Ice Skating: Final Report

Pranav Nair, *Purdue University*

Summer 2020

Abstract

Kernel concurrency bugs have become a focus of kernel testing in recent years, and extensive work has been conducted in detecting and preventing such bugs. Alongside this, coverage-guided fuzzing has presented itself as an effective tool in detecting user space application faults, and has recently found its way to kernel space following the introduction of Syzkaller, primarily focusing on detecting general bugs, most of which are sequential in nature.

This paper proposes *Ice Skating*, a testing tool that leverages the effectiveness of coverage-guided fuzzing and applies it specifically to the domain of kernel concurrency bugs. In particular, *Ice Skating* seeks to generate sequential tests that are effective candidates to be paired in order to create kernel concurrency tests more likely to discover such bugs.

1. Introduction

As systems become increasingly concurrent in both the software they run and the underlying hardware designed to cater to these applications, the problem of detecting concurrency faults has been brought to great importance. Unlike sequential bugs in programs designed to have a single thread of execution, concurrency bugs have been noted to be notorious in terms of detection, prevention, and correction. Such bugs tend to manifest in, unsurprisingly, highly complex concurrent systems. The kernel is one such example.

Given the importance of the kernel and its reliability for systems far and wide, there has been a large effort in detecting, preventing, and correcting such bugs. Prior works have considered various techniques to do so, with examples such as exercising control over the kernel thread scheduler to explore certain error prone interleavings [1], injecting delays to certain instructions to deviate from developer expectations of execution flow [11], and even employing static analysis to identify potential data race locations in the kernel source code [14]. Such methods have proven to be practical, within their own domains (e.g. specifically targeting file systems [11]).

Recently, there have been efforts in bringing coverage-guided fuzzing to the kernel for bug detection. With the method having shown great effectiveness in user-space applications [12], the most notable attempt at kernel fuzzing has been Google's Syzkaller [2], having detected a wealth of bugs, known and unknown alike [13]. However, existing kernel fuzzing attempts have been largely targeted at detecting sequential bugs, without specialized consideration for concurrency errors.

This paper presents *Ice Skating*, a system that intends to bring the effectiveness of coverage guided fuzzing to the domain of kernel concurrency bugs. It is to be complimentary to the works of *Snowboard,* through the optimization of sequential kernel tests for the generation of multi-threaded concurrency tests.

The following subsections will provide relevant background information prior to the discussion of system design and architecture in §2, followed by implementation details in §3. An empirical evaluation of the current state of *Ice Skating* is conducted in §4 and future steps for the project are considered in §5. Some related work is discussed in §6.

1.1. SKI and Snowboard

Snowboard builds upon the earlier works of SKI [1], a systematic kernel interleaving explorer. SKI was introduced as a testing tool to detect previously unknown concurrency bugs by scheduling individual kernel threads in interleavings that are more prone to cause data races and other related bugs, based on certain heuristics. As input to the tool, the current state of the kernel and a concurrent test (e.g. two or more systems calls) must be provided.

However, a testing tool is only as good as the tests themselves. For SKI to be effective at detecting bugs, effective multithreaded tests for the kernel must be provided. This is largely where Snowboard comes to picture. Snowboard is a complimentary tool that aims to generate effective concurrent kernel tests to be used with SKI.

Its functioning can be divided into 4 stages. First, Snowboard leverages existing sequential kernel tests and individually traces their runtime memory accesses to shared memory resources, collecting information such as the address of the access, type of access (i.e. read or write), and the instruction address. Second, Snowboard triages the large number of tests by determining whether they have increased memory coverage. If not, the test is classed as redundant and removed from the pool. Third, each sequential test is compared to another, identifying *channels*, which are unique tuples of the form *(write ip, read ip, memory address)*. For each channel identified, a concurrent test of the form *(test A, test B, channel)* is created. Lastly, these concurrent tests are executed using SKI, with the hopes of detecting a thread interleaving involving the predetermined *channel* that causes a concurrency bug.

