# Brief Industry Paper: Directed Kernel Fuzz Testing on Real-time Linux

Yuheng Shen*, Shijun Chen†, Jianzhong Liu*, Yiru Xu*, Qiang Zhang‡, Runzhe Wang§, Heyuan Shi¶, Yu Jiang*
*Tsinghua University †Central South University ‡Hunan University, §Alibaba Group,
{shenyh20, liujz21, xuyr21}@mails.tsinghua.edu.cn, jiangyu198964@126.com
†M1ldZz@163.com, ‡zhangqiang9413@126.com, §runzhe.wrz@alibaba-inc.com, ¶hey.shi@foxmail.com

***Abstract*—*Rt-Linux* contains critical modifications that are much less tested than the vanilla kernel, thus placing many systems at risk. In this paper, we present *DRLF*, a directed fuzzer targeted towards fuzzing any code area in *Rt-Linux*, thus allowing for more efficient tests on *Rt-Linux*'s unique code sections. *DRLF* performs directed fuzzing through a kernel-level weighted callgraph construction technique, and prioritizing input sequences that exhibit less distance to the target code. Evaluations show that *DRLF* delivers better cover speed while achieving a 24.70% coverage increase for the targeting code areas. *DRLF* also found 11 previously unknown bugs within *Rt-Linux*, and has been integrated into Alibaba's CI/CD pipeline.**

## I. Introduction

Real-Time Linux (*Rt-Linux*) [17] is a derivative of the Linux kernel which is designed to ensure deterministic response times for various tasks. Given the applicability and effectiveness nature of *Rt-Linux*, many industry sectors have deployed *Rt-Linux*, where they have modified the real-time module in *Rt-Linux*, specifically catering to their unique operational requirements, ensuring it meets diverse real-time requirements. As the demand for more diverse applications and enhanced real-time performance grows, the complexity of the *Rt-Linux* codebase increases, presenting challenges in ensuring its robustness and reliability. Therefore, the security of *Rt-Linux* is important, where oversight of any potential bugs within *Rt-Linux* can lead to catastrophic results, such as significant financial losses or, in extreme cases, loss of life.

Given the importance of *Rt-Linux*'s security, many testing methods have been deployed in the industry to ensure its security, such as unit tests and integration tests. Fuzz testing, a.k.a. fuzzing, is known for its ability to detect concrete bugs and has gained traction from academia and industry alike. Kernel *fuzzing tools*, a.k.a. fuzzers, generates testing payloads to test a target kernel. One popular kernel fuzzer, *Syzkaller* [14], generates system call sequences as test payloads using domain-expert-written system call specifications, executes the payload, and checks for any unexpected behaviors. It has successfully uncovered numerous bugs in various kernels, such as Linux, Android and Windows.

While upstream kernels undergo rigorous testing in the CI/CD pipelines, *Rt-Linux*'s additional code does not receive the same level of scrutiny. *Rt-Linux*'s unique code is exceptionally error-prone, as it is often tailored towards various use cases and different real-time requirements. Additionally, industrial scenarios allow for limited resource allocation, thus, testing the entire kernel code is impractical, as it is time-consuming and cannot specifically test the modified code. Therefore, our testing direction should be pointed towards code that contains *Rt-Linux*-specific features, allowing for efficient testing of the *Rt-Linux*-specific code and detecting potential defects within. This calls for the use of *directed fuzzing*, a technique that directs the testing procedure towards a specific code section in the target program.

However, to conduct directed fuzzing on *Rt-Linux*, we encounter the following challenges. First, directed fuzzing requires precise measurements of the differences and directions between execution traces and the target code within *Rt-Linux*, which cannot be applied directly using conventional methods. Second, utilizing the distance and direction information to generate input payloads that test the target code requires certain strategies that are tailored for *Rt-Linux*.

To address these challenges, we propose *DRLF*, a directed kernel fuzzer that is capable of efficiently testing any given code section in a target *Rt-Linux* kernel. *DRLF*'s techniques are introduced as follows. First, we construct a weighted callgraph using kernel-level code analysis that accurately represents the distance between arbitrary code blocks in the kernel and the target code that we wish to test. Then, during fuzzing, we calculate the distance of any executed input and devise a fuzzing strategy that prioritizes the generation and mutation of seeds that exhibit a lower distance to the target code. These techniques allow for more frequent execution of inputs that trigger the execution of the target area, thus delivering better fuzzing effectiveness in our designated area. We evaluated *DRLF* against *Syzkaller*, where the results show that, for the targeting code section, *DRLF* covers the target area faster than *Syzkaller*, and achieves a 24.70% increase in coverage statistics. Furthermore, *DRLF* found 11 new bugs in *Rt-Linux*'s designated code sections, all of which have been confirmed and fixed by the *Rt-Linux* project team.

