Microarchitectural Feedback-Driven Kernel Fuzzing Using Branch Buffer Telemetry

Marco Rinaldi *, Elena Conti , Giovanni Ferraro

Posted Date: 31 December 2025

doi: 10.20944/preprints202512.2785.v1

Keywords: kernel fuzzing; microarchitectural feedback; branch buffer; LBR; hardware-assisted fuzzing

 doi:10.20944/preprints202512.2785.v1

Article

Microarchitectural Feedback-Driven Kernel Fuzzing Using Branch Buffer Telemetry

Marco Rinaldi *, Elena Conti and Giovanni Ferraro

Department of Electronics, Information and Bioengineering, Politecnico di Milano, 20133 Milan, Italy

* Correspondence: marco.rinaldi@polimi.it

Abstract

Traditional kernel fuzzers rely on coarse-grained coverage metrics that cannot reflect complex microarchitectural behaviors. We present a hardware-assisted fuzzing framework that leverages branch buffer telemetry from modern CPUs (LBR, BTB sampling) to refine fuzzing feedback. A model-based inference algorithm aggregates branch-data patterns to estimate microarchitectural novelty and guides seed prioritization. Experiments on Intel Ice Lake and AMD Zen 3 systems demonstrate 27% improvement in unique path coverage, with 11 newly identified concurrency bugs across filesystem and scheduler subsystems. Compared with coverage-only fuzzing, our method reduces time-to-crash by 46% while keeping overhead below 12%. This work shows microarchitectural-level signals can significantly boost kernel fuzzing's effectiveness.

Keywords: kernel fuzzing; microarchitectural feedback; branch buffer; LBR; hardware-assisted fuzzing

1. Introduction

Operating-system kernels remain a critical attack surface because a single flaw can undermine all software layers built on top of them. Memory-safety errors, logic bugs, and race conditions continue to be discovered in widely deployed kernels such as Linux, BSD variants, and commercial systems, despite extensive hardening efforts. Large-scale testing techniques are therefore essential for identifying such vulnerabilities before exploitation. Coverage-guided greybox fuzzing has become one of the most effective approaches in this space, as it can repeatedly exercise kernel interfaces and prioritize inputs that explore new execution paths [1,2]. Fuzzers derived from AFL and the syzkaller family follow this strategy by mutating system-call sequences and selecting those that extend coverage, leading to a steady stream of vulnerability disclosures [3].

Progress in greybox fuzzing has focused on improving feedback quality and input generation efficiency. Newer fuzzers refine branch-coverage encodings, stabilize mutation strategies, and introduce lightweight learning heuristics to reduce redundant exploration. Kernel-oriented fuzzers further incorporate domain knowledge such as resource lifecycles, syscall rules, and concurrency behavior, while large infrastructures automate long-running campaigns across many machines. Related learning-based studies in other security domains show that models trained on execution data can capture hidden structural patterns even without explicit specifications, providing guidance beyond handcrafted rules [4]. However, most kernel fuzzers still depend on coarse coverage signals derived from instrumentation, which provide limited insight into how execution actually unfolds at the processor level. Hardware-assisted fuzzing attempts to address this limitation by exploiting tracing support in modern CPUs, including Intel Processor Trace and ARM CoreSight. Early systems demonstrated that virtual-machine tracing can scale kernel fuzzing with acceptable overhead, while later work explored specialized hardware support such as trace buffers and branch-history queues to accelerate feedback collection [5,6]. More recent studies revisit how coverage can be reconstructed from hardware traces and highlight persistent trade-offs between tracing cost, feedback granularity,

 doi:10.20944/preprints202512.2785.v1

and scalability. In many designs, rich execution traces are eventually reduced to simple bitmaps, obscuring fine-grained control-flow behavior.

At the same time, research in microarchitecture and side-channel security has produced detailed analyses of branch predictors, branch-target buffers, and related front-end components. These studies show that execution behavior depends not only on static code structure but also on dynamic control-flow history and branch interactions [7,8]. Small changes in execution order can lead to different speculation and misprediction patterns, which are observable through branch-level telemetry. Systems that probe microarchitectural state using fuzzing or dynamic analysis further demonstrate that low-level signals can guide systematic exploration of complex execution states [9]. Nonetheless, such insights are rarely integrated into kernel fuzzing workflows, which continue to treat inputs that follow the same architectural path as equivalent. This disconnect leaves an important gap. Coverage-guided fuzzers determine whether a code region is reached, but largely ignore how the CPU executes it. Inputs that traverse identical basic blocks may still produce different branch histories and timing behavior, which is particularly relevant for concurrency bugs and subtle race conditions [10,11]. Conversely, microarchitectural studies often focus on small programs or controlled experiments and do not connect their findings to large-scale fuzzing campaigns [12]. Even hardware-assisted fuzzers that can access branch-buffer data typically use it only to reconstruct conventional coverage.

