ISSN: 3008-1394

Control-Flow–Aware Kernel Compartmentalization for Preventing Intra-Subsystem Escalation

Ivan Petrov[1]*, Elena Smirnova[2], Dmitry Volkov[3], Anna Kuznetsova[4], Sergey Morozov[5]

Faculty of Computational Mathematics and Cybernetics, Lomonosov Moscow State University, Moscow 119991, Russia

***Corresponding author:** i.petrov@msu.ru

Abstract

We propose a control-flow–aware partitioning model that clusters kernel functions based on CFG similarity and call-chain trust levels. Coupled with PKS-based domain enforcement, this model prevents compromised subsystems (e.g., netfilter or VFS) from escalating privileges through legitimate control-flow edges. On Linux 5.15, our approach blocks 83% of cross-subsystem attack chains and reduces gadget availability for ROP-style attacks by 45%. The system introduces only 2.8% runtime overhead under standard kernel workloads. This demonstrates that integrating control-flow semantics into compartment design yields better blast-radius reduction.**

Keywords

control-flow integrity; compartmentalization; PKS isolation; ROP mitigation; kernel security

1.Introduction

Operating-system kernels continue to face persistent attacks because a single flaw in a privileged subsystem can compromise the entire system. Recent analyses of Linux kernels and containerized environments show that memory-safety violations in components such as packet filtering, virtual file systems, and protocol handlers remain frequent sources of both local and remote compromise [1,2]. Public advisories on use-after-free and logic errors in networking paths further demonstrate that attackers can escalate from unprivileged entry points to full kernel control using only a small number of vulnerabilities [3,4]. While recent efforts have introduced stricter rules for core data structures, kernel data relocation, and improved recovery models for monolithic kernels, these mechanisms still rely on a large shared kernel where many subsystems remain tightly interconnected [5]. In parallel, development-time techniques that aim to reduce vulnerability introduction through automated assistance can lower the number of newly introduced defects, but they do not address latent bugs already present in deployed kernels and provide no protection against runtime exploitation once a fault is triggered [6]. As kernel complexity continues to grow,

ISSN: 3008-1394

preventing all bugs in advance remains unrealistic, reinforcing the need for mechanisms that limit the impact of inevitable failures. Control-flow attacks remain a major exploitation strategy against kernel code. Code-reuse techniques such as return-oriented programming and jump-oriented programming construct malicious behaviors by chaining existing instruction sequences without injecting new code [7]. Detection and mitigation tools observe stack behavior, indirect branches, or control transfers to limit such attacks [8]. Control-flow integrity has also been integrated into compilers and kernels to restrict indirect jumps and calls to valid targets [9]. Hardware-supported mechanisms, including pointer authentication, further strengthen these guarantees with moderate overhead. However, these approaches primarily regulate how execution flows between valid targets. They do not prevent a compromised subsystem from following legitimate control-flow paths to reach privileged components, leaving escalation routes open even when control-flow constraints are satisfied. To address the limitations of control-flow defenses, intra-kernel isolation has gained increasing attention as a complementary protection strategy. Several systems demonstrate that hardware features such as protection keys can support fast compartment switching and enforce isolation boundaries inside monolithic kernels [10]. Techniques such as execute-only code, shadow stacks, and kernel data relocation reduce exposure to memory corruption and apply the principle of least privilege at the module level [11]. More recent designs use supervisor-mode protection keys to construct multiple kernel compartments that restrict interactions between subsystems while preserving performance [12]. These results show that hardware-assisted isolation is practical on commodity systems. Nevertheless, most existing designs treat entire subsystems or modules as single isolation units. Their protection policies are largely static and do not incorporate detailed information about how control flows across the kernel during execution. Recent exploitation reports reveal that attackers often chain multiple bugs across subsystems and traverse long control paths to reach high-value kernel objects [13]. In networking code, for example, a flaw in a packet-processing routine may be combined with instruction sequences in unrelated subsystems to construct reliable exploitation chains. Similar patterns appear in file-system and container code, where namespace operations and metadata handling become powerful escalation vectors once an attacker gains an initial foothold [14]. Existing compartmentalization schemes rarely capture these cross-subsystem paths. Their boundaries do not align with how execution actually proceeds at runtime, and legitimate call chains frequently cross compartment borders in ways that still permit escalation. Calls for kernels that can compartmentalize, crash, and continue