*Yu Jiang and Heyuan Shi are the corresponding authors.

## II. Background

**Fuzz Testing**, commonly known as *fuzzing*, is a dynamic software testing technique. Its primary objective is to feed the System-Under-Test (SUT) with large amounts of randomly generated inputs, thus attempting to trigger bugs within the SUT. This method is particularly effective in identifying previously unknown vulnerabilities or unexpected system behaviors. The inputs used in fuzzing can probe potential weak points of the system, ranging from minor service interruptions to significant security violations. Due to its effectiveness in finding bugs, fuzzing is extensively used in various areas, such as databases and browsers, to enhance the safety and reliability of software applications [2], [3], [5], [6], [15], [16].

**Kernel Fuzzing** is an application of fuzzing that feeds random or semi-random inputs to the kernel's interfaces and its system call surface to identify vulnerabilities [4], [7], [8], [10]–[13]. For *Rt-Linux*, which is designed to operate within specific time constraints, it is particularly susceptible to disruptions originating from unexpected behaviors. As such, fuzzing these kernels is extremely beneficial to uncovering any anomalous behavior caused by bugs. It ensures that, not only the identification of vulnerabilities but also the consistent performance of systems, where timely responses are essential. Tardis [10] and RtKaller [9] are kernel fuzzers that perform kernel fuzzing on a wide range of real-time OS kernels.

**Directed Fuzzing** is a derivative method of fuzzing that targets specific code sections within the SUT. As this is beneficial towards testing newly added or recently modified code in an already-extensively-tested system, there have been many attempts to integrate directed fuzzing into CI/CD pipelines in the industry. AFLGo [1] is one such example. Implemented as an extension of AFL, it generates inputs targeting specific code locations in userspace programs. It uses both the control flow graph and the call graph of the program under test to determine the distance from any known basic block to the target basic block, and subsequently uses this distance metric to correspondingly guide further input generation.

## III. Motivation and Challenges

Most of the *Rt-Linux*'s code base comes directly from the upstream Linux kernel, which has undertaken rigorous security testing. The real-time-related code, however, is unique to *Rt-Linux*, and is often modified by different developers and vendors for various real-time requirements, and lacks the same level of scrutiny and testing that the upstream kernel receives. This is further complicated in industrial settings, where tests are often performed under stringent time and compute resource constraints. Therefore, to efficiently test *Rt-Linux*, especially the newly added features and its real-time-relevant code, we can adapt directed fuzzing to the kernel fuzzing domain, whereby provided the position of the target code, we can direct the fuzzer's testing into the target code sections, thus increasing the possibilities in uncovering vulnerabilities within. To perform directed fuzzing on *Rt-Linux*, we need to address the following challenges:

**Accurately Measure the Difference and Direction Between Code Blocks in Kernel Code.** To perform effective directed fuzzing, it is critical to accurately measure the difference and direction between the current execution trace and the target code within the *Rt-Linux*. Given the vast code space of *Rt-Linux*, discerning the relationship between the current input and the target code during execution can be challenging. Moreover, in a system as intricate as the *Rt-Linux*, any misjudgment in this measurement can direct the fuzzing process incorrectly, resulting in degraded fuzzing effectiveness. Thus, a precise approach is required to measure the difference and direction before initiating the fuzzing, ensuring a more informed and effective fuzzing process.

**Effective Guidance of Fuzzing Direction using Distance Information.** The next challenge lies in ensuring that the fuzzer's generated inputs can reach and cover this region. This requires an effective method to compute the distance between the target code region and the execution trace for any given input. Using this information during input generation allows the fuzzer to generate and mutate seeds that trigger code blocks closer to the target code regions, eventually directing the fuzzing process into testing the intended code region, thus achieving our design goals.

