RustShield: Exploring the Feasibility of a Rust-Based Linux Security Module
CS 5264 (Advanced) Linux Kernel Programming Term Project

Charles P. Wiecking
Virginia Tech

1 Introduction

The Linux kernel is one of the most important and fundamental components in computing. It is used in systems ranging from mobile phones and servers to cloud infrastructures and embedded devices. While Linux's popularity and versatility are indisputable, its reliance on C makes it vulnerable to memory safety issues such as buffer overflows, null pointer dereferences, and use-after-free errors. These vulnerabilities expose the kernel to security risks, particularly in file access control, where adversaries can modify or delete critical system files such as `/etc/shadow` (used for storing password hashes), `/etc/passwd` (containing user authentication data), and `/var/log/auth.log` (the system's security audit logs) to escalate privileges or erase forensic evidence.

To address this issue, this project introduces **RustShield**, an experimental Linux Security Module (LSM) written in Rust. RustShield is built using the Rust-for-Linux framework, which brings Rust support to the kernel in an effort to reduce memory-related vulnerabilities through compile-time guarantees. While the initial goal was to implement a fully functional file access control module in Rust, limitations in the current Rust LSM integration APIs have required a shift in scope. The project now focuses on evaluating the feasibility of developing a Rust-based LSM within the constraints of the current kernel infrastructure. As part of this effort, RustShield will be compared conceptually to traditional C-based modules such as SELinux and AppArmor to assess the practicality, safety, and integration challenges involved in using Rust for security-critical kernel components.

2 Background

2.1 Linux Kernel Security and Memory Safety

The Linux kernel serves as the backbone for modern computing, powering systems ranging from embedded devices and mobile phones to large-scale servers and cloud infrastructure. Its versatility and performance are rooted in its low-level implementation in C, which allows for fine-grained hardware control. However, this reliance on C introduces significant security challenges due to the lack of memory safety guarantees. Common vulnerabilities with C implementation include buffer overflows, null pointer dereferences, use-after-free errors, and double-free errors, all of which can lead to privilege escalation, system crashes, and data corruption [5].

2.2 Existing Security Models: SELinux and AppArmor

To address security concerns, the Linux kernel incorporates the LSM framework, which allows developers to load security models at runtime. Two widely used LSMs are SELinux and AppArmor:

- **SELinux**: Initially developed by the National Security Agency, SELinux implements a flexible and fine-grained mandatory access control (MAC) system based on a security policy. It uses a security server and an Access Vector Cache (AVC) to determine and enforce permissions based on subject-object pairs. SELinux effectively restricts unauthorized access, but it is highly complex to configure and maintain due to its extensive policy framework [11, 12].

- **AppArmor**: AppArmor is a simpler alternative that confines programs using path-based rules rather than label-based policies. It offers ease of configuration and reduced complexity, but at the cost of less granular control compared to SELinux [3, 11].

Despite their effectiveness, both SELinux and AppArmor are written in C, making them vulnerable to memory safety issues and potential kernel exploits. Their complexity also contributes to misconfigurations by system administrators, which can reduce overall system security.

2.3 Introduction of Rust for Linux

To mitigate these memory-related vulnerabilities, the Linux kernel community has introduced Rust-for-Linux, which allows developers to write kernel components in Rust rather than C. The Rust programming language offers memory safety guarantees through its ownership and borrowing models, effectively preventing vulnerabilities like buffer overflows and use-after-free errors at compile time [4]. Rust-for-Linux integrates Rust into the kernel by:

- Providing kernel abstractions and bindings for safe access to kernel APIs.
- Wrapping unsafe kernel functions with safe Rust abstractions.
- Leveraging Rust's strong typing and borrow checker to eliminate null pointer dereferences and race conditions.

Early experiments with Rust in the Linux kernel, such as the rOOM (Rust-based Out of Memory) component, demonstrate that Rust can match the performance of C-based components while reducing memory-related bugs [5].

2.4 RustShield: Motivation and Goals

RustShield builds on the momentum of the Rust-for-Linux initiative by exploring whether a LSM can be implemented in Rust to enhance file access control while maintaining memory safety. Existing LSMs like SELinux and AppArmor provide robust access control, but their reliance on C leaves them susceptible to memory-related vulnerabilities. RustShield investigates whether using Rust can offer comparable functionality with improved safety. While Rust support in the kernel continues to expand, the APIs and infrastructure needed for full LSM integration in Rust remain under active development, with minimal documentation available. As a result, RustShield's current goal is to assess the feasibility of such an implementation. Specifically, it aims to:

- Monitor and log unauthorized file access attempts
- Restrict unauthorized modifications and deletions of critical system files.
- Leverage Rust's memory safety features through its ownership model and compile-time checking.