1.2. Syzkaller

Syzkaller [2] is a fuzzer designed for the kernel. It has seen great success in detecting previously unknown kernel bugs. A large part of this is due to coverage-guided design of the fuzzer, which guides fuzzing according to the amount of code that is covered in each fuzzing execution loop.

Syzkaller takes input a corpus of tests containing a sequence of syscalls, which are executed in succession in order to achieve unaccounted kernel states, where bugs often lie. As a coverage-guided fuzzer, the kernel code coverage is evaluated on a per test basis. If the test has increased coverage, the fuzzer inserts it into its corpus, mutates the program (e.g. adding, removing or modifying the sequence of syscalls) and prioritizes these mutated tests for the next generation of tests executions. This process is repeated indefinitely, with the idea that the corpus maintained will saturate with effective tests over time or execution cycles. It is this corpus of sequential tests that Snowboard leverages as input.

1.3. Scope for Optimization and Ice Skating

While Snowboard has shown great promise with its current design, one aspect that seems to be a great contender for optimization are the properties of the sequential tests provided to Snowboard.

This intuition comes from the fact that the corpus of tests maintained by Syzkaller, which are designed to prioritize code coverage upon execution, may not necessarily correspond to being better candidates for the pairing stage in Snowboard where the concurrent tests are generated.

A reason for this is largely due to metrics used to evaluate code coverage in Syzkaller. The fuzzer uses the edge coverage metric, which essentially numerates the number of edges in the control-flow graph that have been explored during the fuzzing process. While this is effective for the use case of Syzkaller (i.e. kernel bugs in general), this metric has little consideration for the concurrency dimension that Snowboard prioritizes. Specifically, the metric does not consider data-flow dependencies in its criteria, an aspect that is crucial to concurrency faults (as most concurrency bugs are due to memory being accessed in an interleaving that has not been considered by the developer).

Ice Skating seeks to modify this metric such that Syzkaller saturates its corpus with tests not necessarily effective at increasing code coverage, but at being effective sequential pairing candidates for concurrent test generation in Snowboard. In short, this aims to be accomplished by modifying Syzkaller such that data-flow dependencies discovered during test runtime are considered when evaluating the coverage of each test, which in turn will cause the corpus to saturate with tests more dependent on shared memory regions, allowing Snowboard to discover channels more prone to concurrency bugs.

2. Design and Architecture

This section will detail the design choices for Ice Skating and the system's architecture. In particular, §2.1 introduces the concept of Define-Use (DU) pairs and how Ice Skating leverages them to guide Syzkaller to generating more relevant sequential tests for Snowboard. The method used to collect this information from the kernel is discussed in §2.2, and an overall summary of the architecture is presented in §2.3.

2.1. DU Pair Coverage Metric

There exists a wealth of research conducted on coverage metrics, and more recently metrics designed for evaluating coverage in concurrent applications. While not directly applicable to the need for a sequential metric (as Ice Skating deals with sequential tests, not concurrent ones), there is much to be inspired and potentially ported for Ice Skating's use case.

Regular coverage metrics designed for single-threaded applications are largely syntactical, often based on control-flow graphs and abstractions like code blocks and methods (such as the metric used in Syzkaller, edge coverage, which observes which edges in the CFG have been covered). When the concurrent dimension is considered however, these metrics are often misleading as various thread interleavings must be considered. Additionally, data-flow dependencies are more prevalent to execution safety due to these interleavings. There has been prior work on metrics that consider both these aspects [3][4][5][6].

A key idea present in most prior works regarding this topic builds on the concept of Definition–Use (DU) paths [7]. A *DU-path* is defined as the execution path starting at a *define* instruction (write to an address) and ending at a *use* instruction (read from an address), where both instructions access the same memory address. The pair of instructions are an example of a *DU-pair.* Several memory aware metrics use this simple abstraction of a data-flow dependency. For example, the *All-Defs* [15] coverage metric states that 100% coverage has occurred when an execution has exercised at least one DU-path for every definition instruction. When full coverage is not observed, it is possible that erroneous interleavings have not been tested, possibly leading to a concurrency bug such as a data race.

