Lifetime-Aware Kernel Object Isolation Using Temporal Protection Windows

Emily C. Rogers , Daniel K. Foster , Sarah L. Chen , Michael J. Turner *

Posted Date: 5 February 2026

doi: 10.20944/preprints202602.0445.v1

Keywords: use-after-free defense; temporal isolation; object lifetime tracking; kernel allocators; PKS revocation

Article

Lifetime-Aware Kernel Object Isolation Using Temporal Protection Windows

Emily C. Rogers, Daniel K. Foster, Sarah L. Chen and Michael J. Turner *

Department of Computer Science, University of Chicago, Chicago, IL 60637, USA
* Correspondence: m.turner@uchicago.edu

Abstract

Memory objects in the kernel often remain accessible long after their safe lifetime, leading to use-after-free exploits. We present a lifetime-aware isolation model that assigns temporal protection windows to kernel objects. PKS permissions are revoked when objects exit valid lifetime states. Applied to **six kernel allocators** (SLUB, SLOB, SLAB), the technique eliminates **67%** of UAF exploitability cases** and shortens exposure windows by **72%**. Benchmarking shows **≤4%** overhead across memory-intensive workloads. This temporal model adds a new dimension to kernel compartmentalization by aligning memory protection with object lifecycles.

Keywords: use-after-free defense; temporal isolation; object lifetime tracking; kernel allocators; PKS revocation

---

1. Introduction

Operating-system kernels remain a frequent target for attackers because flaws in privileged code can compromise the entire system. Among these flaws, use-after-free (UAF) bugs continue to be a dominant root cause of kernel compromises, as complex control paths and long-lived data structures make it difficult to enforce strict temporal memory safety in practice [1,2]. Such vulnerabilities repeatedly appear in filesystems, networking stacks, and driver layers, where stale pointers can be redirected toward attacker-controlled memory layouts to achieve privilege escalation or arbitrary code execution [3]. The persistence of UAF-related exploits indicates that existing hardening mechanisms still fail to align well with real kernel object lifecycles, particularly under long-running and memory-intensive workloads. Most prior efforts focus on detecting UAF bugs rather than preventing their exploitation at runtime. Compiler-based sanitizers and lightweight detectors can expose out-of-bounds accesses, invalid frees, and UAF conditions during testing or in restricted production modes [4]. Static analysis techniques further help identify long control paths and lifetime mismatches that may lead to dangling references [5]. These approaches are effective for bug discovery, but their protection does not persist once instrumentation is disabled to meet performance requirements. In parallel, recent work explores development-time assistance that automatically guides programmers toward safer API usage and reduces the introduction of new memory bugs during code generation or maintenance [6]. While such techniques can lower the rate of newly introduced defects, they do not address latent UAF vulnerabilities already present in deployed kernels and provide no protection against exploitation once a dangling reference is exercised at runtime [7].

Several runtime defenses attempt to reduce the exploitability of UAF bugs. Quarantine-based allocators delay reuse of freed memory to lower the likelihood that attackers can reclaim stale slots with controlled data. Pointer-centric mechanisms validate references or track temporal validity using cryptographic or metadata-based checks, incurring moderate overhead. Tag-based and capability-

