Kernel Fuzzing

Mahbub Raton Nicholas Baron Benjamin Rutkowski

June 2024

1 Introduction

Kernel fuzzing is a testing strategy where an automated tool generates random input and feeds it to the kernel under the test. The goal is to discover the unspecified behaviors of the kernel. The responsibility of the kernel is to act as an interface between processes and hardware. It is inconceivable to find kernel bugs by manual inspection because of the sheer complexity. Kernel fuzzing comes to the rescue in this case. In this project, we will review some state-of-the-art techniques for kernel fuzzing. We will mainly focus on the Linux kernel since most of the research focuses on Linux.

2 Motivation

The most important part of any system for testing is its interface, as that is the area where all internal errors will escape and attacks from malicious agents are most frequently made. However, since many system calls allow arbitrary data to be passed in the form of pointers to strings or structs, the attack surface is rather large. Additionally, many operations require multiple system calls. For example, reading a file properly requires at least three (`open`, `read`, and `close`), with the `read` call repeated when the destination buffer is too small or to allow the file to be loaded from disk in smaller chunks.

A common technique to explore large input spaces is fuzzing, which (at its simplest) is using random bytes as input and iteratively modifying them to find bugs or errors. Many fuzzers use a heuristic of new paths discovered to help determine better inputs, allowing them to explore the control flow under test. For fuzzing a kernel, the best fuzzer would be able to mutate the order of calls and the values (both the identifying number and the other parameters) passed in. The fuzzer would thus simulate a very erratic program and show that a kernel is resilient against such unusual behavior. A more intelligent fuzzer (one with some knowledge of syscalls) would be able to give more "pointed" inputs and act more like an adversary.

2.1 List of Topics

Our survey covers a few subfields of kernel fuzzing, with the goal of understanding the general approaches to fuzzing kernels and some of the more advanced techniques. Data

races are a particular field of interest, as they are difficult to detect and may often occur in modern multithreaded environments. The layer below kernels, namely hypervisors, is both an interesting tool and a target for fuzzing. Finally, the complexity and flexibility of the kernel API often require special techniques to fuzz effectively.

3 General Kernel Fuzzing

Kernels are internally complex, with a large state space that makes accessing all paths difficult. This has led to a variety of approaches to fuzz kernels, including manually detecting dependencies [7], statically analyzing the kernel [26], symbolically executing system calls [11], and even applying machine learning [21]. Additionally, certain subsystems are vital to fuzz, as these may silently error [12, 19].

3.1 Finding Semantic Bugs in File Systems

File systems are large and complex parts of a kernel that are both essential in correctness and under-tested, with the paper blatantly stating that "Filesystems are too large to be bug free". The authors go on to propose fuzzing the implementations to see what bugs exist and build *Hydra* [12], a framework for this exact purpose.

3.1.1 Background

The bugs of file systems are broken into four major types, with a smaller miscellaneous type trailing along. The first is crash inconsistency, where a kernel panic or power loss leaves a file system in an invalid state. The second is a specification violation, which occurs when a file system does something other than what was written in, for example POSIX. The third is implementation-specific logic bugs, which are often caught by assertions in debug builds and silently failed in release builds. The final major category is memory errors, a class that has additional scrutiny applied in the form of various sanitizers. The two issues mentioned in the miscellaneous group are concurrency and hardware failures, both of which Hydra does not check for.

The existing practices are mostly reactive. Regression tests are written after a bug has been found, thus forming an ad-hoc collection and result in insufficient coverage. Per-bug checkers have a limited effectiveness, while formal verification (the only proactive method mentioned) still falls short, with two bugs discovered by Hydra in a verified file system.

3.1.2 Design

The process for fuzzing starts by selecting a seed input. This input has both a file system image and a sequence of system calls. The input is then mutated, with the image maintaining a correct initial state. The image is mounted, and the calls are executed to generate a fuzzing report. This report has both a coverage map and a list of assertions if any were triggered. The aforementioned process is repeated, mutating the file system image at first. If nothing interesting happens and coverage remains the same, the syscalls are mutated. If, still, the report is uninteresting, syscalls are appended to explore more intricate bugs. A "interesting

thing" is a seed which causes any bug, so a crash consistency checker was implemented to emulate syscalls and enumerate the possible crash states.

3.1.3 Evaluation

Hydra went on to report 91 bugs in the Linux 5.0 kernel. 32 of these were semantic bugs, which the paper authors argue could not have been found by prior work. The speed of fuzzers is measured in executions per second, which in Hydra's case comes out to ∼104. This approaches the theoretical maximum of the Linux Kernel Library at 122 and soundly beats VM-based speeds of ∼0.7. Coverage is reported as 1.5 times more than Syzkaller [23] alone.

3.2 Using learning to fuzz syscalls

A critical aspect of effectively fuzzing the kernel through syscalls is incorporating the dependencies that may intentionally and unintentionally exist across syscalls. Previous fuzzers such as Syzkaller [23] and Moonshine [15] require handwritten syscall specifications or real-world application logs. However, these tools only are ever aware of the dependencies that have been manually described in the specification. The majority of dependencies across syscalls are perhaps unintentional and unspecified; they are merely emergent from the implementation in question. Hao Sun et al. present *HEALER* [21], a fuzzing tool that uses learning to discover any of these hidden dependencies. At runtime, *HEALER* will analyze relationships between syscalls to both generate new fuzzing inputs and dynamically improve its model of dependency relations.

3.2.1 Overview of *HEALER*

The present tool expands the framework that was already established by Syzkaller. *HEALER* utilizes the syscall description language `SyzLang` to represent syscall specifications. The authors define a syscall C_i to have influence over syscall C_j if the execution of C_i can modify the kernel state and change the execution path of C_j. *HEALER* stores the influence relation between any two syscalls in the *relation table*. The values of the table entries are first evaluated statically using the `SyzLang` descriptors. This table is then dynamically refined during runtime by analyze the amount of coverage per sequence of inputs using the learning algorithm. The learning algorithm first removes superfluous fuzzed inputs in the sequence (inputs whose absences do not affect coverage) to get a *minimized* sequence. Then it will remove each syscall in the minimized sequence one at a time and analyze the change in coverage. It uses this analyze to update the relation table. The new relation table is then used to generate and mutate new test inputs for the next cycle.