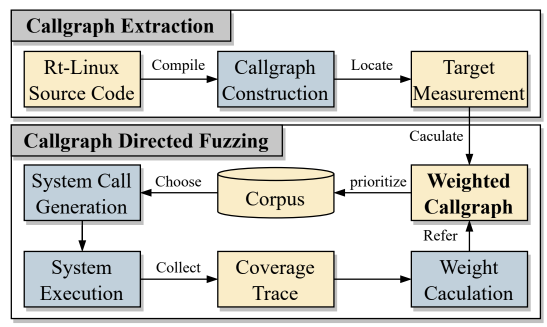



Fig. 1. The overview workflow of *DRLF*. *DRLF*'s fuzzing procedure consists of two steps: callgraph extraction and callgraph-directed fuzzing.

## IV. DRLF Design

We propose *DRLF*, a directed kernel fuzzer tailored for testing any specific code region within *Rt-Linux*. The overall workflow of *DRLF* can be found in Figure 1. For an *Rt-Linux* kernel under test, *DRLF* first performs callgraph extraction during the kernel's compilation process. *DRLF* extracts the kernel's control graph and the address for each basic block, then calculates and derives the weighted callgraph based on the control graph and the address of the provided target code. When *DRLF* conducts directed kernel fuzzing, during each fuzzing loop, *DRLF* generates system call sequences as inputs for the SUT and collects the execution code coverage trace. Then, based on the weighted callgraph, *DRLF* calculates the weight for the current input sequence, and prioritizes seeds in the corpus based on the weight information. For the next fuzzing iteration, *DRLF* selects the input with the highest priority for mutation, thereby steering the fuzzing process toward the target code region.

## A. Callgraph Extraction

To effectively conduct directed fuzzing, our primary task is to construct a weighted callgraph for the target *Rt-Linux* kernel. This contains the extraction of the intricate control flow spanning the entire kernel and the computation of the distance between each basic block and the target code location.

**Callgraph Construction.** For an *Rt-Linux* kernel under test, we first need to construct its callgraph. However, constructing the complete callgraph for the *Rt-Linux* kernel is complicated, especially considering the modular nature of *Rt-Linux* and its vast array of files. Therefore, to construct the system-wise callgraph, *DRLF* initiates the process by compiling the kernel with Clang. During this phase, *DRLF* systematically adjusts the compile configurations across every tier of the *Rt-Linux* directory. This is to extract specific compilation commands and then modify them to dump the corresponding LLVM bitcode. Consequently, a bitcode file for each kernel file is generated. These bitcode files encapsulate the control flow of each file, represented as a set of inter-linked basic blocks. Once all individual bitcodes are generated, we use *llvm-link* to link the bitcode files progressively in a bottom-up manner. The linking process aggregates the separate bitcode files, creating a unified and comprehensive representation of the kernel's control flow and structure. This detailed callgraph is essential for extracting the intricate control flow from the entire kernel.

**Target Measurement.** After capturing the control flow of the target kernel, we can then compute the distance from the target position to each basic block. This distance measurement is based on the number of linked edges within the callgraph. To achieve this, we start by disassembling the kernel binary. This disassembly process reveals the address of our target code region, which could either be a specific function or a distinct basic block. With this address, we traverse the callgraph using Dijkstra's algorithm to iterate over each edge. For every iteration, we calculate the number of steps required to reach the target code from the current position. This step count is then assigned as the weight of the edge. By following this methodical approach, we obtain a callgraph where each edge is weighted, reflecting the distance to the target code, thereby providing us with a detailed weighted callgraph.

## B. Callgraph Guided Fuzzing

Once *DRLF* acquires the weighted callgraph, we can start the guided fuzzing process. Specifically, our aim is to generate high-quality inputs that effectively direct our testing efforts towards the target region. *DRLF* uses the following steps: calculate the weight for the current input and harness the weight information to steer subsequent input mutations.

**PC Weight Caculation.** Within each fuzzing iteration, we retrieve the current coverage bitmap from `KCOV`, which enumerates the address of basic blocks encountered during the test. Leveraging this coverage bitmap, we further consult our weighted callgraph to ascertain the distance associated with each basic block's address. Specifically, we employ Equation 1 to convert this distance into program priority.

$$\text{Weight Value} = \frac{\text{atan}(distance) + \frac{\pi}{2}}{\pi} \tag{1}$$

This bounds the weight value between 0 and 1, ensuring a consistent range regardless of the actual distance. Also, its growth tendency emphasizes more on the shorter distance, therefore, sequences closer to the target have higher priority. This is achieved by the very nature of the atan function, which slows down for larger distances, ensuring that shorter sequences are prioritized.

