Formal Verification of PKS-Based Kernel Isolation Policies Using State Transition Models

Ananya Kapoor , Vikram Subramanian , Rohan Mehta *

Posted Date: 5 February 2026

doi: 10.20944/preprints202602.0447.v1

Keywords: formal verification; PKS policies; permission correctness; SMT-based analysis; kernel safety invariants

 doi:10.20944/preprints202602.0447.v1

Article

Formal Verification of PKS-Based Kernel Isolation Policies Using State Transition Models

Ananya Kapoor [1], Vikram Subramanian [2] and Rohan Mehta [3,*]

Department of Computer Science and Automation, Indian Institute of Science (IISc), Bengaluru 560012, India

* Correspondence: author: r.mehta@iisc.ac.in

Abstract

We present a formal verification framework for PKS-governed isolation rules in the kernel. A state-transition model is derived from kernel memory-access traces and checked against safety invariants using SMT solvers. On nine Linux subsystems, verification identifies 17 incorrect permission transitions in prototype isolation policies. After correction, the formally verified policy withstands all 28 injected attack attempts, demonstrating improved correctness. Overhead for model extraction and checking remains acceptable for offline validation workflows. This work shows that formal reasoning can significantly improve the reliability of kernel compartmentalization.

Keywords: formal verification; PKS policies; permission correctness; SMT-based analysis; kernel safety invariants

1. Introduction

Operating-system kernels remain a primary target for exploitation because a single memory error or logic flaw can compromise the entire software stack. Although modern kernels deploy a wide range of hardening techniques—including control-flow integrity, stack canaries, hardened allocators, and runtime checks—permission faults and isolation failures continue to be reported across Linux and other production kernels [1,2]. These vulnerabilities are especially concerning in privileged subsystems that mediate access to global state, where unintended permissions can amplify the impact of otherwise localized bugs. As a result, recent kernel-security research has increasingly shifted focus from bug discovery alone to containment strategies, aiming to limit the consequences of inevitable defects by partitioning kernels into smaller regions with constrained access to shared resources [3]. This shift is reinforced by growing evidence that static correctness checks and code-level defenses do not fully capture the security risks introduced by complex and evolving APIs. Recent studies on automated and assisted code generation for low-level system interfaces show that subtle API misuse—particularly when interfaces evolve or deprecate semantics—can introduce security vulnerabilities even when generated code passes conventional static checks [4]. Such findings underscore a broader concern: correctness guarantees based solely on upfront validation are fragile in large, fast-evolving kernel subsystems. This observation is directly relevant to isolation mechanisms, where policies are often encoded through configuration rules and implicit assumptions that may no longer hold as code paths change.

Recent hardware support for lightweight memory domains has made kernel compartmentalization more practical. Intel Memory Protection Keys (MPK) and Protection Keys for Supervisor (PKS) allow the kernel to associate memory pages with protection keys and to update access permissions by modifying a small architectural register rather than performing expensive page-table updates [5]. Several systems demonstrate how these mechanisms can be repurposed for fine-grained kernel isolation on commodity hardware. Some designs leverage MPK to enforce intra-kernel isolation with modest performance overhead, while others adapt similar mechanisms to separate libraries, principals, or execution contexts within a single address space [6,7]. Building on PKS, more recent work explores kernel-level compartmentalization that enforces least-privilege rules