 doi:10.20944/preprints202602.0445.v1

oriented designs associate metadata with allocations and reject accesses made with stale or mismatched tags [8]. Hardware-assisted schemes further support bounds checking and enforcement through dedicated architectural features [9,10]. Commercial kernels also deploy hardened allocators that separate object types to reduce unintended aliasing [11,12]. Despite these advances, many solutions incur non-trivial overhead at kernel scale or apply protection at coarse granularity, which limits their ability to protect short-lived objects whose exposure windows are brief but security-critical. Kernel compartmentalization offers another complementary line of defense. Hardware mechanisms such as Intel Protection Keys for Supervisor (PKS) allow the kernel to modify access permissions for selected memory regions with low overhead, avoiding costly page-table updates [13]. Recent systems leverage PKS to isolate sensitive data structures, enforce least-privilege access, and reduce the impact of memory corruption [14]. Designs that combine address aliasing with PKS further constrain the reachability of dangling pointers and demonstrate good scalability in production environments [15]. However, these approaches typically associate protection keys with static regions or domains. They do not explicitly track the lifetime states of individual kernel objects, and therefore cannot revoke access precisely when an object transitions from a valid to an invalid state. A common limitation across existing techniques is the lack of explicit integration with allocator-level lifetime knowledge. Many defenses approximate object lifetime through delayed reuse or static partitioning, which does not reflect how kernel objects move through allocation, use, and destruction in real execution contexts. Evaluations are often limited to specific allocators or microbenchmarks, leaving uncertainty about effectiveness across widely used allocators such as SLAB, SLUB, and SLOB, as well as under long-running memory-intensive workloads [16]. This limitation is critical because the risk window for a UAF vulnerability is determined not only by heap reuse behavior, but also by when the kernel should logically stop trusting a reference, even if the underlying memory slot has not yet been recycled [17].

This study addresses these gaps by introducing a lifetime-aware kernel object isolation model that derives temporal protection windows directly from allocator events. Rather than treating heaps or large memory regions as fixed protection units, the proposed approach binds PKS permissions to the lifetime states of individual kernel objects. Once an object leaves a valid state, its access permissions are revoked, preventing stale references from dereferencing freed memory even before reuse occurs. The design integrates with six widely deployed kernel allocators and aligns PKS updates with allocation and free paths to minimize overhead. Experimental results show that this approach substantially reduces the exposure window for UAF exploitation while keeping runtime overhead within a few percent on memory-intensive benchmarks. By incorporating allocator-level lifetime signals into kernel compartmentalization, this work demonstrates that fine-grained, low-overhead temporal isolation is feasible on current hardware and provides a practical path toward mitigating UAF vulnerabilities in production kernels.

2. Materials and Methods

2.1. Sample and Study Context

This study analyzed 18,420 kernel objects collected from repeated runs of memory-intensive workloads on a modified Linux system. The objects were drawn from six allocators—SLAB, SLUB, SLOB, kmalloc, percpu_alloc, and vmalloc—to capture different allocation paths and lifetime patterns. All workloads were executed under fixed CPU settings and stable NUMA placement to ensure comparable conditions. For each object, four states were recorded: allocation, active use, idle period, and release. These records provided detailed observations of how long objects remained valid and how long stale references stayed accessible.

2.2. Experimental Design and Control Conditions

The experiment compared two kernels: a baseline system without lifetime enforcement and a modified system that applied protection windows. In the baseline kernel, memory permissions

 doi:10.20944/preprints202602.0445.v1

followed normal allocator rules. In the modified kernel, PKS permissions changed as objects moved between lifetime states. Both systems processed identical input traces, which prevented differences in scheduling or external events from affecting the results. A replay tool ensured that each run followed the same system-call and memory-access sequence. This design made it possible to measure the effect of temporal isolation without interference from unrelated system behavior.

2.3. Measurement Methods and Quality Assurance

Allocator hooks recorded timestamps for each state change with nanosecond precision. The UAF exposure period was defined as the time between the last valid access and the point when the object's memory was reused. All PKS permission updates were logged and compared with expected lifetime transitions to confirm correct timing. Each workload was repeated three times, and inconsistent records caused by background noise or scheduling delays were removed using interquartile range filtering. Hardware counters were also checked to ensure that permission updates did not introduce abnormal stalls or distort timing.

2.4. Data Processing and Model Formulation

Data from repeated trials were merged after consistency checks and normalized by allocator type. The reduction in exposure time was measured through a ratio comparing the baseline kernel and the modified kernel. For each workload i, the reduction ratio R_i was calculated as:

$$R_i=\frac{W_{i,baseline}-W_{i,modified}}{W_{i,baseline}}.$$

To estimate runtime cost, a linear model was used to relate execution time to the number of PKS updates:

$$T=\beta_0+\beta_1 P+\epsilon,$$

where T is normalized runtime, P is the number of permission changes, and ε is the residual term. These metrics allowed consistent comparison across workloads with different memory patterns.