3.2.2 Evaluation and future work

Hao Sun et al. performed evaluations on *HEALER* to determine if its coverage is superior to its previous tools: Syzkaller and Moonshine. They also performed evaluation to determine how significant the dynamic relation table actively affects the generation of fuzzed inputs. The performance tests were evaluated on multiple Linux kernel versions, and each fuzzer

was given the same fixed amount of time to run. The presented data showed that the rate of coverage achieved by *HEALER* outperformed the other two fuzzers after around four to eight hours across all test kernels. The authors attribute this "speed-up" as a result of the relation table improving over time. This phenomenon was reinforced in their second set of tests in which *HEALER*'s performance was compared against *HEALER*, an identical copy of the original tool with the exception that this tool does not use dynamic learning to improve the relation table. It was found that the original *HEALER* experienced an increase in coverage rate that was more than double the rate of coverage *HEALER* experienced. The original tool also achieved 34% more coverage than *HEALER*.

Using dynamic learning to improve knowledge of syscall dependency, which is then used to actively seed fuzzing, is evidently effective. However, the present tool still is reliant on the manually written syscall descriptors from Syzkaller. But human written descriptors are laborious, error-prone, and may be incomplete. Thus, *HEALER* might benefit from the aid of a tool that automatically generates syscall descriptors, like SyzGen [3] or KSG [22]. These tools would only aid with the static step of initializing the first step. A more difficult and yet potentially fruitful task would be to incorporate the dynamic learning of syscall relations that *HEALER* uses into a tool like FuzzNG [2] which does not rely on handwritten syscall descriptors and instead dynamically modifies user space inputs at runtime to coincide with syscall specifications.

3.3 Hybrid Fuzzing

Hybrid fuzzing combines symbolic execution and fuzzing to increase code coverage. General fuzzing techniques use randomly generated inputs to guide the fuzzers. However, they often fail to pass the complex tests because random inputs are unlikely to satisfy those conditions. On the other hand, symbolic execution can reliably determine the satisfying values for all the conditions. Therefore, combining these two techniques is very promising in coverage-guided fuzzing. In this section, we will discuss *HFL* [11] that took a hybrid approach to improve the code coverage.

3.3.1 Technical Details

The use of a function pointer table in the Linux Kernel makes it very difficult to guide the fuzzer toward more code coverage. Because, to cover all the functions, traditional fuzzers have to enumerate all indices to the function pointer table correctly. However, random input generation used by most fuzzers is less likely to produce all such values. A symbolic execution engine can solve this issue. However, it suffers from a search space explosion problem because it can not differentiate whether a pointer is a code pointer or a data pointer. *HFL* [11] solves the issue by converting the function pointer table into a more explicit control flow transfer.

HFL follows the traditional coverage-guided fuzzing technique. It tries to generate inputs that extend the execution paths. When it detects hard branches, it passes those to a symbolic analyzer to find a satisfiable assignment to its arguments. A hard branch is defined as a branch that is hard to take due to so many constraints. To maintain the system call's state, it first tries to infer the sequence of the system calls that is required to make the system call

successful. *HFL* performs inter-procedural point analysis to infer the order of the system calls.

HFL utilizes two of the most popular tools to implement its intended functionality. First, it harnesses the power of Syzkaller [23] for fuzzing. Second, it incorporates S2E [4] symbolic execution engine to drive its symbolic execution.

3.3.2 Evaluation

Four major goals are set for the evaluation: effectiveness in finding bugs in Linux, coverage enhancement, efficiency in bug finding compared to other fuzzers, and performance. *HFL* was able to find 51 vulnerabilities in total, and 24 of them were previously unknown. The most common reasons for these vulnerabilities are integer overflow, memory access violation, and uninitialized variable access. *HFL* is more efficient in finding bugs compared to other fuzzers, notably Syzckaller [23]. It was able to detect the same number of vulnerabilities in less time than Syzkaller. Moreover, *HFL* can cover more number of blocks relatively faster than state-of-the-art tools such as SyzKaller and Moonshine. In numbers, it improves the coverage by 15% and 26% over Moonshine and Syzkaller respectively.

3.4 Detecting Complex Dependencies in Kernel Fuzzing

A paper by Hao et al. [7] addresses the limitations of fuzzing a kernel or any other program that is highly dependent on global state. Global state can lock certain codepaths from the fuzzer and may never be discovered during unaided fuzzing. Two previous works cited as "tackling the challenge" are MoonShine and HFL. However, the former is not directly focused on this paper's challenge and the latter had limited success and was unclear which dependencies it addressed.

3.4.1 Plan and Definitions

The researchers plan to (1) determine the coverage of state-of-the-art kernel fuzzers, (2) discover and categorize the unresolved dependencies, and (3) propose methods of handling them. The definition of an unresolved dependency is a condition that has not changed under fuzzing and accesses global memory. In order to manipulate the kernel's state to open a hidden codepath, the system detects effective write statements, which are statements that modify kernel state to the expected value of the unresolved dependency that hides the codepath.

Syscall information for fuzzers is condensed into templates, so that fuzzers have a starting point to work from. The paper focuses on modules with complete syscall templates, as template formats often cannot encode all dependency information.

The pipeline that the authors construct is a semi-automated dependency detector, where the automated parts find unresolved dependencies and emit their associated write statements. Manual review is need to determine the effectiveness of the write statements and the root causes of the unresolved dependencies.

3.4.2 Implementation and Evaluation

The pipeline maps the binary instructions to LLVM bitcode, which is easier to do if the bitcode is generated *after* the optimization pipeline. Since the bitcode is more "exotic" after optimization, Dr. Checker [14] was improved to handle these new cases. Additionally, Dr. Checker also needs domain knowledge of global reachability to taint the appropriate values.

The pipeline was tested on four modules: `cdrom`, `snd_seq`, `ptmx`, and `kvm`. These were picked for their rather complete set of templates and that they are not all from the same category of module. This was useful in finding the various root causes of unresolved dependencies, of which the paper lists five categories: dead code, environment dependency, partially unobserved, incomplete templates, specialized search. The dead code is hidden from compilers due to the global memory accesses, preventing definitive analysis. Some kernel modules (e.g., `cdrom`) interact with hardware, making some codepaths hidden on other hardware. A dependency can be partially observed when code that is not marked as part of some testcase opens or even executes a previously closed codepath. An example of this is the paper's Figure 6, where there are two kinds of user, but the syscall only uses one, leaving a branch unexplored. The templates that are the base of knowledge for the fuzzers may be incomplete, which they found to be rare, but one case was recorded in `kvm`. The specialized search encompasses special cases like interleaving threads or a specific sequence of calls.

3.4.3 Limitations and Future Direction

The paper marks its own limitations in its use of only four modules, the manual inspection required to prune hardware dependent code, and the manual analysis required. The two major directions for future research are cross-syscall propagation analysis and fuzzing hardware and syscalls simultaneously.

