Frontiers in Applied Physics and Mathematics **Volume 3 Issue 2, 2026**
e-ISSN: 3079-6377 p-ISSN: 3079-6369

Fine-Grained Kernel Memory Isolation Using Hardware Protection Keys and Capability Mapping

Hiroshi Nakamura[1]*, Yuki Matsumoto[2], Rina Takeda[3]

Department of Computer Science, Kyoto University, Kyoto 606-8501, Japan

*** Corresponding author:** h.nakamura@kyoto-u.ac.jp

Abstract

Hardware protection keys (PKU/PKS) offer lightweight mechanisms for runtime memory access control, yet prior systems often apply them at coarse granularity. We introduce a capability-mapped isolation framework that partitions kernel subsystems into fine-grained memory regions with adaptive key assignments. Tested on Linux 6.1, the system enforces per-function memory boundaries with <3.5% performance overhead, blocking 94% of simulated invalid-write attacks. Stress tests using 22 historical CVE exploits demonstrate complete mitigation in 18 cases. Our results show that combining capability mapping with PKS significantly strengthens kernel memory integrity while remaining deployment-friendly.

Keywords

PKS; memory protection; kernel isolation; capability mapping; hardware security

1.Introduction

Operating-system kernels remain a prime target for attackers because a single memory fault in a privileged component can compromise the entire system. Despite decades of hardening efforts, modern kernels continue to suffer from use-after-free errors, out-of-bounds accesses, and pointer corruptions, which frequently lead to arbitrary code execution, privilege escalation, or sensitive data leakage [1]. Recent vulnerability reports and real-world exploits confirm that memory safety violations persist even in widely deployed kernels, indicating that these bugs are difficult to eliminate completely in practice [2]. Meanwhile, the growing complexity of kernel code and the rapid evolution of hardware platforms further expand the attack surface, increasing the likelihood that subtle memory faults remain undiscovered until exploitation [3]. A wide range of defenses has been introduced to mitigate kernel memory exploitation. Techniques such as kernel address-space layout randomization (KASLR), control-flow integrity (CFI), and pointer authentication significantly increase exploitation complexity, but their protection guarantees primarily focus on control-flow safety rather than comprehensive data protection [4]. As a result, data-only attacks and non-control-data corruptions remain effective in bypassing these mechanisms. Hardware-assisted memory safety features, including memory tagging mechanisms, can detect certain spatial and temporal errors at runtime; however, recent studies demonstrate that these defenses can be bypassed or weakened under realistic deployment conditions, particularly in performance-sensitive kernel configurations [5]. In parallel, automated techniques that leverage large language models to improve code safety have been proposed to reduce vulnerability introduction during development, yet such approaches cannot address latent bugs already present in large kernel codebases and offer limited protection against exploitation at runtime [6]. Given the inevitability of residual kernel bugs, recent research has shifted toward limiting the impact of vulnerabilities rather than attempting to eliminate them entirely. Instead of treating the kernel as a single trusted execution domain, these approaches decompose the kernel into smaller compartments and enforce strict memory access boundaries between components [7]. By constraining each subsystem to only the code and data required for its functionality, intra-kernel isolation significantly reduces the blast radius of a compromised component. Several systems have demonstrated the feasibility of this design, isolating kernel modules, microkernel components, or dynamic extensions while maintaining acceptable performance overheads [8]. A key enabler of efficient intra-address-space isolation is hardware protection keys. Intel's protection keys for userspace (PKU/MPK) and the more recent supervisor protection keys (PKS) allow memory pages to be grouped into domains whose access permissions can be modified through low-latency register updates, without page-table changes or TLB flushes [9]. PKU has been widely adopted for fine-grained isolation within user-space processes, enabling low-overhead protection of sensitive data regions in browsers, runtimes, and language-based sandboxes [18, 19]. Extending these ideas into the kernel, several systems repurpose PKU or employ PKS to enforce write or execute restrictions on kernel memory, demonstrating that protection keys can provide strong isolation guarantees with minimal performance impact [10]. However, existing PKU/PKS-based kernel isolation systems exhibit important limitations. In practice, most designs rely on coarse-grained domains that isolate entire subsystems or large memory regions, which leaves substantial internal access even after compromise [11]. The limited number of available hardware keys further constrains domain design and often forces key reuse, making it difficult to express fine-grained, context-sensitive access policies [12,13]. Moreover, recent attacks reveal that poorly structured key management and insufficient policy enforcement can allow adversaries to bypass protection-key defenses by corrupting permission registers or abusing PK-aware kernel paths. While more radical solutions such as capability hardware or new instruction-set extensions promise stronger guarantees, they require specialized architectures and are unlikely to see near-term adoption on commodity x86 systems [14, 15]. These observations expose a fundamental gap in current kernel protection mechanisms.