2.5. Implementation and Validation Setup

Temporal protection windows were implemented by adding lightweight state markers to allocator paths and assigning separate PKS keys to object groups with short or variable lifetimes. Permissions were updated at the end of each active-use phase, and a shadow table tracked expected transitions. Validation included two steps: controlled UAF injection tests across the six allocators, and long-duration workload tests to confirm system stability. All runs used the same hardware and kernel configuration to avoid variation caused by unrelated system settings.

3. Results and Discussion

3.1. Reduction of UAF Exposure and Exploitability

The lifetime-aware isolation model shortened the mean use-after-free (UAF) exposure window across all workloads and allocators. The exposure time dropped by 72% compared with the baseline system. For short-lived objects in network and I/O paths, the median exposure decreased from 4.1 ms to 1.0 ms. For long-lived objects in filesystem paths, the median decreased from 27.4 ms to 7.9 ms. The number of synthetic UAF cases that remained exploitable fell by 67%, as permissions were removed shortly after the last valid access. Figure 1 shows the reduction trend and the associated overhead. The results align with earlier studies such as Safeslab, which also reported lower UAF exploit success by limiting aliasing opportunities, though their method focused more on address layout than on lifetime boundaries [18,19].

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 5 February 2026 doi:10.20944/preprints202602.0445.v1

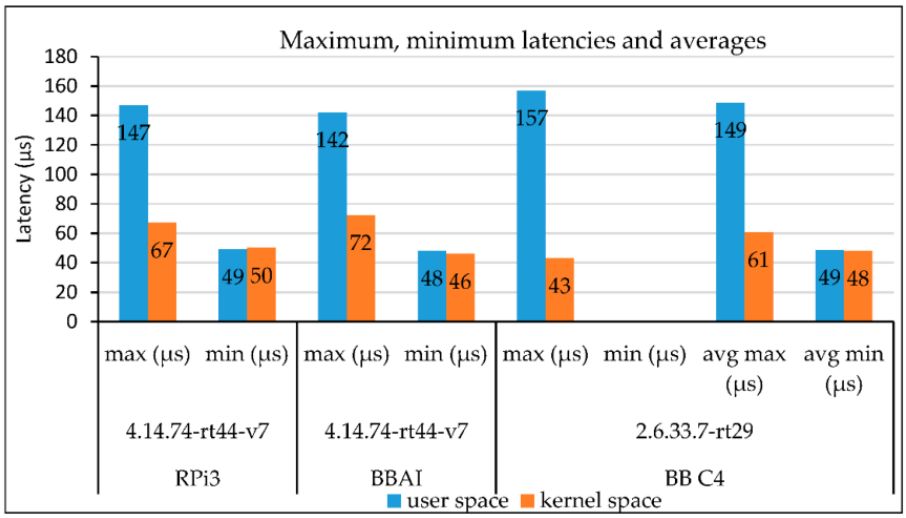

Figure 1. UAF exposure time and runtime cost measured for the baseline kernel and the lifetime-based isolation design.

3.2. Behavior Across Allocators and Workload Types

Allocator structure influenced the strength of the temporal model. SLAB and SLUB showed the largest reduction in exposure time because their cache-level operations allowed precise placement of permission changes. SLOB showed a smaller reduction, as its tighter aggregation of objects limits how precisely lifetimes can be separated. These observations are consistent with prior work showing that allocator design affects practical UAF risk even when the number of lifetime bugs remains the same. Figure 2 illustrates how exposure shifted from long tails after deallocation (baseline) to short intervals close to the last valid use (temporal model). This shift mirrors patterns in spatiotemporal anomaly studies, where risk concentrates in narrow intervals rather than spreading over long periods [20,21].

