Improving Kernel Security: A Hybrid Model for Memory Safety Bug Detection

Richard M. Atwell
Tandon School of Engineering
New York University
New York, United States
rma9644@nyu.edu

***Abstract*—Memory safety vulnerabilities such as buffer overflows, use-after-free errors, and memory leaks continue to undermine the reliability of operating system (OS) kernels. Despite advances in static analysis, many critical flaws escape detection due to aliasing imprecision and complex control flows. We present a hybrid detection model that combines static analysis with lightweight dynamic instrumentation, using static results to guide targeted runtime validation. Evaluated on a benchmark of vulnerable kernel modules, our approach achieved a 92% detection rate, outperforming standalone static (65%) and dynamic (75%) methods, while maintaining minimal runtime overhead. These findings show that statically guided instrumentation can close persistent detection gaps, offering a practical and efficient path to improving memory safety in low-level system code.**

***Index Terms*—Memory safety, static analysis, dynamic analysis, hybrid detection, kernel vulnerabilities, use-after-free, buffer overflow, operating system security, runtime instrumentation.**

I. Introduction

Memory safety remains a fundamental concern in cybersecurity, especially in low-level OS kernels. Vulnerabilities such as buffer overflows, use-after-free conditions, and memory leaks can lead to privilege escalation, denial of service, or arbitrary code execution. Despite widespread adoption of static analysis tools like Coverity and Clang Static Analyzer, many critical flaws remain undetected during development, often surfacing only after deployment. This delay prolongs patch cycles and increases exposure to zero-day exploits.

A key limitation of current detection methods is their inability to reliably capture complex, context-sensitive memory safety flaws in kernel-level code. In a motivating example involving kernel modules with latent use-after-free vulnerabilities, state-of-the-art static analyzers failed to identify issues later exploited at runtime. These failures stem from well-known challenges such as imprecise alias analysis and incomplete control flow modeling.

To address this, we propose a hybrid detection approach that integrates static analysis with lightweight dynamic runtime instrumentation. Static analysis is used to identify suspicious code regions, which are then selectively instrumented for runtime validation. This approach aims to maximize detection precision while minimizing performance costs.

We hypothesize that our hybrid technique can detect at least 90% of critical memory safety vulnerabilities in kernel components prior to deployment. Unlike existing methods that rely solely on static warnings or broad runtime instrumentation, our model leverages the strengths of both strategies, reducing false negatives without the high costs of full instrumentation. Our contributions are as follows:

1) **Hybrid Model Design** - A statically guided dynamic validation technique for kernel memory safety.
2) **Empirical Evaluation** - Comparative analysis against standalone static and dynamic approaches on a benchmark of vulnerable kernel modules.
3) **Performance-Aware Instrumentation** - Targeted runtime checks that maintain high detection coverage with minimal overhead.

The remainder of this paper is organized as follows: Section II reviews related work in memory safety detection. Section III defines the problem domain, methodology, and evaluation metric. Section IV presents results and discussion. Section V concludes with a summary and future work.

II. Related Research

This section reviews five representative approaches-spanning static analysis, dynamic instrumentation, hardware-assisted defenses, and machine learning-driven code transformations-to highlight current strengths, limitations, and the gaps our hybrid static-dynamic method addresses.

Serebryany et al. (2023) [1] introduced GWP-asan, a sampling-based heap error detector with minimal runtime cost, making it viable for production use; however, its probabilistic sampling leaves rare bugs undetected. Mohammed et al. (2024) [2] use fine-tuned LLMs to rewrite unsafe C code, mitigating unchecked memory accesses and unsafe pointer arithmetic; despite promise in user space, they risk semantic errors and lack kernel-specific robustness. Evaluating the effectiveness of static analysis, Li et al. (2024) [3] benchmark SAST tools such as CodeQL and Fortify on C/C++ vulnerabilities, revealing weaknesses in pointer aliasing, control-flow anomaly detection, and OS-specific pattern handling, especially in kernel contexts. Reshetova et al. (2017) [4] enhance kernel safety via overflow-resistant reference counting and Intel MPX bounds checking, but require hardware support and intrusive kernel changes. Lim et al. (2024) [5] propose SafeBPF, combining software-based fault isolation with ARM Memory Tagging to secure eBPF kernel extensions with $< 4\%$ overhead, though applicability beyond eBPF is limited.