3.5 Fuzzing through network protocols

A critical part of an operating system is the network stack, in which it uses to communicate with any other foreign computer. This presents a large attack surface that can compromise the entire operating system and protocols may contain vulnerabilities. Ensuring that your kernel is secure against malicious network activity and patching that large attack surface is crucial to the security of the entire operating system. Thus, there is a demand for a fuzzer that generates network protocol test cases. Producing an effective network protocol fuzzer is difficult because states must be accounted for in stateful protocols (e.g., File Transfer Protocol), dependencies may exist between individual messages (e.g., encryption keys may be present in some messages that are necessary for future messages), and lastly coverage is difficult; many previous methods require human decided templates.

Song et al. present *SPFuzz* [19], a tool for fuzzing network protocols that address all three of the above problems, it maintains state, manages dependencies and increases coverage. The mutation of the inputs synthesizes AFL mutation with knowledge of the structure of each protocol. The original use case presented in the paper prescribes the use of *SPFuzz* to fuzz a target server, however, this tool can be repurposed to target the network implementation of an operating system.

3.5.1 Design Overview

There are five main components of *SPFuzz*: the description file, the interpreter, the core scheduler, the mutation engine and the target server. The description file is in charge of specifying the protocol specification for the given protocol, as well as handling state transitions descriptors. This is implemented by extending the descriptive language Boofuzz [16] to understand protocol specifications (and the specific field bounds and values within a message), state transitions and dependencies within and across messages.

Next, the interpreter will take both a protocol specification file and a state description file, configured by the previous component, and translate them into the format used throughout the rest of the tool; that is, it creates message entities and message state sequences. The core scheduler takes the message entities and state sequences and mutates them according to a hierarchical set of fields: head, content and sequence. The head of a message has a very limited range of possible values within an implementation of the protocol. It is within the category of either *length*, *marker*, *version* or *command*; each of which may be an integer or string within a specific range. The content field of a message does not have a definite bound and is normally random within the fuzzing process. This randomness is influenced by AFL's full-strategy for mutation. Lastly, sequence refers to the order that the messages are initiated. Each of these three categories are utilized by the mutation scheduler to create new test cases. The strategy of which is dictated by a message weight and a strategy weight that is maintained for each message and strategy. Once a test case is completed, the dependency handler checks if the header violates any dependencies of previous messages or within the message (e.g., the length specified in the header does not correspond to the length of the content).

Finally, the communication scheduler will send the generated messages to the target server and gather the responses. The responses are sent to back into the mutation scheduler to feed the generation engine.

3.5.2 Related Works

Previous fuzzing tools have been developed for the use of fuzzing network protocols while incorporating a sense of state. For example, SNOOZE [1] represents test cases using XML (similar to *SPFuzz* using its own descriptive language) and performs fuzzing on that class of objects. Boofuzz, as mentioned above, uses its own descriptive language to represent objects as well as state; it is implemented in Python. The present work extends the framework of Boofuzz to address the fact that it does not support dependencies within and across test messages. Other fuzzing tools like AutoFuzz [6] (which is somewhat stateful) and SecFuzz (which has a three-layer mutation strategy, similar to *SPFuzz*) addressed some nuances needed for protocol fuzzing however they fell short. The former failed to manage the encryption of the FTP process, and the latter's mutation strategy was too narrow. *SPFuzz* manages to take elements of previous works that succeed in a novel manner and synthesize a broad and versatile network protocol fuzzer.

3.5.3 Evaluation

Song et al. tested the evaluation of *SPFuzz* on three target severs running `Proftpd`, `Oftpd` and `OpenSSL`. The target servers were running on a virtual machine instance of Debian-i386 2.6.32-5-68. They aimed to measure three dimensions of coverage: function coverage, basic block coverage, and edge coverage. The two protocols used within the evaluation were FTP (for `Proftpd` and `Oftpd`) and TLS (for `OpenSSL`). *SPFuzz* tested twice using two mutation strategies within the AFL mutation stage: content mutation and full mutation strategy. Its performance was compared against that of Boofuzz. *SPFuzz* with full mutation strategy outperformed the other two strategies in all three metrics on all three test cases. This shows the necessity to mutation both head and sequence test cases, as well as maintain state and dependencies; *SPFuzz* with full mutation strategy outperforms the "ignorant" *SPFuzz*, that which only mutates the content of a test message.

3.6 System Call-Based State-Aware Linux Driver Fuzzing

Previous fuzzing techniques do not take the state of the system into account. Each run of the fuzzer is independent of the previous run. However, complex software like the Linux kernel consists of numerous interconnected components. Therefore, it is logical to consider the effect of a fuzzing run on the next run of the same fuzzer. The state fuzzing techniques provide feedback to the next run while exploring the test space. In this section, we will discuss *StateFuzz* [26], which defines a program state and how to detect such a state. It also provides techniques to utilize the program states to guide the fuzzer to previously unexplored code space.

3.6.1 Technical Innovation

First, the static analysis identifies some critical variables as program states. To minimize the combination of the state variables, they track the pairs of relevant variables in the program state. Moreover, they only track the ranges of value each variable can take. First, *StateFuzz* recognizes the program action, which is the functions or syscalls. Second, they analyzed the call graph and marked the accessed global variables as candidates. From the candidate set, it marked those variables as program variables that are updated by program action and loaded by another program action.

StateFuzz's fuzzing loop tries to cover as much code as possible, like the existing fuzzer Syzkaller [23]. It also tries to trigger different value ranges to trigger a new state. It uses three-tier feedback dimensions: code coverage, value range dimension, and extreme value dimension. Based on the signal it received from the feedback, *StateFuzz* retains the seed and reruns the test cases in a new context.

StateFuzz has three components. First, it uses a modified *DIFUZE* to identify the program actions. Second, it drives the static analyzer SVF to detect program states precisely. Finally, for fuzzing, it utilizes the existing fuzzer engine Syzkaller to fuzz the kernel driver.

3.6.2 Evaluation

They mainly compared their tool with the two state-of-the-art fuzzers: Syzkaller and HFL [11]. They primarily focused on determining whether their state representation is expressive enough to cover more codes than existing techniques. State variables recognition costs 6 hours, point to analysis by SVF takes 2 hours, and symbolic execution takes 7 hours. It is able to identify around six thousand state variables. The accuracy of *StateFuzz* in recognizing program actions is 99% and program states is 90%. *StateFuzz* is able to detect 19% more code edges than $Syzkaller_D$ and 15% more code edges than HFL. It detected 20 vulnerabilities, of which 19 are confirmed.

4 Advanced Fuzzing Techniques

The elementary techniques of standard fuzzing algorithms may be insufficient for the applied task of fuzzing the kernel. This is a difficult task because many bugs or control flow paths may be locked behind unprecedented sequences of input that may be inaccessible by the standard fuzzing algorithms. One solution taken in the rest of this paper is to use knowledge of the structure of the kernel to effectively fuzz the kernel. However, in this section we present some general fuzzing solutions that are ambiguous to it's use-case but may be applicable in further research on kernel fuzzing.