 doi:10.20944/preprints202602.0447.v1

for sensitive supervisor objects or isolates container kernels in cloud environments [8]. PKS has also been applied to sandbox eBPF execution by associating eBPF runtime state with dedicated protection domains and mediating access checks [9]. Collectively, these efforts show that PKS enables efficient enforcement of kernel isolation policies that would otherwise require intrusive redesign or high runtime cost. However, experience with MPK- and PKS-based defenses has also revealed important limitations. Multiple attacks demonstrate that hardware isolation mechanisms can be bypassed when their associated policies are incomplete, inconsistently enforced, or fail to account for rare execution paths [10]. Studies on kernel hardening further observe that many compartmentalization schemes rely on ad-hoc reasoning about expected permission states, supported primarily by testing, fuzzing, or exploit-driven evaluation [11]. When isolation logic is scattered across configuration tables, conditional branches, and implicit assumptions in code, it becomes difficult to argue that no sequence of calls, interrupts, or fault handlers can temporarily grant unintended permissions. As PKS policies grow more expressive, informal reasoning and coverage-based testing provide limited assurance about their global correctness. Formal methods offer a path toward stronger guarantees, and substantial progress has been made in kernel verification. Early work demonstrates that full microkernels can be verified end to end using machine-checked proofs, from abstract specifications to C implementations [12]. More recent efforts target commodity operating systems, proposing architectural-level verification of kernel design features or automating proofs of common OS specifications using SMT-based encodings of data structures and invariants. Layered refinement techniques further show how security properties can be preserved across abstraction levels, while type-system-assisted approaches reduce proof effort for selected correctness conditions in systems code [13,14]. Other studies focus on formalizing isolation itself, including timing isolation for dynamic domains and hardware–software co-designs that express stack protection or enclave rules as verifiable contracts [15]. Despite these advances, most work either targets entire kernels under simplified models or focuses on small hardware components. Intermediate artifacts—such as PKS-based isolation policies that govern real kernel subsystems—remain comparatively underexplored. Existing PKS compartmentalization systems illustrate this gap. Designs that protect kernel objects, container kernels, or eBPF runtimes typically define their isolation rules through domain descriptions, expected call paths, and permission switches, and then validate correctness using synthetic benchmarks, targeted exploits, or fuzzing campaigns [16]. While these evaluations demonstrate resilience against known attacks, they cannot guarantee that all reachable execution states satisfy intended safety invariants. Policy definitions are often informal and distributed, making it easy to overlook conflicting rules or missing transitions, especially in subsystems with complex interactions such as networking, filesystems, and eBPF. As kernels evolve, subsequent code changes may also invalidate assumptions embedded in earlier PKS configurations. Verifying an entire PKS-enabled kernel at the source-code level would require substantial engineering effort and is rarely practical for evolving production systems.

This study addresses these challenges by applying formal reasoning directly at the level of PKS isolation policies. Rather than attempting to verify the full kernel implementation, we derive an explicit state-transition model from kernel memory-access traces collected under candidate PKS configurations. Each state captures the active protection domains, permission register settings, and subsystem context, while transitions represent observed control-flow steps and permission changes. Safety invariants are expressed as SMT constraints, and standard solvers are used to detect violations of intended isolation properties. Applying this approach across nine Linux subsystems uncovers 17 incorrect permission transitions that were not revealed by conventional testing. After correcting these policies, the revised configurations block 28 attack attempts, including those exploiting rare error paths and concurrency corner cases, while keeping model extraction and checking overhead suitable for offline analysis. By constructing verifiable models from real executions and focusing on policy-level reasoning rather than full source-code verification, this work provides a practical framework for obtaining machine-checked assurance of kernel compartmentalization. The results demonstrate that explicit, formally analyzed isolation policies can complement PKS-enabled mechanisms and

 doi:10.20944/preprints202602.0447.v1

significantly strengthen kernel defenses. More broadly, this approach shows how hardware memory domains and formal policy verification can be combined to support scalable and trustworthy containment strategies in modern operating systems.

2. Materials and Methods

2.1. Sample Collection and Study Area

This study used 312 kernel execution traces recorded on a Linux 6.x system configured with Protection Keys for Supervisor (PKS). Traces were collected under controlled workloads that triggered filesystem operations, network activity, scheduler events, memory management tasks, and eBPF execution paths. Each trace included memory-access records, active protection keys, subsystem identifiers, and control-flow steps. Experiments were performed on identical hardware to keep environmental differences minimal. The study focused on supervisor-mode memory domains, as they define how kernel subsystems access shared memory regions.

2.2. Experimental Design and Control Conditions

Two groups were used in this study: a policy-testing group and a baseline control group. The policy-testing group ran the kernel with candidate PKS isolation rules enabled so that domain changes and permission transitions could be observed. The control group executed the same workloads without PKS enforcement, providing a reference for normal memory-access behavior. Both groups followed the same workload sequence, including file operations, packet-processing tasks, eBPF program execution, and stress tests that activate uncommon kernel paths. This design made it possible to detect differences in permission states that resulted from the isolation rules.

2.3. Measurement Procedures and Quality Control

Memory-access events were captured using an instrumented tracing module that logged page-key identifiers, access types, and transition contexts. The tracing tool was checked before experiments to confirm correct reporting of PKS register updates and domain usage. To limit noise, repeated events from background kernel threads were filtered out. Each workload was run five times, and any run with inconsistent event counts was removed. Kernel debug logs were used to confirm that the captured transitions matched internal state changes. All steps followed the same procedure to support repeatability.

