Domains` and the users who start and manage processes. The RBAC enforcement model maps SELinux users to policy defined `Roles`, and each `Role` is approved to operate across a limited set of domains. For instance, `systemd` runs as the SELinux `system_u` user. The `system_u` user has the `system_r` SELinux role, which maps to a number of type label domains such as `init_t`, `kernel_t`, and `sshd_t` for system processes. These domain labels are then matched against type labels to define what system resources a `systemd` process like `sshd` is authorized to interact with. These policies define how `systemd` manages system processes.

SELinux Booleans are checked by a subset of SELinux enforcement hook functions to deny certain behaviors. These enforcement decisions are not necessarily controlled by the SELinux type matching model but rather by the state of the Boolean value.

2.2.2 *BPF/KRSI.*

Linux's BPF was originally designed to make policy decisions on network packets. Nevertheless, BPF was quickly used for more than network filtering. For example, the secure computing system call, `seccomp`, on Linux can load a BPF program to determine which system calls a process can make. Unfortunately, `seccomp` operates at the system call level, must be enabled per process, and only supports DAC functionality. Moreover, `seccomp` has been documented to incur as much as a 25% overhead [5].

Modern BPF benefits from hardware improvements and uses just-in-time (JIT) compilation [8]. The limitations of `seccomp` and improvements to BPF motivated the KRSI patchset [20], which hooks into LSM and enables BPF-based policy decisions. KRSI describes itself as a unified audit and MAC mechanism that does not require kernel recompilation. KRSI implements its policies by inserting a trampoline in kernelspace; the trampoline transfers control to the BPF policy decision (see Figure 2).

3 MICROBENCHMARKS

We began with Wilson's BPF/KRSI security programs [24] and developed enforcement equivalent SELinux security profiles. This set of BPF programs and SELinux policies address network access, file

```
/* attach to the "bprm_check_security" LSM hook */
LSM_PROBE(bprm_check_security, struct linux_binprm *bprm) {
  u64 gid_uid, pid_tgid;
  unsigned short mode;
  /* check if executable has its SUID bit set */
  mode = bprm->file->f_inode->i_mode;
  if (mode & S_ISUID) {
    /* gather event data */
    struct data_t data = {};
    u64 gid_uid = bpf_get_current_uid_gid();
    u64 pid_tgid = bpf_get_current_pid_tgid();
    bpf_get_current_comm(&data.comm, sizeof(data.comm));
    data.uid = gid_uid & 0xffffffff;
    data.gid = gid_uid >> 32;
    data.pid = pid_tgid >> 32;
    data.dev = bprm->file->f_inode->i_sb->s_dev;
    data.inode = bprm->file->f_inode->i_ino;
    data.mode = mode;
    /* optional filters */
    if (1 UID_FILTER FILE_FILTER DEFAULT_ALLOW) {
      data.allowed = 0;
      events.perf_submit(ctx, &data, sizeof(data));
      return -EPERM;
    } else {
      data.allowed = 1;
      events.perf_submit(ctx, &data, sizeof(data));
    }
  }
  return 0;
}
```

Figure 2: Example BPF function (adapted from Wilson [24]) preventing `setuid` executables via LSM hooks and KRSI.

access, and user privilege administration. We then created a suite of microbenchmarks to test the performance of these policies and used BPF probes to identify the time for each security check. We analyzed these microbenchmarks with equivalent SELinux and BPF/KRSI policies for comparison. The following paragraphs summarize our microbenchmarks.

Networking. LSM provides `security_socket_connect`, a hook for making networking enforcement decisions. In SELinux, we use `lsm_socket_connect` to attach to this LSM hook and prevent `guest_u` SELinux users from binding to any sockets. For network enforcement testing, we add a test user to the `guest_u` user role. In KRSI, we load a BPF program which trampolines off the `bpf_lsm_socket_connect` function attached to the LSM hook; this BPF program prevents the user from making socket connections to a deny list of IP addresses. Enforcement is triggered in both cases by the user executing `curl` against a public IP address. Thus, this microbenchmark prevents creating IPv4 sockets while maintaining Unix Domain Sockets (UDS).

User Privileges. `setuid` programs run as the user who owns the executable. `sudo` and `passwd` are `setuid` programs because users must inherit the root user's permissions to perform the operations in these two programs. Malicious programs can make use of `setuid` executables to mutate or escalate their execution privileges.

