Hybrid Taint-Guided Kernel Fuzzing with Selective State Propagation

Arjun Mehta *, Rohan Srinivasan , Neha Kapoor

Posted Date: 31 December 2025

doi: 10.20944/preprints202512.2745.v1

Keywords: taint analysis; hybrid fuzzing; selective propagation; symbolic constraint; kernel security

 doi:10.20944/preprints202512.2745.v1

Article

Hybrid Taint-Guided Kernel Fuzzing with Selective State Propagation

Arjun Mehta [1,*], Rohan Srinivasan [2] and Neha Kapoor [3]

Department of Computer Science and Engineering, Indian Institute of Technology Bombay, Mumbai 400076, India

* Correspondence: arjun.mehta@iitb.ac.in

Abstract

We integrate static taint analysis with dynamic fuzzing to target high-impact kernel code paths. A pruning mechanism removes irrelevant taint propagation, while symbolic constraints are applied only to tainted regions to control overhead. Evaluated on 18 kernel subsystems, the hybrid fuzzer achieves 44% more taint-relevant path hits, identifying 13 bugs, including buffer overflows and pointer dereferences. Symbolic overhead remains limited (≤18%) through selective propagation. This hybrid design efficiently directs fuzzing toward semantically meaningful kernel logic, demonstrating a productive balance of taint tracking and dynamic mutation.

Keywords: taint analysis; hybrid fuzzing; selective propagation; symbolic constraint; kernel security

1. Introduction

Operating-system kernels remain high-risk targets because a single defect can compromise all software layers that depend on them. Despite long-term hardening efforts, recent studies continue to report a growing number of kernel vulnerabilities, particularly in subsystems such as filesystems, networking stacks, and device drivers [1,2]. Coverage-guided greybox fuzzing has become a primary technique for uncovering these flaws at scale, with systems derived from AFL and syzkaller contributing to a steady stream of vulnerability disclosures [3]. However, the kernel's privileged execution context, extensive interface surface, and complex internal state make systematic testing difficult. In many cases, security-critical code is protected not by shallow control-flow checks but by deep state conditions that are hard to satisfy through coverage feedback alone [4].

To improve exploration of state-dependent behavior, recent work has shifted toward kernel-aware fuzzing techniques that model interfaces, state transitions, and environmental constraints more explicitly. Relation-learning approaches construct dependency graphs over system-call sequences to guide fuzzers toward deeper kernel states [5]. Other systems extract or approximate kernel state variables using static analysis or symbolic reasoning, allowing state changes to serve as feedback signals instead of raw code coverage [6]. Additional efforts focus on kernel-specific execution environments and crash handling, for example by combining fuzzing with symbolic execution or refining crash deduplication using stack-trace clustering [7,8]. Related learning-based studies in security analysis further suggest that data-driven models can capture latent structural patterns and guide exploration even when explicit specifications are incomplete or unavailable [9]. Nevertheless, surveys consistently show that most kernel fuzzers still rely primarily on coverage and struggle to reach state-dependent paths where many vulnerabilities reside [10].

Hybrid fuzzing, which combines coverage-guided mutation with symbolic or concolic execution, has been proposed to overcome such limitations. General-purpose hybrid fuzzers use symbolic execution selectively to bypass hard-to-reach branches and then feed generated inputs back into the fuzzing loop [11,12]. While effective in some contexts, their benefits vary widely across workloads, and symbolic execution often becomes prohibitively expensive when applied to large, stateful programs such as kernels [13]. Studies of hybrid systems further note that their success depends

 doi:10.20944/preprints202512.2745.v1

strongly on invocation frequency, path prioritization, and careful coordination between symbolic and mutation-based components [14]. Kernel-oriented hybrids demonstrate similar trade-offs: they can reveal new bugs, but only under strict limits on symbolic exploration and with careful management of kernel execution environments [15].