This work aims to contribute to ongoing efforts to improve kernel security and offers insight into the current boundaries and future potential of Rust-based LSM development.

3 Related Works

Numerous efforts have explored topics related to LSMs, kernel security, and the integration of Rust into the Linux kernel. However, no publicly available documentation or academic work currently describes an attempt to implement an LSM entirely in Rust. As a result, this section focuses on three relevant areas: the design and evolution of LSMs, the progress of Rust-for-Linux development, and the lack of publicly documented Rust-based LSM implementations.

3.1 LSM Implementation

The LSM framework was introduced to provide a general-purpose security infrastructure within the Linux kernel, allowing developers to implement access control policies without enforcing a single security model. Foundational work by Smalley et al. outlines the architecture of LSMs and how the framework enables multiple security models, such as SELinux and AppArmor, to coexist through pluggable hooks and modular policy enforcement logic [12].

Later research has evaluated the performance and maintainability of LSMs. Zhang et al. analyzed the runtime overhead introduced by LSM hooks in file system protection and found measurable, but acceptable, performance costs associated with increased granularity [14]. Other works have focused on the correctness and long-term reliability of the LSM framework. For example, methodologies have been proposed to verify the stability of LSM hook placement and detect regression errors during kernel updates [2].

Collectively, these works highlight both the security benefits and design challenges involved in implementing and maintaining LSMs, particularly in C-based environments where memory safety remains a persistent concern.

3.2 Rust-For-Linux Progress

The Rust-for-Linux initiative began gaining momentum around 2020 and has since established Rust as a supported language for kernel module development. Li et al. conducted an in-depth empirical study examining the adoption of Rust in the kernel. Their findings show that Rust significantly reduces memory and concurrency-related bugs, though it introduces development overhead when bridging to existing C infrastructure [4].

Panter and Eisty provide a broader perspective on the ecosystem of Rust-for-Linux in which they show improvements in developer tooling, kernel abstractions, and interoperability challenges [10]. In a practical demonstration of Rust's capabilities, Miller et al. developed Ekiben, a Rust-based kernel framework for agile scheduler development, showing that complex and performance-sensitive subsystems can be implemented safely in Rust [7].

The Rust-for-Linux documentation continues to evolve and now includes experimental APIs, kernel build instructions, and integration guidance for new developers [6]. In addition, individual contributors and kernel maintainers (e.g., Wedson

Almeida Filho, Miguel Ojeda) regularly share progress reports and technical proposals, contributing to the language's adoption in the kernel space [1, 8, 9, 13].

3.3 Other Considerations

While numerous projects have explored using Rust to implement kernel subsystems, drivers, and even full operating systems (e.g., Redox, Theseus), there is currently no publicly available documentation or academic work describing a Rust-based Linux Security Module. This gap suggests that Rust-based LSM development remains largely unexplored in both the literature and community practice—a space that RustShield aims to begin investigating.

The most comparable effort is rOOM, a Rust-based reimplementation of the kernel's out-of-memory (OOM) component. rOOM demonstrates that Rust can offer both safety and performance in memory-critical kernel subsystems [5]. However, it does not involve the LSM framework or kernel-level access control.

RustShield contributes to this emerging area by examining the feasibility of developing an LSM using Rust. Even without full integration support, the project provides preliminary insights into the technical barriers, design considerations, and future directions for memory-safe kernel security development.

4 Goals and Progress

4.1 Original Goals

The RustShield project aimed to develop a LSM that leverages Rust's memory safety features to monitor and restrict unauthorized file access. The project intended to address the inherent vulnerabilities present in existing C-based LSMs, such as SELinux and AppArmor, by implementing a Rust-based module. The specific goals outlined in the project proposal were as follows:

- **Goal 1: Implement a Rust-based LSM using Rust-for-Linux.** The module would integrate seamlessly with the existing Linux Security Module framework.
- **Goal 2: Hook into file access syscalls.** The module aimed to intercept file operations (e.g., `open`, `unlink`) to monitor and log unauthorized actions.
- **Goal 3: Log and restrict unauthorized file modifications.** Attempts to modify critical system files (e.g., `/etc/shadow`) would be logged and blocked.
- **Goal 4: Ensure memory safety through Rust's ownership model.** Rust's compile-time safety guarantees would help mitigate vulnerabilities like buffer overflows.
- **Goal 5: Evaluate performance and stability.** The LSM would be benchmarked against existing C-based modules to assess runtime overhead.