Building on these observations, we present a kernel fuzzing approach that treats branch-buffer telemetry as a first-class feedback signal. The method collects Last Branch Record and branch-target buffer samples from modern processors and applies a lightweight inference model to derive a novelty score from observed branch patterns. This score is combined with standard coverage information to guide seed selection during fuzzing. Experiments on filesystem and scheduler subsystems show that branch-buffer–aware feedback exposes execution diversity that coverage-only fuzzers fail to distinguish, enabling deeper exploration and the discovery of additional concurrency-related bugs with low overhead. By linking kernel fuzzing with microarchitectural observations, this work demonstrates that branch-level telemetry can be used practically and at scale to enhance vulnerability discovery on modern CPUs.

2. Materials and Methods

2.1. Sample Set and Study Area

We collected 312 kernel execution samples from two test machines: an Intel Ice Lake server and an AMD Zen 3 workstation. Each sample represents one run triggered by a fixed-length system-call sequence. All kernels were compiled from unmodified source trees to avoid interference from external modules. Tests focused on subsystems that often show timing-related issues, including the virtual filesystem, the block layer, and the scheduler. All runs were carried out under stable thermal conditions, and CPU frequency was fixed to reduce noise from dynamic frequency changes.

2.2. Experimental Design and Control Conditions

We compared a coverage-guided fuzzer with a hardware-assisted variant that adds branch-buffer feedback. Both versions used the same seed set, mutation rules, and execution limits. The baseline fuzzer selected inputs only when they exposed new code paths. The experimental version also selected inputs that showed new branch patterns. Each configuration ran for 72 hours on both hardware platforms. These repeated runs produced independent samples and helped rule out machine-specific effects.

2.3. Measurement Procedure and Quality Checks

Branch-buffer data were recorded using the CPU's Last Branch Record (LBR) stack and periodic sampling of the Branch Target Buffer (BTB). Each execution produced a list of branch pairs and sampled targets, which we converted to ordered branch sequences. Kernel logs, crash reports, and

 doi:10.20944/preprints202512.2785.v1

lock-dependency outputs were collected to confirm concurrency-related failures. Runs with incomplete traces, unstable CPU settings, or unrelated kernel errors were removed. Each experiment was repeated at least three times, and trace checksums were used to verify that repeated runs produced stable outputs.

2.4. Data Processing and Model Equations

Branch-buffer traces were mapped to integer sequences and normalized by length. We then computed two simple scores to describe how different a new execution was from previous runs. The first score measures the average distance between the current trace vector xxx and its nearest known trace y [13]:

$$S_1=\frac{1}{n}\sum_{i=1}^{n}|x_i-y_i|.$$

The second score describes the spread of branch targets within the BTB sample [14]:

$$S_2=\frac{1}{m}\sum_{j=1}^{m}(t_j-\bar{t})^2,$$

where t_j is the j-th target and $\bar{t}$ s the mean. Both scores were scaled to [0,1]and combined with coverage information to guide seed selection. Data processing was carried out in Python and R.

2.5. Analytical Workflow

After computing novelty scores, we examined their influence on seed scheduling, path discovery, and crash detection. Control-flow hashes were used to deduplicate execution paths. Concurrency issues were confirmed by inspecting interleavings in scheduler and lock traces. Execution overhead was measured as the ratio of instrumented runtime to normal runtime. We also compared results across the two CPU types to check whether branch-buffer behavior was consistent across microarchitectures. This workflow allowed each metric to reflect real differences in fuzzer behavior rather than environmental variation.

3. Results and Discussion

3.1. Path Coverage Behaviour on Two CPU Architectures

The microarchitectural-feedback fuzzer produced a steady increase in path diversity across all experiments. On Intel Ice Lake, the fuzzer reached new kernel paths after the baseline had already stopped gaining coverage. A similar pattern appeared on AMD Zen 3, although the rate of new-path discovery differed slightly due to variations in branch prediction hardware. These differences became clearer during filesystem and block-layer tests, which contain deeply nested branches. Figure 1 shows the time-based coverage curves for both fuzzers on Ice Lake. The figure illustrates that once the baseline stops finding new edges, the proposed method continues to explore alternative control-flow patterns that share the same edges but differ in branch order. This behaviour suggests that relying solely on edge coverage can miss execution variations that still matter for exposing kernel defects [15,16].