Taint analysis provides another mechanism for directing exploration toward security-relevant behavior. Static taint analysis has been applied to track how external inputs propagate through kernel data structures, highlighting potential attack surfaces [16]. Dynamic and hybrid taint-guided approaches show that taint information can reduce time spent on irrelevant paths and help distinguish exploitable faults from benign ones [17]. Outside the kernel domain, taint-based guidance has been used to approximate path constraints and reduce reliance on full symbolic reasoning [18]. Recent system-level fuzzers also incorporate taint signals to steer exploration across complex hardware–software interactions [19]. However, in kernel fuzzing, taint analysis is often applied only during triage or exploitability assessment and rarely integrated into the main exploration loop at scale. Several challenges therefore remain unresolved. Coverage-centric feedback is insufficient for routines guarded by complex state variables. Hybrid fuzzers face scalability limits due to symbolic overhead and path explosion, especially on long kernel execution paths. Taint analysis introduces its own practical difficulties: static methods tend to over-approximate in pointer-rich kernel code, while dynamic tracking incurs nontrivial runtime overhead and complicates deployment across kernel versions. As a result, taint-guided fuzzing is typically confined to narrow scenarios and does not systematically guide overall path exploration.

This work investigates how selectively filtered taint information can be integrated into hybrid kernel fuzzing to improve exploration efficiency. We combine subsystem-aware static taint analysis with a dynamic fuzzing loop and introduce a pruning mechanism that removes taint flows unlikely to influence attacker-controlled inputs or key control decisions. Symbolic execution is then applied only to execution segments associated with these pruned taint regions, while the remaining exploration relies on coverage-guided mutation. This design treats symbolic execution as a targeted aid rather than a general search strategy, reducing overhead and limiting path explosion. We evaluate the proposed approach on 18 kernel subsystems and assess its impact on path coverage, bug discovery, and symbolic cost relative to a coverage-only baseline. The results show that selective taint guidance enables the fuzzer to reach more security-relevant paths and uncover memory-safety vulnerabilities while maintaining practical performance, demonstrating a balanced and scalable approach to kernel fuzzing.

2. Materials and Methods

2.1. Sample Scope and Kernel Targets

This study evaluated the taint-guided hybrid fuzzer on 18 Linux kernel subsystems chosen to represent different interface types and state behaviors. The targets included filesystems, memory-management modules, network stacks, device drivers, and IPC components. All subsystems were built from the same kernel version with identical configuration flags to keep testing conditions consistent. For each subsystem, we identified system-call entry points and reachable internal routines that could be affected by external inputs. Test inputs were produced in a controlled environment that varied system-call sequences, argument ranges, and scheduling conditions. Each fuzzing run lasted 48 hours to allow time to reach deeper kernel states while keeping results comparable across subsystems.

2.2. Experimental Design and Control Setup

Three configurations were evaluated: a coverage-guided baseline, a taint-only variant, and the full hybrid fuzzer with selective state propagation. All configurations used the same mutation engine, execution harness, and system-call scheduler so that differences in results could be attributed to taint analysis and hybrid components rather than unrelated factors. In the baseline configuration, no taint

 doi:10.20944/preprints202512.2745.v1

information was used and exploration depended only on coverage growth. In the taint-only variant, static taint summaries were used for seed selection, but symbolic execution was disabled. The full hybrid configuration applied symbolic constraint solving only to branches linked to pruned taint regions. All experiments were run under the same hardware and system load, and each configuration was tested three times to reduce random variation.

2.3. Measurement Procedures and Quality Control

We recorded path coverage, taint-relevant path counts, symbolic execution time, and detected faults during each run. Kernel execution traces were collected with a lightweight instrumentation layer that logged branch decisions, state variables connected to taint sources, and propagation edges. To ensure consistent taint tracking, we compared static taint results with hand-annotated samples from four subsystems and confirmed that major dependency paths were correctly captured. All crashes were processed through a deduplication pipeline that grouped faults by stack-trace similarity and reproduced them under controlled replay. Crashes that could not be reproduced twice were removed from analysis. Quality checks also verified that symbolic solvers received only short path fragments rather than complete execution traces, confirming that selective propagation was applied as intended.