4.2 Evolved Goals

As the project progressed, significant challenges necessitated a shift from full implementation to feasibility exploration. The Rust-for-Linux framework, while promising, lacks comprehensive support for LSM integration. The following obstacles led to goal adjustments:

- The function `register_lsm` required for LSM registration is not accessible via Rust.
- Essential LSM hooks (e.g., `inode_permission`) are not available within the Rust-for-Linux APIs.
- The `kernel::security` module is currently a stub, limiting direct integration.

To accommodate these limitations, the goals evolved as follows:

- **Revised Goal 1: Demonstrate the feasibility of Rust-based LSM integration.**
- **Revised Goal 2: Develop foundational Rust kernel modules.**
- **Revised Goal 3: Document integration challenges and potential solutions.**
- **Revised Goal 4: Propose future steps for enhancing Rust-for-Linux support.**

By shifting to an exploratory focus, RustShield provides insights into the current capabilities and limitations of Rust-based LSM development.

5 Implementation Status

5.1 Development Environment

The RustShield project was developed within an Oracle VirtualBox environment running Ubuntu 24.04.2 LTS. The Linux kernel version used was v6.14.0-rc5 (rust-next branch), configured to support Rust as a first-class language. The Rust toolchain was the nightly version `rustc 1.87.0`, enabling experimental features needed for kernel module development.

The development environment configuration included:

- **Virtual Machine Platform:** Oracle VirtualBox
- **Operating System:** Ubuntu 24.04.2 LTS
- **Kernel Version:** v6.14.0-rc5 (rust-next branch)
- **Rust Toolchain:** `rustc 1.87.0-nightly`

- **Kernel Build Tools:** GCC, Clang, make, dpkg
- **Kernel Configuration:**
  - Enabled Rust support: `CONFIG_RUST=y`
  - Enabled securityfs support: `CONFIG_SECURITYFS=y`
  - Included RustShield as a built-in module: `CONFIG_RUSTSHIELD=y`

Figure 1 shows the successful integration of RustShield in the kernel configuration menu, confirming that the module is recognized as part of the kernel build. As implementation progressed, the RustShield module was relocated to the Security options submenu within `menuconfig` as seen in Figure 2.

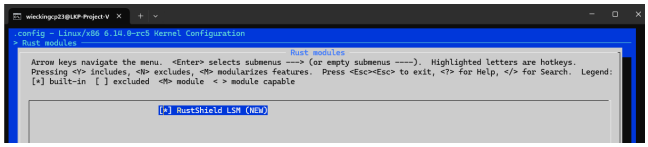

Figure 1: Kernel configuration showing RustShield as a built-in module (From Mid-Report).

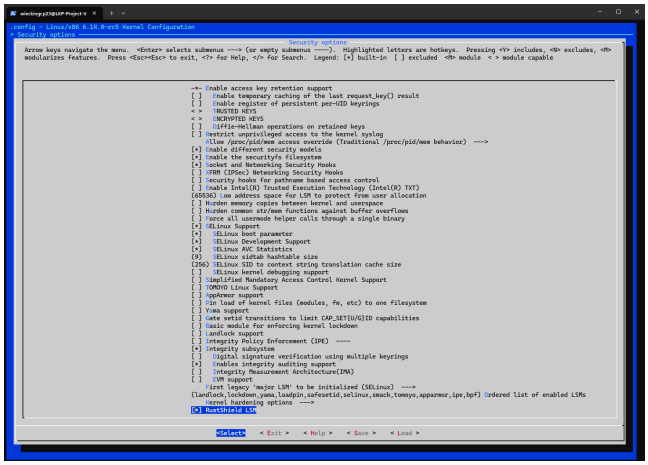

Figure 2: Kernel security options with RustShield enabled (Final Iteration).

5.2 Module Structure

The RustShield project follows a hybrid architecture combining Rust and C components to overcome integration challenges:

- **Rust Core Module:** Implements core logic, including logging via `pr_info!` to indicate module load and unload events.
- **C Helper Module:** Handles LSM registration and security subsystem interaction, using `DEFINE_LSM()` and `security_add_hooks()`.

