SyzRetrospector: A Large-Scale Retrospective Study of Syzbot

Joseph Bursey
Computer Science Dept.
University of California, Irvine
Irvine, California
jbursey@uci.edu

Ardalan Amiri Sani
Computer Science Dept.
University of California, Irvine
Irvine, California
ardalan@uci.edu

Zhiyun Qian
Computer Science and Engineering Dept.
University of California, Riverside
Riverside, California
zhiyunq@cs.ucr.edu

***Abstract*—Over the past 7 years, Syzbot has fuzzed the Linux kernel day and night to report over $6{,}700$ bugs, of which nearly $5{,}500$ have been patched. While this is impressive, we have found that 25% of bugs take longer than 738 days to find. Moreover, we have found that current metrics commonly used, such as time-to-find and number of bugs found, are inaccurate in evaluating Syzbot since bugs often spend the majority of their lives hidden from the fuzzer. In this paper, we set out to better understand and quantify Syzbot's performance and improvement in finding bugs. Our tool, SyzRetrospector, takes a different approach to evaluating Syzbot by finding the earliest that Syzbot was capable of finding a bug, and why that bug was revealed. We use SyzRetrospector on a large scale to analyze 695 bugs and find that 40% of bugs are hidden for more than 258 days before Syzbot is even able to find them. We further present findings on why bugs were revealed to Syzbot (*i.e.*, their revealing factors), the effort required to reveal bugs, the trends in delays, and how the location of bugs affects these delays. We also provide key takeaways for improving Syzbot's delays.**

I. Introduction

Over the past 7 years, Syzbot, one of the largest continuous fuzzing projects, has managed to find over 6,700 bugs [1] in the Linux kernel. Despite this number, the question "How well has Syzbot performed?" cannot be precisely answered. Syzbot has found about 2.6 bugs per day over its lifetime, but we do not know how many bugs were missed. Is Syzbot finding bugs faster than they are being introduced? What is limiting it from finding more bugs?

In the past, the aggregate time-to-find of bugs has been used to evaluate and compare fuzzer performance. This metric considers the period of time from when the bug was introduced to when it was found by the fuzzer, regardless of change to the fuzzer or target code base. However, in a small pilot study (§IV), we find that time-to-find is inaccurate. Bugs are not always findable by Syzbot, sometimes due to the fuzzer's capability or code in the kernel hiding the bug. Indeed, 68% of bugs found by Syzbot have spent some amount of their lifetimes hidden from the fuzzer.

Based on this observation, we divide the overall delay into two parts: one where the bug is hidden from the fuzzer, and another where the bug is revealed, but not yet found. We call these delays D_1 and D_2 respectively, and use them as key metrics in our study. These delays have the advantage of being rooted in the context of a bug's lifetime, while still being true to the fuzzer's capabilities.

We present SyzRetrospector, a tool purpose-built to identify D_1 and D_2 by finding the dividing line between them. SyzRetrospector is capable of going back in time to faithfully recreate Syzbot's fuzzing environment at a given point in time in order to find the exact day and reason Syzbot was first able to find a bug. The resulting reason, which we call the *revealing factor*, is the dividing line between the two delays. Across 4.5 months, we used SyzRetrospector on a large scale to find the revealing factors of 695 bugs[1]. With a solid understanding of how long bugs are hidden for and how long until they are found afterwards, we present our evaluation of Syzbot's performance over the past 7 years.

We identify 5 different revealing factors, each with different behaviors, and highlight that some may be hard to improve such that the bug finding delay decreases. We present a breakdown of the bug finding delay as D_1 and D_2, and show how they can be used to evaluate the fuzzer over time. Importantly, we demonstrate that D_1 and D_2 follow power law distributions and give strong evidence that they are independent of each other. This means that the delays must be improved by separate means. We also give evidence that Syzbot is approaching a local ideal state of fuzzing where most found bugs are revealed shortly after they are introduced. We then provide strategies for improving D_1 and D_2 such as using CVEs to find gaps in Syzbot's syscall description set.

II. Background: Continuous Fuzzing and Syzbot

Continuous fuzzers run non-stop alongside development of long-standing projects like Linux, finding bugs as they are introduced. Such fuzzers can amass a large corpus of test cases and maintain a complex fuzzer state across years of testing, giving them a distinct advantage with time.

Syzbot, Google's coverage-guided continuous fuzzer, began fuzzing the Linux kernel in 2017, and continues today with 27 instances called *managers*, which are instances of Syzkaller (a Linux kernel fuzzer). Each manager in turn fuzzes using 20

[1]We have open-sourced SyzRetrospector and the results of this study with instructions on how to set up and run the tool. The repository can be found on github: https://github.com/trusslab/syzretrospector

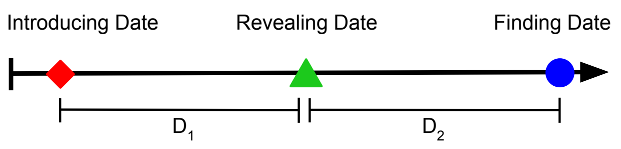


Fig. 1: The lifetime of a bug.

CPU cores spread across 10 virtual machines. To be clear, Syzkaller interfaces with and fuzzes the kernel, while Syzbot is the apparatus that allows Syzkaller to fuzz continuously. Syzbot oversees each instance of Syzkaller, sharing test cases between them and periodically updating both Linux and Syzkaller. This way, Syzbot always fuzzes relevant kernel versions and has the most up to date version of Syzkaller to fuzz with.

A Syzkaller instance begins by generating a number of test cases from *syscall descriptions*, a manually written API describing the kernel syscalls available to Syzkaller. Syzkaller runs these inputs on the target kernel and adds to the corpus those that provide new coverage or generate a crash. It then selects inputs from the corpus to be mutated and run again, and the cycle repeats.

Sanitizers instrument the kernel and allow the fuzzer to find specific types of bugs such as memory bugs. They are particularly useful to fuzzers because they generate crashes from bugs that would normally go unnoticed. For instance, a pointer use-after-free would not normally generate a crash, thus hiding it from a fuzzer. Instead, KASAN (Kernel Address SANitizer) [2], identifies the use-after-free and generates a crash, which the fuzzer reports. Syzbot makes use of many such sanitizers to help it find undefined behavior, uninitialized values, race conditions, deadlocks, and memory leaks.

III. Definitions

At any given time, Syzbot fuzzes a specific Linux commit using a specific commit of Syzkaller. We call the combination of Syzkaller and Linux Syzbot's *fuzzing environment*. This will be key for faithfully recreating Syzbot in our study.