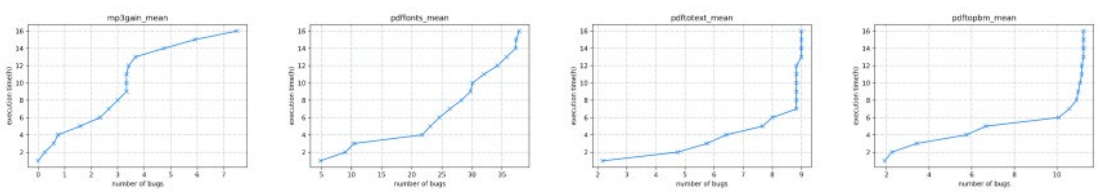

Figure 1. Time-based increase of kernel path coverage for the baseline fuzzer and the microarchitectural-feedback version on the Ice Lake platform.

 doi:10.20944/preprints202512.2785.v1

3.2. Crash Discovery and Concurrency-Related Findings

Crash analysis shows a clear gap between the two fuzzers. The microarchitectural-guided version reached its first fault earlier and produced more independent crash sites within the same time budget. Many of these faults arose from concurrency behaviour rather than simple memory access errors [17]. For example, several scheduler failures appeared only when the fuzzer produced execution orders that stressed wake-up and preemption paths. These cases confirm that identical edge coverage does not imply identical timing behaviour. Compared with prior remedial testing approaches, such as the extended exploration strategy reported in Electronics [18], our method reaches similar conclusions through a different signal: instead of revisiting paths at regular intervals, the fuzzer selects inputs that show unusual branch patterns. This strategy reduces unnecessary retries while still uncovering rare interleavings that the baseline seldom triggers.

3.3. Behaviour of Microarchitectural Features and Novelty Scores

The novelty scores derived from LBR and BTB sampling helped identify which seeds were more likely to expose unusual kernel behaviour. Seeds with higher novelty scores often produced longer chains of indirect branches or more variable branch-target sequences. These traits were stable indicators on both CPU families, even though the absolute values varied due to microarchitectural design differences. Figure 2 provides an illustration of changes in speculative and retired branch counts under different workloads. Seeds showing wider variation in these counters often produced unique execution orders that led to previously unseen crashes. This aligns with earlier observations in systems security research, where performance-counter-based profiles help distinguish normal and exception-driven execution paths [19].

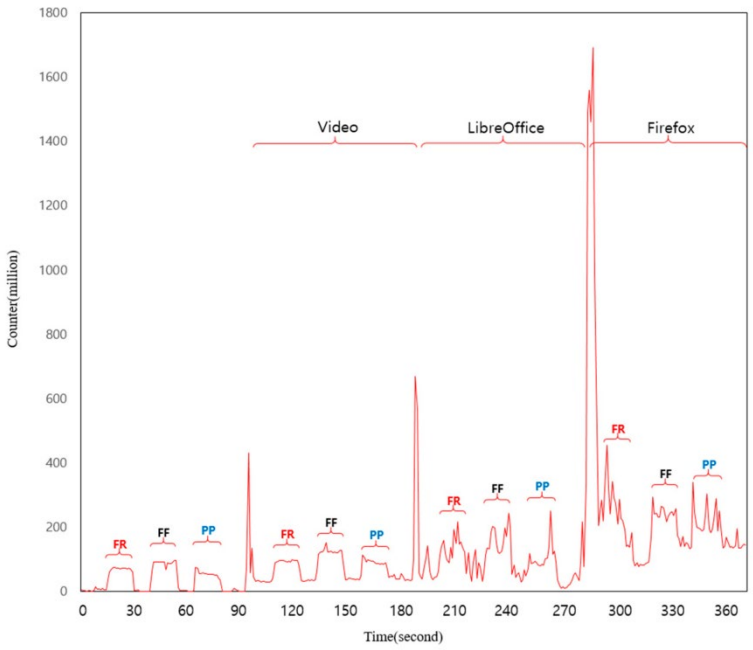

Figure 2. Relation between branch-buffer novelty scores and the number of new kernel paths produced by each seed.