The hybrid approach was necessary due to the lack of direct LSM function exposure in the Rust-for-Linux API. The C component facilitates the registration of hooks while the Rust component provides safer and more modular logging functionality. The C helper module implements:

- Registration of LSM hooks using `DEFINE_LSM()`.
- Initialization of a sysfs interface to toggle debug mode.
- Proper LSM descriptor definition with flags: `LSM_FLAG_LEGACY_MAJOR | LSM_FLAG_EXCLUSIVE`.

Figure 3 shows the successful loading of the C helper module, as indicated in the kernel log.

Figure 3: Confirmation that the RustShield C helper code was successfully loaded.

5.3 Key Features Implemented

The RustShield project achieved several key technical milestones despite the challenges:

- **Module Registration:** The RustShield module was successfully loaded at boot, verified via kernel logs.
- **Basic Logging:** Module activities were recorded using `pr_info!`, allowing traceability through `dmesg`.
- **Sysfs Debug Interface:** A sysfs directory was created for toggling debug mode, although its stability varied.

Figure 4 shows a kernel log entry confirming the successful loading of RustShield. This output was among the most significant results, as it was generated by a Rust-based kernel module.

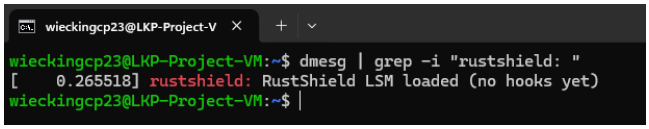

Figure 4: First Instance of a Rust-based kernel module loading

5.4 Key Challenges

Integrating Rust into the kernel security API presented several critical challenges. The primary issues and corresponding solutions are as follows:

5.4.1 LSM Hook Integration

Attempts to register LSM hooks directly using Rust resulted in unresolved symbol errors. The core problem was the absence of direct access to functions such as `register_lsm` and `security_hook_heads` within the Rust-for-Linux API. Figure 5 displays the errors encountered during compilation.

```
make clean
make LLVM=1 -j$(nproc) > full-build.log 2>&1
grep -Ei "error|failed|panicked" full-build.log
  CLEAN   arch/x86/entry/vdso
  CLEAN   arch/x86/kernel/cpu
  CLEAN   arch/x86/kernel
  CLEAN   arch/x86/realmode/rm
  CLEAN   arch/x86/tools
  CLEAN   certs
  CLEAN   init
  CLEAN   rust
  CLEAN   security/selinux
  CLEAN   usr
  CLEAN   .
  CC      /home/wieckingcp23/projects/rustshield/linux/tools/objtool/str_error_r.o
  RUSTC L rust/build_error.o
error[E0725]: the feature `allocator_api` is not in the list of allowed features
error[E0433]: failed to resolve: could not find `Hook` in `security`
error[E0405]: cannot find trait `KernelModule` in this scope
error[E0412]: cannot find type `Hook` in module `security`
error[E0425]: cannot find function `add_hooks` in module `security`
error: unused attribute
error: aborting due to 6 previous errors
Some errors have detailed explanations: E0405, E0412, E0425, E0433, E0725.
For more information about an error, try `rustc --explain E0405`.
make[4]: *** [scripts/Makefile.build:263: rust/rustshield/rustshield.o] Error 1
make[3]: *** [scripts/Makefile.build:465: rust/rustshield] Error 2
make[2]: *** [scripts/Makefile.build:465: rust] Error 2
  CC      kernel/trace/error_report-traces.o
make[1]: *** [/home/wieckingcp23/projects/rustshield/linux/Makefile:1989: .] Error 2
make: *** [Makefile:251: __sub-make] Error 2
```

Figure 5: First set of errors revealing the last of security support for Rust

To circumvent this issue, the project employed a C helper for LSM registration. Although this approach enabled basic functionality, it compromised the goal of a purely Rust-based module by introducing C code back into the module logic.