Every bug has a commit in which it was introduced to the kernel – its *Introducing Commit*. Our study focuses on bugs that exist in the upstream (*i.e.,* mainline) Linux repository, so the introducing commit is the one that introduces the bug to upstream. We mark the date of the introducing commit as the *Introducing Date*.

At the Introducing Date, the bug may or may not be findable by the fuzzer. We say a bug is findable if Syzbot is able generate and run a test case that triggers the bug. For bugs that are not yet findable, some change has to take place either in Syzkaller or the Linux kernel in order for them to become findable. This study revolves around identifying the precise code changes or commits that caused bugs to be revealed. We call these commits *Revealing Commits*.

Based on what the revealing commits change, we enumerate 5 reasons a bug could be revealed. These are (1) changes to kernel code, (2) the removal of another kernel bug that was blocking the fuzzer from reaching this bug (*i.e.,* a blocking bug), (3) an addition of or improvement to a kernel sanitizer, (4) a change to how Syzkaller fuzzes, and (5) a change to Syzkaller's syscall description set. We refer to these as *Revealing Factors* as they are the reason a bug has been revealed. The day a bug is revealed marks its *Revealing Date*, after which Syzbot can find it.

Lastly, the *Finding Date* is the date when a bug is first found by Syzbot. On the finding date, Syzbot was fuzzing a particular Linux commit, which we will call the *Finding Commit*. This commit both marks the end of the bug's finding delay and is an anchor where we know the bug should reproduce.

Using these dates, we get two periods of time in a bug's life cycle. We define D_1 as the period of time where the bug is hidden (*Revealing Date - Introducing Date*). In order to accurately evaluate Syzbot, D_1 is truncated by the date Syzbot first began fuzzing. Syzbot could not have fuzzed before it existed, so it does not make sense to evaluate Syzbot on this time. In this sense, the beginning of Syzbot acts as a revealing factor for bugs that pre-date it and are not otherwise hidden. Since we do not know the exact date Syzbot was turned on, we will use the date of the first bug found by it: July 22nd, 2017. D_2 is the delay while the bug is revealed, but the fuzzer has not found it yet (*Finding Date - Revealing Date*). This delay is the actual time it takes for the fuzzer to find the bugs.

IV. Pilot Study and Revealing Factors

We performed a pilot study of 20 bugs in which we manually identified their revealing factors in order to motivate our study. We found that $9/20$ bugs were not findable at their introducing commits and were later revealed due to changes in either Linux or Syzkaller. 3 bugs were revealed by changes in kernel code, 2 bugs were hidden behind other bugs (*i.e.,* blocking bugs), 3 bugs were hidden until a syscall description update, and 1 was found due to a general Syzkaller update. The bugs in our pilot study often have an overall delay ($D_1 + D_2$) of well over 300 days, but the bugs were only revealed for a median of 46 days (D_2). Next, we list each of the possible revealing factors for a bug and provide real-world examples of each.

1) Generic Kernel Commit: There are some cases where the root cause of a bug and the kernel commit that reveals it are not the same. By our definitions: the introducing commit is the commit that introduces the root cause of a bug, while the revealing commit reveals an already present flaw to the fuzzer. This can be seen clearly in the following bug: `WARNING in rtl28xxu_ctrl_msg/usb_submit_urb` [3]. The bug arises when the message pipe directions do not match between host and USB device, which can happen for some zero-length control requests. This behavior had existed since February 3rd, 2015, but was not findable by Syzbot since it would not generate a crash; the control request would simply fail. On May 22nd, 2021, a warning was added to check the pipe direction before handling the request, and thus the bug was revealed. Syzbot found the bug 2 days after the revealing commit was pushed. Added warnings and checks are the easiest reveal to explain, but there are many others, such as an additional function call that completes the code path to the crash site [4]. In general, the kernel can already enter a buggy state, and the revealing kernel commit turns this state into a crash.

2) Blocking Bug: A blocking bug is a bug in the kernel that somehow prevents Syzbot from finding another bug. It is often a crash earlier in the call stack of the hidden bug that triggers most of the time, thus preventing the fuzzer from continuing. A good example is seen in the bug `WARNING in exception_type` [5]. This bug occurs in KVM when the guest's maximum physical address (`MAXPHYADDR`) is set to 1. However, another bug in KVM: `WARNING in x86_emulate_instruction`, is triggered before the bug in `exception_type` and prevents Syzbot from finding it. The blocking bug was patched on May 28th, 2021, and the warning in `exception_type` was found on August 29th, 2021.

3) Sanitizer Commit: A sanitizer commit is a commit that specifically introduces or improves a sanitizer. Since sanitizers are built into the kernel, it is a type of kernel commit. Despite this, sanitizers are developed independently of the kernel and are not intended to be built into consumer releases such as Ubuntu. So, we consider them to be a separate revealing factor from other kernel commits. We note that there were no bugs revealed by sanitizer commits in our pilot study. However, since sanitizers play such a huge role in finding bugs, it stands to reason that their development could reveal bugs. We will explain this revealing factor's rarity in §IX-B2.

4) Syzkaller Commit: Bugs are sometimes revealed by changes to Syzkaller itself. This occurs when Syzbot gains additional hardware or software support, or changes its fuzzing method for a specific module. In the case of `KASAN: slab-out-of-bounds Read in packet_recvmsg` [6], Syzbot was already capable of fuzzing the buggy module (WireGuard), but needed assistance in setting up network devices. Setting up complex scenarios can be difficult for fuzzers as their input is entirely random and may never create something as complex as a valid network. A Syzkaller commit on February 13th, 2020 changed how Syzkaller fuzzed WireGuard by initializing its virtual network with 3 devices. With this support, Syzbot was able to find the bug on March 12th, 2022.