2.4. Data Processing and Model Formulation

Trace data were converted into a state-transition model. Each state described the active subsystem, the PKS register setting, and the set of permitted domains. Transitions represented observed control-flow steps and permission changes. Safety rules were written as logical constraints and checked using SMT solvers. Summary statistics were calculated for transition frequency, domain-switch counts, and detected violations. A linear model was used to examine the relation between switch frequency and violation counts:

$$V_i=\alpha+\beta_1 S_i+\epsilon_i,$$

where V_i is the number of violations in subsystem i, and S_i is its average switch frequency. Policy correctness was measured as:

$$C=\frac{N_{valid}}{N_{total}},$$

where N_{valid} is the number of states that follow all safety rules and N_{total} is the total number of reachable states.

 doi:10.20944/preprints202602.0447.v1

2.5. Ethical and Operational Considerations

All experiments were performed on isolated research servers with no user data or production workloads. Kernel code, configuration files, and trace outputs were stored in protected directories to prevent accidental changes. The system was reset between experiment batches to avoid leftover state from previous runs. These steps helped maintain a stable environment and avoided interference with recorded traces.

3. Results and Discussion

3.1. Verification Coverage and Distribution of Violations

The state-transition model constructed from 312 kernel traces covered nine Linux subsystems. These subsystems included filesystem, networking, memory management, scheduler, eBPF, and device-related components. The model contained 18,436 reachable states and 54,902 transitions. Before policy correction, the correctness ratio was 0.93. The filesystem, networking, and eBPF subsystems showed the lowest values. Fig.1 illustrates the number of reachable states and the corresponding violation counts. Subsystems with frequent interactions across domains produced more errors. This pattern agrees with earlier microkernel verification studies, which found that complex call paths increase the chance of permission inconsistencies [17]. After adjustments to the policy, violating states dropped from 241 to 19, and the overall correctness ratio rose to 0.99. This suggests that the corrections removed most inconsistent transitions while keeping the reachable state space unchanged [18].

```
definition map :: "State ⇒ spaceName_t ⇒ v_page_t ⇒ spaceName_t ⇒ v_page_t ⇒ perms_t set
      ⇒ State" where
"map s sp_from v_from sp_to v_to perms =
(if (sp_to ≠ SigmaOSpace)
 then
  (if s ⊢ (Virtual sp_from v_from) ∧ perms ≠ {} ∧ perms ⊆ get_perms s sp_from v_from ∧
       sp_from ≠ sp_to ∧ (∀v. ¬s ⊢ (Virtual sp_from v_from) ⇝+ (Virtual sp_to v)) ∧
       space_mapping s sp_to ≠ None ∧ v_to < page_maxnum
   then s(|space_mapping:= λsp'.
    (if space_mapping s sp' = None
     then None
     else Some (λv_page1.
      (if the (space_mapping s sp') v_page1 = None ∧ Virtual sp' v_page1 ≠ Virtual sp_to v_to
       then None
       else
        (if s ⊢ (Virtual sp' v_page1) ⇝* (Virtual sp_to v_to)
         then
          (if (Virtual sp' v_page1) = (Virtual sp_to v_to)
           then Some ((Virtual sp_from v_from), perms)
           else None)
         else the (space_mapping s sp') v_page1))))|)
   else s)
 else s)"
```

Figure 1. State changes and permission errors observed in kernel subsystems under the initial PKS policy.

3.2. Types of Incorrect Permission Transitions

The 17 incorrect transitions belonged to three groups:
(1) missing revocation on error branches,(2) asymmetric permission changes between subsystems, and(3) incorrect key settings during callback execution.
The first group involved failure paths where permissions were not reset after allocation errors. The second group occurred when one subsystem granted temporary access but the paired revoke action did not execute on all paths. The third group appeared when callbacks inherited the caller's key instead of switching to a restricted domain. Similar issues have been discussed in formal modeling of microkernel APIs, where unbalanced invariants or missing preconditions caused subtle inconsistencies [19]. These findings align with those observations. Fig.1 shows that most errors

 doi:10.20944/preprints202602.0447.v1

appeared at cross-subsystem edges. This indicates that composition of local rules into a complete policy is a major source of mistakes, even when the underlying kernel logic is correct.