5.4.2 Sysfs Interface Issues

The sysfs debug mode interface was intended to allow toggling of logging behavior. Initial implementations using `kobject_create_and_add()` failed, prompting a switch to `securityfs_create_dir()`. Despite this change, the sysfs directory occasionally failed to initialize, as seen in Figure 6, where excessive logging led to instability.

```
[   12.836556] RustShield: Dummy inode permission check
[   12.836567] RustShield: Dummy inode permission check
[   12.836569] RustShield: Dummy inode permission check
[   12.836573] RustShield: Dummy inode permission check
[   12.836711] RustShield: Dummy inode permission check
[   12.836966] RustShield: Dummy inode permission check
[   12.837114] RustShield: Dummy inode permission check
[   12.837307] RustShield: Dummy inode permission check
[   12.837309] RustShield: Dummy inode permission check
[   12.837342] RustShield: Dummy inode permission check
[   12.837413] RustShield: Dummy inode permission check
[   12.837495] RustShield: Dummy inode permission check
[   12.837664] RustShield: Dummy inode permission check
[   12.837831] RustShield: Dummy inode permission check
[   12.837969] RustShield: Dummy inode permission check
[   12.838121] RustShield: Dummy inode permission check
[   12.838134] RustShield: Dummy inode permission check
[   12.838137] RustShield: Dummy inode permission check
[   12.838138] RustShield: Dummy inode permission check
[   12.838139] RustShield: Dummy inode permission check
[   12.838140] RustShield: Dummy inode permission check
[   12.838165] RustShield: Dummy inode permission check
[   12.838166] RustShield: Dummy inode permission check
[   12.838167] RustShield: Dummy inode permission check
[   12.838167] RustShield: Dummy inode permission check
```

Figure 6: Excessive logging captured multiple times during boot up

5.4.3 Boot Stability Issues

During early testing, the system would occasionally fail to boot due to excessive log outputs overwhelming the `systemd-journald` service. Figure 7 shows the system hanging at the Ubuntu splash screen, a direct result of unregulated log volume. Mitigation involved reducing log verbosity and adjusting the GRUB configuration to prevent kernel log overflow, ultimately stabilizing the boot process.

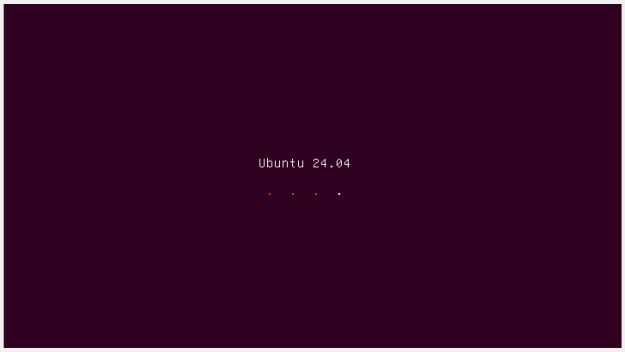

Figure 7: Seeing Ubuntu heavily suggested that RustShield overwhelmed the `systemd-journald` service

5.4.4 Kernel Configuration Challenges

Integrating RustShield required iterative kernel configuration updates. Despite enabling Rust and LSM support, inconsistencies remained, necessitating manual adjustments via `menuconfig`.

5.5 Conceptual Performance Analysis

Due to the incomplete implementation of file access control, performance benchmarking of the RustShield LSM was not feasible. The primary reason is that without fully functional hooks, the module did not enforce file access policies or generate significant runtime workload to measure. However, some conceptual insights can be drawn:

- **Minimal Overhead for Logging:** The use of `pr_info!` for basic logging introduces negligible performance impact, as shown in successful load tests.

- **Expected Overhead for Hook Integration:** If LSM hooks were fully implemented, the primary performance consideration would involve the frequency and volume of log entries. The observed log spamming during the dummy inode permission check suggests that rate-limiting mechanisms would be necessary to maintain system stability.

- **Kernel Impact:** Kernel panics were avoided during basic load and unload operations, indicating that the integration of the C helper itself does not introduce significant overhead. The instability primarily arose from runtime behavior rather than module integration.

In summary, while direct benchmarking was not possible, the conceptual analysis indicates that the primary performance challenge would be managing log verbosity rather than computational overhead from Rust integration itself. This conclusion is drawn from empirical observations during testing, where the primary cause of system instability was identified as excessive log spamming rather than computational overhead. The core module, when functioning without excessive logging, demonstrated stable behavior with minimal impact on system performance. Therefore, the primary challenge lies in managing log verbosity rather than addressing computational inefficiencies inherent to Rust integration.

6 Experimental Results

6.1 Logging and Kernel Messages

One of the primary objectives of the RustShield project was to verify that the module loaded correctly and produced kernel log entries, indicating successful integration. Upon loading, the RustShield module generated logs confirming its activation, as verified via the `dmesg` command.