5) Syscall Description Commit: Syscall descriptions are manually written descriptions for the kernel syscalls that Syzkaller can use to generate test cases. Bugs are often hidden behind syscall descriptions that have not yet been implemented into Syzkaller. Consider a memory leak in `kobject_set_name_vargs` [7]. In this case, the bug is not in the `kobject` library, but a misuse of it in the `nilfs` file-system. After deleting a `kobject` using `kobject_del`, the calling function must call `kobject_put` in order to free the object. Not doing so constitutes a memory leak. This bug was introduced on August 8th, 2014, but even once Syzbot began, it still lacked the proper syscall descriptions to fuzz the `nilfs` module for another 3 years. Support for this file-system was added along with support for 19 other file-systems

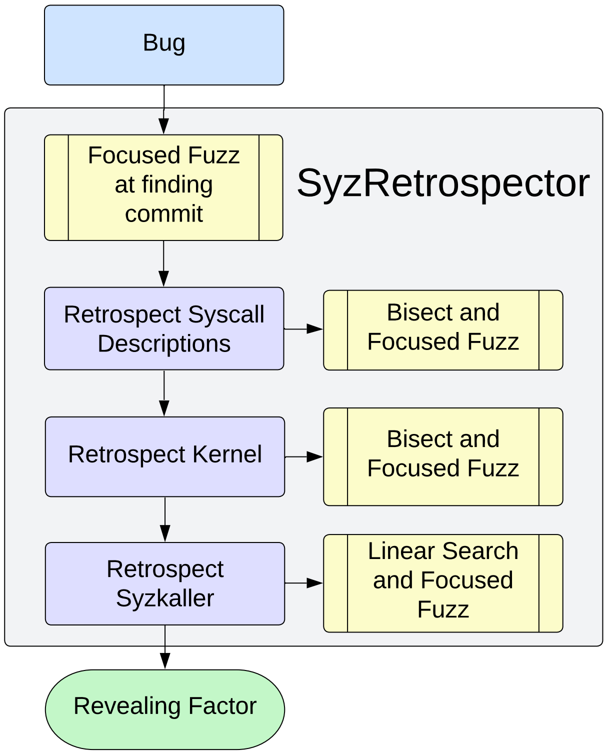


Fig. 2: SyzRetrospector Workflow.

on September 20th, 2020. With the new descriptions built in, Syzbot was able to find the bug 57 days later on November 16th, 2020. We refer to the commit that added these new descriptions to Syzkaller as a *syscall description commit*.

For bugs revealed this way, it is important to note that *hidden* refers to Syzbot's ability to find the bug. Syzbot could not find the bug because it lacked the syscall descriptions to fuzz the vulnerable module. A bug like this may be exploitable before it is revealed, as well as after; a 6 year window in this case.

6) Never Hidden: A bug that could have been found on its introducing date was never hidden from Syzbot. This is the case for the bug `WARNING in futex_requeue` [8], which existed for 13 days until Syzbot found it. Syzbot was capable of finding it the entirety of this time.

V. Overview

Building on our pilot study, we construct SyzRetrospector, an analysis tool capable of going back in time to faithfully recreate Syzbot's fuzzing environment at any point in time. In this section, we provide a high-level overview of SyzRetrospector's workflow (Fig. 2) and how it goes back in time to identify a single revealing commit.

Given any bug, SyzRetrospector starts by fuzzing at that bug's finding commit, where we know the bug exists. Fuzzing here tells SyzRetrospector whether finding the bug is feasible, and how long the bug takes to reproduce. A feasible bug is one that can be reproduced by SyzRetrospector in a reasonable amount of time. If a bug is feasible, then its findability is determined by whether it reproduces in a particular fuzzing session. §IX-A demonstrates that SyzRetrospector uses this implication to reach the correct revealing factor. Bugs that are not feasible, often hard-to-find races, are skipped. We will justify this decision in §XI.

In order to determine whether a bug is findable in a commit, SyzRetrospector faithfully recreates Syzbot's fuzzing environment on the day of that commit and then fuzzes for the bug. Syzbot's fuzzing environment is comprised of *a kernel commit, a Syzkaller commit, and a syscall description commit*. Each of our 5 revealing factors is the result of one of these types of commits, so SyzRetrospector will analyze each of these commit types in turn. Whenever SyzRetrospector chooses a commit to test, it completes the fuzzing environment with commits of the other types from the same date as the chosen commit. So, if SyzRetrospector is testing a syscall description commit, it chooses Syzkaller and Linux commits from the same day to complete the environment. This means that as it switches from one commit type to the next, the shrinking date range of where the revealing factor could be carries over.

SyzRetrospector first gathers and retrospects all of the relevant syscall description commits. It does this by parsing the bug's reproducer and Syzkaller's git to see which commits changed or added the syscall descriptions in it. SyzRetrospector then uses binary search to find the earliest syscall description commit that is able to reproduce the bug, and confirms or rules out whether this commit is the revealing factor by rolling back the description set, but keeping the same Syzkaller and Linux commits. This process is depicted in Fig. 3. If the bug is found after the description commit, but not before, the description commit must be the revealing factor. This provides a simple algorithm to isolate and then confirm whether a syscall description commit revealed the bug in question. Otherwise, the revealing factor is either a Linux or Syzkaller commit between the dates of this description commit and the one before it.

Next, SyzRetrospector searches the kernel commits for a revealing factor. Again, it uses binary search to narrow down the commits. And once again, if the bug is found after a kernel commit, but not before, then that commit is the revealing factor. Otherwise, the revealing commit must be a Syzkaller commit. Since kernel commits are pushed upstream daily, the date range has shrunk to a single day.

Lastly, there are relatively few Syzkaller commits each day, so SyzRetrospector searches them linearly going back in time. Once the bug reproduces after, but not before a commit, that commit must be the revealing factor.

VI. Design

In this section, we will explain in more detail the workflow laid out in §V, including how we focus Syzkaller to a specific set of syscalls and how it arrives at a correct result.

A. Retrospection Preparation

Before retrospection begins, SyzRetrospector gathers everything it needs related to the bug from Syzbot, including the bug's reproducer, its finding and introducing commits, and the exact kernel configuration used by Syzbot. It also makes note of any bugs which share the same patch as these are duplicates, or different manifestations of the same bug. They are treated as the same bug during retrospection.

Maximum Fuzzing Time: As shown in Fig. 2, SyzRetrospector begins by attempting to trigger the bug at the finding commit. Here, SyzRetrospector fuzzes 3 times for 30 minutes in order to understand how long it takes for the bug to reproduce. It then recalculates the maximum fuzzing time by using the mean + standard deviation of the 3 trials, which can end up being greater than 30 minutes. For most bugs, the new time is 10 minutes (a minimum set by us), and tops out at around 33 minutes. SyzRetrospector uses this time going forward except when the bug is harder to reproduce than it originally thought. If any bug takes 80% of its maximum time to reproduce, SyzRetrospector resets the maximum time to 30 minutes and fuzzes 5 times rather than 3. Calculating the maximum time further decreases SyzRetrospector's overall run time by letting it better decide when a bug is not findable. From our experience, the time needed to find a bug does not change as we move back in time. So, the maximum fuzzing time is not recalculated past the first fuzzing session.