2.4. Data Processing and Model Formulation

Coverage values, execution counts, and taint-propagation metrics were aggregated in 30-minute windows to examine progress over time. Symbolic overhead was calculated as the fraction of runtime spent on constraint solving. To study the relationship between taint-guided exploration and bug detection, we fitted a linear model linking taint-relevant coverage C_t to the number of unique faults B:

$$B=\alpha+\beta C_t+\varepsilon,$$

where α andβ were estimated for each subsystem, and ε represented unexplained variation. We also measured improvement from hybrid execution using the normalized scoreI:

$$I=\frac{P_h-P_b}{P_b},$$

where P_h is the number of taint-relevant paths reached by the hybrid fuzzer and P_b is the corresponding number from the baseline. All statistical analyses were performed using Python tools with fixed random seeds to ensure reproducibility.

2.5. Implementation Details and Execution Environment

All experiments were run on a dedicated machine with a 32-core CPU, 128 GB RAM, and hardware virtualization support. The kernel executed inside a lightweight virtualized environment that allowed quick snapshot restoration after each test input, ensuring that every run started from the same initial state. The hybrid symbolic component used a constraint solver integrated at the basic-block level, which enabled selective solving without reconstructing full paths. Static taint analysis relied on a kernel-aware control-flow graph builder that included pointer aliasing rules adapted to kernel memory layouts. The fuzzing harness isolated subsystems to avoid cross-effects during exploration. All seeds, taint summaries, and crash logs were stored in version-controlled repositories for traceability.

3. Results and Discussion

3.1. Coverage growth and taint-relevant exploration

Across all 18 kernel subsystems, the hybrid fuzzer increased both total branch coverage and the number of taint-relevant executions. Over the 24-hour runs, median edge coverage rose modestly, but the main effect appeared in taint-guided exploration: the hybrid configuration reached about 44%

 doi:10.20944/preprints202512.2745.v1

more taint-related paths than the baseline. Several networking and storage subsystems showed larger gains because their internal states depend on multi-step system-call sequences [20].

A similar trend has been reported in studies that refine input scheduling to avoid early stagnation in greybox fuzzing. The curve in our Fig.1 displays a related pattern: the baseline fuzzer stops discovering new taint-relevant edges early, while the hybrid version continues to expand coverage. This indicates that filtering taint information and applying constraints only at selected points helps the fuzzer use its time more effectively than a purely coverage-driven approach [21].