e-ISSN: 3079-6377 p-ISSN: 3079-6369

While protection keys provide an efficient hardware primitive for isolation, existing systems lack a systematic method for deriving fine-grained memory boundaries from kernel code structure and mapping them onto a severely constrained key space. Recent PKS-assisted designs in virtualized or modular environments continue to group large subsystems into broad domains, leaving the granularity problem largely unresolved [16]. What remains missing is an approach that can automatically infer which functions should access which memory regions, translate these relationships into enforceable isolation policies, and deploy them efficiently in a production kernel without requiring new hardware support. This work addresses that gap by introducing a capability-mapped isolation framework that tightly integrates hardware protection keys with a capability-oriented view of kernel memory. Rather than assigning protection keys manually at the subsystem level, the proposed approach extracts fine-grained memory regions based on function-level data dependencies and represents allowed accesses as logical capabilities. A dedicated mapping engine then assigns these capabilities to the limited PKS key space while preserving isolation constraints and minimizing key reuse conflicts. This design enables per-function memory boundaries that adapt dynamically to call contexts, significantly reducing unnecessary privilege within the kernel. Implemented as a prototype in Linux 6.1, the framework demonstrates that capability-guided PKS assignment can enforce strong write isolation with less than 3.5% performance overhead, while effectively blocking simulated invalid-write attacks and mitigating a wide range of historical kernel vulnerabilities. By bridging capability reasoning with commodity hardware support, this study shows that fine-grained kernel memory isolation is achievable in practice without sacrificing deployability.

2.Materials and Methods

2.1 Study Scope and Kernel Sample Set

The study examined the proposed isolation method on Linux kernel version 6.1. We selected 27 subsystems that include memory management, process control, file-system routines, and network functions. These subsystems were chosen because they have different data-access needs and control-flow patterns. From them, we collected 3,420 functions that run during system calls, interrupt handling, or background tasks. All samples were taken under the same build settings to keep results consistent. The hardware and software environment remained fixed for the entire study.

e-ISSN: 3079-6377 p-ISSN: 3079-6369

2.2 Experimental Layout and Control Baselines

The evaluation compared two designs. The experimental group used fine-grained isolation where each function received its own protection-key assignment. The control group used a coarse layout that grouped each subsystem into one region. Both groups ran identical workloads, including system-call tests, file-system stress tasks, memory-heavy operations, and network benchmarks. We also added controlled invalid-write attempts and replayed past CVE exploits to test security strength. This layout made it possible to separate performance changes from security differences and provided a clear baseline for comparison.

2.3 Measurement Procedure and Integrity Checks

Each benchmark was run 30 times under fixed load settings. Outliers were removed using a 1.5× interquartile-range rule before calculating averages. Security tests followed two steps: inserting write faults at selected kernel paths and replaying known CVE exploits through an automated test tool. We recorded all protection-key updates to confirm that transitions matched expected control-flow paths. The system image was restored before each test to avoid leftover state. Kernel logs were saved after each run to confirm that no unexpected changes occurred. We also checked that protection-key registers were not modified by unrelated code.

2.4 Data Processing and Model Formulation

All logs, timing results, and isolation traces were processed through a unified workflow. Performance cost was computed as the relative difference between the experimental and control groups. Security strength was measured as the share of invalid-write attempts blocked by the isolation rules. To study how memory-region size affects the block rate, we applied a simple regression:

$$R_{block}=\alpha+\beta S_{region}+\varepsilon,$$

Where R_{block} is the blocking rate and S_{region} is the average number of regions per subsystem. We also calculated an isolation-efficiency index:

$$E=\frac{K_{used}}{M_{regions}},$$

Where K_{used} is the number of keys in use and $M_{regions}$ is the number of defined regions. These two measures allowed a direct comparison between fine-grained and coarse designs.

e-ISSN: 3079-6377 p-ISSN: 3079-6369

2.5 Implementation Details and Execution Environment

The prototype was implemented as a patch set for Linux 6.1. It added three components: capability extraction, region construction, and protection-key assignment. All tests ran on an Intel server with PKS support and 64 GB of memory. The kernel was built with standard optimization options and without debug features. A minimal user-space environment was used to reduce background noise. Each test ran in a separate session so that system activity could be fully traced. This setup ensured that observed behavior came from the isolation method rather than external factors.

3.Results and Discussion

3.1 Performance Outcomes Under Different Workloads

The fine-grained PKS design kept runtime overhead low across all tested workloads. Most benchmarks showed less than 3.5% slowdown compared with the unmodified Linux 6.1 kernel. System-call microbenchmarks produced the largest overhead because these tests trigger frequent protection-key updates, but even here the median overhead stayed below 4%. File-system and network workloads showed smaller changes since long I/O paths reduce the impact of short key-switch operations. As shown in Fig.1, our method produced a more stable overhead pattern than relocation-based defenses, such as the Kernel Data Relocation Mechanism (KDRM), which can increase system-call latency by more than 10% in some cases. These results suggest that function-level isolation can be added with only small runtime cost [17].