4.1 Fuzzing by Iterative Deepening

The creation of novel test inputs is one of the goals of fuzzing. Most of the existing work attempts this by first fuzzing and then reducing the large test case into a smaller input. The authors of *Bonsai* [24] propose that, instead of this "fuzz-then-reduce approach", a concise test input can be generated from iterative deepening, which they liken to growing a bonsai tree. Their particular case is that of generating well-typed programs for testing student compilers. However, this method could be applied to generating fuzzing programs in general, and for kernels in particular.

4.1.1 Goals

The paper describes five high level goals for bonsai fuzzing. Firstly, test creation needs to be automatic from a grammar and reference implementation. Secondly, code coverage and fault detection should be sufficiently high from the generated test corpus. Third, test cases should be minimal in terms of individual size and number of cases. Next, a large portion of cases should be "semantically valid". In the paper's example, validity is defined by the language the compilers accept, but could be defined for another use case. Finally, the technique should be generalizable to other testing targets.

4.1.2 Construction of a Bonsai Fuzzer

A bonsai fuzzer is bounded over some set of features to keep the size of the generated inputs small. This also allows for increasing the size of the input iteratively, ensuring that more

of the larger programs are semantically valid. Since these features are described by an n-dimensional vector, a lattice forms from the fuzzers, as each has successors and predecessors. Two "special" cases are named top and bottom in the lattice, top being the fuzzer with no successors and bottom being the one with no predecessors. Bottom is the start of fuzzing and is given a random seed to generate the first corpus. All of its successors are added to a worklist, which is then iterated to generate new seeds and corpora. The worklist is appended with the successors of its current contents, thus doing a breadth-first traversal of the lattice until there are no more successors.

4.1.3 Evaluation

The paper divided the evaluation into four axes: conciseness, semantic validity, comprehensiveness, and fault detection. Conciseness showed significant double-digit improvements in file size and test count. Validity showed similar, though smaller, improvements over the baseline "fuzz-then-reduce" system. Comprehensiveness showed little change, while fault detection is similarly affected.

4.2 Discovering magic bytes

Certain pieces of software contain bugs or crashes are triggered by a sequence of "magic bytes". Many fuzzers struggle to accurately discover these magic bytes because many times they are very long and specific. For example, in the case where a bug was hidden behind multiple conditional branches each, comparing a character in the magic byte sequence, a coverage-based fuzzer like AFL might successfully find the magic sequence by mutating bytes of the input and fixing a byte anytime that input increased the amount of coverage, i.e., one more branch leading to the bug was taken. However, if this bug was hidden behind one condition, these fuzzers would struggle to succeed. This is because mutating all but one byte correctly would still not increase the coverage because the condition would fail. Thus, the fuzzer would not ever converge to the magic sequence. *Steelix* [13] addresses this problem by responding to both coverage and comparison progress.

4.2.1 Technical Details

Steelix is a grey-box mutation-based fuzzing algorithm. It uses mutation and expands the coverage of the input range, as well as react to direct comparison results at runtime. The latter mechanism enables *Steelix* to converge on magic byte sequences even when path coverage does not expand. By grey-box, the authors mean that access to the binary is required, not the source code. There are three phases of *Steelix*; static analysis, binary instrumentation, and the fuzzing loop. The static analysis filters out "uninteresting" comparisons and organizes instructions into *basic blocks*. During the binary instrumentation stage, *Steelix* modifies the binary in the basic blocks to record the progress of comparisons in shared memory, as well as record the actual value of the comparison [mention definition of value]. Lastly, the fuzzing loop with perform mutations on an input that is adaptive based on both coverage and comparison result.

The static analysis step first marks every comparison within the binary; the implementation presented in Yuekang Li et al. uses the x86 32-bit instruction set, thus the static analysis step marks the `cmp` and `test` instructions as well as string comparison instruction: `strcmp`, `strncmp`, `memcmp`. Then, the "uninteresting comparisons" are ignored. These are one-byte comparisons and comparisons of function return values. An explanation as to why the latter is excluded briefly surmounts the fact that the input bytes being flipped does not directly map to the behavior of the output of a function; think hash functions, bytes in one dimension affect the output in other dimensions. The text states that this does not reduce complexity. The remaining comparisons are collected into a set of lists. The authors then present a way to compress the comparison progress, which is defined as which bytes "succeed" in the comparison. The binary is then instrumented to record the comparison progress during the runtime of the program.

4.2.2 Evaluation

Steelix was evaluated on the two benchmarks (`LAVA-M` and `DARPA CGC`) and five programs: `tiff2pdf`, `tifcp`, `pngfix`, `gzip`, `tcpdump`. *Steelix*'s performance was compared against AFL, VUzzer and AFL-lafintel. On the `LAVA-M` dataset, *Steelix* outperformed every other fuzzer in all but one program in the set, `uniq`. Otherwise, *Steelix* found at least around twice the amount of bugs than the other fuzzers. On `CGC`, *Steelix* slightly outperformed its competitor, AFL-dyninst, as *Steelix* finds seven out of eight bugs and AFL-dyninst finds only six.

Steelix and AFL-dyninst were also compared on their coverage ability on the five selected programs. The programs were fuzzed for either one or three days (consistently between each fuzzer). It was found that the coveraged achieved by *Steelix* significantly out-measured that of AFL-dyninst.

5 Data Races

A data race bug occurs when two or more concurrent threads try to access the same shared resource, and their execution sequence or time may lead to some undefined behavior. This is very hard to detect because one has to not only control the shared memory access but also order the threads in a particular sequence to reproduce the behavior. The traditional coverage metric is not sufficient to capture the complete picture in the concurrency world. For example, a data race can only be triggered by the execution of some instructions in a particular order. Since most coverage-guided fuzzers do not care about the order of the instructions, they would probably not find the race bugs. To this end, *Razzer* [10] is the first that focuses on finding data race bugs in the Kernel. *Razzer* combines static analysis and deterministic thread interleaving techniques to find the data race bugs. Another approach suggested by *Krace* [25] improves the performance.

5.1 Finding Kernel Race Bugs through Fuzzing

5.1.1 Design of *Razzer*

Razzer [10] finds data race bugs in two stages. First, it statically analyzes the kernel to point out some of the potential targets that may lead to race conditions. Second, it then uses dynamic analysis to actually find out the race bug. In this step, they run two threads concurrently in different order to trigger the potential race bugs.