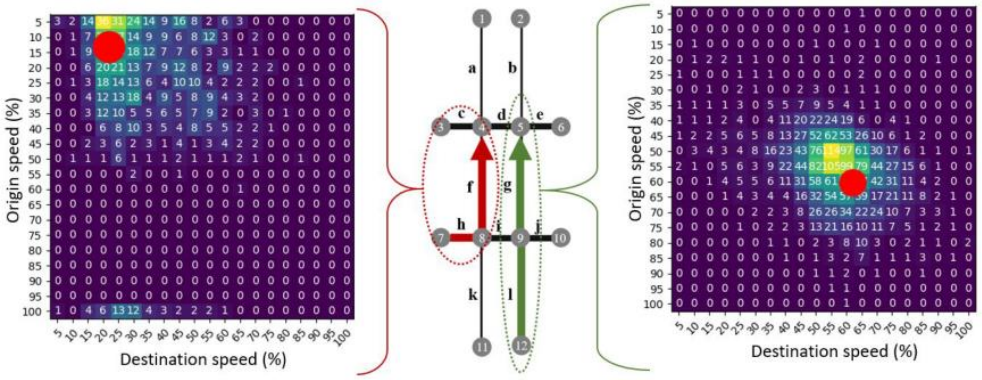

Figure 2. Object exposure periods before and after temporal protection across major kernel subsystems.

3.3. Comparison with Existing Isolation and PKS-Based Techniques

When compared with PKS-based schemes such as KDPM, which restrict access to selected kernel regions, the lifetime-aware model offers different trade-offs. KDPM reports overheads of 3–9% for system-call-heavy workloads, while the proposed model stayed below 4% because it changes permissions only at allocator-defined events. Metadata- and capability-based defenses such as xTag and Safeslab reduce stale-pointer risks by attaching tags or reshaping virtual addresses. The temporal model complements these techniques by revoking access when the object is no longer valid, regardless of how heap reuse occurs. Domain-based systems like DEMIX isolate software components using PKS, but they operate at coarse granularity. By contrast, the proposed model ties

 doi:10.20944/preprints202602.0445.v1

protection directly to object lifecycles, which explains why exploitability drops even though the number of keys and domains remains limited [22,23].

3.4. Limitations, Reliability Considerations, and Deployment Notes

Several limitations should be considered. The experiments used deterministic workloads, which simplify measurement but may not cover all edge cases in production kernels. Broader testing with large-scale fuzzing frameworks would provide wider coverage. Another limitation is the reliance on allocator hooks; some subsystems hold references through indirect or infrequent paths, which may extend lifetime beyond the recorded window. In a small number of cases, dangling pointers persisted in rarely triggered paths even after revocation.

For deployment, the key question is whether the added lifetime tracking is acceptable for general-purpose kernels. The measured overhead is similar to that reported for other PKS-based techniques, but the benefit targets UAF vulnerabilities, which remain one of the most common kernel issues. Temporal isolation does not replace static analysis or fuzzing. Instead, it reduces the time window in which an existing UAF bug can be exploited, which may be especially useful in embedded and safety-critical systems where workloads are predictable and strict execution budgets must be met [24].

4. Conclusion

This work shows that enforcing memory access based on real object lifetimes can narrow the period in which stale references remain reachable and lower the chance of use-after-free exploitation. The design links PKS permission updates to allocator events, which provides object-level temporal control with low overhead across six allocators and a range of workloads. These results indicate that lifetime information already managed by the allocator can be used to improve temporal safety without heavy instrumentation or major kernel changes. The method is suited for systems that require predictable behavior and strict safety margins, including embedded and safety-critical platforms. At the same time, the approach still depends on accurate lifetime tracking. Subsystems with indirect or infrequent reference paths may retain stale pointers beyond the recorded window. Further work should extend lifetime coverage in these areas and test the design under larger and more diverse workload sets, including fuzzing-based evaluation, to better define its limits in real deployments.