Figure 4 shows a successful log entry indicating that the RustShield LSM loaded without hooks, confirming that the kernel recognized the module. Additionally, the C helper module, responsible for registering LSM hooks, was correctly loaded, as shown in Figure 3. This confirmation demonstrates that the LSM registration process, although partially successful, was sufficient for basic logging functionality.

Log Spamming Issue: A significant challenge encountered during the project was the excessive logging from the dummy inode permission check. Figure 6 shows the kernel logs becoming overwhelmed with repetitive entries, causing system instability and, in some cases, kernel stalls.

Mitigation Strategies: To address this issue, the following measures were implemented:

- Reduced verbosity of the dummy inode permission check.
- Introduced log rate-limiting where feasible.
- Adjusted GRUB configuration to reduce the number of active LSMs.

While these mitigations reduced log spamming, they did not completely resolve the issue, as kernel logging remains a resource-intensive process when triggered frequently.

6.2 Stability Testing

Stability testing was conducted to ensure that the RustShield module could be loaded and unloaded without causing kernel panics or other system instabilities.

Module Load and Unload: The module was loaded using: `sudo insmod rustshield.ko` and unloading was performed via: `sudo rmmod rustshield`. The module consistently loaded without causing kernel crashes, as confirmed by the successful log entry shown earlier in Figure 4. However, instability issues emerged when the module generated excessive logging during file access attempts. Figure 7 shows the system hanging at the Ubuntu splash screen after rebooting, which was traced to the excessive log output overwhelming the `systemd-journald` service.

Recovery Procedure: To regain system control after a failed boot, the recovery mode was utilized to disable the RustShield module. Figure 8 shows the recovery menu accessed after a system hang. This method allowed the system to revert to a previous kernel configuration, disabling the problematic module.

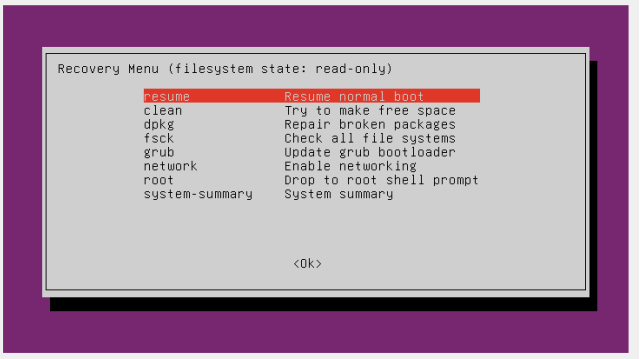

Figure 8: The Linux recovery menu was a lifesaver and progress saver during the development process

Improving Stability: The primary instability issue stemmed from unregulated log output, particularly when the inode permission hook was triggered repeatedly. To stabilize the module:

- Logging verbosity was reduced, limiting the frequency of kernel log entries.
- Unnecessary LSMs (SELinux) were disabled via GRUB to minimize conflicts.

These adjustments significantly improved boot stability, although the module's logging functionality remained sensitive to high-frequency file operations.

6.3 Conceptual Performance Analysis

Due to the incomplete implementation of comprehensive file access hooks, direct performance benchmarking was not con-

ducted. However, several conceptual insights can be drawn from the observed behavior:

Minimal Overhead for Basic Logging: The use of `pr_info!` for basic load/unload messages had negligible impact on system performance during controlled testing. However, when used excessively (as in the inode permission check), it led to system instability, indicating that even lightweight logging can become a bottleneck if not properly managed.

Expected Overhead for Full Hook Integration: If the LSM hooks were fully integrated, the primary performance consideration would be the frequency of log entries. Given the observed log spamming during partial testing, future implementations would require robust rate-limiting mechanisms to maintain performance.

Kernel Impact Considerations: The kernel did not experience panics when the module loaded and unloaded correctly, indicating that the integration of the C helper did not inherently destabilize the system. Instabilities primarily arose from runtime logging, rather than from the module's static integration.

Proposed Performance Enhancements: To accurately assess performance in future iterations, the following improvements are suggested:

- Implement a rate-limited logging mechanism to prevent service overload.
- Develop test scenarios that simulate realistic file access patterns to better evaluate the LSM's impact.
- Compare Rust-based logging with existing C-based LSMs like SELinux and AppArmor to quantify any performance differences.

Although direct benchmarking was not feasible, the conceptual analysis indicates that managing kernel log output, rather than computational overhead, is the primary challenge for Rust-based LSMs.