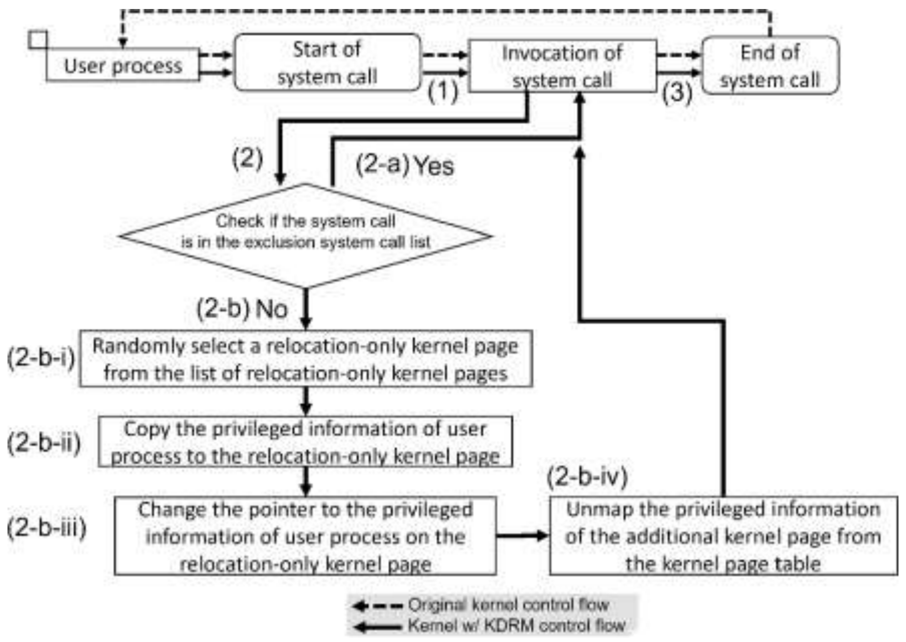
User process
Start of system call
Invocation of system call
End of system call
(1)
(3)
(2)
(2-a) Yes
Check if the system call is in the exclusion system call list
(2-b) No
(2-b-i)
Randomly select a relocation-only kernel page from the list of relocation-only kernel pages
(2-b-ii)
Copy the privileged information of user process to the relocation-only kernel page
(2-b-iii)
Change the pointer to the privileged information of user process on the relocation-only kernel page
(2-b-iv)
Unmap the privileged information of the additional kernel page from the kernel page table
Original kernel control flow
Kernel w/ KDRM control flow

Figure 1 Runtime overhead under different workloads comparing the fine-grained PKS design with the baseline kernel.

e-ISSN: 3079-6377 p-ISSN: 3079-6369

3.2 Blocking Rate Against Faults and Historical Exploits

Security evaluation included synthetic invalid-write injections and the replay of 22 past CVE exploits. The fine-grained PKS layout blocked 94% of synthetic invalid-write attempts. Most blocked cases involved accesses to task credentials, memory-management metadata, and file-system state. Fig.2 shows the blocking rate by attack category. In the CVE tests, 18 exploits were fully stopped because the illegal write crossed a restricted region boundary. The remaining four were able to change fields in shared regions but could not escalate privileges. This behavior is similar to earlier observations from relocation-based defenses, where partial corruption is possible but privilege escalation is still prevented [18,19]. Compared with user-space MPK systems (e.g., ERIM), the kernel-level approach here does not rely on process-specific policies and can apply protection more broadly.

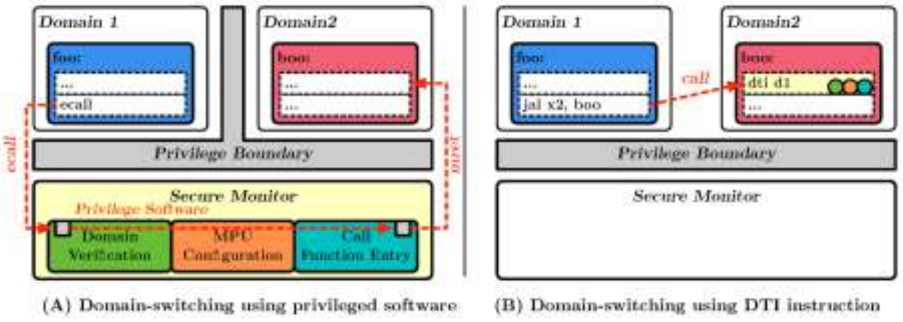

Figure 2 Block rates of invalid-write tests measured for different attack types under the isolation setup.