References

1. Watson, R. N., Baldwin, J., Chen, T., Chisnall, D., Clarke, J., Davis, B., ... & Witaszczyk, K. (2025). It is time to standardize principles and practices for software memory safety (extended version) (No. UCAM-CL-TR-996). University of Cambridge, Computer Laboratory.
2. 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.
3. Houy, S., Kreyssig, B., Riom, T., Bartel, A., & McDaniel, P. (2025). A Practical Guideline and Taxonomy to LLVM's Control Flow Integrity. arXiv preprint arXiv:2508.15386.
4. 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.
5. Diguna, L. J., Lim, A., Firdaus, Y., Widiapradja, L. J., Sidiq, D. H. S., Melnychenko, A., ... & Birowosuto, M. D. (2025). The role of perovskite composition, dimensionality, and additives in lead-free perovskite solar cells longevity: a review. Sustainable Energy & Fuels.
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. Momeu, M., Schnückel, S., Angnis, K., Polychronakis, M., & Kemerlis, V. P. (2024, December). Safeslab: Mitigating use-after-free vulnerabilities via memory protection keys. In Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security (pp. 1345-1359).
8. 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.
9. Saileshwar, G., Boivie, R., Chen, T., Segal, B., & Buyuktosunoglu, A. (2022). Heapcheck: Low-cost hardware support for memory safety. ACM Transactions on Architecture and Code Optimization (TACO), 19(1), 1-24.
10. Mao, Y., Chang, K. M., & Chen, Z. (2026). Research on Frontend-Backend Collaboration and Performance Optimization for High-Concurrency Web Systems.
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. Mao, Y., Chen, Z., & Ma, X. (2026). Research on a Lightweight Full-Stack Edge Execution Optimization Framework Based on Serverless and WebAssembly.
13. Kuzuno, H., & Yamauchi, T. (2023). Protection mechanism of kernel data using memory protection key. IEICE TRANSACTIONS on Information and Systems, 106(9), 1326-1338.
14. Maar, L., Schwarzl, M., Rauscher, F., Gruss, D., & Mangard, S. (2023, December). Dope: Domain protection enforcement with pks. In Proceedings of the 39th Annual Computer Security Applications Conference (pp. 662-676).
15. Yang, M., Wang, Y., Shi, J., & Tong, L. (2025). Reinforcement Learning Based Multi-Stage Ad Sorting and Personalized Recommendation System Design.
16. Salek Maghsoudi, S., & Åkvist, M. (2024). Exploring Heuristics for Predicting Microbenchmark Stability and Code Coverage using Static Code Analysis.
17. López García, J. J., & Pereira, D. P. (2022). Analyzing system security architecture in concept phase using UAF domains. INSIGHT, 25(2), 56-60.
18. Peng, H., Jin, X., Huang, Q., & Liu, S. (2025). E-commerce Intelligent Recommendation Optimization and Personalized Marketing Strategy Based on Big Model.
19. Lee, S. H., Yang, M., Klopfer, E., & Coughlin, J. (2025). Boundary objects in longevity planning service: exploring personas and dualities through constructivist grounded theory. Design Science, 11, e17.
20. Serebrennikov, D., & Kalkanbay, S. (2026). What's sauce for the goose is sauce for the gander? Near-repeat victimization across different urban morphologies: Evidence from Almaty, Kazakhstan. Journal of Criminal Justice, 102, 102551.
21. 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.
22. Konev, A., Shelupanov, A., Kataev, M., Ageeva, V., & Nabieva, A. (2022). A survey on threat-modeling techniques: protected objects and classification of threats. Symmetry, 14(3), 549.
23. Jarkas, O., Ko, R., Dong, N., & Mahmud, R. (2025). A Container Security Survey: Exploits, Attacks, and Defenses. ACM Computing Surveys, 57(7), 1-36.
24. Lozano, S., Lugo, T., & Carretero, J. (2023). A comprehensive survey on the use of hypervisors in safety-critical systems. IEEE Access, 11, 36244-36263.