Preventing this behavior proved challenging. In KRSI, we load a BPF program attached to the `security_binrpm` LSM hook which allows us to deny `setuid` program execution for a set of users. SELinux does not have an equivalent mechanism for dynamically preventing `setuid` program execution. The closest similar functionality occurs in the LSM hook `security_capable` when a user attempts to execute `sudo` or `su`. At this point, SELinux performs a Linux capabilities check against the calling user's SELinux role. Our test user has passwordless sudo privileges for all operations but is assigned to the `guest_u` SELinux role, and this role precludes `sudo` operations. Testing each enforcement engine included executing `sudo` operations as the `guest_u` role for SELinux and including the user in a deny list for the BPF program. Thus, this microbenchmark prevented the creation and execution of `setuid` files while other file accesses were permitted.

Process Administration. The `ptrace` system call facilitates process tracing and debugging. While this functionality is critical for debuggers like `gdb` and `lldb`, the `ptrace` system call can potentially be used by attackers to control or to steal data from processes.

LSM has two hooks `security_ptrace_traceme` and `security_ptrace_access_check` that must fail in order to prevent `ptrace` attempts. SELinux provides a Boolean flag, `deny_ptrace`, which is checked at the SELinux enforcement function for each of the LSM hooks. For KRSI, we load a BPF program at each LSM hook function; this BPF program checks a local Boolean flag to determine if the `ptrace` attempt should be permitted. These enforcement hooks were triggered by executing an `strace` operation. Thus, this microbenchmark blocks the `ptrace` system call.

4 EVALUATION

We evaluated SELinux and BPF/KRSI using the aforementioned microbenchmarks and provide a qualitative analysis of usage and flexibility differences observed during testing. We tested two scenarios in our evaluation. The first scenario had no delay between invocations whereas the second scenario had a ten-second delay (i.e., sleep) between invocations. The delay allows us to estimate the effectiveness of caching policy decisions.

4.1 Performance Comparison

Figures 3, 4, and 5 contain histograms of our performance results. Tests used an Amazon Web Services (AWS) t.micro instance with 1 vCPU and 1 GiB of RAM running Red Hat Enterprise Linux (RHEL) 8. Each figure has separate histograms for allow and deny decisions with each scenario (i.e., no delay and a ten-second delay between invocations). In many cases, the histograms clearly illustrate a performance difference between SELinux and BPF/KRSI.

We use Welch's t-test [23] with $\alpha = 0.05$ to identify if the differences between SELinux and BPF/KRSI samples are statistically significant—i.e., the likelihood two independent samples have equal means. The results of this statistical analysis are captured in Table 1.

Our analysis reveals several interesting patterns:

- Neither BPF/KRSI nor SELinux outperforms the other in all tasks although BPF/KRSI typically outperforms SELinux.
- BPF/KRSI is superior when there is no delay.
- There appear to be asymmetries in some policies. For example, the BPF/KRSI policy is better than SELinux when actions are denied, but not when they are allowed.

Table 1: Statistical analysis of the performance comparison

Decision	Microbenchmark	Delay (s)	p	Best
Allow	socket	0	.000	BPF/KRSI
		10	.010	SELinux
	`setuid`	0	.000	BPF/KRSI
		10	.000	—
	`ptrace`	0	.000	BPF/KRSI
		10	.000	SELinux
Deny	socket	0	.000	BPF/KRSI
		10	.011	BPF/KRSI
	`setuid`	0	.000	BPF/KRSI
		10	.000	BPF/KRSI
	`ptrace`	0	.000	BPF/KRSI
		10	.000	SELinux

Even when there is a statistically significant difference, the magnitude of the difference is minute. BPF/KRSI and SELinux were within 12–54 microseconds of each other when there is a delay between policy decisions and 15–92 microseconds when there is no delay. However, these minute differences can be large in aggregate. For a workload with a million policy decisions, BPF/KRSI will be 12–92 seconds faster than equivalent SELinux policies.

Because BPF/KRSI outperforms SELinux when there is no delay between policy decisions, we infer that BPF is able to better cache a policy decision. We hypothesize two reasons for this behavior. First, the kernel's BPF implementation uses JIT compilation. The policy enforcement code generated by JIT compilation may be deallocated or swapped out after a certain period of time. Second, SELinux requires configuration files to be specified; processing the configuration likely requires many conditional jumps. However, BPF/KRSI routines are implemented as kprobes; only a jump to the JIT-ed BPF code and a jump back are needed. Therefore, BPF/KRSI policies may be friendlier with CPU instruction caches.