During static analysis, they marked a region as $RacePair_{cand}$, a potential site for a race condition, if they access the same memory region, one of them is a write instruction and they execute concurrently. After collecting $RacePair_{cand}$ set, *Razzer* finds out the $RacePair_{true}$ through a dynamic analysis. To perform a dynamic analysis, they utilized a modified hypervisor. The modified hypervisor allowed them to set per core breakpoint. Therefore, they can control the execution sequence of the threads more precisely. The breakpoint facility was provided by the hardware debug register. However, one of the key challenges was determining whether their intended guest program hit the breakpoint because some other programs might also hit the breakpoint. In this case, *Razzer* utilizes virtual machine introspection (VMI) to find out the kernel thread ID. The next task is to determine the order of the threads' execution that results in the data race. To achieve this, they have added a call to `hcall_set_order` to set the order of the execution of the thread. Finally, they checked the result by confirming that the threads were accessing the same memory location.

Each of the phases of the fuzzer is implemented in two parts: generator and executor. For the first stage, the single thread generator generates a user program with random syscalls. It then passes the program to the executor, which runs the program and annotates the area of the program if it covers a $RacePair_{cand}$. In the second stage, a multithreaded generator generates a multithreaded user program and passes it to the executor. Then, the executor runs the program to figure out the $RacePair_{true}$ from the annotated program.

5.1.2 Evaluation

Razzer[10] was implemented as a pass in LLVM. They used a modified QEMU for the hypervisor. Their evaluation showed that the race bugs found by the *Razzer* had a severe impact on the reliability and security of the system. For instance, a bug with buffer overflow or use-after-free can allow an attacker to overwrite a credential storage and gain root privilege. *Razzer* was able to detect some harmful bugs that previously existed in the Linux kernel. Their analysis revealed that the chance to observe these bugs is very low in practice; therefore these bugs remained potent for so long. In addition to finding bugs, *Razzer* generated a detailed report that was very useful in analyzing the root cause of the bugs. *Razzer* was very effective in finding bugs. For 578 million memory accesses, it was able to produce only 300 thousand $RacePair_{cand}$. The overhead of the hypervisor was around 5 μs, which is rather small. The predominant hyper-calls that incurred the overhead the most were `hcall_set_bp` and `hcall_nop`.

5.2 Alias Fuzzing Data Race

5.2.1 Design of *Krace*

Krace [25] introduces a new metric called alias coverage for the concurrency dimension. It records accesses if the accesses are to the same memory address from different threads.

For the input generation, *Krace* utilizes a specification for each syscall. Other than that, it uses the usual mutation, addition, deletion, and shuffling to evolve the input. Thread scheduling is very hard to control. *Krace* manually injects some delay instruction to weakly control threads. At first, the execution of fuzzing starts and tries to cover as much as branch possible. The logs of this execution are dumped and checked for data races. A few separate background threads work on checking the potential data races. The reason behind this separation of executions is that the checking tasks required a lot of time. This slows down the overall speed of the fuzzing. *Krace* filters out the benign data races using several heuristics. In each run of fuzzing, *Krace* freshly booted the Kernel in the emulator. This helps *Krace* avoid the aging OS problem.

Krace suffers from replaying the end-to-end execution it conducted before. This means it provides limited support for debugging. However, it provides a full call stack as well as a branching history to compensate for its limitation.

5.2.2 Evaluation

The authors of *Krace* [25] evaluated their fuzzing technique on two popular file systems: `btrfs` and `ext4`. *Krace* is able to find nine new harmful and 11 benign data races. Their evaluations show that `btrfs` is inherently more concurrent than `ext4`. *Krace* incurs a significant overhead on the execution. One execution took at least seven seconds to complete. This is quite slower than compared to other fuzzers such as Syzkaller [23]. It took 168 hours to discover all of these bugs. Another reason for the overhead is its offline checking processes that check the execution logs for data races. The delay injection mechanism helps *Krace* to cover 28.7% and 12.3% more in `btrfs` and `ext4`.

6 Hardware-Related Fuzzing

Hypervisors are potentially more essential to fuzz, due to being the layer below OS kernels and increasingly used. One essential use of hypervisors in modern cloud infrastructure is the security and separation they provide between operating systems. They also have large attack surfaces that are difficult to secure fully, or even discover vulnerabilities. Additionally, fuzzing user space components often relies on feedback mechanisms to discover new inputs. Kernels do not have such mechanisms and will have a difficult time implementing them due to non-determinism and the risk of crashes. Hardware is thus both needed to improve fuzzing [18], and is a fuzzing surface itself [17, 20].

6.1 High-Dimensional Hypervisor Fuzzing

An interface can be termed a "dimension of fuzzing". In kernel fuzzing, the dominant dimension has been the syscall interface, due to the large amount of functions it exposes. However, the hardware a kernel runs on is a second, more difficult dimension to fuzz over. Since changing hardware frequently enough for good fuzzing speed is near impossible, hypervisors also need to be fuzzed to ensure they model the hardware correctly. *Hyper-Cube* [17] is such an attempt to fuzz hypervisors.

6.1.1 Background and Design

The authors target the x86 platform, as it is the dominant architecture across servers and desktops. x86 has three mechanisms for input (ports, memory mapping, and direct memory access), all of which need to be handled by the fuzzing system. The threat model for the systems assumes a malicious kernel that is attempting to "break out" via hypercalls that could either gain control over another virtual machine or the hypervisor itself.

To achieve this behavior while maintaining performance, a custom operating system is needed to control the software interface. This operating system is the eponymous *Hyper-Cube OS*, which is a minimal kernel around a bytecode interpreter. This bytecode interpreter (named *Tesseract*) has a "fuzzer-friendly" input, in that it is difficult to create an invalid program. Additionally, the interpreter can generate its own pseudorandom instruction stream and can minimize a test case for human readability.

6.1.2 Evaluation

Four research questions were discussed in the evaluation. The first was the ability to uncover bugs in different hypervisors, which was demonstrated by finding 54 bugs across six hypervisors. The second was whether the fuzzer could rediscover old vulnerabilities. VDF was chosen as the specific competitor in this case and was beaten by *Hyper-Cube* in only 10 minutes when VDF's [8] search took 60 days. The third question was coverage achievable by *Hyper-Cube*. With VDF as the comparison and *Hyper-Cube* given only 10 minutes per device, VDF has the lesser coverage in all except one two devices. The difference is by the authors ascribed to *Hyper-Cube*'s ability to interleave operations over memory and handing the device under test a pointer to said memory. The final question was the overall performance compared to other fuzzers. The opponent this time was CFAFT [5], a compiler-based approach. The paper conservatively showed that generating test inputs and restarting after crashes were significantly faster for *Hyper-Cube*, while memory footprint was only 5 times smaller with *Hyper-Cube*. One limitation noted for *Hyper-Cube* was the lack of ISA device enumeration.