B. Building the Fuzzing Environment

SyzRetrospector begins a fuzzing session by cutting down the number of syscall descriptions to only those needed to find the bug. If Syzkaller ran without any form of guidance, it would simply try to maximize coverage and may never find the bug in question. At the same time, we do not want to change how Syzkaller fuzzes as that could invalidate our results. Instead, we restrict what areas of code Syzkaller can fuzz by focusing the syscall descriptions built into it. We call this contribution *focused fuzzing*, and it is one of the core reasons SyzRetrospector is able to scale to a large number of bugs. Our focused fuzzing scheme greatly reduces the time taken to find a single bug from many days to only a few hours. To enable focused fuzzing, SyzRetrospector parses the large, unfocused description set that Syzkaller pulls from randomly, and narrows it down to only the descriptions and dependencies required to reproduce the bug. After this, Syzkaller has a fully functional subset of the original descriptions, usually around 20 syscall descriptions. Importantly, none of the descriptions or underlying structures are changed. We have only limited the ones Syzkaller is allowed to use.

We further decrease the time to reproduce the bug by inserting the reproducers into the corpus as seeds. This creates the beneficial scenario where Syzkaller will only fuzz a small window of the kernel, and has just generated the test cases to exercise the buggy code. By encouraging these test cases, Syzkaller finds most bugs in 2 minutes. Importantly, we leave Syzkaller's mutation and scheduling algorithms unchanged. Since mutation is left untouched, Syzkaller is still capable of finding new paths to the same bug if the reproducer fails to trigger it. We reason that using the reproducers does not interfere with SyzRetrospector's faithful recreation of Syzbot's

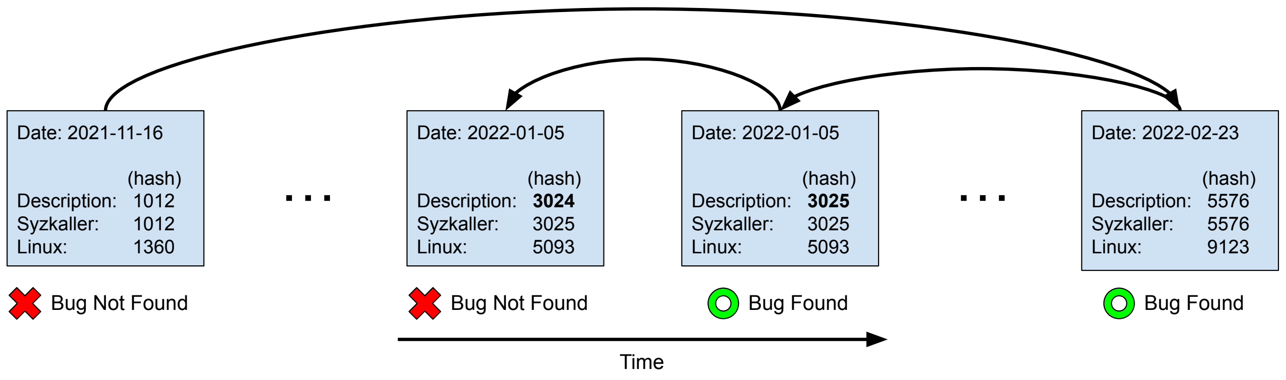


Fig. 3: SyzRetrospector determines which commit revealed the bug in question as the bug reproduces after, but not before, the revealing commit with hash 3025. For ease of reading, the example hashes increase monotonically.

fuzzing environment since it does not change how the fuzzer operates.

Next, the kernel is built. SyzRetrospector uses the exact configuration used by Syzbot when the bug was found. Then, SyzRetrospector chooses a suitable compiler based on the date of the kernel commit from set of compilers that Syzbot has used. This ensures the kernel builds without error.

C. Result Collection

Once retrospection is completed, SyzRetrospector outputs a final report with a log of all the fuzzing sessions, the exact revealing commit, and important dates in the bug's lifetime. SyzRetrospector differentiates between syscall description and Syzkaller commits, but a kernel commit could represent a sanitizer commit, a blocking bug, a generic kernel commit, or the introducing commit. SyzRetrospector identifies bugs that are never hidden (*i.e.,* are findable at their introducing commit), and lists potential blocking bugs that appear many times throughout retrospection, especially when the original bug is not found. The duty falls to the researcher to read the log and determine if a listed bug is a blocking bug or not. Similarly, sanitizer commits can be recognized by their commit name. If all other options are ruled out, then a generic kernel commit is what revealed the bug.

VII. Implementation

We faced and solved many implementation challenges to successfully realize SyzRetrospector, which is comprised of 7,800 lines of C++ code and BASH script. Many of these challenges stem from looking back in time to fuzz many commits, and matching the scale of Syzbot.

A. Duplicate Bugs

Duplicate bugs can be difficult to identify automatically, even though SyzRetrospector has some simple deduplication built into it. When gathering a bug from Syzbot, it will check for any other bugs with the same patch as they share the same root cause, and are thus the same bug, a strategy used by a previous study [9]. However, some duplicates are not found by Syzbot. During analysis, if SyzRetrospector finds bugs that appear to be duplicates, we manually study them and mark them as such. We use heuristics such as similar stack trace, same crashing function, same crash type, same sanitizer, etc. to consistently mark duplicate bugs.

B. Syscall Description Parsing

In order to achieve focused fuzzing, SyzRetrospector needs to be able to parse the syscall descriptions of any Syzkaller commit. While more recent versions of Syzkaller allow a user to enable only a subset of syscall descriptions, older versions do not have this feature. SyzRetrospector must carve out a new description set for each bug, and for each version of the description set.

Syzlang, the custom language of the syscall descriptions, is relatively straightforward to parse thanks to the rigorous and consistent writing of the Syzkaller team. Each syscall depends on a number of structures, resources, and other dependencies that describe how data should be passed to the syscall. Most of these are easy enough to find and add to the new description, keeping in mind that those items may depend on yet more items. Resources, however, pose a greater challenge.

Each resource must be produced, or provided as output from a syscall in the description, and consumed, or used as an input by another syscall, otherwise Syzkaller will not build. The nuance arises because resources are not always easily identified as input or output. A pointer in a syscall may be labeled as `in` or `out` or `inout` (both in and out) [10]. Items pointed to by that pointer will be used as input, output, or both, which must be accounted for. In Fig. 4 for example, `parent` is passed to `consumer` as an `inout` pointer, `parent` houses the structure `child` as an `in` (input) structure, and finally `child` has a member `my_resource`. So `consumer` uses `my_resource` as an input.

Following the rules above, SyzRetrospector begins by parsing all of the syscall descriptions from Syzkaller and classifying each object. It then consults the reproducer to determine which syscall descriptions are required and adds those to a new description. SyzRetrospector then adds the dependencies of the objects already in the new set. If a resource is added, it adds up to two new syscalls: one that produces it, and one that consumes it. While choosing these syscalls, SyzRetrospector tries to select those that do not depend on other resources to reduce the total number of syscalls in the end.