**Distance-guided Program Evaluation.** With the priorities determined, the next phase of distance-guided fuzzing takes over. Each program within our corpus is assigned a priority. During mutation, the fuzzer prefers programs with the highest priority. This strategic selection ensures that our fuzzing endeavors are consistently oriented toward the target region, maximizing the efficacy of our testing process.

# V. Evaluation

**Implementation.** We implemented *DRLF* using Golang and Python, with some components borrowed from *Syzkaller*. To extract the kernel's callgraph before fuzzing, *DRLF* compiles the target kernel using Clang and emits its corresponding intermediate representation. Then, *DRLF* automatically constructs the weighted callgraph using the IR information, where the weights are based on the address of the target code.

*DRLF* is integrated into Alibaba's continuous fuzz testing pipeline called *ABACI* Robot. Changes to a specific part of the kernel trigger the CI/CD process, which invokes the automatic script that generates the weighted graph, and subsequently sends the result to the fuzzer, which then uses this information to focus its testing on the to-be-tested part in the kernel.

**Experiment Setup.** We tested *DRLF* on four versions of the *Rt-Linux* kernel, namely 5.10, 5.11, 5.14, 5.19, as they are widely deployed in Alibaba. We choose the functions within `io_uring` module as the directed fuzzing target due to its significance in real-time operations. Specifically, the `io_uring` uses two lock-free ring buffers: one for managing submission entries, allowing concurrent request handling, and another for completion events. The primary system call used is `io_uring_submit()`, designed for efficient multi-operation handling, making it highly relevant for real-time operations. We instrument the target kernel with KCOV for coverage collection, with Kernel Address SANitizer (KASAN) and Kernel Concurrency SANitizer (KCSAN) enabled for bug detection. We perform our evaluation on an AMD EPYC 7742 CPU at 2.25GHz with 64 cores running Ubuntu 20.04. All experiments are conducted on the same hardware for 24 hours, with each experiment repeated three times to establish statistical significance.

## A. Bug Detection Capabilities

**Found New Bugs.** *DRLF* found a total of 11 previously unknown bugs within the target module `io_uring`, as listed in Table I. Within the 11 bugs, 4 of which are memory-related, 4 of which are concurrency-related, whereas the rest are logic bugs. All listed bugs have been fixed by kernel maintainers. More information can be found using "git log –grep abaci" in the kernel's code base.

TABLE I
PREVIOUSLY UNKNOWN BUGS DETECTED BY *DRLF*

| Versions | Operations | Risk | Status |
|---|---|---|---|
| 5.11 | __io_clean_op | logic error | fixed |
| 5.11 | __io_req_task_submit | deadlock | fixed |
| 5.11 | __io_uring_sq | null ptr deref | fixed |
| 5.11 | io_cqring_overflow_flush | logic error | fixed |
| 5.11 | io_uring_poll | deadlock | fixed |
| 5.10 | io_sq_thread_stop | deadlock | fixed |
| 5.10 | __io_clean_op | null ptr deref | fixed |
| 5.10 | io_wq_submit_work | deadlock | fixed |
| 5.10 | io_uring_create | out of bound | fixed |
| 5.10 | io_commit_cqring | double free | fixed |
| 5.10 | io_rw_reissue | logic error | fixed |

***Rt-Linux* Related Bugs.** *DRLF* identified 11 bugs in *Rt-Linux*, each of which is located specifically within the targeted `io_uring` module and its designated functions. This precision is attributed to our directed fuzzing approach, which not only directs the fuzzing process towards the target area but also ensures a comprehensive and in-depth testing of the target code. Furthermore, the bugs we found are critical, as many are concurrency-related, causing potential slowdowns or hangs, which is especially detrimental to a system like *Rt-Linux*, where the timing and the system performance are of great importance.