6.2 Hardware-Assisted Fuzzing for Kernels

Hypervisors not only need to fuzz their hardware interface; they may also be useful in fuzzing kernels, given some hardware support. An initial implementation of this is *kAFL* [18], which is an Intel Labs project to fuzz kernels.

6.2.1 Background and Overview

The hardware requirements for *kAFL* are hardware virtualization (Intel's VT-x) and the previously mentioned Processor Trace. Intel VT-x gives hardware acceleration for VMs, while Processor Trace uses main memory to generate traces for the kernel currently under fuzzing.

The fuzzing process involves three components: the fuzzing logic, the modified KVM/ QEMU combo, and the fuzzing agent that runs in the user mode of the kernel being fuzzed. When the fuzzing target is loaded, a hypercall sends the address of the panic handler to the QEMU, which patches it to improve response times to crashes. The main fuzzing logic produces inputs, which are passed to the agent via QEMU. While the fuzzing occurs, the trace data is decoded to keep the buffer size under control. The hypervisor is notified when the fuzzing run is complete and sends the fuzzing map to the main logic.

Some features of the components are particular to the use case of fuzzing kernels. For example, *kAFL* is more parallel than normal AFL, which is useful for the non-CPU-bound tasks. Another example is the user agent, which is implemented in two components. The first is a loader that sets up a binary passed to it and monitors it for crashes. This "input binary" is the second component and the part which actually executes the syscalls. Finally, challenges of non-determinism, statefulness, and asynchrony are handled via two mechanisms: filtering interrupts from the trace data and ignoring blocks that are sometimes executed.

6.2.2 Evaluation

The system was evaluated against the three dominant operating systems: Windows, macOS, Linux. Windows seems to have been a rather difficult target to fuzz, as mounting volumes was rather slow and the generic syscall fuzzing encountered user mode callback related crashes. Linux was a much more performant run and more fruitful, as 160 bugs were found in ext4. macOS was similar to Linux, with 150 in the HFS driver and 220 in APFS. *kAFL* was able to detect previously known bugs, like CVE-2016-0758, and surprised the authors by finding CVE-2016-8650, a previously unknown bug. All bugs found in the paper were reported by the time of writing, but seemingly only the Linux ones have been acknowledged. Performance comparisons were made against TriforceAFL [9], which *kAFL* handily beats in terms of discovery. Syzkaller [23] was not used for comparison, as it starts with a lot of domain knowledge in the form of templates, and it only works on Linux.

6.3 A driver fuzzing framework

There is a risk of trust between the user space device APIs provided by the operating system and potentially compromised devices. Dukyong Song et al. present *PeriScope* [20] limits driver accessibility to minimize security risks within device drivers that may compromise the system to malicious hardware. Along with the *PeriScope* implementations, Song et al. introduce *PeriFuzz* to discover and simulate such attacks on the device drivers which is built off the mechanisms or *PeriScope*. This serves as the most useful discovery for our research as *PeriFuzz* can but used to test and measure the vulnerability of device drivers within the kernel. This paper presents a general framework of how *PeriScope* can be utilized to fuzz drivers, however the choice of fuzzer (AFL in the present paper) is open to the researcher.

6.3.1 Technical Details

PeriScope works by monitoring the interaction between device drivers and the rest of the system through *memory-mapped I/O* and *direct memory access*. When a device driver is configured, *PeriScope* will keep track of all MMIO and DMA mappings performed by the driver and removes these pages from the kernel memory space. Thus any driver use of this page will trigger a page fault which gives *PeriScope* a signal in the form of an exception to respond before then resolving the page fault exception. Upon every page fault, *PeriScope* first checks if that memory region is mapped by the driver. If it is, it will call user set hook functions with the given information. It will then execute the instruction, this time allowing the page fault to resolve. Upon the instruction completion, the driver-set hook is called for that given instruction. Lastly, *PeriScope* removes the page from the kernel page table once again and continues instruction; this last step allows *PeriScope* to retain the ability to catch any memory accesses to this page. In order to return control back to *PeriScope* after executing the faulting instruction, the processor's single-stepping support is utilized.

The implementation and design of *PeriScope* is utilized by *PeriFuzz* to fuzz for security risks and bugs in a device driver that result from malicious peripherals or unknown undefined behaviour of the peripheral. The three components used in the design of *PeriFuzz* in Song et al. is composed of the fuzzer, the executor and the injector (the design and implementation, however, were created to be modular). The fuzzer runs in user space and in the present implementation uses AFL. The executor acts as a bridge between the user space fuzzer and the kernel space injector. It exists in user space and communicates with the injector through designated shared memory. The executor is also in charge of checking if a crash occurred by remembering the last input delivered before a crash. Lastly, the injector directly interacts with *PeriScope* by setting a hook before a potential memory access instruction by the driver that gives *PeriFuzz* full knowledge of data received from the device. It also intervenes and replaces the output of the device with the generated input from the executor (if one is ready).

6.3.2 Evaluation

The present *PeriFuzz* implementation was tested on two Wi-Fi chips by the vendors Broadcom and Qualcomm. The test machines containing these devices were a Google Pixel 2 with Qualcomm's chip and `qcacld-3.0` device driver and the Samsung Galaxy S6 with Broadcom's chip and `bcmdhd4358` device driver. An investigation of high traffic memory regions mapped by the two drivers and fuzzing was enabled for these regions. Fuzzer seeds were generated based off of monitored traffic and *PeriFuzz* was used to fuzz for bugs. In their tests, fifteen bugs were found total, nine of which were novel. It was also found that *PeriFuzz* was somewhat performant with room to be optimized. Most of the execution time was spent waiting for a new input to be generated. This incurred a greater number of page faults to be "unfuzzed".

7 Syscall Specification Generation

In order to fuzz a kernel through the interface of it's syscalls, it is optimal that a fuzzing tool has knowledge of the syntax and domain of arguments for each syscall; this way only

valid or feasible inputs are generated and fuzzing computation isn't wasted on meaningless input. Traditionally, the task writing descriptors for syscalls was performed manually. In this section we explore methods to fuzz the kernel through syscalls by either automatically generating syscall descriptors or avoiding the need of descriptors all together.

7.1 Specification Generation

Traditional fuzzers require the specifications of the system calls. One could collect specifications by inspecting the relevant documentation manually. However, it requires tremendous effort and expertise in the relevant fields. The alternative approach is to enumerate automatically, which will be much more preferable. However, the key challenges to this approach are identifying the appropriate submodules and inputs to the submodules. In Linux, each of the submodules implements a specific operation. A submodule might register itself in runtime dynamically, in which case it is impossible to detect entry by static analysis. This paper presents a method to generate specifications for Linux Kernel system calls automatically. With these new specifications, they evaluated the state-of-the-art fuzzers. The experimental results showed that *KSG* [22] achieved more code coverage than before.