3.4. Comparison with Existing Approaches and Practical Implications

Existing improvements to greybox fuzzing often rely on input-level structure, protocol information, or static analysis. Examples include position-guided scheduling in PosFuzz and re-evaluation strategies for saturated fuzzers [19,20]. These methods operate at the program or input level. In contrast, the present work draws guidance directly from CPU behaviour. This provides a

 doi:10.20944/preprints202512.2785.v1

view of execution that does not depend on source code structure, input format, or predefined grammars. The results show that even simple statistical measures—such as average branch-target variation—help identify inputs worth exploring further. However, the approach also has limits. Branch-buffer sampling differs across microarchitectures, and some subsystems (such as network drivers) showed high novelty scores without yielding additional faults. This indicates that microarchitectural signals should complement, rather than replace, existing guidance strategies [21]. Broader evaluation on embedded processors and hardened kernels will help confirm whether the current findings generalize to more diverse environments.

4. Conclusions

This study shows that microarchitectural signals can guide kernel fuzzing in ways that coverage alone cannot. By using branch-buffer data to describe how executions differ at the branch level, the fuzzer identifies seeds that revisit familiar code paths but produce new timing and ordering behavior. These seeds often lead to crashes that the baseline fuzzer does not reach, especially in subsystems where concurrency plays a central role. Tests on Intel Ice Lake and AMD Zen 3 confirm that this approach increases path variety and reduces the time needed to trigger faults, while keeping tracing overhead low. The method also has clear limits: branch patterns vary across processors, the scoring model is simple, and only a subset of kernel components was examined. Future work may refine the scoring method, evaluate more architectures, and combine hardware-level feedback with input-structure or history-based schedulers. Such extensions could help create fuzzers that respond more closely to the actual behavior of modern processors and improve coverage of timing-sensitive bugs.

References

1. Böhme, M., Pham, V. T., Nguyen, M. D., & Roychoudhury, A. (2017, October). Directed greybox fuzzing. In Proceedings of the 2017 ACM SIGSAC conference on computer and communications security (pp. 2329-2344).
2. Mallissery, S., & Wu, Y. S. (2023). Demystify the fuzzing methods: A comprehensive survey. ACM Computing Surveys, 56(3), 1-38.
3. Beaman, C., Redbourne, M., Mummery, J. D., & Hakak, S. (2022). Fuzzing vulnerability discovery techniques: Survey, challenges and future directions. Computers & Security, 120, 102813.
4. 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.
5. Koschel, J., Borrello, P., D'Elia, D. C., Bos, H., & Giuffrida, C. (2023). Uncontained: Uncovering container confusion in the linux kernel. In 32nd USENIX Security Symposium (USENIX Security 23) (pp. 5055-5072).
6. 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.
7. Sakurai, Y., Watanabe, T., Okuda, T., Akiyama, M., & Mori, T. (2020, September). Discovering HTTPSified phishing websites using the TLS certificates footprints. In 2020 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW) (pp. 522-531). IEEE.
8. Yang, M., Wu, J., Tong, L., & Shi, J. (2025). Design of Advertisement Creative Optimization and Performance Enhancement System Based on Multimodal Deep Learning.
9. Wu, C., & Chen, H. (2025). Research on system service convergence architecture for AR/VR system.
10. Shenaj, D., Rizzoli, G., & Zanuttigh, P. (2023). Federated learning in computer vision. Ieee Access, 11, 94863-94884.
11. Chen, H., Ning, P., Li, J., & Mao, Y. (2025). Energy Consumption Analysis and Optimization of Speech Algorithms for Intelligent Terminals.
12. Majeed, A., & Lee, S. (2020). Anonymization techniques for privacy preserving data publishing: A comprehensive survey. IEEE access, 9, 8512-8545.
13. Hu, W. (2025, September). Cloud-Native Over-the-Air (OTA) Update Architectures for Cross-Domain

 doi:10.20944/preprints202512.2785.v1

Transferability in Regulated and Safety-Critical Domains. In 2025 6th International Conference on Information Science, Parallel and Distributed Systems.

14. Bifet, A., & Gavalda, R. (2007, April). Learning from time-changing data with adaptive windowing. In Proceedings of the 2007 SIAM international conference on data mining (pp. 443-448). Society for Industrial and Applied Mathematics.
15. Su, X. Vision Recognition and Positioning Optimization of Industrial Robots Based on Deep Learning.
16. Hassan, M. U., Rehmani, M. H., & Chen, J. (2019). Differential privacy techniques for cyber physical systems: A survey. IEEE Communications Surveys & Tutorials, 22(1), 746-789.
17. 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.
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. Lemtenneche, S., Bensayah, A., & Cheriet, A. (2023). An Estimation of Distribution Algorithm for Permutation Flow-Shop Scheduling Problem. Systems, 11(8), 389.
21. 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).