The concept of Location Pairs (LPs) [3] build on DU-paths by modifying the definition such that no accesses to the memory address in question can be made between the pair of instructions. This has shown to catch certain interleavings that cause concurrency bugs that DU-paths do not account for, although increasing the effort needed to saturate (i.e. reach 100% coverage).

Inspired by these works, the coverage metric Ice Skating implements *alongside* regular edge coverage can be defined as follows:

A DU-pair has been covered if:

- It has not been encountered before (i.e. it is unique).
- Both the *define* and *use* instructions refer to a shared memory object (as local accesses are not relevant to testing concurrent aspects).
- There are no other *write* accesses within the corresponding DU-path than the initial *define* instruction of the pair. The reason for this is because in an uninterrupted sequence of write instructions, assuming single-threaded execution, the value of the latest write is all that will be read in later instructions – thus we need not consider prior write instructions.

Given this definition, modifications to Syzkaller's coverage evaluation algorithm can build upon this concept to introduce data-flow awareness and guide sequential test generation to create better pairing candidates. This is because a test that has shown considerable DU-pair coverage indicates the presence of many shared data dependencies, and using these tests for pairing allow context switches (controlled by *Snowboard*) to interrupt a greater number of DU-paths during execution. As this transfer of thread execution occurs between memory being written and read, there is potential for this memory to be modified unexpectedly before being read again by the original thread, i.e. a concurrency bug.

2.2. Memory Access Tracing

To compute DU-pair coverage, a trace of memory accesses during the execution of the test is required, alongside information such as the address of the instruction performing this access, as well as whether it was a write or read instruction.

Many methods to trace memory accesses during test runtime were discussed, including the use of the perf-mem tool or VMM memory tracing through QEMU modifications. However, these options were inviable, only sampling memory accesses (which would be detrimental to finding uncommon accesses where bugs may manifest) or introducing too much overhead, respectively.

However, Google's KASAN [10] seems to fit Ice Skating's needs to a much greater degree. KASAN was introduced as a tool to detect memory errors such as use-after-free or out-of-bounds exceptions in the kernel. This is implemented by instrumenting every memory access in the kernel [10] with a check to a previously allocated shadow region, which contains information allowing KASAN to validate the memory access. KASAN observes a 2x reduction in performance, which is relatively fast compared with other options.

Ice Skating leverages this instrumentation function and modifies it such that every memory dereference in the kernel source code is written to a buffer during runtime. This method allows the collection of the memory address being accessed, the access type (i.e. read or write), the size of the access, and the instruction address accessing the memory, fitting our needs to compute DU pair coverage.

2.3. Ice Skating Architecture

Ice Skating largely builds upon Syzkaller's current architecture, as well as providing an interface for the kernel to push memory access information so that Syzkaller may retrieve it.

As Syzkaller targets the kernel for fuzzing, with the aim of producing errors of which include kernel crashes, it is imperative that Syzkaller is not completely deployed on the kernel it is testing. This issue is overcome through a divided approach [8]. *Syz-manager* is a process running on a host OS (unaffected by the kernel fuzzing) that monitors and deploys several virtual machines running the target kernel. All these VMs run the *Syz-fuzzer* process, which is in charge of guiding the fuzzing through coverage evaluation, as well as communicating with *Syz-manager,* which holds the corpus. *Syz-fuzzer* deploys transient *Syz-executor* processes, which execute tests and collect coverage information from the kernel.

Modifications to Syzkaller will largely keep this architecture unchanged. Kernel modifications to provide memory access coverage information on the other hand will follow the architecture of KCOV [9] as closely as possible, which exposes code coverage information through a debugfs virtual file API (which Syzkaller uses to extract edge coverage information).