Thankfully, the language of Syzkaller's syscall descriptions, Syzlang, largely remains unchanged through the past 7 years, and most changes are simply additions of features. The only functional change we have had to account for is the meaning of `inout` as an object attribute. This keyword currently means that children objects, specifically resources, will be labeled as either `in` or `out` as needed. This nuance allows the developers to have members of the same structure function as either producers or consumers in a single syscall. However, in older versions of Syzkaller, `inout` meant that every child object should be treated as both in and out. In this case, the description parser needs to gather all children objects, regardless of attribute.

C. Patches

SyzRetrospector goes back in time to look at Syzkaller and Linux commits alike. However, large swathes of commits in both Linux and Syzkaller either do not build or do not boot. In order to obtain accurate retrospection, we have written a series of patches for both the kernel and Syzkaller. Through manual effort, we have denoted exact commit ranges to which each patch needs to be applied.

We took care to patch the kernel as little as possible, but some bugs required modifying the kernel. In one example, the kernel built itself with the incorrect page size, and failed boot. Our patch is based off of the one used in mainline Linux and simply forces a 2 MB size page [11]. SyzRetrospector has 7 patches that it checks before building a kernel commit.

We also develop and maintain 9 patches for Syzkaller that either build Syzkaller correctly, or ensure that Linux boots properly. This usually meant modifying boot parameters or changing the makefile. In total, there are 16 patches SyzRetrospector considers when building the fuzzing environment.

While we tried to keep these patches small and non-invasive, they often dealt with core components such as memory. Indeed, the very nature of applying patches means that we are removing bugs from the kernel. We reason that if left unpatched, thousands of commits would be unusable. We also did not observe any of these patches introducing or removing bugs during testing.

D. Unstable Commits

The patches in the previous section are a key part of allowing SyzRetrospector to function over wide ranges of Linux and Syzkaller commits, but it is infeasible to patch every boot error or incompatibility. Instead, we implement SyzRetrospector with the ability to work around errors.

SyzRetrospector is able to detect issues in two ways: it can detect boot failures by reading the Syzkaller logs, and it can detect incompatibility crashes by parsing the crashes. If SyzRetrospector identifies an issue, it marks the fuzzing environment as unstable. SyzRetrospector continues retrospection, but attempts to work around the unstable commits by fuzzing nearby commits. In the case that the revealing factor is inside a range of unstable commits, SyzRetrospector reports this in the final report. In most cases, SyzRetrospector successfully works around the unstable commits and identifies the revealing factor outside of them. We observe that without this feature, SyzRetrospector would incorrectly assume that a bug was hidden, negatively impacting our results.

```
resource my_resource[intptr]

producer(num int32) my_resource
consumer(p ptr[inout, parent])

parent {
  child  child  (in)
}

child {
  rec    my_resource
}
```

Fig. 4: An example syscall description

E. Compiler Selection

In addition to choosing Linux commits to fuzz, SyzRetrospector multiplexes 6 versions of the C compiler GCC. Using the correct compiler version is crucial to building older kernel versions. In some cases, compiler warnings in older GCC versions are treated as errors in newer versions. Older kernel versions may not comply with the rules that were made after them, so we need to use older GCC versions to facilitate compilation. The GCC version used by Syzbot also naturally changed over time. We have looked back on the history of GCC compilers used by Syzbot and compiled a collection of them to be multiplexed by SyzRetrospector. We choose the compiler based on the date of the kernel commit we want to build, and compare that to the compiler Syzbot used during the same time frame.

F. Architecture

In addition to the x86_64 architecture, Syzbot also fuzzes i386, using cross-compiled test cases. In these cases, kernel compilation does not change, but parts of Syzkaller must be cross-compiled. SyzRetrospector uses syz-env [12], a docker container provided with Syzkaller, in order to handle cross-compilation. We note Syzbot has a spread of 6 managers that fuzz RISC-V and ARM versions. However, these managers only fuzz auxiliary repositories, so their crash instances are left out of our study. If their bugs are found in upstream, the upstream crash is used for retrospection.

VIII. Data Set

Each of the 695 bugs in our data set meets strict requirements to ensure valid results. We gathered the most recent batch of bugs on May 17th, 2024, so recently fixed bugs are not included. First, every bug needs a patch that contains a `Fixes` tag. This tag is added by the kernel developer to denote a bug's introducing commit. Since we have found erroneous cases where a bug was found before its introducing commit or found after it was marked as fixed, we also perform a sanity check that all of the dates for a bug are in order. The bug also needs at least one reproducer to be used throughout retrospection.

We found that auxiliary repositories such as `linux-next` often undergo major changes, such as deleting branches, leaving many commits dangling. These commits are often the introducing and/or finding commits that SyzRetrospector needs to study a bug. Due to the transient nature of these repositories, we only study bugs that appear upstream.

We gathered a total of $1{,}407$ bugs for SyzRetrospector to study. However, many of these bugs failed to reproduce even with their reproducer, and others encountered build errors that remain unsolved. As we justify in §XI, these bugs were left out of the study.

IX. Findings

Here we present our findings after performing a large scale retrospection of 695 bugs found by Syzbot. This process took over 4.5 months and over $12{,}400$ CPU hours.

A. Correctness Study

Due to the non-deterministic nature of both fuzzing and some of the bugs in our data set, the bugs may not always reproduce as we expect them to. This results in false negatives for individual fuzzing sessions, and possibly incorrect results overall. However, manually verifying each of the 695 bugs in our data set is infeasibly labor-intensive. So, we *randomly* sampled 50 bugs from our 695 bug data set, and check this subset's correctness. In this case study, we carefully retrospected the revealing factors and dates given by SyzRetrospector in order to check their correctness. In addition, we manually looked for other possible revealing factors. Based on the results of this study, we can infer the accuracy of SyzRetrospector for all bugs in our data set.

Out of the 50 bugs studied, 45 of them had correctly identified revealing factors. So, we extrapolate that SyzRetrospector reaches the correct result 90% of the time. The bugs that did fail were hard-to-find bugs that managed to reproduce at their finding commits, and then resulted in false negatives during retrospection. On average, the incorrect results were off by 19 days. Compared to the relatively large D_1 and D_2, we argue that the error is small enough that SyzRetrospector's overall results are reliable.

The following findings are derived from the entire data set of 695 bugs, which we take to be accurate based on our manual verification of the 50, randomly sampled bugs.

B. Revealing Factors

Fig. 5 shows the revealing factors and the percent of bugs they are responsible for revealing. We see 68% of bugs were hidden for some portion of their lives, the most prominent categories being generic kernel commits, syscall description commits, and blocking bugs. Each of these revealing factors requires unique effort to induce the reveal.