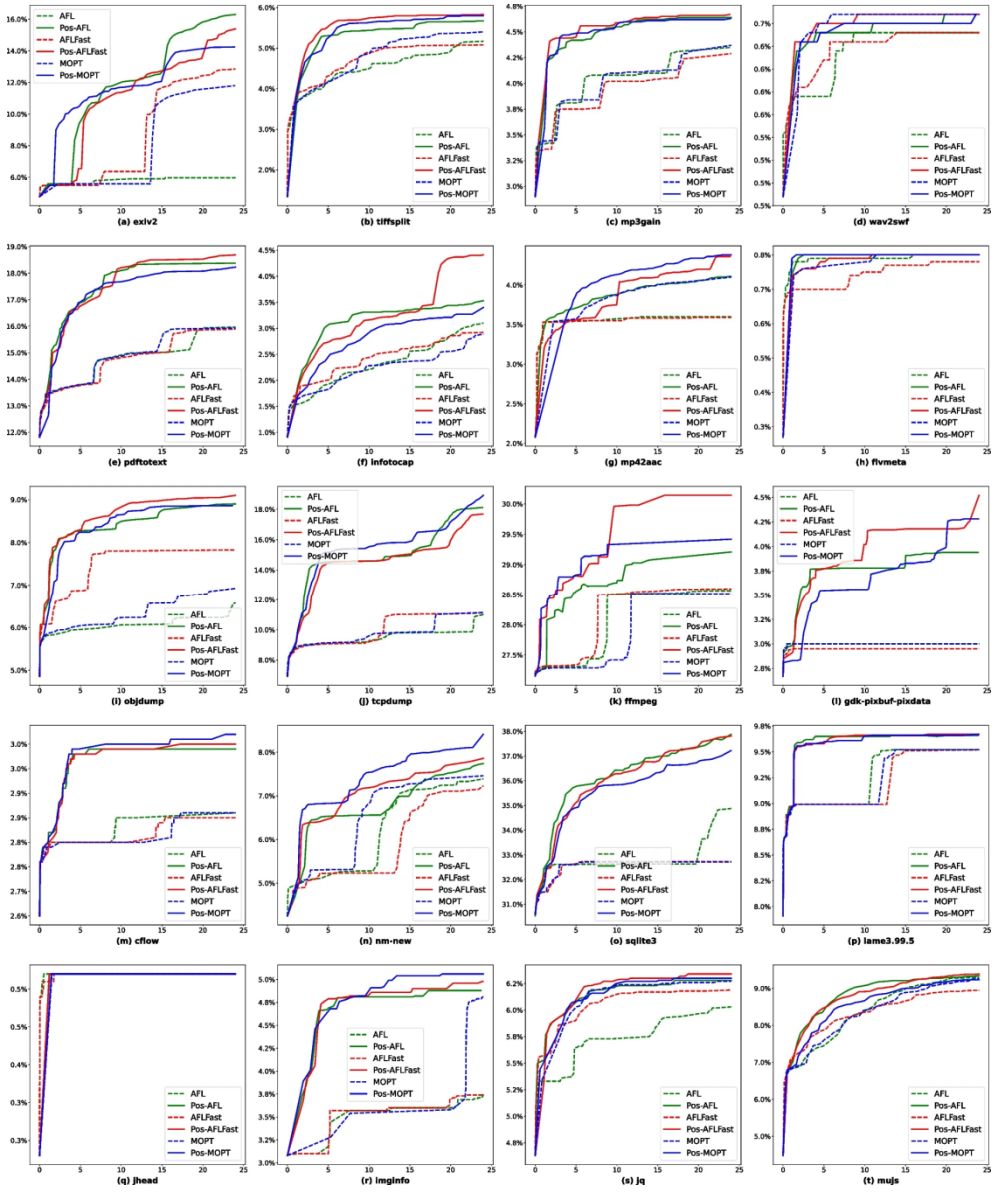

Figure 1. Edge coverage and taint-related path counts of the baseline and hybrid fuzzers over time.

3.2. Bug discovery outcomes and characteristics

The hybrid fuzzer found 13 unique kernel bugs, while the baseline found 7 under identical conditions. The additional bugs mainly came from subsystems where nested pointer structures and multi-step state transitions make unsafe memory operations hard to reach with mutation alone. These included buffer overflows, out-of-bounds reads, and pointer misuse.This observation aligns with prior taint-aware studies. Our Fig.2 shows a similar relationship: subsystems with more taint-relevant coverage also produced more distinct crashes. Unlike methods designed for a single bug

 doi:10.20944/preprints202512.2745.v1

class, our approach uses general taint-based signals, yet still detects a wide range of memory faults, suggesting that taint-guided path selection is broadly useful in kernel settings.