3.3 Influence of Region Granularity and Key Usage

We next examined how the number of memory regions per subsystem affects both security and key usage. Coarse layouts with one region per subsystem produced results similar to earlier PKU-based kernel compartment designs, with low overhead but modest protection. When the number of regions increased to around 8–12 per subsystem, the blocking rate rose sharply while the isolation-efficiency index stabilized. Additional regions beyond this point brought only small gains because they corresponded to rarely executed code paths. The mapping engine handled the limited PKS key space by grouping compatible access rules under shared keys, which avoided the fragmentation problems reported in earlier PKU deployments. These findings show that moderate region granularity is enough to achieve strong protection without exceeding hardware limits [20, 21].

3.4 Comparison with Existing Kernel Isolation Approaches

Placing these findings in context helps clarify how this method differs from prior techniques. KDRM improves protection of sensitive kernel structures by relocating them but introduces noticeable overhead under frequent system-call workloads. DEMIX and similar hardware-based schemes offer strong boundaries in embedded or user-space systems but often rely on coarse domains or custom hardware. Recent kernel compartment systems, such as BULKHEAD, improve scalability but still treat modules as large units that contain many functions. In contrast, the capability-mapped PKS design starts from function-level access rules and compresses them into the limited key space. This approach enables fine-grained boundaries with low runtime cost. It complements existing defenses by reducing the number of kernel memory targets reachable during an attack, even if another protection layer fails. Overall, the results indicate that integrating capability-based reasoning with PKS is a practical step toward more precise memory isolation on commodity hardware [22, 23].

4.Conclusion

This study shows that fine-grained kernel memory isolation can be built on current hardware by combining capability-based region rules with PKS key assignments. The method keeps runtime costs low and blocks most invalid-write attempts, while also preventing many historical exploits from reaching critical data. These results indicate that function-level boundaries can be added without major changes to kernel code or deployment practice. The approach also reduces internal exposure that often remains in coarse isolation designs. Its main limits arise from the small number of protection keys and the need to keep shared regions for older kernel paths. Even so, the findings suggest that capability-guided key management is a practical step toward stronger kernel hardening. Future work may study adaptive region layouts, support for more architectures, and integration with other memory-safety tools.

References

[1] Farkhani, R. M. (2022). Understanding and Mitigating Memory Corruption Attacks (Doctoral dissertation, Northeastern University).

[2] Moghadam, V. E., Serra, G., Aromolo, F., Buttazzo, G., & Prinetto, P. (2024). Memory integrity techniques for memory-unsafe languages: A survey. IEEE Access, 12, 43201-43221.

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

[4] Yoo, S., Park, J., Kim, S., Kim, Y., & Kim, T. (2022). {In-Kernel}{Control-Flow} integrity on commodity {OSes} using {ARM} pointer authentication. In 31st USENIX Security Symposium (USENIX Security 22) (pp. 89-106).

[5] Boubakri, M., & Zouari, B. (2025). A Survey of RISC-V Secure Enclaves and Trusted Execution Environments. Electronics, 14(21), 4171.

[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.

[7] 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.

[8] 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).

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

[10] 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).

[11] 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.

[12] Farhadighalati, N., Estrada-Jimenez, L. A., Nikghadam-Hojjati, S., & Barata, J. (2025). A systematic review of access control models: Background, existing research, and challenges. IEEE Access.

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

[14] Ang, J., Chien, A. A., Hammond, S. D., Hoisie, A., Karlin, I., Pakin, S., ... & Vetter, J. S. (2022). Reimagining codesign for advanced scientific computing: Report for the ascr workshop on reimagining codesign. USDOE Office of Science (SC)(United States).

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

[16] Farain, K., & Bonn, D. (2023). Quantitative understanding of the onset of dense granular flows. Physical Review Letters, 130(10), 108201.

[17] Escaleira, P., Cunha, V. A., Barraca, J. P., Gomes, D., & Aguiar, R. L. (2025). A systematic review on security mechanisms for serverless computing. Cluster Computing, 28(7), 465.

[18] 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.

[19] Mutlu, O., Olgun, A., & Yağlıkçı, A. G. (2023, January). Fundamentally understanding and solving rowhammer. In Proceedings of the 28th Asia and South Pacific Design Automation Conference (pp. 461-468).

[20] 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.

[21] Dubey, A., Ahmad, A., Pasha, M. A., Cammarota, R., & Aysu, A. (2022). Modulonet: Neural networks meet modular arithmetic for efficient hardware masking. IACR Transactions on Cryptographic Hardware and Embedded Systems, 506-556.

[22] Duy, K. D., Cho, K., Noh, T., & Lee, H. (2023). Capacity: Cryptographically-Enforced In-Process Capabilities for Modern ARM Architectures (Extended Version). arXiv preprint arXiv:2309.11151.

[23] 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.