3. Implementation

The implementation of Ice Skating can be cleanly divided into two aspects: kernel modifications to collect and expose memory access coverage information, and Syzkaller modifications to read and consider this data in its coverage evaluation algorithm. Within the kernel modifications, the prior work done on KASAN [10] (Kernel Address Sanitizer) is leveraged and an API named KMCOV (Kernel Memory Coverage) is introduced as a parallel to KCOV.

In total, this implementation of Ice Skating added 578 lines of *Go* source code to Syzkaller, 434 lines of code to the existing KASAN implementation in the Linux Kernel v5.4.0, and 187 lines of code to create the KMCOV debugfs interface.

3.1. KASAN instrumentation

KASAN instrumentation is applied either through an LLVM pass or through GCC's GIMPLE iterators, an API that allows instrumentation using the intermediate language GIMPLE used by the compiler. The default implementation in the v5.4.0 kernel uses the GCC method.

Leveraging this API, KASAN instruments every GIMPLE statement that is either a 'store' or 'load' (corresponding to write and read instructions). Ice Skating modifies the instrumenting function such that upon being called, information such as the memory address, instruction address, etc. are stored to a buffer for later access by KMCOV.

```
def computeCoverage(mem_addrs, ip_addrs, access_type_buf):
    declare coverage_map[]
    declare ip_map[]

    for index, addr in mem_addrs:
        map[addr].append(index)

    for mem_addr, ip_indicies in ip_map:
        read_indicies[]
        for (i = len(ip_indicies); i >= 0; i--):
            if (access_type_buf[i] == 0): // Read instuction
                read_indicies.append(i)
            else: // Write instruction
                for read_index in read_indicies:
                    pair = {
                        mem_addr,
                        ip_addrs[i],
                        ip_addrs[read_index]
                    }

                    if !coverage_map[pair]:
                        coverage_map[pair] = true

            read_indicies.clear()

    return coverage_map
```

Figure 1*:* Procedure for generating DU-pair coverage given shared memory addresses (*mem_addrs*), instruction addresses (*ip_addrs*) and access types (*access_type_buf*).

3.2. KMCOV API

Similar to KCOV, Ice Skating implements a debugfs virtual file, KMCOV, that acts as an interface to access to the memory information collected by the modified KASAN instrumentation. It is a module loaded into memory during kernel boot time.

Currently, KMCOV dynamically allocates buffers of user defined size to store memory addresses, instruction addresses, and access types, ensuring a temporal relation such that entries prior to a certain entry in the buffers are guaranteed to have occurred before in the execution trace. Writes to this buffer are then enabled, disabled, and reset through an IOCTL interface, to be used before and after test execution for precise tracing of memory accesses made by the test.

Additionally, KMCOV only collects memory accesses that have occurred beyond the stack, due to local accesses not being considered in the definition of a DU-pair. This is performed by checking whether the memory address points to inside the stack, the offsets of which are found by adding *2 * PAGE_SIZE* to the address returned by *current_thread_info()* representing the *thread_info* struct found at the bottom (lowest memory region) of the stack.

3.2. Syzkaller Modifications

In order for Syzkaller to access memory coverage information, it must interface with KMCOV during test runtime. Modifications to *Syz-executor* have been made, retrieving memory coverage on a per syscall basis, enabling memory tracing just before a target syscall executes and collecting the trace after the syscall returns. This information is then made available to a modified *Syz-fuzzer* and is collected through a shared memory region with the executor.

KCOV makes available only raw instruction addresses to Syzkaller, however internally, the fuzzer uses a hash table data structure that maps address values to a boolean, which determines whether the address was covered during execution. A similar approach was taken with DU-Pair coverage, as it is both space efficient (as pairs not covered will not be present in the data structure, and a value of *false* will be returned in case it is provided as a key to the hash table) and keeps an amortized O(1) access time complexity.