Solve the variable aliasing using CSA because CSA gives the same symbol value or allocates the same memory.

7.1.1 Technical Details

First, *KSG* [22] compiles the Kernel source code into a bootable image and generates source code information in Clang AST. Then, *KSG's* entry extraction module hooks probe before and after the different syscall functions. These probes are installed utilizing Linux Kernel's eBPF functionality.

Second, *KSG* [22] triggers execution for different Linux modules. When the execution is going on, *KSG's* probe collects trace information such as the entry for each syscall, signature of the function, various registration points, input data type, etc. Next, the trace information is used to execute the symbolic execution engine of the Clang Static Analyzer(CSA) to identify the variables and parameters data types. To solve the aliasing between variables, *KSG* maps each symbol to a memory region. The aliased variables will get the same symbol name or be mapped to the same memory region. Additionally, *KSG* collects range constraints for each variable.

Third, *KSG* utilizes the previously collected information about submodules, variables, and parameters to generate the specifications in Syzlang. First, it generates a resource type for each submodule. Next, it generates the specifications for the system call that creates the resource.

7.1.2 Evaluation

KSG [22] defines three goals for its evaluation: (1) performance in generating system calls specifications, (2) effectiveness of the generated specifications in improving the code coverage, and (3) effectiveness in finding bugs. It took *KSG* 5 hours to scan 78 sockets and 1,098 device files. It was able to generate 1,460 new specifications. The generation process was automatic,

and no manual intervention was needed. *KSG* improves the code coverage of *Syzkaller* [23] and *MoonShine* [15] by 22% and 23% respectively. Moreover, *KSG* detected 138 unique vulnerabilities, out of which 26 were confirmed to be unknown at that time.

7.2 Fuzzing syscalls without descriptors

User-space applications interface with the kernel through syscalls which are called by executing the syscall instruction; the arguments are set to certain registers before the instruction is to be executed. These arguments may be specific constant values important for that specific call, or they may be a pointer or file-descriptor that refer to other objects for the syscall to use as arguments. Syscalls with pointers as arguments are expecting that pointer to reference a specific data-type dependent on the syscall as well other arguments. Furthermore, chunks of data within that referenced object can internally instruct the syscall implementation to call further syscalls to cover a highly dependent or unique control flow. Similarly, syscall implementation or control flow may be dependent on file-descriptor arguments; that is, the type of file being referenced will lead to different implementations of the `write` syscall for example. Thus, while there are hundreds of syscalls in the Linux kernel, the expected data-type referenced by points and types of file referenced by file-descriptors drastically increase the possible control flow paths.

Evidently, creating an effective syscall fuzzer for the Linux kernel is difficult. Passing randomly fuzzed pointer or file-descriptor values will be ineffective because they will most like not refer to a valid object or file for the syscall. Thus, the fuzzer must have knowledge of the valid data-types to be reference for a given context, device, module, etc. Previous fuzzers like Syzkaller [23] rely on manually written descriptors to get around this problem. However, human written descriptors are error-prone and labor-intensive.

The present tool achieves substantial coverage without the need for handwritten descriptors.

7.2.1 Critique on Previous Work

There have been previous attempts to solve the problem present in Syzkaller in which syscall descriptors must be written by hand. That is, other projects focus on generating such descriptors through some sort of static or dynamic analysis. Such examples are Difuze, IMF, SyzGen [3] and KSG [22]. The former three do not require source code analysis and generate their descriptors off of the API. However, this vastly limits any the ability to generate descriptors for any internal syscalls.

The fundamental issue present with current syscall fuzzers is that as the scale of the Linux kernel and it's modules increase, Syzkaller hinges on many engineers to manually keep the descriptors up to date; the performance of Syzkaller depends on it. And that cannot efficiently scale forever.

7.2.2 *FuzzNG* Design Overview

Bulekov et al. present *FuzzNG*, a fuzzing tool that does not require any prior knowledge of syscall descriptors and achieve better coverage than Syzkaller. The tool works by dynamically

modifying the input-space to adhere to the syscalls expected objects that are referenced. There are two separate types of arguments that *FuzzNG* must address: pointers and file-descriptors.

To handle the issue of pointer arguments, upon any memory access within a dereferenced pointer argument from the kernel, *FuzzNG* will "prepare" the memory before the kernel access it. That is, *FuzzNG* will first populate the accessed region with fuzzed data before returning control to the kernel. Whenever the kernel access user-space memory, it does so through safe kernel functions to retain the security of the kernel from a potentially malicious user-space application. For example, this can be implemented using the `copy_from_user` or `get_user` kernel functions. Thus, *FuzzNG* is able to hook into these API calls to properly handle the memory reshaping.

To handle the issue of file-descriptors, *FuzzNG* remaps a randomly fuzzed file-descriptor (`fd`) values to existing `fd`s of files that are currently opened by the process. *FuzzNG* keep tracks of every open file that the process has. Then every time that a (probably invalid) fuzzed `fd` is passed into a syscall argument, the kernel must call `fdget()` to find the associated open file in the kernel. This is where *FuzzNG* will hook, and replace that with a valid file. Then, to maintain the fuzzing process, *FuzzNG* will use `dup2` to map the invalid fuzzed `fd` to the correct open file.

There are three components of the *FuzzNG* implementation; the fuzzer, the kernel modifications and the user-space agent. *FuzzNG*'s fuzzing component is called *QEMU-Fuzz*, which is based off of the QEMU-KVM hypervisor and uses the *libFuzzer* as an input fuzzing engine. By separating the fuzzing engine and the test kernel with virtualization, crashes and timeouts are no longer threatening to any other component of the fuzzer. The kernel module introduced by *FuzzNG* is called *mod-NG*. This features all the kernel-space hooks necessary for the fuzzer. Lastly, they user-space component is called *NG-Agent*. This component will call the syscalls to be tested in the test kernel. Upon initialization, *NG-Agent* first must configure `kcov` which allows *NG-Agent* to share memory with the kernel to receive coverage-data for the kernel, inflate its process memory which permits *FuzzNG* to place user-space hooks, create necessary threads and make a connection with *QEMU-Fuzz*. The *NG-Agent* then interprets fuzzed inputs from *QEMU-Fuzz* and feeds syscalls to the test kernel.