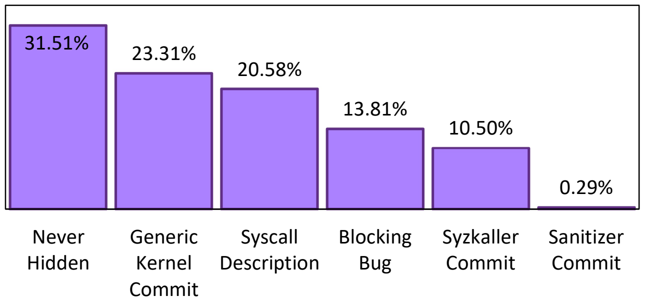


Fig. 5: Percent of bugs revealed by each revealing factor.

1) Kernel-based Reveals: Among the 5 reveal groups, generic kernel commits and blocking bugs are the hardest to intentionally cause. For instance, any of the open bugs in Linux could be a blocking bug. A developer has no way of knowing which of the open bugs are blocking bugs before they are patched - a circular dependency. The only way to improve D_1 for these bugs is to speed up the patching process, which is challenging due to the large number of players involved.

Generic kernel reveals are difficult to predict since it is unknown what code change will reveal a bug. Consider a slab-out-of-bounds bug in the Thrustmaster joystick driver [13]. The driver holds a pointer to a USB endpoint that has already been freed, but the driver fails to check the endpoint before attempting to use it. The bug was hidden for a long time since even though the pointer enters a buggy state, it usually does not point out of bounds. We now shift to the revealing factor: a commit in the B-Tree Filesystem (BTRFS) module [14]. Periodically, BTRFS compresses inodes to save memory, but prior to this commit, did not compress single-page inodes. The Thrustmaster pointer happens to point into one of these single-page inodes. Once the inode is compressed, the Thrustmaster pointer, which uses fixed offsets, points outside the valid memory range, and can trigger KASAN. Many generic kernel commit reveals are similar in that it is hard to predict what change will reveal a bug. However, we will provide suggestions on how to improve the chances of inducing generic kernel commit reveals in our takeaways.

Finding 1. We find that 37% of bugs are hidden by generic kernel commits (23.31%) and blocking bugs (13.81%), and that it is hard to improve the D_1 of these bugs.

Sanitizers are an interesting revealing factor since it appears they do not reveal many bugs. Despite 42% of bugs in our data being found with the assistance of sanitizers, only 2 were actually revealed by commits to sanitizers. A deeper look into the sanitizers' development reveals what is actually going on. Take KASAN for example, one of the first sanitizers in Linux. It has only seen around 500 commits since it was introduced in 2015. Furthermore, most of the commits since 2019 are minor fixes and selftests. It makes sense that none of these commits would reveal many, if any, bugs.

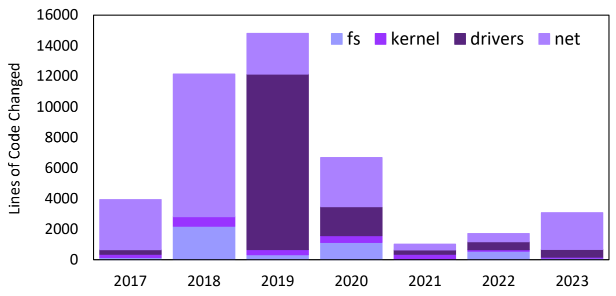


Fig. 6: Lines of code changed in syscall descriptions each year.

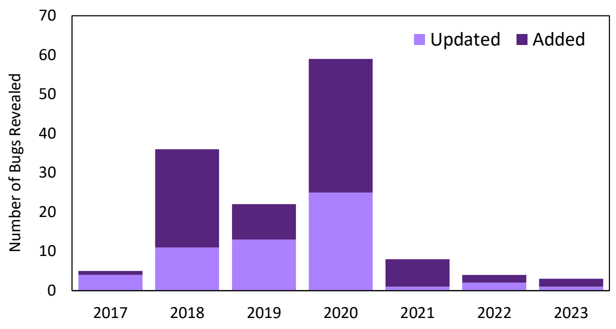


Fig. 7: Syscall description reveals by year based on whether an added syscall or an updated syscall caused the reveal.

KMSAN, however, is unique in that it was introduced to Linux after Syzbot had already matured. To show the impact of adding new sanitizers to the kernel, we manually studied 8 bugs found by KMSAN around the time it was introduced. Until recently, Syzbot only used KMSAN in an auxiliary repository, so these 8 bugs are not included in the retrospection data set. We traced $4/8$ of the bugs back to the commit that added KMSAN. The remaining 4 bugs were revealed some time after KMSAN was added. So, adding a new sanitizer reveals an entirely new class of bugs to the fuzzer, but the little development that occurs after does not reveal many bugs.

Finding 2. We find that new sanitizers bring a jump in fuzzing capability, revealing several bugs. However, the little development after this does not reveal many bugs.

2) Syzkaller-based Reveals: Contrary to the revealing factors above, syscall description commits and Syzkaller commits reveal bugs more predictably. In order to reveal a bug that is hidden behind some syscall, that syscall must be added to the description set. It follows that fewer lines of code changed would result in a lack of bugs being revealed. We observe this trend over the past 3 years. As syscall description development has slowed since 2021 (Fig. 6), reveals attributed to those years have also slowed (Fig. 7). As a result, there likely exists a large group of bugs that Syzkaller is unable to find, hidden behind syscall description development.

This is the primary challenge behind improving Syzkaller: it must remain consistently up to date with kernel development to achieve its best performance. Not only are more syscalls being added to Linux via ioctl commands, but old syscall descriptions become out-of-date. We looked into the syscall description reveals identified by SyzRetrospector and found that 42% of the reveals were a result of updating old syscalls, not adding new ones. We note that this ratio stays fairly consistent over time, and predict that it is unlikely to change anytime soon.

Finding 3. We find the greatest challenge of improving Syzkaller is developing its description set, and that maintaining old syscalls is just as important as adding new ones.

Despite this challenge, there is a silver lining. The number of bugs reported by Syzbot has not dropped in recent years [15]. In fact, despite fewer reveals coming from Syzkaller development, we see a higher ratio of bugs that were never hidden. This ratio has been slowly climbing from 28% of bugs found in 2020, to 53% of bugs found in 2023. This shows that all of the effort put into Syzkaller years ago continues to pay off.