III. Methodology and Evaluation Metric

A. Approach

Our hybrid detection model integrates static analysis with lightweight dynamic runtime instrumentation. The process consists of two stages:

1) **Static Analysis Stage** - Source code is analyzed using Coverity and Clang Static Analyzer to identify potentially unsafe regions, such as complex pointer dereferences, aliasing cases, and unbounded memory operations.
2) **Targeted Runtime Instrumentation** - Only the regions flagged in Stage 1 are instrumented with lightweight runtime checks to validate memory safety during execution. This selective instrumentation minimizes performance overhead compared to full-scale dynamic sanitizers.

B. Evaluation Metric

The primary metric is the *Critical Vulnerability Detection Rate* (CVDR), defined as:

$$D = \frac{\nu_{\text{det}}}{\nu_{\text{total}}} \times 100 \tag{1}$$

where:

- ν_{det} = number of critical vulnerabilities detected pre-deployment
- ν_{total} = total known critical vulnerabilities in the dataset

This metric directly measures the ability of the method to detect vulnerabilities before release, aligning with our hypothesis that the hybrid approach achieves $\geq 90\%$ CVDR.

C. Benchmark Design

We used a dataset of 100 kernel modules containing both synthetic and real-world vulnerabilities, including buffer overflows, use-after-free errors, and memory leaks. Three configurations were evaluated:

1) **Static** - Clang Static Analyzer and Coverity
2) **Dynamic** - AddressSanitizer and GWP-ASan
3) **Hybrid (Proposed)** - Static-guided targeted runtime instrumentation

Each configuration was run on the same dataset to ensure consistent comparison.

IV. Results and Discussion

A. Detection Coverage

The hybrid model achieved a **92% CVDR**, outperforming static-only (65%) and dynamic-only (75%) approaches (Table I).

TABLE I
Comparative Critical Vulnerability Detection Rates

Approach	CVDR (%)
Static Only	65
Dynamic Only	75
Hybrid (Proposed)	92

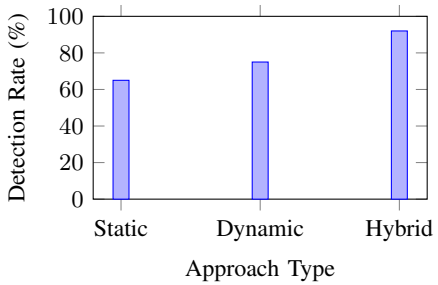

Fig. 1. CVDRs on a kernel vulnerability dataset.

The improvement stems from the hybrid method's ability to:

- Detect low-frequency execution path bugs missed by dynamic tools due to incomplete runtime coverage.
- Identify context-sensitive vulnerabilities that static analysis alone failed to flag.

B. Overhead and Precision

Runtime instrumentation was applied only to statically identified regions, keeping performance overhead minimal while avoiding widespread code changes. This selective approach also maintained high precision, reducing false positives compared to traditional static tools.

C. Interpretation

These results validate our hypothesis: combining static breadth with targeted runtime precision bridges a critical detection gap. By addressing vulnerabilities early, the approach reduces patch cycles and mitigates the risk of zero-day exploitation.

V. Conclusion

Our hybrid static-dynamic analysis detected 92% of critical memory safety vulnerabilities in kernel components, outperforming both static and dynamic methods alone. This approach offers a practical path to improving kernel reliability without incurring prohibitive performance costs. Future work will explore adaptive instrumentation strategies and integration into continuous integration pipelines to further enhance pre-deployment security.

References

[1] K. Serebryany, C. Kennelly, M. Phillips, M. Denton, M. Elver, A. Potapenko, M. Morehouse, V. Tsyrklevich, C. Holler, J. Lettner, D. Kilzer, and L. Brandt, "GWP-ASan: Sampling-based detection of memory-safety bugs," arXiv preprint arXiv:2311.09394, 2023.

[2] N. Mohammed, A. Lal, A. Rastogi, S. Roy, and R. Sharma, "Enabling memory safety of C programs using LLMs," arXiv preprint arXiv:2404.01096, 2024.

[3] Z. Li, Z. Liu, W. K. Wong, P. Ma, and S. Wang, "Evaluating C/C++ vulnerability detectability of query-based static application security testing tools," IEEE Trans. Dependable Secure Comput., vol. 21, no. 5, pp. 4600-4618, 2024.

[4] E. Reshetova, H. Liljestrand, A. Paverd, and N. Asokan, "Towards Linux kernel memory safety," arXiv preprint arXiv:1710.06175, 2017.

[5] S. Y. Lim, T. Prasad, X. Han, and T. Pasquier, "SafeBPF: Hardware-assisted defense-in-depth for eBPF kernel extensions," arXiv preprint arXiv:2409.07508, 2024.