4.2 Qualitative Comparison

Aside from performance differences, we observed qualitative differences between SELinux and BPF/KRSI. BPF/KRSI offered much more range in its security enforcement decision making. Essentially, any security logic which can be articulated as code can be loaded as a BPF program attached to an LSM hook. Conversely, SELinux functionality is constrained by its rigid system architecture and enforcement models. It was much easier to get the desired enforcement behavior of deny-`setuid` from BPF/KRSI than SELinux. Some researchers are using the extended flexibility of BPF/KRSI to create behavioral security enforcement mechanisms, for example by building system call profiles for normal functioning processes [6].

The flexibility of BPF/KRSI imposes a higher administrative overhead. Unlike SELinux's policy framework interface and security server enforcement engine, BPF/KRSI does not have an abstracted

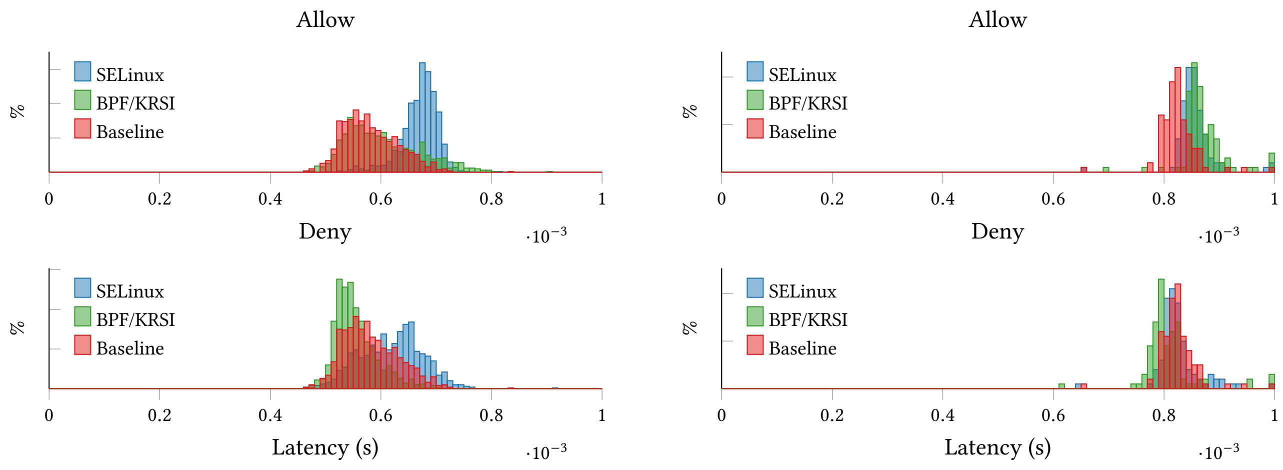

Figure 3: Histograms of latencies for IPv4 socket creation with no delay between invocations (left) and a 10-second delay between invocations (right). The baseline demonstrates the latency of the system without a MAC mechanism enabled.