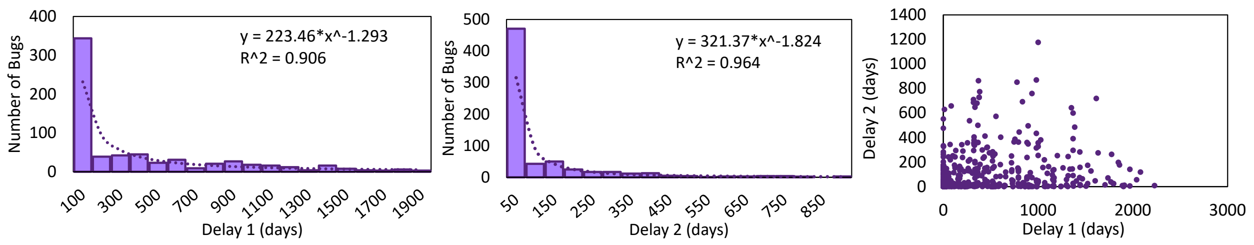
Finding 4. We find that Syzbot continues to find bugs based on past improvements to its description set.

C. Bug Finding Delays

Figs. 8a and 8b show histograms of D_1 and D_2, where they have been fit to a power law distribution with R^2 values of 0.906 and 0.964 respectively. Following the power law means many bugs will have shorter delays, but there is a long tail of less common bugs with very large delays. For instance, D_1 has a 25th percentile of 0 days, yet a median of 100 days, and a 75th percentile of 531 days. It is this long tail that we are concerned with. In this section, we will disprove any correlation between D_1 and D_2, and then take a deeper dive into the trends of each delay.

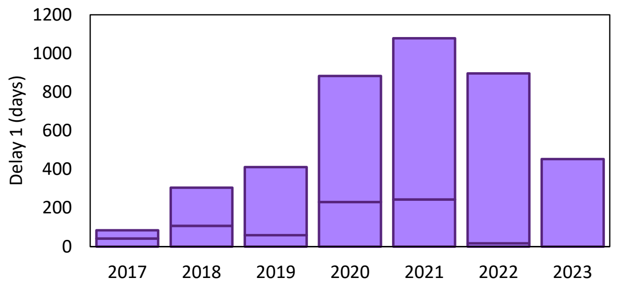
We investigate whether D_1 and D_2 are independent. To do so, we present Fig. 8c, a scatter plot of D_1 and D_2, and observe that there is no obvious correlation. We also perform regression analysis and find that no regression line, linear or curved, has an R^2 value greater than 0.136. For any D_1 we cannot predict the value of D_2. This is strong evidence that D_1 and D_2 are independent. This has an important implication: it shows that separate efforts are needed to reduce each delay and provides a bound on how much each solution can reduce the delay.

Finding 5. We find strong evidence that D_1 and D_2 are independent components of the overall delay. This implies that separate efforts are needed to reduce D_1 and D_2.

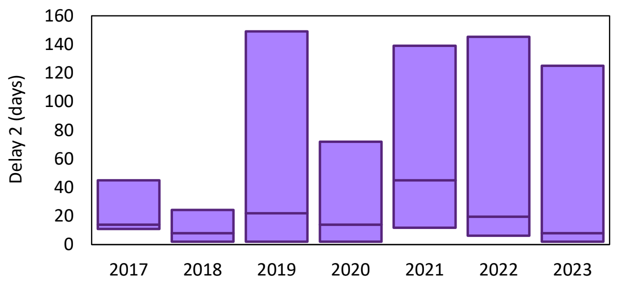


(a) The distribution of D_1. (b) The distribution of D_2. (c) A scatter plot of D_1 and D_2.

Fig. 8: Plots of D_1 and D_2.



(a) Box plot of D_1 by year.



(b) Box plot of D_2 by year.

Fig. 9: Box plots showing quartiles of D_1 and D_2 by year. The minimums and maximums are omitted for readability.

1) Delay 1: We first consider D_1 as shown in Fig. 9a. The increase in D_1 is expected as Syzbot begins fuzzing, noting that D_1 is measured from the time Syzbot began fuzzing for bugs older than it. A general increase in D_1 means that more bugs are being found that were once hidden, and that those bugs are often older than the fuzzer. We know that when Syzbot first began in 2017, Linux was already much older than it and had many bugs. So, Syzbot's increasing D_1 shows that it is improving.

The upward trend of D_1 comes to an abrupt halt in 2022 when it takes a nose-dive. We observe that a decrease in D_1 means the fuzzer is finding fewer previously hidden bugs, and that bugs are being revealed much earlier in their lifespans. Indeed, Table I shows that bugs found in 2022 also largely originate from 2022. In fact, the median D_1 for 2022 is only 17 days. This is in stark contrast to the previous years, which have a more even spread between bugs whose introducing commits lie in the same year and those that are much older. The same is true to 2023. So, Syzbot is finding fewer old bugs in recent years, and the bugs it does find are not hidden as long.

TABLE I: Number of bugs found in each year (columns) by the year they were introduced (rows). We note that 2005 is a hard barrier as this is the year Linux began using git.

Year	2017	2018	2019	2020	2021	2022	2023
2005	0	0	9	9	3	2	0
2006	0	11	1	2	0	1	0
2007	0	2	2	2	0	0	0
2008	0	1	1	4	3	0	0
2009	0	0	3	1	4	2	0
2010	0	4	5	4	2	2	2
2011	0	0	2	3	2	2	1
2012	0	4	3	2	5	2	1
2013	0	5	6	4	2	2	0
2014	0	3	2	7	2	0	1
2015	4	4	0	1	1	2	1
2016	3	12	5	9	2	4	2
2017	9	8	11	20	5	2	1
2018		63	19	17	3	1	1
2019			66	32	6	3	0
2020				68	20	5	1
2021					32	11	3
2022						45	5
2023							32

From this, we reason that Syzbot is reaching an ideal state of fuzzing. We define an *ideal state of fuzzing* as one where all bugs are either revealed shortly after they are introduced or are never hidden at all. In such a state, the fuzzer is able to search for all bugs uninhibited by revaling factors which would otherwise hide some bugs. However, based on our own observations, we believe this ideal state is merely a local ideal, with more improvement being possible. We observe that Syzbot has not yet achieved 100% coverage of the Linux kernel, and that syscall description development has slowed from its peak in 2020 (Fig. 6). These clues point to the conclusion that there are more hidden bugs lying in wait in the kernel. In this sense, Syzbot has caught up on finding many of the currently revealed bugs and is ready to have its capabilities expanded and improved again. This behavior is important as it demonstrates one thing: *Syzbot is capable of converging to an ideal state of fuzzing with the Linux kernel.*

Finding 6. We find that based on the decreasing D_1, Syzbot is approaching a local ideal state of fuzzing, but that more improvement is still possible.