7 Evaluation and Analysis

7.1 Feasibility of Rust-based LSMs

The RustShield project aimed to assess the feasibility of implementing a LSM using Rust to enhance kernel security. While the project demonstrated partial integration, significant challenges were encountered, indicating that a fully functional Rust-based LSM remains impractical with the current Rust-for-Linux support.

Comparison with SELinux and AppArmor: SELinux and AppArmor are both mature C-based LSMs that integrate seamlessly with the Linux kernel. They implement comprehensive access control policies through extensive syscall hooks, offering robust file access monitoring and process isolation. In contrast, RustShield faced considerable barriers to replicating this functionality due to the following limitations:

- **LSM Hook Registration:** Unlike SELinux and AppArmor, RustShield could not directly register hooks using Rust. Essential functions, such as `register_lsm` and related security symbols, were not exposed to the Rust-for-Linux API.
- **Hybrid Design Complexity:** While SELinux and AppArmor maintain a unified C-based structure, RustShield's hybrid design introduced complexity. Combining Rust logic with C helpers compromised the modularity and safety advantages inherent to Rust.
- **Performance and Stability Challenges:** Due to the inability to fully integrate file access controls, performance comparisons with SELinux and AppArmor were not feasible. SELinux and AppArmor maintain stability through careful management of security checks, which RustShield's logging functionality could not fully replicate.

Assessment of Feasibility: While Rust theoretically offers significant benefits in memory safety and modern programming practices, the current Rust-for-Linux framework is not equipped to support comprehensive LSM functionality. The lack of critical kernel API support means that even partial implementation requires extensive kernel modifications or reliance on C-Rust interop. Consequently, the primary advantage of using Rust, memory safety, is undermined when reintroducing C code to bridge gaps.

The hybrid approach, combining Rust-based logic with a C registration helper, was the most practical solution within the current framework. However, this method inherently reintroduces the potential for memory safety issues, which defeats the purpose of a purely Rust-based security module.

7.2 Lessons Learned

The RustShield project presented several challenges associated with developing Rust-based LSMs. The most significant issue was the lack of direct support for LSM functions within the Rust-for-Linux framework, necessitating hybrid solutions that compromise Rust's safety advantages. Integrating Rust and C components introduced complexity and potential memory safety risks, undermining the modularity benefits expected from using Rust. Additionally, managing log output within kernel modules proved essential for stability, as excessive logging overwhelmed system services, especially `systemd-journald`. Unregulated log output during runtime caused system instability, underscoring the need for effective rate-limiting mechanisms. These challenges indicate that without broader support from the Linux kernel community, achieving fully Rust-based LSMs remains impractical.

7.3 Takeaways for the Community

Advancing Rust-based LSM development requires upstreaming essential functions like `register_lsm` and related hooks to the Rust API, reducing the need for C-based intermediaries. Improving documentation on hybrid module development would also support developers in minimizing complexity and safety risks. Establishing clear standards for C-Rust interoperation would further streamline development and reduce inconsistencies. Additionally, integrating built-in rate-limiting tools within the Rust-for-Linux API would help manage kernel log output more effectively. Finally, fostering community collaboration through dedicated working groups focused on Rust LSM development would accelerate progress and encourage best practices. The RustShield project demonstrates the potential of Rust-based LSMs, but significant upstream development remains necessary to make them practical alternatives to traditional C-based modules.

8 Future Work

8.1 Short-Term Improvements

To enhance the feasibility of Rust-based LSM development, several short-term improvements should be prioritized. Kernel patching is essential to expose functions like `register_lsm` within the Rust API, reducing reliance on C helpers and promoting a more modular and memory-safe approach. Improving C-Rust interoperability by developing standardized Rust macros for LSM registration would streamline the integration process, reducing the complexity of hybrid modules. Addressing logging challenges requires implementing rate-limiting mechanisms within the Rust-based logging framework to prevent kernel log buffer overload, particularly during high-frequency file access events. Stability testing should cover a range of workloads to better understand how the RustShield module performs under typical and stress conditions. Additionally, refining the sysfs-based debug mode toggle would ensure consistent directory creation and more reliable log control, reducing operational inconsistencies.