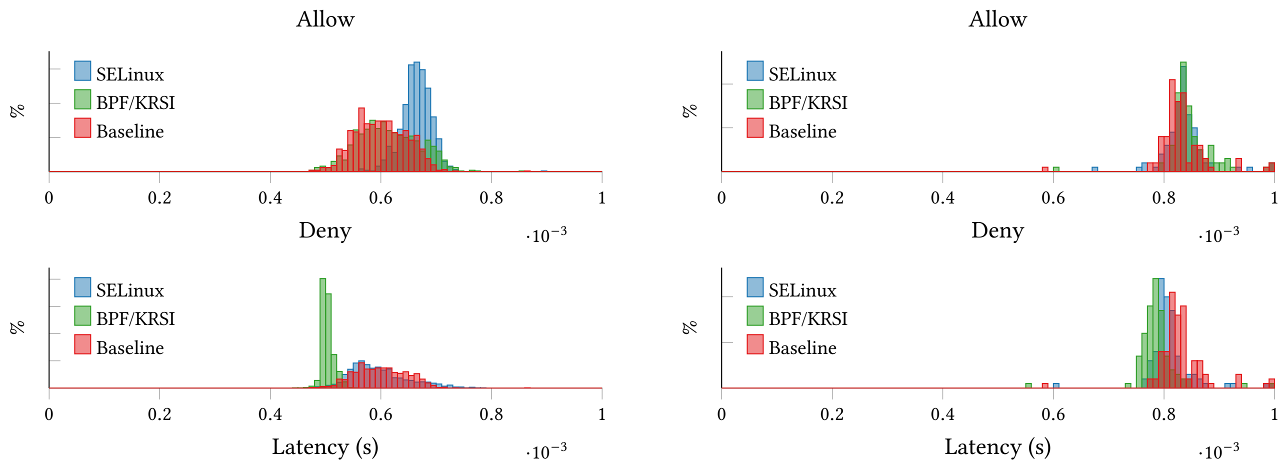

Figure 4: Histogram of latencies of `setuid` file execution policies with no delay between invocations (left) and a 10-second delay between invocations (right). The baseline demonstrates the latency of the system without a MAC mechanism enabled.

interface for administration, and decision-making logic spans enforcement functions (i.e., it is decentralized). To develop BPF/KRSI enforcement functions requires proficiency in kernel programming, increasing the cost of adoption of BPF/KRSI for most organizations.

5 RELATED WORK

Research in the infant stages of LSM [25] explains the use of LSM as a general framework for access control along with the performance implications on the kernel as performance is a critical aspect of LSM adoption. Comparing the stock kernel performance with one where LSM was compiled but no enforcement modules loaded resulted in 0–2% overhead for microbenchmarks with the worst being 7.2% for file deletion. More recent work [26] focuses on the overhead of LSM for file access. Performance was worse with SELinux hooks compared to not having one, as performance is related to the connection of the hooks and the policy enforcement placed on LSM. This research complements our findings in the sense that MAC policies degrade performance of a system, but our work extends the discourse by offering a comparative performance analysis.

Chen et al. [2] analyze the vulnerability surface exposed by SELinux and AppArmor. They find significant differences between targeted and reference policies for SELinux; the reference policy in some cases is little better than not using MAC at all! Schreuders et al. [18] analyze the usability of systems for application confinement, including SELinux and AppArmor. They conclude that SELinux "suffers from major usability deficiencies and is in need of significant usability improvements" whereas AppArmor is substantially better in this regard. Nevertheless, the security decisions required to correctly confine applications remain beyond the reach of end users with both tools. Our qualitative comparison of SELinux and BPF/KRSI raises similar issues, and Schreuders et al.'s usability study serves as a model for future work in this vein.

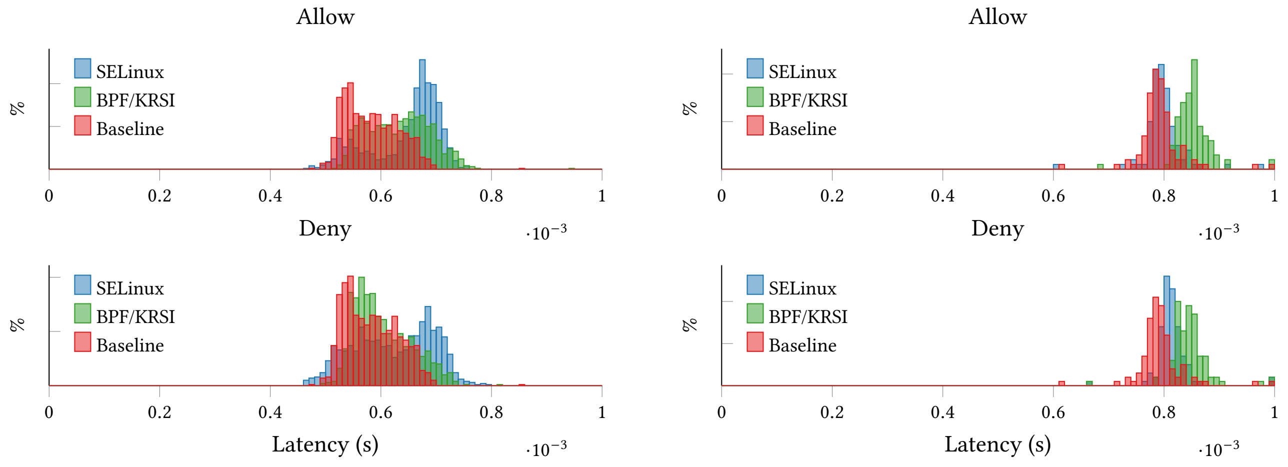

Figure 5: Histogram of latencies of `ptrace` system calls with no delay between invocations (left) and a 10-second delay between invocations (right). The baseline demonstrates the latency of the system without a MAC mechanism enabled.