Computing DU-pair coverage, however, requires further analysis of the memory coverage information retrieved from KMCOV. To analyze DU-pairs encountered during the test execution, individual memory addresses are linked (e.g. a corresponding list) to a list of instruction addresses that were found to access the same address. Generating this list of instruction pointers per shared memory address has then effectively computed a list of possible candidates for a DU-pair. For example, if three instruction addresses ip_1, ip_2, and ip_3 (with corresponding access types of write, read, read) are contained in the list corresponding to the same memory address, two potential DU-pairs are (ip_1, ip_2) and (ip_1, ip_3). However, care must be given to the order in which these instructions were executed in, as a read before a write instruction does not qualify as a DU-pair. The exact procedure is presented in Figure 1.

The next stage of implementation focuses on modifying Syzkaller's coverage evaluation algorithm such that DU-pair coverage is also taken into consideration. Currently, Syzkaller executes the same test multiple times and merges the coverage collected in each test to produce a maximal coverage. This merged coverage is compared to the overall coverage in the corpus, and depending on whether the test has shown new coverage, it is either inserted into the corpus or discarded. The same approach is taken with DU-pair coverage – the corpus contains a maximal DU-pair coverage map which represents the coverage collected during Syzkaller's execution, and only tests showing new coverage are added to the corpus. At the same time, tests showing new edge coverage but not DU-pair coverage are similarly discarded.

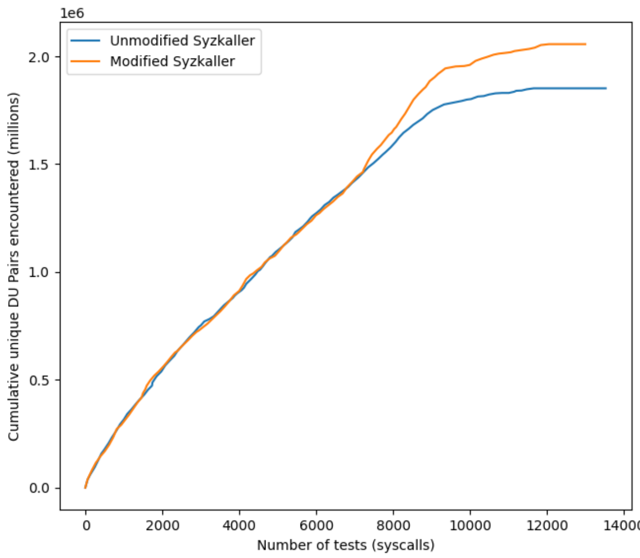

Figure 2*:* Comparison of DU Pair coverage in unmodified Syzkaller and Syzkaller with Ice Skating's modifications over test executions.

4. Evaluation

In this section an evaluation of the current state of Ice Skating is presented. §4.1 looks at the DU-pair coverage in both base Syzkaller and the modified Syzkaller, and the performance of the system is evaluated against unmodified Syzkaller in §4.2.

The experiments were conducted on a local machine with an Intel i5 6500 processor and 16GB of RAM, running an Ubuntu VM on VMWare Workstation as the host machine. Syzkaller was set to target the v5.4.0 Linux Kernel.

4.1. DU Pair Coverage

To get a sense of the impact of the modifications done by Ice Skating to Syzkaller's coverage evaluation algorithm, the DU pair coverage over approximately 14,000 syscalls conducted by base Syzkaller and modified Syzkaller was observed, the results of which are shown in Figure 2. The coverage of both versions of Syzkaller seem to be undisguisable up until about 7,000 syscall executions, at which point the Ice Skating modified Syzkaller seems to increase in DU Pair coverage, observing over 2 million unique DU-pairs compared to approximately 1.85 million DU-pairs. While these results seem to imply the generation of tests with greater DU pair coverage in the modified Syzkaller, further testing will be required confirm this.