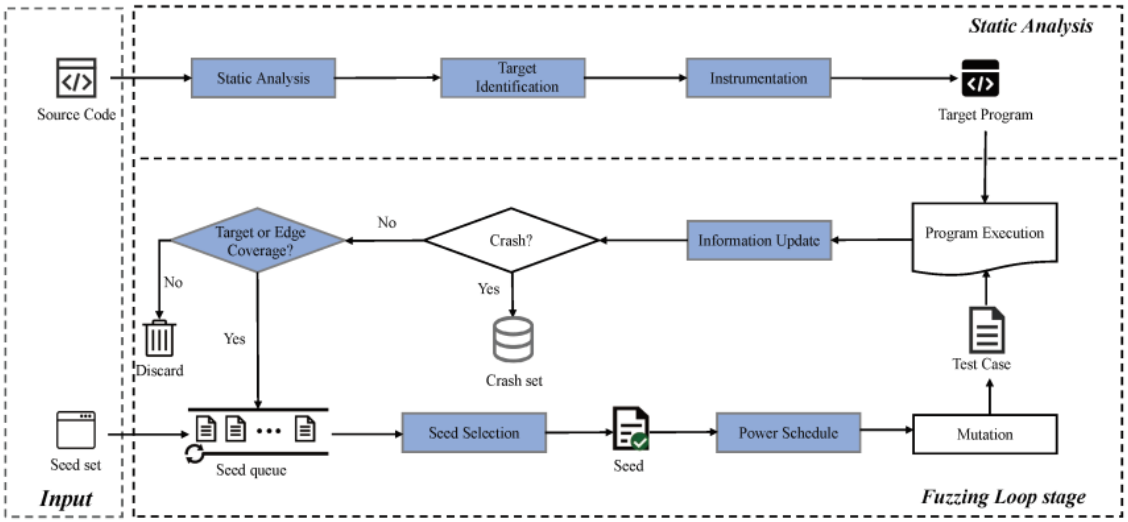

Figure 2. Solver time and unique crash cases observed across kernel subsystems under the hybrid setup.

3.3. Cost of selective symbolic execution and solver overhead

One goal of the design was to constrain solver usage so that symbolic execution would not dominate runtime. In the experiments, symbolic solving accounted for 11–18% of total runtime, depending on subsystem complexity. This level is lower than many hybrid fuzzers that run symbolic execution on full paths and often encounter long delays.

Prior evaluations of taint-driven systems also noted that heavy online analysis can slow execution. For instance, GREYONE and related tools report added runtime cost from extensive data-flow tracking. In our case, the selective propagation mechanism limited solver calls to a narrow set of taint-linked branches. As shown in Fig.2, solver time increases at a slower rate than the number of taint-relevant paths, indicating that pruning rules were effective in keeping constraint solving small relative to overall fuzzing effort.

3.4. Comparison with existing taint-guided and hybrid fuzzers

Compared with coverage-only fuzzers such as syzkaller-based setups, the hybrid method reduces the gap between code coverage and the reachability of deep state-dependent logic. Prior reports, including UAF-focused fuzzers and directed fuzzers, often rely on vulnerability-specific indicators or target-driven heuristics. These methods work well for selected bug types but may not generalize across kernel modules.

Our approach differs by using static taint analysis to identify fields influenced by untrusted input and applying symbolic reasoning only along these paths. This avoids the heavy state tracking found in full symbolic executors and keeps performance closer to greybox fuzzers. At the same time, the method inherits known limits of static taint analysis, such as incomplete alias handling. Broader evaluations—similar to large-scale tests on UNIFUZZ or LAVA-M—will be useful to assess how the method behaves across more kernels and longer campaigns.

4. Conclusion

This study shows that static taint analysis combined with selective symbolic execution can guide fuzzing toward kernel paths that carry higher security relevance. By reducing unnecessary taint flows and applying constraints only where input-dependent behavior is present, the hybrid fuzzer increases taint-related coverage and uncovers more memory-safety issues than the coverage-only

 doi:10.20944/preprints202512.2745.v1

baseline. The added cost of symbolic execution remains controlled, which makes the approach suitable for longer campaigns and for subsystems where deep state interactions are common. These findings suggest that taint-based path selection offers a practical middle ground between fast mutation-driven fuzzing and more precise but expensive symbolic analysis. The method may be useful for large kernel codebases and for testing environments where stateful behaviors limit the reach of greybox fuzzers. However, the approach still depends on the accuracy of static taint analysis and has been tested on a limited set of kernel modules. Further work should improve taint precision, broaden subsystem coverage, and study how the method behaves under larger and more diverse workloads.