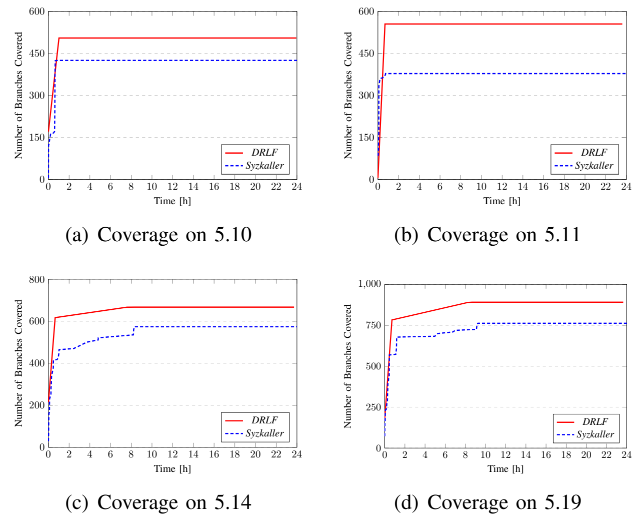
### B. Coverage Comparison

To further evaluate the effectiveness of *DRLF*, we compare the branch coverage within the target code section. The detailed statistics are listed in Table II. As shown in the table, Syzkaller achieves an average of 534.75 branch coverage on the respective kernel versions, while *DRLF* achieves statistics of 654.50 branch coverage on the respective four *Rt-Linux* versions. In comparison, *DRLF* gains an average of 24.70% coverage improvement in the `io_uring` module.

TABLE II
COVERAGE COMPARISON ON `IO_URING`

| Versions | 5.10 | 5.11 | 5.14 | 5.19 | Average |
|---|---|---|---|---|---|
| **Syzkaller** | 425.00 | 378.00 | 573.67 | 762.33 | 534.75 |
| ***DRLF*** | **505.00** | **555.00** | **667.00** | **891.00** | **654.50** |
| **Impr** | 18.82% | 46.83% | 16.27% | 16.88% | 24.70% |

Furthermore, the growth of the coverage is shown in Figure 2. As shown in the figure, *DRLF* can achieve a higher code coverage at a faster speed compared to *Syzkaller*. The graph also shows that most of the coverage growth saturates at around 4-8 hours, which is because we only target the `io_uring` modules, resulting in a short time to coverage saturation. The above improvement is attributed to the Target-related code coverage.




(a) Coverage on 5.10 (b) Coverage on 5.11

(c) Coverage on 5.14 (d) Coverage on 5.19

Fig. 2. Coverage Statistics for Syzkaller and *DRLF* on *Rt-Linux* Kernel Versions 5,10, 5.11, 5.14, and 5.19, respectively.

To conduct a more fine-grained coverage comparison, we analyzed the coverage on each file. Specifically, we choose the *Rt-Linux* v5.11, and we compare the *DRLF* with *Syzkaller*. We collect the coverage percentage for all the files within the `io_uring` module and compare the file coverage percentage between the *DRLF* and *Syzkaller*.

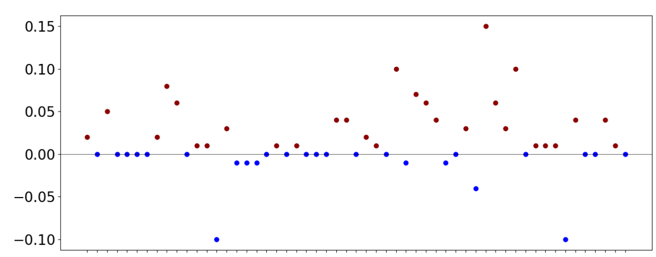


Fig. 3. Coverage Comparison for To-be-tested Files. The y-axis represents the percentage difference between *DRLF* and *Syzkaller*.

The results are shown in Figure 3, where positive values represent *DRLF* covers more code, and vice versa. We percieve that positive values are the majority, indicating that *DRLF* generally covers more code than *Syzkaller*, clearly demonstrating *DRLF*'s effectiveness in directed fuzzing. For the instances where *Syzkaller* covers more than *DRLF*, our analysis shows that the source files do not contain the target code section and, therefore are not concerning to us.

## VI. CONCULSION

In this paper, we propose *DRLF*, a directed kernel fuzzer tailored for *Rt-Linux*. By constructing a weighted callgraph and leveraging dynamic distance calculations, *DRLF* prioritizes testing toward newly-added code, ensuring an efficient and targeted fuzzing. Our evaluations demonstrate its efficiency in covering target code space, with 24.70% coverage improvement compared to *Syzkaller*, and 11 previously unknown bugs were detected.

## VII. ACKNOWLEDGMENT

This research is sponsored in part by the National Key Research and Development Project (No. 2022YFB3104000), NSFC Program (No. 62022046, 92167101, U1911401, 62021002, U20A6003, 62202500), Hunan Provincial Natural Science Foundation (No: 2023JJ40772), and Hunan Provincial Sci-Tech Talents Program (No: 2023TJ-X40).