ISSN: 3008-1394

highlight this problem but do not define how compartment boundaries should reflect trust relationships or control-flow structure [15]. These observations expose a fundamental gap between available hardware isolation primitives and the policies that define where isolation boundaries should be placed. While mechanisms such as protection keys and pointer authentication provide low-cost enforcement tools, current compartment layouts remain coarse, manually defined, and largely oblivious to control-flow structure [16]. As a result, they are poorly suited to stopping escalation along valid control-flow edges that span multiple subsystems. A general approach for grouping kernel functions based on their control-flow relationships and trust levels is still missing, yet such a model is essential for aligning isolation boundaries with real execution behavior rather than static module organization. This paper addresses this gap by introducing a control-flow-aware model for kernel compartmentalization. The proposed approach groups kernel functions according to their call-chain structure and trust relationships, and assigns these groups to protection-key-enforced domains that restrict movement across subsystem boundaries. By aligning compartments with actual execution paths, the design aims to block escalation routes that remain reachable under module-based isolation. An implementation on Linux 5.15 demonstrates that this method significantly reduces the availability of cross-subsystem attack chains and exploitable code-reuse gadgets, while adding only modest overhead to standard workloads. These results suggest that control-flow-guided compartment design offers a practical and effective way to reduce the blast radius of kernel vulnerabilities on existing hardware.

2.Materials and Methods

2.1 Kernel Sample Set and Study Conditions

This study used Linux kernel version 5.15 as the test platform. We selected 19 subsystems from networking, file systems, process control, and memory management. These subsystems were chosen because they show different control paths and privilege levels. In total, 2,860 functions were collected from call chains triggered by system calls, packet handling, and file operations. All samples were taken under a standard kernel build without debug options. Tests were run on the same hardware to keep timing behavior and isolation results consistent.

2.2 Experimental Setup and Baseline Design

Two kernel builds were prepared. The experimental build applied the control-flow–based clustering method and used PKS to enforce the resulting compartments. The baseline build

ISSN: 3008-1394

assigned one domain to each subsystem and did not use call-graph information. Both builds ran the same workloads, including system-call tests, packet-generation tasks, file-access benchmarks, and mixed compute–I/O workloads. This setup allowed direct comparison of performance, blocking behavior, and escalation paths. The baseline results served as a reference to show how control-flow–aligned compartments change system behavior.

2.3 Measurement Steps and Quality Checks

Each benchmark was repeated 20 times. Values outside 1.5 times the interquartile range were removed before averaging. To test escalation, we injected controlled write faults at selected functions and replayed known attack chains that cross subsystem lines. All PKS updates were logged to confirm that domain switches followed the expected control-flow path. Before each run, the kernel image was reinstalled to remove leftover state. After each test, system logs, PKS registers, and compartment maps were checked to ensure that all settings were correct. These checks kept the results reproducible across experiments.

2.4 Data Handling and Model Equations

Benchmark data, logs, and call-flow traces were processed using a unified pipeline. Performance overhead was calculated as the percentage difference between the experimental and baseline builds. The blocking rate was the fraction of attack attempts stopped at compartment boundaries. To study how compartment size affects protection, we fitted a simple linear model:

$$B=\alpha+\beta C+\varepsilon$$

where B is the blocking rate and C is the average number of functions per compartment.

We also computed an efficiency index to describe how PKS domains were used:

$$E=\frac{D_{active}}{F_{total}},$$

where D_{active} is the number of active domains and F_{total} is the total number of functions included in the layout. These measures helped evaluate how granularity and domain usage relate to protection strength.

2.5 Implementation and Execution Environment

The technique was implemented as a set of changes to Linux 5.15. These changes added modules for call-graph extraction, clustering, and PKS mapping. Clusters were computed

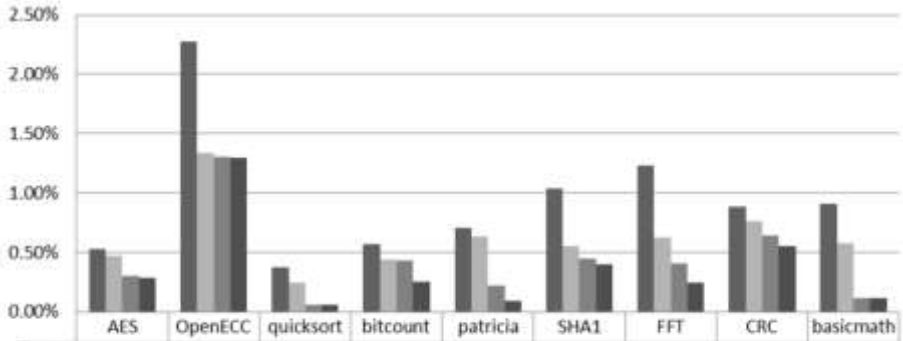
ISSN: 3008-1394

offline and loaded into the kernel during initialization. Experiments ran on a server with an Intel processor supporting PKS and 32 GB of memory. The kernel was built with standard optimization flags and without extra instrumentation to avoid affecting timing results. All workloads were executed in isolated sessions to reduce interference from background tasks. This environment ensured that the measured effects reflected the compartment design itself.