7.2.3 Evaluation

FuzzNG was tested to evaluate the coverage and edges. *FuzzNG* was compared against Syzkaller and HEALER. It was found that on average *FuzzNG* covered 102.5% of the coverage that Syzkaller achieved. And in all parts of code that were tested (except one), *FuzzNG* covered code regions that were not covered by either Syzkaller or HEALER. The lead in coverage achieved by *FuzzNG* was initially outperformed by Syzkaller because of it's intentionally written descriptors for syscalls. However, Syzkaller's coverage eventually reached an upper bound and *FuzzNG* proceeded to outperform. Furthermore, *FuzzNG* found nine previously undiscovered bugs in the Linux kernel. Five of these components were already (falsely) confirmed by Syzkaller to be safe.

8 Conclusion

The large interface of the kernel has let research explore many options of fuzzing it. The early need for information on system calls led to template-focused systems, as they were quicker at gaining coverage of the kernel from the interface. Hardware-based setups have also been implemented to gain coverage from the other side of the kernel "abstraction". The multithreaded nature of some programs would expose implementation bugs that would otherwise go undetected, necessitating specialized fuzzers. Many fuzzers use test cases of multiple system calls to expose bugs in the internal state of the kernel, as many "operations" are composed of multiple calls.

References

[1] Greg Banks et al. "SNOOZE: toward a Stateful NetwOrk prOtocol fuzZEr". In: *Information Security: 9th International Conference, ISC 2006, Samos Island, Greece, August 30-September 2, 2006. Proceedings 9*. Springer. 2006, pp. 343–358.

[2] Alexander Bulekov et al. "No Grammar, No Problem: Towards Fuzzing the Linux Kernel without System-Call Descriptions". In: *NDSS*. 2023. URL: https://www.ndss-symposium.org/wp-content/uploads/2023/02/ndss2023_f688_paper.pdf.

[3] Weiteng Chen et al. "SyzGen: Automated Generation of Syscall Specification of Closed-Source macOS Drivers". In: *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. CCS '21. Virtual Event, Republic of Korea: Association for Computing Machinery, 2021, pp. 749–763. ISBN: 9781450384544. DOI: 10.1145/3460120.3484564. URL: https://doi.org/10.1145/3460120.3484564.

[4] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. "S2E: A platform for in-vivo multi-path analysis of software systems". In: *Acm Sigplan Notices* 46.3 (2011), pp. 265–278.

[5] YoungHan Choi et al. "Call-Flow Aware API Fuzz Testing for Security of Windows Systems". In: *2008 International Conference on Computational Sciences and Its Applications*. 2008, pp. 19–25. DOI: 10.1109/ICCSA.2008.32.

[6] Serge Gorbunov and Arnold Rosenbloom. "Autofuzz: Automated network protocol fuzzing framework". In: *Ijcsns* 10.8 (2010), p. 239.

[7] Yu Hao et al. "Demystifying the dependency challenge in kernel fuzzing". In: *Proceedings of the 44th International Conference on Software Engineering*. 2022, pp. 659–671.

[8] Andrew Henderson et al. "VDF: Targeted Evolutionary Fuzz Testing of Virtual Devices". In: *International Symposium on Recent Advances in Intrusion Detection*. 2017. URL: https://api.semanticscholar.org/CorpusID:32357905.

[9] Jessie Hertz and Tim Newsham. *Project Triforce: Run AFL On Everything*. Tech. rep. NCC Group, 2017.

[10] Dae R Jeong et al. "Razzer: Finding kernel race bugs through fuzzing". In: *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2019, pp. 754–768.

[11] Kyungtae Kim et al. "HFL: Hybrid Fuzzing on the Linux Kernel." In: *NDSS*. 2020.

[12] Seulbae Kim et al. "Finding semantic bugs in file systems with an extensible fuzzing framework". In: *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 2019, pp. 147–161.

[13] Yuekang Li et al. "Steelix: program-state based binary fuzzing". In: *Proceedings of the 2017 11th joint meeting on foundations of software engineering*. 2017, pp. 627–637.

[14] Aravind Machiry et al. "DR. CHECKER: A Soundy Analysis for Linux Kernel Drivers". In: *26th USENIX Security Symposium (USENIX Security 17)*. Vancouver, BC: USENIX Association, Aug. 2017, pp. 1007–1024. ISBN: 978-1-931971-40-9. URL: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/machiry.

[15] Shankara Pailoor, Andrew Aday, and Suman Jana. "MoonShine: Optimizing OS fuzzer seed selection with trace distillation". In: *27th USENIX Security Symposium (USENIX Security 18)*. 2018, pp. 729–743.

[16] J. Pereyda. *Boofuzz: Network Protocol Fuzzing for Humans*. URL: https://boofuzz.readthedocs.io/en/latest.

[17] Sergej Schumilo et al. "HYPER-CUBE: High-Dimensional Hypervisor Fuzzing." In: *NDSS*. 2020.

[18] Sergej Schumilo et al. "kAFL: Hardware-Assisted feedback fuzzing for OS kernels". In: *26th USENIX security symposium (USENIX Security 17)*. 2017, pp. 167–182.

[19] Congxi Song et al. "SPFuzz: a hierarchical scheduling framework for stateful network protocol fuzzing". In: *IEEE Access* 7 (2019), pp. 18490–18499.

[20] Dokyung Song et al. "Periscope: An effective probing and fuzzing framework for the hardware-os boundary". In: *2019 Network and Distributed Systems Security Symposium (NDSS)*. Internet Society. 2019, pp. 1–15.

[21] Hao Sun et al. "HEALER: Relation Learning Guided Kernel Fuzzing". In: *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. SOSP '21. Virtual Event, Germany: Association for Computing Machinery, 2021, pp. 344–358. ISBN: 9781450387095. DOI: 10.1145/3477132.3483547. URL: https://doi.org/10.1145/3477132.3483547.

[22] Hao Sun et al. "KSG: Augmenting Kernel Fuzzing with System Call Specification Generation". In: *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. Carlsbad, CA: USENIX Association, July 2022, pp. 351–366. ISBN: 978-1-939133-29-20. URL: https://www.usenix.org/conference/atc22/presentation/sun.

[23] *syzkaller: Linux syscall fuzzer*. https://github.com/google/syzkaller.

[24] Vasudev Vikram, Rohan Padhye, and Koushik Sen. "Growing a test corpus with bonsai fuzzing". In: *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE. 2021, pp. 723–735.

[25] Meng Xu et al. "Krace: Data race fuzzing for kernel file systems". In: *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2020, pp. 1643–1660.

[26] Bodong Zhao et al. “StateFuzz: System Call-Based State-Aware Linux Driver Fuzzing”. In: *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, Aug. 2022, pp. 3273–3289. ISBN: 978-1-939133-31-1. URL: https://www.usenix.org/conference/usenixsecurity22/presentation/zhao-bodong.