Findlay et al. [7] propose bpfbox, a proof-of-concept system for application confinement that uses eBPF to confine applications at userspace function, system call, LSM hook, and kernel function boundaries. bpfbox imposes an overhead of 15% in the worst case compared to AppArmor's 110%. Our work is similar, but we compare "raw" BPF/KRSI policies with SELinux rather than introducing a new policy language.

6 CONCLUSION

Unix systems traditionally use a DAC paradigm, but this paradigm for access control can be augmented with MAC. Our comparison of SELinux—the de facto MAC mechanism for Linux—with the newer BPF/KRSI indicates that policies in BPF/KRSI generally offer enhanced performance and are more expressive than comparable policies in SELinux. Our results suggest that BPF/KRSI is preferable to SELinux for new MAC deployments.

There are cases where SELinux outperforms BPF/KRSI such as when MAC policy decisions are computed infrequently. Thus, performance-critical workloads should benchmark both SELinux and BPF/KRSI to determine which MAC mechanism provides the least overhead. Furthermore, SELinux policies may be easier to create and manage in some cases because SELinux does not require an understanding of the C programming language or kernel data structures. SELinux also offers administrative tooling that is superior to BPF/KRSI. Nevertheless, the performance overhead, ease of creating and managing policies, and development of administrative tools for BPF/KRSI will likely improve over time, as BPF/KRSI remains a relative newcomer in this space.

Future Work

While our research provides recommendations for MAC policy enforcement on Linux, further research is needed to fully understand the impact of MAC in the real world. For example, relatively little research has been done to identify the real-world security gains that MAC provides. Neither SELinux nor BPF/KRSI (nor any other MAC mechanism on any operating system) prevent kernel vulnerability exploitation. However, MAC can help reduce the blast radius of a vulnerable userspace application. Future work should examine recent CVEs and what (if any) policies could thwart them.

Research is also needed to understand the performance tolerances for common workloads. While Wilson [24] suggests a 1% overhead may be unacceptable for high-performance computing, it is likely that this overhead is acceptable in other contexts. Understanding the causes of the performance differences between SELinux and BPF/KRSI—specifically why BPF/KRSI provides worse performance when policies are exercised infrequently and result in an allow decision—should be another area of active research. Extending this analysis to include additional MAC mechanisms, such as AppArmor [4], is also worthwhile. Like SELinux, AppArmor is installed as an LSM module and does not require kernel recompilation, but unlike SELinux, AppArmor policies are written in flat files with no labeling.

ACKNOWLEDGMENTS

The views expressed in this paper are those of the authors and do not necessarily reflect the official policy or position of the United States Air Force Academy, the Air Force, the Department of Defense, or the U.S. Government.

REFERENCES

[1] Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young. 1986. Mach: A New Kernel Foundation For UNIX Development. In *USENIX Summer Conference*. USENIX.

[2] Hong Chen, Ninghui Li, and Ziqing Mao. 2009. Analyzing and Comparing the Protection Quality of Security Enhanced Operating Systems. In *Proceedings of the 2009 NDSS Symposium* (NDSS 2009). (Feb. 2009).

[3] Jonathan Corbet. 2019. KRSI - the other BPF security module. (Dec. 2019). https://lwn.net/Articles/808048/.

[4] Crispin Cowan, Steve Beattie, Greg Kroah-Hartman, Calton Pu, Perry Wagle, and Virgil Gligor. 2000. SubDomain: Parsimonious Server Security. In *Proceedings of the 14th Systems Administration Conference* (LISA 2000). The USENIX Association, (Dec. 2000). https://www.usenix.org/legacy/event/lisa2000/full_papers/cowan/cowan.pdf.

[5] Jake Edge. 2015. A seccomp overview. (Sept. 2015). https://lwn.net/Articles/656307/.

[6] William Findlay. 2020. Security Applications of Extended BPF Under the Linux Kernel. (Apr. 2020). https://www.cisl.carleton.ca/~will/written/findlay20bpfsec.pdf.

[7] William Findlay, Anil Somayaji, and David Barrera. 2020. Bpfbox: Simple Precise Process Confinement with eBPF. In *Proceedings of the 2020 ACM SIGSAC Conference on Cloud Computing Security Workshop* (CCSW'20). Association for Computing Machinery, Virtual Event, USA, 91–103. ISBN: 9781450380843. DOI: 10.1145/3411495.3421358.

[8] Matt Fleming. 2017. A thorough introduction to eBPF. (Dec. 2017). https://lwn.net/Articles/740157/.