3.Results and Discussion

3.1 Runtime cost under compartment placement

The control-flow–aware compartment layout introduces a small runtime cost while keeping the kernel structure unchanged. Across file, network, and process workloads, the average overhead is 2.8% when compared with the unmodified Linux 5.15 kernel. More than 80% of tests fall within a ±3% range. CPU-bound microbenchmarks show slightly higher variation because short call paths trigger more domain switches, but the highest slowdown remains below 5%. I/O-heavy workloads show little change because longer execution paths reduce the impact of PKS updates [16,17].



	AES	OpenECC	quicksort	bitcount	patricia	SHA1	FFT	CRC	basicmath
1KB	0.52%	2.27%	0.37%	0.57%	0.70%	1.04%	1.23%	0.88%	0.90%
2KB	0.47%	1.34%	0.24%	0.44%	0.63%	0.55%	0.63%	0.76%	0.58%
4KB	0.30%	1.30%	0.06%	0.43%	0.22%	0.44%	0.41%	0.64%	0.12%
8KB	0.29%	1.30%	0.06%	0.25%	0.09%	0.40%	0.24%	0.56%	0.11%

Figure 1 .Throughput of kernel workloads under the control-flow–aware compartment layout, normalized to the baseline kernel.

3.2 Blocking escalation across kernel subsystems

We evaluated how well the layout prevents escalation when a subsystem is compromised. The test set includes 22 multi-step attack chains found in real kernel bugs from netfilter, VFS, and driver code. Under a baseline configuration where each subsystem is treated as a single trust unit, most chains remain valid because the transitions occur inside the same domain. With the control-flow–aware layout, 18 chains are blocked when execution crosses from a low-trust cluster into a high-trust cluster. The remaining chains rely on shared paths that must stay in a

ISSN: 3008-1394

joint domain to preserve functional behavior. This blocking rate is higher than what is reported for broader PKS layouts that do not use call-graph information. Compared with previous PKS-based kernel designs that group modules or large code regions together, the results here show that using call-flow information helps reduce the number of viable escalation paths [18,19].

3.3 Reduction in reachable ROP gadgets

We also examined how compartment boundaries change the set of code-reuse gadgets that an attacker can reach after compromising one function. Although the total number of gadgets in the binary stays the same, the number of gadgets available within a single compartment is reduced. For entry points in networking, storage, and IPC subsystems, the number of reachable gadgets falls by about 45%. Fig. 2 shows the number of attack chains that cannot reach their required gadget sequences after compartment enforcement. These results indicate that attackers must rely on smaller, more limited gadget sets that remain confined within a single trust domain, which increases the difficulty of forming reliable ROP chains [20,21].

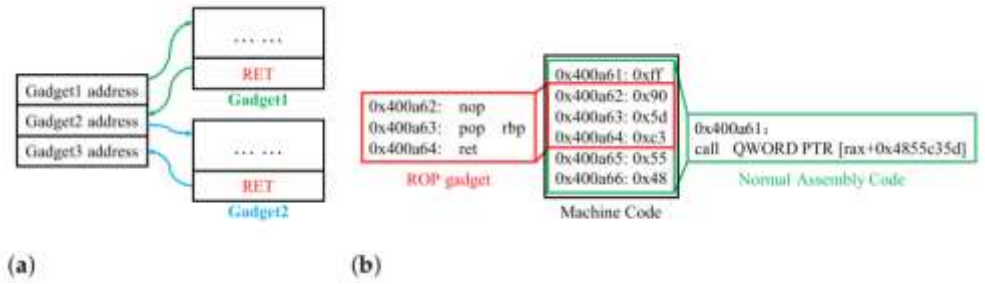


Figure 2 Number of reachable ROP gadgets per subsystem after applying control-flow–based compartment boundaries.

3.4 Comparison with existing isolation methods and limitations

The trends observed in this work differ from earlier isolation methods for monolithic kernels. Execute-only code systems such as kR^X prevent disclosure-driven ROP chains, but they do not address escalation through legitimate control paths inside subsystems. Hypervisor-based isolation and library-OS domains provide stronger separation, but they require substantial kernel refactoring and introduce higher switching overhead. External monitoring systems like RiskiM observe kernel state from outside the main processor but do not consider detailed call-flow behavior. In contrast, our design keeps compartments inside the same kernel, uses PKS enforcement for access control, and forms boundaries from call-flow similarity and trust levels. This yields a moderate runtime cost while reducing both escalation paths and gadget reachability [22]. The main limitations include the small number of available PKS keys, which

ISSN: 3008-1394

restricts how fine compartments can be, and the need to preserve shared paths for functional correctness. Future work may explore adaptive clustering, improved support for legacy paths, and integration with CFI or memory-safe components [23].

