Ice Skating: Status Report

Pranav Nair, *Purdue University*
Summer 2020

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. This report aims to provide a status update on *Ice Skating,* a component of a larger project named *Snowboard,* which intends to contribute to detection of concurrency bugs in the kernel.

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 mutations 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. 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 have high code coverage, 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 evaluated 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, this metric has little consideration for the concurrent 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 programmer).

Ice Skating seeks to optimize 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 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

At this stage, several coverage metrics and their respective practicalities have been researched, with the hope of inspiring a method to modify Syzkaller's metric.

2.1. Memory Aware Coverage Metrics

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 our 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 our use case.

Regular coverage metrics designed for single-threaded applications are largely syntactical, often based on control-flow graphs and execution abstractions like code blocks and methods. 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 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*[waterloo slides] 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 an erroneous interleaving has 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.

Modifications to Syzkaller's coverage evaluation algorithm can build upon these concepts to introduce data-flow awareness when generating new sequential tests. A naïve approach may simply enumerate shared memory accesses and prioritize tests that show increased coverage in this sense for further mutation. Another approach being considered computes the number of DU-pairs/LPs in the post-execution memory trace, and in turn prioritizes tests that discover new instruction pairs and consequently data dependencies.

2.2. Proposed Architecture

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.

Syzkaller overcomes this issue 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.

3. Implementation Plan

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 we call KMCOV (Kernel Memory Coverage) is introduced as a parallel to KCOV.

3.1. KASAN Instrumentation

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, KASAN seems to fit our 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, in theory, 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.

3.2. KMCOV API

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

Currently, KMCOV dynamically allocates a buffer to store memory addresses based on user input. 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.

However, this format may not be the most efficient or useful way to represent this data. Discussions of utilizing a bitmap are being considered, allowing constant time checks to see if a certain memory address was accessed. Additionally, several bitmaps could be used to expose information such as access type or instruction addresses in a similar fashion. Deduplication of memory addresses is another optimization being considered that can reduce space usage.

3.3. 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, enabling memory tracing just before a test executes and collecting the trace after the test concludes. This information is then made available to *Syz-fuzzer* and collected through a shared memory region with the executor.

One question that comes into mind is how the memory coverage should be represented within Syzkaller for optimal usage when evaluating and comparing coverage. 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 is being considered with memory coverage, as it is both space efficient (as addresses 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.

The next stage for implementation will be focused on modifying Syzkaller's coverage evaluation algorithm such that data-flow 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. Based on the outcome of the discussion in §2.1, the algorithm will have to be modified accordingly.

The naïve approach can be implemented by simply enumerating the number of accesses to shared memory addresses per test, and prioritizing tests that show greater coverage in this regard (this will probably be implemented anyway to see the impact of this change to the algorithm). Another approach of calculating DU-pairs will require a more complex implementation. To compute DU-pairs, individual memory addresses will need to be linked in some form (e.g. a corresponding list) to the instruction addresses that were found to access these addresses, as well as whether their corresponding access types (as a DU-pair must contain at least one write instruction). 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 taken to the order in which these instructions were executed in, as a read before a write instruction does not qualify as a DU-pair. Although, if the concept of a Location Pair were used instead, write after read instruction pairs are still considered (LPs however introduce the restriction that no accesses to the same memory address can be made in the execution path connecting the instruction pair, reintroducing the need of maintaining execution order – perhaps a hybrid approach between DU-pairs and LPs can be considered).

4. Evaluation of Plan

Several aspects need to be considered before Ice Skating can be a practical contribution, in terms of effectiveness and efficiency.

4.1. 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. Though the modifications being made to collect memory coverage information is not too dissimilar to the checks KCOV makes, experiments must be conducted to compare performance. Initial tests have shown around 50 million memory accesses caught by KASAN per Syzkaller test, although this number greatly depends on the composition of the test itself.

A larger problem is the performance overhead of computing DU-pairs/LPs post-execution, per test. If the approach described in §3.3 is used, it is likely that most memory addresses written to the KMCOV buffer have been accessed by multiple instruction addresses. On top of that, computing these instruction pairs will involve quadratic time complexities (at its most naïve implementation) to check every instruction address against the others. This results in a very demanding process per executed test. Though limiting memory addresses to just those in shared regions will cause a considerable reduction, numbers will need to be generated to completely see the impact of this computation.

4.2. Mutation Policy

Another aspect to consider is how Syzkaller mutates its programs with the goal of increasing coverage from the original test. When considering edge coverage (or any syntactical coverage metric for that matter), slight random mutations to the inputs that observed the most coverage will often lead to better *code* coverage. However, it does not seem to be so straightforward when dealing with data-flow dependencies, as slightly mutating the input (e.g. introducing a new syscall to the test, mutating syscall arguments, etc.) does not seem to directly influence the number nor 'quality' of data dependencies that will be discovered in the mutated program (apart from the natural data dependencies that are discovered alongside increased code coverage). This will need to be studied further such that the mutation policy of Syzkaller can be accordingly modified to provide more relevant mutations to data dependencies.

5. Related Work

One similar work that was found is KRACE [11], which 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 practically 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 is not practical when fuzzing the entire kernel.

6. 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 multi-processor. 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