[9] J. Freyensee. 2020. Security Module Infrastructure for Linux and macOS. (Apr. 2020). https://medium.com/macos-is-not-linux-and-other-nix-reflections/security-module-infrastructure-for-linux-and-macos-cf677fa520b7.

[10] Mirek Jahoda, Barbora Ančincová, Ioanna Gkioka, and Tomáš Čapek. 2019. *SELinux User's and Administrator's Guide.* (Aug. 2019). https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/7/html-single/selinux_users_and_administrators_guide/index.

[11] Poul-Henning Kamp and Robert N. M. Watson. 2000. Jails: Confining the omnipotent root. https://papers.freebsd.org/2000/phk-jails.files/sane2000-jail.pdf.

[12] Kernel Development Community. 2022. Linux Security Module Usage. (June 2022). https://docs.kernel.org/admin-guide/LSM/index.html.

[13] Peter Loscocco and Stephen Smalley. 2001. Integrating Flexible Support for Security Policies into the Linux Operating System. In *2001 USENIX Annual Technical Conference (USENIX ATC 01)*. USENIX Association, Boston, MA, (June 2001). https://www.usenix.org/conference/2001-usenix-annual-technical-conference/integrating-flexible-support-security-policies.

[14] M.K. McKusick, G.V. Neville-Neil, and R.N.M. Watson. 2014. *The Design and Implementation of the FreeBSD Operating System.* Addison Wesley. ISBN: 978-0321968975.

[15] Daniel Price and Andrew Tucker. 2004. Solaris Zones: Operating System Support for Consolidating Commercial Workloads. In *18th Large Installation System Administration Conference (LISA 04)*. USENIX Association, Atlanta, GA, (Nov. 2004). https://www.usenix.org/legacy/publications/library/proceedings/lisa04/tech/price.html.

[16] Dennis M. Ritchie and Ken Thompson. 1974. The UNIX Time-Sharing System. *Commun. ACM*, 17, 7, (July 1974), 365–375. DOI: 10.1145/361011.361061.

[17] Vandana Salve. 2020. Inside the Linux Security Module (LSM). (July 2020). https://elinux.org/images/0/0a/ELC_Inside_LSM.pdf.

[18] Z. Cliffe Schreuders, Tanya McGill, and Christian Payne. 2011. Empowering End Users to Confine Their Own Applications: The Results of a Usability Study Comparing SELinux, AppArmor, and FBAC-LSM. *ACM Transactions on Information and System Security*, 14, 2, Article 19, (Sept. 2011), 28 pages. DOI: 10.1145/2019599.2019604.

[19] Red Hat Customer Content Services. 2022. Using SELinux: Writing a Custom SELinux Policy. (Aug. 2022). https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/8/html/using_selinux/writing-a-custom-selinux-policy_using-selinux.

[20] KP Singh. 2019. Kernel Runtime Security Instrumentation. (Sept. 2019). https://lwn.net/Articles/798918/.

[21] Stephen Smalley. 2005. Configuring the SELinux Policy. (Feb. 2005). https://www.nsa.gov/portals/75/images/resources/everyone/digital-media-center/publications/research-papers/configuring-selinux-policy-report.pdf.

[22] Stephen Smalley, Chris Vance, and Wayne Salamon. 2006. Implementing SELinux as a Linux Security Module. (Feb. 2006). https://www.nsa.gov/portals/75/documents/resources/everyone/digital-media-center/publications/research-papers/implementing-selinux-as-linux-security-module-report.pdf.

[23] B. L. Welch. 1947. The generalization of 'student's' problem when several different population variances are involved. *Biometrika*, 34, 1–2, (Jan. 1947), 28–35. DOI: 10.1093/biomet/34.1-2.28.

[24] Billy Wilson. 2020. Mitigating Attacks on a Supercomputer with KRSI. *SANS GIAC Certifications*, (Dec. 2020).

[25] Chris Wright, Crispin Cowan, James Morris, Stephen Smalley, and Greg Kroah-hartman. 2002. G.: Linux security module framework. In *In: Ottawa Linux Symposium, Citeseer*.

[26] Wenhui Zhang, Peng Liu, and Trent Jaeger. 2021. Analyzing the Overhead of File Protection by Linux Security Modules. In *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security* (ASIA CCS '21). Association for Computing Machinery, Virtual Event, Hong Kong, 393–406. ISBN: 9781450382878. DOI: 10.1145/3433210.3453078.