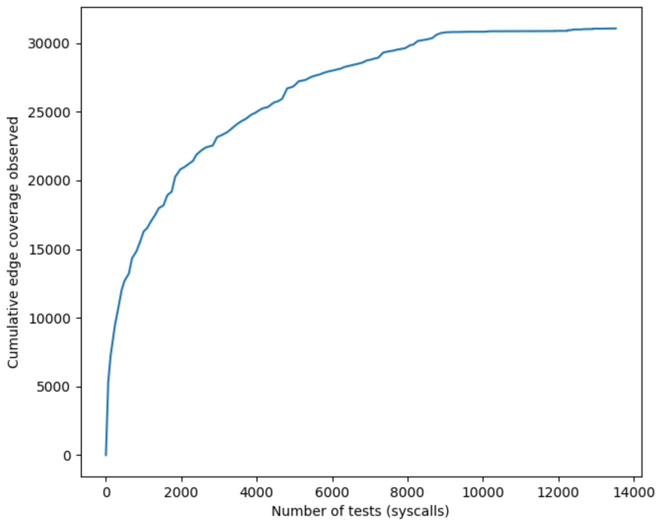

Figure 3: Comparison of DU Pair coverage in unmodified Syzkaller and Syzkaller with Ice Skating's modifications over test executions.

For one, both instances of Syzkaller started with an empty corpus, meaning the first seed input would be randomly generated in both cases. Due to this, it is possible that the modified Syzkaller received a seed that would later on manifest as greater DU-pair coverage in subsequent mutated tests. However, this does not seem likely as both instances had similar coverage for the first 7000 tests, by then which such a difference would have had an effect on the observed coverage.

Another characteristic that will need to be investigated is the plateauing of DU pair coverage in both instances at around 12,000 executed syscalls. Several tests were conducted, with all of them exhibiting the same plateau of DU Pair coverage. Additionally, the edge coverage observed by Syzkaller also plateaued, after around 9000 syscalls, which can be seen in Figure 3.

4.2. Performance Impact on Syzkaller

A large part of Syzkaller's effectiveness relies on its high throughput of test executions, and being able to mutate and generate tests efficiently to continue the fuzzing cycle without any bottlenecks.

Fortunately, KASAN is already required by Syzkaller on most target architectures, and such the performance overhead of KASAN does not seem to be detrimental to Syzkaller's effectiveness. The modifications being made to collect memory coverage information is not too dissimilar to the checks KCOV makes, however the performance bottleneck is undoubtedly found during the computation of the DU pair coverage per syscall.

Syzkaller Modifications	Throughput (syscalls/minute)
Unmodified	**1080**
Modified (Computation of unique DU pairs)	**567**

Table 1: Comparison of throughput between different instances of Syzkaller, both captured upon executing 7000 syscalls.

Table 1 presents the test throughput between the different instances of Syzkaller. As shown, the modified Syzkaller observes an approximate 50% reduction in test throughput. This is a non-insignificant impact on the performance of Syzkaller, and needs to be addressed in subsequent iterations of Ice Skating.

5. Future Work

Various possibilities and finalizations are still left for Ice Skating. The most pending of which is analyzing the reason behind the coverage plateau observed during experimentation. Specifically, experimentation against a clean Syzkaller tree will be needed to conducted, as mentioned in §4.2. Once overcome, the next phase of experimentation will require using the Ice Skating corpus of sequential tests as input to *Snowboard,* observing the impact on concurrent endpoint coverage and potentially bug detection.

Another scope for improvement is the performance of the current system. As discussed in §4.2, the performance impact of computing DU-pairs is quite sizable on Syzkaller, and such, potential other forms of coverage metrics are being considered.

One such performance–friendly idea that is being explored employs hashing to reduce the sizable address space that are characteristic of 64 bit addresses. The idea is to define a coverage metric based on a value that incorporates the instruction address, memory address and type of access per shared memory access in the execution trace. As the IP is included in the value, the memory address can be truncated or hashed as all 64 bits worth of information is not required, instead only a couple bits worth of information may be all that is needed to distinguish between memory accesses. This method avoids the costly computation of DU pairs and should theoretically be more performant.

Another interesting idea to consider is utilizing the coverage per test and identifying sequential tests that have relatively large intersections in the DU pairs they have encountered. Such tests, when paired to create a concurrent test, may exhibit large data dependencies between each other, as many of the shared memory accesses will be to the same shared object. This may further increase the possibilities of discovering certain concurrency bugs.