## REFERENCES


[1] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*, pages 2329–2344, 2017.

[2] lcamtuf. American fuzzy lop, 2013. https://lcamtuf.coredump.cx/afl/.

[3] J. Liang, M. Wang, C. Zhou, Z. Wu, Y. Jiang, J. Liu, Z. Liu, and J. Sun. PATA: Fuzzing with Path Aware Taint Analysis. In *2022 2022 IEEE Symposium on Security and Privacy (SP) (SP)*, pages 154–170, Los Alamitos, CA, USA, may 2022. IEEE Computer Society.

[4] Jianzhong Liu, Yuheng Shen, Yiru Xu, Hao Sun, and Yu Jiang. Horus: Accelerating kernel fuzzing through efficient host-vm memory access procedures. *ACM Trans. Softw. Eng. Methodol.*, aug 2023. Just Accepted.

[5] Valentin JM Manes, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J Schwartz, and Maverick Woo. Fuzzing: Art, science, and engineering. *arXiv preprint arXiv:1812.00140*, 2018.

[6] Lucas McDonald, Muhammad Ijaz Ul Haq, and Ashley Barkworth. Survey of software fuzzing techniques.

[7] Shankara Pailoor, Andrew Aday, and Suman Jana. MoonShine: Optimizing OS Fuzzer Seed Selection with Trace Distillation. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 729–743, Baltimore, MD, August 2018. USENIX Association.

[8] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. kAFL: Hardware-Assisted Feedback Fuzzing for OS Kernels. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 167–182, Vancouver, BC, August 2017. USENIX Association.

[9] Yuheng Shen, Hao Sun, Yu Jiang, Heyuan Shi, Yixiao Yang, and Wanli Chang. Rtkaller: State-Aware Task Generation for RTOS Fuzzing. *ACM Trans. Embed. Comput. Syst.*, 20(5s), sep 2021.

[10] Yuheng Shen, Yiru Xu, Hao Sun, Jianzhong Liu, Zichen Xu, Aiguo Cui, Heyuan Shi, and Yu Jiang. Tardis: Coverage-guided embedded operating system fuzzing. *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, 41(11):4563–4574, nov 2022.

[11] Heyuan Shi, Runzhe Wang, Ying Fu, Mingzhe Wang, Xiaohai Shi, Xun Jiao, Houbing Song, Yu Jiang, and Jiaguang Sun. Industry practice of coverage-guided enterprise linux kernel fuzzing. In Marlon Dumas, Dietmar Pfahl, Sven Apel, and Alessandra Russo, editors, *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019*, pages 986–995. ACM, 2019.

[12] Hao Sun, Yuheng Shen, Jianzhong Liu, Yiru Xu, and Yu Jiang. Ksg: Augmenting kernel fuzzing with system call specification generation. In *USENIX Annual Technical Conference*, 2022.

[13] Hao Sun, Yuheng Shen, Cong Wang, Jianzhong Liu, Yu Jiang, Ting Chen, and Aiguo Cui. *HEALER: Relation Learning Guided Kernel Fuzzing*, page 344–358. Association for Computing Machinery, New York, NY, USA, 2021.

[14] Dmitry Vyukov and Andrey Konovalov. Syzkaller: an unsupervised coverage-guided kernel fuzzer, 2015. https://github.com/google/syzkaller.

[15] Mingzhe Wang, Jie Liang, Yuanliang Chen, Yu Jiang, Xun Jiao, Han Liu, Xibin Zhao, and Jiaguang Sun. SAFL: Increasing and Accelerating Testing Coverage with Symbolic Execution and Guided Fuzzing. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, ICSE '18, page 61–64, New York, NY, USA, 2018. Association for Computing Machinery.

[16] Mingzhe Wang, Jie Liang, Chijin Zhou, Yu Jiang, Rui Wang, Chengnian Sun, and Jiaguang Sun. RIFF: Reduced Instruction Footprint for Coverage-Guided Fuzzing. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 147–159. USENIX Association, July 2021.

[17] Victor Yodaiken and Michael Barabanov. Rt-linux. *White paper, Departament of Computer Science, New Mexico Institute of Technology, avalaibe at http://www. rtlinux. org/documents*, 2001.