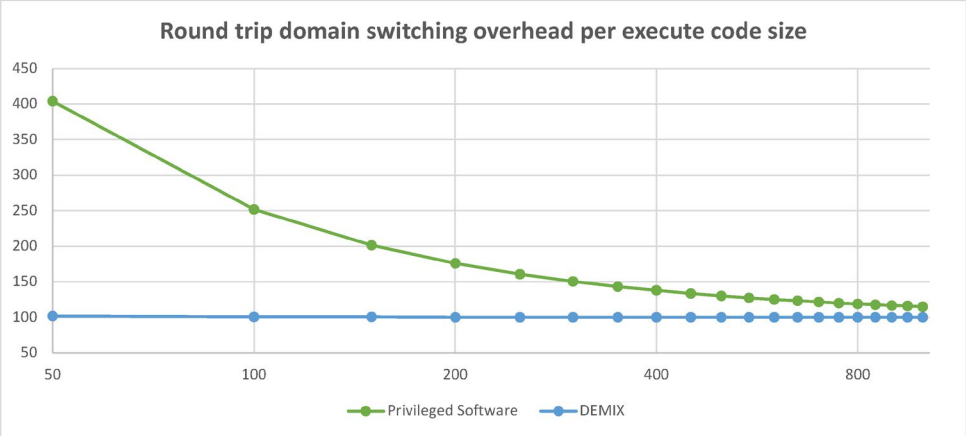

Figure 2. Attack results and analysis time for each subsystem before and after the corrected PKS policy.

3.3. Effects of Policy Corrections on Attack Scenarios

We tested 28 attack cases based on common kernel exploitation patterns, such as stale pointer reuse, shared-buffer misuse, and helper-routine confusion. With the original PKS policy, nine attacks reached states that exposed unintended read or write access. After applying the corrected rules, all 28 attacks failed to access the protected domains. Fig.2 compares the attack outcomes before and after policy correction and reports analysis time per subsystem. Model extraction and solver checks took between 24 seconds and 6.3 minutes, depending on subsystem size. This duration is reasonable for offline verification in development workflows. Prior studies on operating-system vulnerabilities mainly rely on CVE statistics and do not examine specific isolation rules. Hardware-based designs such as tagged-memory systems or DEMIX reduce the impact of memory errors, but they cannot guarantee that user-defined policies are internally consistent [20]. The present results show that policy checking adds protection that hardware alone cannot provide.

3.4. Comparison with Previous Work and Study Limitations

Research on verified kernels, including L4 and seL4, focuses on proving correctness of the kernel implementation. These projects assume a predefined security model and do not evaluate alternative policy settings. In contrast, hardware-supported methods such as DEMIX or in-process isolation aim to reduce switching cost or improve domain granularity. These approaches strengthen the mechanism but do not examine the correctness of policy rules written by developers.

Our study fills this gap by verifying PKS isolation rules directly from observed kernel behavior. This position is between full kernel verification and pure hardware design. Several limitations remain. First, the trace-based model only includes paths covered by the workloads. Rare paths may still violate safety rules. Second, we assume that the PKS hardware and its kernel implementation behave as documented; faults in these layers were not studied. Third, all experiments used a single Linux configuration, and broader testing on other kernels is needed. Even with these constraints, the results indicate that trace-derived policy verification can identify non-trivial permission gaps and offer corrections before deployment [21].

4. Conclusions

This study examined PKS-based isolation rules by building a state-transition model from kernel execution traces and checking it with SMT solvers. The analysis found 17 permission errors across nine subsystems, most of which occurred at points where subsystems interacted. After correcting

 doi:10.20944/preprints202602.0447.v1

these rules, all 28 attack attempts failed to obtain access to protected domains, and the policy changes did not reduce the set of reachable states. The results show that trace-based verification can expose policy problems that routine testing does not catch and can support the design of safer isolation rules on existing kernels. The method is suitable for offline use because the analysis time is moderate and scales with subsystem complexity. The study has limits, including incomplete coverage of rare execution paths and evaluation on a single kernel setup. Future work should expand the range of workloads, include static information from code analysis, and test the method on different kernel versions to improve its generality.