6. Related Work

Of the various publications discussing kernel testing, one similar piece of work that was found is KRACE [11], which also uses a fuzzing approach to detect concurrency bugs in the kernel. It presents a delay-based thread scheduler that is utilized to explore different thread interleavings similar to SKI. What is interesting is that the tool uses a data-flow coverage metric alongside regular branch coverage to guide the fuzzing. The coverage metric is named ‘Alias’ coverage, which is functionally the same as enumerating the number DU-paths covered during execution. KRACE is implemented through the use of custom instrumentation, however it only targets filesystems, not the entire kernel. It states that the approach used (instrumentation) is worthwhile for the limited scope of filesystem fuzzing, although not practical when fuzzing the entire kernel.

7. References

[1] P. Fonseca, R. Rodrigues, B. Brandenburg: SKI: Exposing Kernel Concurrency Bugs through Systematic Schedule Exploration. In OSDI ’14. https://www.usenix.org/system/files/conference/osdi14/osdi14-paper-fonseca.pdf.

[2] Syzkaller, an unsupervised coverage-guided kernel fuzzer. https://github.com/google/syzkaller.

[3] Tasiran, S., Keremoğlu, M.E. & Muşlu, K. Location pairs: a test coverage metric for shared-memory concurrent programs. Empir Software Eng 17, 129–165 (2012). https://doi.org/10.1007/s10664-011-9166-8

[4] Jie Yu and Satish Narayanasamy. 2009. A case for an interleaving constrained shared-memory multiprocessor. SIGARCH Comput. Archit. News 37, 3 (June 2009), 325–336. DOI:https://doi.org/10.1145/1555815.1555796

[5] Chao Wang, Mahmoud Said, and Aarti Gupta. 2011. Coverage guided systematic concurrency testing. In Proceedings of the 33rd International Conference on Software Engineering (ICSE ’11). Association for Computing Machinery, New York, NY, USA, 221–230. DOI:https://doi.org/10.1145/1985793.1985824

[6] O. Edelstein, E. Farchi, Y. Nir, G. Ratsaby and S. Ur, "Multithreaded Java program test generation," in IBM Systems Journal, vol. 41, no. 1, pp. 111-125, 2002, doi: 10.1147/sj.411.0111.

[7] Shan Lu, Weihang Jiang, and Yuanyuan Zhou. 2007. A study of interleaving coverage criteria. In The 6th Joint Meeting on European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering: companion papers (ESEC-FSE companion ’07). Association for Computing Machinery, New York, NY, USA, 533–536. DOI:https://doi.org/10.1145/1295014.1295034

[8] Syzkaller Internals. https://github.com/google/syzkaller/blob/master/docs/internals.md#overview

[9] KCOV: Code coverage for the kernel. https://www.kernel.org/doc/html/latest/dev-tools/kcov.html

[10] The Kernel Address Sanitizer. https://www.kernel.org/doc/html/latest/dev-tools/kasan.html

[11] M. Xu, S. Kashyap, H. Zhao, T. Kim: KRACE: Data Race Fuzzing for Kernel File Systems. Pre-publication. https://www.cc.gatech.edu/~mxu80/pubs/xu:krace.pdf

[12] AFL, American Fuzzy Lop https://lcamtuf.coredump.cx/afl/

[13] Syzkaller, Found Bugs https://github.com/google/syzkaller/blob/master/docs/linux/found_bugs.md

[14] D. R. Jeong, K. Kim, B. Shivakumar, B. Lee and I. Shin, "Razzer: Finding Kernel Race Bugs through Fuzzing," *2019 IEEE Symposium on Security and Privacy (SP)*, San Francisco, CA, USA, 2019, pp. 754-768, doi: 10.1109/SP.2019.00017.

[15] A. Gurfinkel, L. Tan. 2018, Testing: Dataflow Coverage https://ece.uwateloo.ca/~agurfink/ece653/assets/pdf/W04-DataflowCoverage.pdf