8.2 Long-Term Enhancements

Long-term efforts should focus on upstreaming essential LSM support in Rust-for-Linux by working with the Linux kernel development community to integrate key functions directly into the Rust API. This would reduce the need for C-based components, making Rust-based security modules more practical and modular. Developing a standardized Rust LSM framework would further facilitate modular and secure implementation, abstracting common LSM operations into reusable libraries. Once comprehensive LSM support becomes available, extending RustShield to implement advanced access control features similar to SELinux and AppArmor would demonstrate the practical viability of Rust-based security. Comparing RustShield's performance with traditional LSMs through benchmarking would provide data-driven insights into the performance trade-offs of using Rust in kernel security. Further exploration into kernel hacking techniques for Rust LSMs could expose LSM hooks safely, minimizing the risks associated with combining Rust and C in kernel space.

8.3 Broader Impact

Developing Rust-based LSMs could significantly improve kernel security by leveraging Rust's memory safety features, reducing risks associated with common vulnerabilities including buffer overflows and use-after-free errors. Successfully upstreaming LSM support in Rust-for-Linux would represent a substantial step forward in how Linux handles security-critical modules. As the community continues to explore Rust integration within the kernel space, the lessons learned from RustShield can guide future efforts, fostering safer and more modular approaches to kernel security.

9 Conclusion

The RustShield project set out to assess the feasibility of implementing a Linux Security Module (LSM) using Rust, aiming to enhance kernel security through memory safety features. While the original goal was to create a fully functional Rust-based LSM, limitations within the Rust-for-Linux framework required a shift to feasibility exploration. Despite these challenges, RustShield successfully demonstrated partial integration by combining Rust logic with a C helper for LSM registration. Key achievements included successful module loading, basic logging through `pr_info!`, and a partially functioning sysfs debug interface.

However, the project revealed critical challenges, including the lack of direct support for LSM functions such as `register_lsm`, which necessitated C-based workarounds. Integrating Rust and C components introduced complexity and potential safety risks, while uncontrolled logging during file access checks led to system instability. Despite these hurdles, RustShield shows the potential of Rust-based kernel security modules and serves as a foundational case study for future developments. Addressing the current gaps in Rust-for-Linux support will be crucial for advancing memory-safe LSM implementations.

References

[1] CLAYTON, J. Linux kernel development, 2023. Accessed: March 26, 2025.

[2] JAEGER, T. Maintaining the correctness of the linux security modules framework. *Security Conference Proceedings* (2002), 223–241.

[3] LEE, R. *AppArmor Documentation*, 2025. Accessed: March 4, 2025.

[4] LI, H., GUO, L., YANG, Y., WANG, S., AND XU, M. An empirical study of rust-for-linux: The success, dissatisfaction, and compromise. In *2024 USENIX Annual Technical Conference* (2024), USENIX Association, pp. 425–438.

[5] LI, L., ZHANG, Q., XU, Z., ZHAO, S., SHI, Z., AND GUAN, Y. room: A rust-based linux out of memory kernel component. *IEICE Transactions on Information and Systems E107-D*, 3 (2024), 245–255.

[6] LINUX KERNEL DOCUMENTATION. *Rust for Linux Documentation*, 2024. Accessed: March 4, 2025.

[7] MILLER, S., KUMAR, A., VAKHARIA, T., ANDERSON, T., CHEN, A., AND ZHUO, D. Agile development of linux schedulers with ekiben, 2023.

[8] OJEDA, M. Kangrejos 2024: The rust for linux workshop, 2024. Accessed: March 27, 2025.

[9] OJEDA, M. Linux kernel development, 2024. Accessed: March 27, 2025.

[10] PANTER, S. K., AND EISTY, N. U. Rusty linux: Advances in rust for linux kernel development. *arXiv preprint arXiv:2407.18431* (2024).

[11] RED HAT. Apparmor vs selinux: Understanding linux security modules, 2022. Accessed: March 25, 2025.

[12] SMALLEY, S., VANCE, C., AND SALAMON, W. Implementing selinux as a linux security module. Tech. rep., NAI Labs, 2006.

[13] WEI, W. Y. Rust kernel module: Getting started, 2022. Accessed: March 25, 2025.

[14] ZHANG, W., JAEGER, T., AND LIU, P. Analyzing the overhead of filesystem protection using linux security modules. *arXiv preprint arXiv:2101.11611* (2021).

Notes

The introduction, background, and related works sections were adapted from the original project proposal and midterm report, which explains the similarities in wording and structure. Every effort was made to address the project's objectives, challenges, and outcomes as outlined in the original plan. Any content omitted was unintentional and due to oversight. All major deviations from the initial goals have been acknowledged and thoroughly discussed in the relevant sections of not only the midterm report but also the final report.