References

1. Muñoz, A. (2024). Cracking the core: Hardware vulnerabilities in android devices unveiled. Electronics, 13(21), 4269.
2. 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.
3. Luo, D., Gu, J., Qin, F., Wang, G., & Yao, L. (2020, October). E-seed: Shape-changing interfaces that self drill. In Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology (pp. 45-57).
4. Wu, C., & Chen, H. (2025). Research on system service convergence architecture for AR/VR system.
5. Yang, M., Wu, J., Tong, L., & Shi, J. (2025). Design of Advertisement Creative Optimization and Performance Enhancement System Based on Multimodal Deep Learning.
6. Miné, A. (2017). Tutorial on static inference of numeric invariants by abstract interpretation. Foundations and Trends® in Programming Languages, 4(3-4), 120-372.
7. Feng, H. (2024, October). Design of Intelligent Charging System for Portable Electronic Devices Based on Internet of Things (IoT). In 2024 5th International Conference on Big Data, Artificial Intelligence and Internet of Things Engineering (ICBAIE) (pp. 568-571). IEEE.
8. Fonseca, P., Rodrigues, R., & Brandenburg, B. B. (2014). {SKI}: Exposing Kernel Concurrency Bugs through Systematic Schedule Exploration. In 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14) (pp. 415-431).
9. Bai, W. (2020, August). Phishing website detection based on machine learning algorithm. In 2020 International Conference on Computing and Data Science (CDS) (pp. 293-298). ieee.
10. Touqir, A., Iradat, F., Iqbal, W., Rakib, A., Taskin, N., Jadidbonab, H., & Haas, O. (2025). Systematic exploration of fuzzing in IoT: techniques, vulnerabilities, and open challenges: A. Touqir et al. The Journal of Supercomputing, 81(8), 877.
11. Chen, H., Ning, P., Li, J., & Mao, Y. (2025). Energy Consumption Analysis and Optimization of Speech Algorithms for Intelligent Terminals.
12. Jeong, D. R., Kim, K., Shivakumar, B., Lee, B., & Shin, I. (2019, May). Razzer: Finding kernel race bugs through fuzzing. In 2019 IEEE Symposium on Security and Privacy (SP) (pp. 754-768). IEEE.
13. 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.
14. Aichernig, B. K., Brandl, H., Jöbstl, E., & Krenn, W. (2009, November). Model-based mutation testing of hybrid systems. In International Symposium on Formal Methods for Components and Objects (pp. 228-249). Berlin, Heidelberg: Springer Berlin Heidelberg.
15. Su, X. Vision Recognition and Positioning Optimization of Industrial Robots Based on Deep Learning.
16. Norollah, A., Beitollahi, H., Kazemi, Z., & Fazeli, M. (2022). A security-aware hardware scheduler for modern multi-core systems with hard real-time constraints. Microprocessors and Microsystems, 95, 104716.
17. Wang, Y., & Sayil, S. (2024, July). Soft Error Evaluation and Mitigation in Gate Diffusion Input Circuits. In 2024 IEEE 6th International Conference on Power, Intelligent Computing and Systems (ICPICS) (pp. 121-128). IEEE.

 doi:10.20944/preprints202512.2745.v1

18. Resing, W. C., & Elliott, J. G. (2011). Dynamic testing with tangible electronics: Measuring children's change in strategy use with a series completion task. British Journal of Educational Psychology, 81(4), 579-605.
19. Tan, L., Liu, X., Liu, D., Liu, S., Wu, W., & Jiang, H. (2024, December). An Improved Dung Beetle Optimizer for Random Forest Optimization. In 2024 6th International Conference on Frontier Technologies of Information and Computer (ICFTIC) (pp. 1192-1196). IEEE.
20. Kotenko, I., Gaifulina, D., & Zelichenok, I. (2022). Systematic literature review of security event correlation methods. Ieee Access, 10, 43387-43420.
21. Askar, A., Fleischer, F., Kruegel, C., Vigna, G., & Kim, T. (2025). MALintent: Coverage Guided Intent Fuzzing Framework for Android. In 32nd Annual Network and Distributed System Security Symposium, NDSS (pp. 24-28).