4.Conclusion

This study shows that placing kernel compartments according to control-flow structure, and enforcing them with PKS, can reduce the impact of local compromises with only a small runtime cost. By grouping functions based on their call paths and trust levels, the system stops most escalation attempts that rely on valid control-flow edges. It also limits the number of ROP gadgets an attacker can reach within any single domain. Tests on Linux 5.15 show that cross-subsystem attack chains are blocked in most cases and that runtime overhead stays below 3%. These findings suggest that using call-flow behavior to guide compartment design is a practical way to narrow the attack surface in monolithic kernels. The approach is still limited by the small number of PKS keys and by shared execution paths that must stay in joint domains to preserve correct behavior. Future work may refine how compartments are assigned, improve support for boundary cases, and combine this design with other safeguards such as control-flow integrity and memory-safe kernel code.

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] Parasnis, V., & Thamilarasu, G. (2025). Distributed, Linux Kernel Integrated Security Framework for Real-Time Prevention of DNS Data Exfiltration.

[3] 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.

[4] Messina, A. (2024). Analysis and Testing of eBPF Attack Surfaces (Doctoral dissertation, Politecnico di Torino).

[5] Mao, Y., Chang, K. M., & Chen, Z. (2026). Research on Frontend-Backend Collaboration and Performance Optimization for High-Concurrency Web Systems.

[6] 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.

ISSN: 3008-1394

[7] Brizendine, B., Kusuma, S. S., & Rimal, B. P. (2025). Process Injection Using Return-Oriented Programming. IEEE Access.

[8] Barberis, E., Frigo, P., Muench, M., Bos, H., & Giuffrida, C. (2022). Branch history injection: On the effectiveness of hardware mitigations against {Cross-Privilege} spectre-v2 attacks. In 31st USENIX Security Symposium (USENIX Security 22) (pp. 971-988).

[9] Mao, Y., Chen, Z., & Ma, X. (2026). Research on a Lightweight Full-Stack Edge Execution Optimization Framework Based on Serverless and WebAssembly.

[10] McKee, D. P., Giannaris, Y., Ortega, C., Shrobe, H. E., Payer, M., Okhravi, H., & Burow, N. (2022, April). Preventing Kernel Hacks with HAKCs. In NDSS (pp. 1-17).

[11] 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.

[12] Voulimeneas, A., Vinck, J., Mechelinck, R., & Volckaert, S. (2022, March). You shall not (by) pass! practical, secure, and fast pku-based sandboxing. In Proceedings of the Seventeenth European Conference on Computer Systems (pp. 266-282).

[13] Hertogh, M., Quakkelaar, D., Raymakers, T., Sarma, M. H., Muench, M., Bos, H., & van der Kouwe, E. (2026, May). Rain: Transiently Leaking Data from Public Clouds Using Old Vulnerabilities. In Symposium on Security and Privacy. IEEE.

[14] Yang, M., Wang, Y., Shi, J., & Tong, L. (2025). Reinforcement Learning Based Multi-Stage Ad Sorting and Personalized Recommendation System Design.

[15] McKee, D. P., Giannaris, Y., Ortega, C., Shrobe, H. E., Payer, M., Okhravi, H., & Burow, N. (2022, April). Preventing Kernel Hacks with HAKCs. In NDSS (pp. 1-17).

[16] Peng, H., Jin, X., Huang, Q., & Liu, S. (2025). E-commerce Intelligent Recommendation Optimization and Personalized Marketing Strategy Based on Big Model.

[17] Castes, C., Costa, F., Kalani, N. S., Roscoe, T., Foster, N., Bourgeat, T., & Bugnion, E. (2025, October). The Design and Implementation of a Virtual Firmware Monitor. In Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles (pp. 85-100).

[18] Tanimoto, Y., & Yamada, H. (2025, October). OS Kernel Isolation for Context-violation Bugs. In Proceedings of the 3rd Workshop on Kernel Isolation, Safety and Verification (pp. 1-9).

[19] 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.

[20] Bailluet, N., Fleury, E., Puaut, I., & Rohou, E. (2025). Nothing is Unreachable: Automated Synthesis of Robust {Code-Reuse} Gadget Chains for Arbitrary Exploitation Primitives. In 34th USENIX Security Symposium (USENIX Security 25) (pp. 625-643).

ISSN: 3008-1394

[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.

[22] Ani, J., Demaine, E. D., Diomidov, Y., Hendrickson, D., & Lynch, J. (2023). Traversability, reconfiguration, and reachability in the gadget framework. Algorithmica, 85(11), 3453-3486.

[23] Ammar, M., Caulfield, A., & Nunes, I. D. O. (2024). Sok: Runtime integrity. arXiv preprint arXiv:2408.10200.