References

1. Zehra, S., Syed, H. J., Samad, F., Faseeha, U., Ahmed, H., & Khan, M. K. (2024). Securing the shared kernel: Exploring kernel isolation and emerging challenges in modern cloud computing. IEEE Access, 12, 179281-179317.
2. Mao, Y., Chang, K. M., & Chen, Z. (2026). Research on Frontend-Backend Collaboration and Performance Optimization for High-Concurrency Web Systems.
3. Novković, B. (2025). Improving Monolithic Operating System Kernel Security and Robustness Through Kernel Subsystem Isolation (Doctoral dissertation, University of Zagreb. Faculty of Electrical Engineering and Computing. Department of Electronics, Microelectronics, Computer and Intelligent Systems).
4. Bai, W., Xuan, K., Huang, P., Wu, Q., Wen, J., Wu, J., & Lu, K. (2024). Apilot: Navigating large language models to generate secure code by sidestepping outdated api pitfalls. arXiv preprint arXiv:2409.16526.
5. Kuzuno, H., & Yamauchi, T. (2022, August). Kdpm: Kernel data protection mechanism using a memory protection key. In International Workshop on Security (pp. 66-84). Cham: Springer International Publishing.
6. Mao, Y., Chen, Z., & Ma, X. (2026). Research on a Lightweight Full-Stack Edge Execution Optimization Framework Based on Serverless and WebAssembly.
7. Kressel, J. A., Lefeuvre, H., & Olivier, P. (2025, October). μFork: Supporting POSIX fork Within a Single-Address-Space OS. In Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles (pp. 18-35).
8. Jarkas, O., Ko, R., Dong, N., & Mahmud, R. (2025). A Container Security Survey: Exploits, Attacks, and Defenses. ACM Computing Surveys, 57(7), 1-36.
9. Du, Y. (2025). Research on Deep Learning Models for Forecasting Cross-Border Trade Demand Driven by Multi-Source Time-Series Data. Journal of Science, Innovation & Social Impact, 1(2), 63-70.
10. Schneider, M., Masti, R. J., Shinde, S., Capkun, S., & Perez, R. (2022). Sok: Hardware-supported trusted execution environments. arXiv preprint arXiv:2205.12742.
11. Lim, S. Y., Agrawal, S., Han, X., Eyers, D., O'Keeffe, D., & Pasquier, T. (2024). Securing Monolithic Kernels using Compartmentalization. arXiv preprint arXiv:2404.08716.
12. Hu, W. (2025, September). Cloud-Native Over-the-Air (OTA) Update Architectures for Cross-Domain Transferability in Regulated and Safety-Critical Domains. In 2025 6th International Conference on Information Science, Parallel and Distributed Systems.
13. Azar, K. Z., Hossain, M. M., Vafaei, A., Al Shaikh, H., Mondol, N. N., Rahman, F., ... & Farahmandi, F. (2022). Fuzz, penetration, and ai testing for soc security verification: Challenges and solutions. Cryptology ePrint Archive.
14. Yang, M., Wang, Y., Shi, J., & Tong, L. (2025). Reinforcement Learning Based Multi-Stage Ad Sorting and Personalized Recommendation System Design.
15. Winderix, H., Piessens, F., & Daniel, L. A. (2024). Efficient Enforcement Mechanisms for the Preservation of Control-Flow Confidentiality.
16. Peng, H., Jin, X., Huang, Q., & Liu, S. (2025). E-commerce Intelligent Recommendation Optimization and Personalized Marketing Strategy Based on Big Model.
17. Bhushan, R. C., & Yadav, D. K. (2022). A survey on formal verification of separation kernels. Recent Advances in Computer Science and Communications (Formerly: Recent Patents on Computer Science), 15(6), 832-850.

 doi:10.20944/preprints202602.0447.v1

18. Buchhold, M., Mueller, T., & Diehl, S. (2022). Revealing measurement-induced phase transitions by pre-selection. arXiv preprint arXiv:2208.10506.
19. Du, Y. (2025). Research on Digital Quality Traceability System for Temperature-Controlled Supply Chain of Foreign Trade Wine Driven by Blockchain and IoT. Business and Social Sciences Proceedings, 4, 57-65.
20. Ducasse, Q. (2024). Hardware security for just-in-time compilation in language virtual machines (Doctoral dissertation, ENSTA Bretagne-École nationale supérieure de techniques avancées Bretagne).
21. Liu, S., Feng, H., & Liu, X. (2025). A Study on the Mechanism of Generative Design Tools' Impact on Visual Language Reconstruction: An Interactive Analysis of Semantic Mapping and User Cognition. Authorea Preprints.