2) Delay 2: D_2 is the delay for Syzbot to find a bug while the bug is revealed. We reason that D_2 is largely related to the compute power given to Syzbot. If a fuzzer is given more

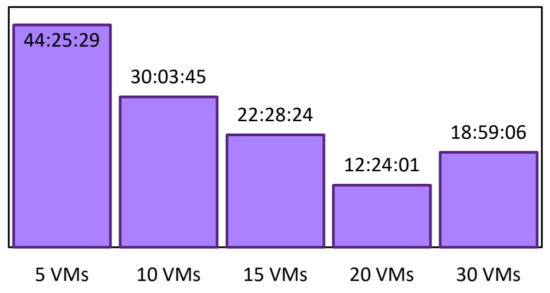


Fig. 10: 75th percentile of D_2 with varying numbers of VMs.

resources (*i.e.,* compute power), it will fuzz the target with greater throughput, which in turn will find bugs faster. In a sense, this defines the rate at which Syzbot can find bugs. To demonstrate this, we performed an experiment in which we varied the amount of compute power given to Syzkaller.

Our study consisted of a single Syzkaller manager using a varying number of VMs, each with 2 CPU cores. In this setup, more VMs means more compute power. We ran Syzkaller 3 times for each number of VMs, each time with an identical starting corpus. The D_2 shown in Fig. 10 is the 75th percentile D_2 of bugs found in all runs - 39 bugs total. The Figure shows a clear and significant decrease in D_2 up until 30 VMs. Thus, D_2 can be roughly thought of as Syzbot's fuzzing power, or how fast it can find bugs in a target.

Concerning the increase in delay for 30 VMs, we believe it is caused by a bottleneck in the host machine. With so many VMs running on a single machine, it is likely that the host's KVM became bogged down and slowed the execution of test cases. Syzbot, which runs VMs on separate machines, may not have this same bottleneck.

Finding 7. We find that D_2 is largely related to the compute power given to Syzbot.

D. Delays by Location

Here we look at how bug finding delays vary for different locations in the kernel. We use the root directories of Linux (*i.e.,* drivers, net, kernel, *etc.*) to describe the possible locations where a bug could exist, focusing on the directories for which we have more than 50 bugs in our data set. Those directories are drivers, the file-system directory (fs), kernel, and network (net).

Consider drivers and net as shown in Fig. 11. These two directories alone account for over 68% of the compiled kernel, and the median D_1 of each is 275 and 208 days respectively. Fuzzer development for these areas struggles greatly due to the niche requirements of drivers and net. Drivers, for example, often require emulated or actual hardware in addition to unique syscalls for each driver. Net sees a similar issue where the fuzzer often requires assistance in setting up networks. These additional requirements mean it will take longer for Syzkaller to improve enough to reveal bugs. Compare this to fs and kernel, which only combine for 21% of the compiled kernel

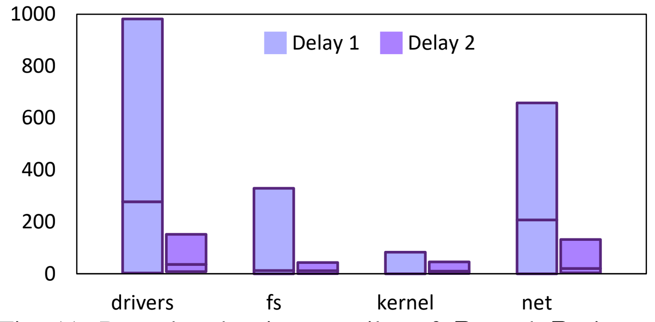


Fig. 11: Box plot showing quartiles of D_1 and D_2 in root directories of Linux.

and have a median D_1 of less than 15 days. From this we see that the size of a location is positively correlated with the time it takes to reveal bugs in that location.

On top of this, description set development for these complex locations does not always yield a proportional amount of payoff. Recall Figs. 6 and 7, which show syscall description development and syscall description reveals over time. In 2018, there was a huge effort to improve drivers related descriptions, yet this effort revealed relatively few bugs. Again in 2023, there was an uptick in development to the BPF related descriptions, though we have yet to observe the payoff for this. This serves to highlight the importance of research into tools which aim to automatically generate syscall descriptions for drivers [16]–[20]. We believe that this area of research is promising and will help fuzzer developers keep up with driver development.

Finding 8. We find that the D_1 of a location is determined by both the size and unique challenges of that location.

Fig. 11 also reveals an interesting finding about D_2. We see that drivers, FS, and net all share a similar median D_2 (about 15 days), meaning Syzkaller does not struggle to find bugs in one location or another. But we also see that the kernel directory has roughly half the D_2 of other directories at only 8 days. Kernel is a small directory and most test cases go through it during their execution, so it has a higher "coverage frequency" (*i.e.,* how often code is fuzzed). This correlation shows that coverage frequency plays a role in determining D_2.

Finding 9. We find that the fuzzer's compute power is not distributed equally between different locations in the kernel. Some locations see a higher "coverage frequency" and hence a lower D_2.

X. Takeaways

Based on our experimental results and understanding of continuous fuzzing, we provide some suggestions to improve Syzbot's bug finding delay.

1) Improving D_1: In order to improve the D_1 of generic kernel commit reveals, we look to the core reason why these bugs are hidden. There exists a buggy state in the kernel that does not trigger a crash, thus the fuzzer cannot find it. So, we propose using warning and bug assertions that explicitly turn these buggy states into crashes. One could imagine a tool which uses static analysis to insert checks in high-risk areas. Such a tool could be opportunistic and would likely require less development effort than a full-blown sanitizer. Recall a bug from §IV-1 which was revealed and found due to an added warning. Since the buggy state is a pipe direction mistake that quietly fails, it is hard to imagine the bug would ever be revealed except through a warning or bug assertion. We believe there are many more bugs like this that could be revealed through similar means. In addition, bugs found this way may be easier to patch as the exact buggy state is already known. We believe this line of work is promising for reducing the D_1 of otherwise hidden bugs.

Takeaway 1. We suggest the use of automation to insert more warning and bug assertions and reveal more bugs.

In order to improve Syzkaller, members of the community must identify areas that need work and use their domain knowledge to create syscall descriptions. Here, we look for a method to better focus developers' time. We surveyed 30 recent CVEs and found that 13 were hidden from Syzbot. Consider CVE-2022-29156, a double free in the Infiniband RDMA Transport Server (RTRS) module. The correct syscall, `write$RDMA_USER_CM_CMD_CONNECT`, is in Syzkaller's description set, but it does not have the proper flag, `RTRS_MSG_CON_REQ`, to reach the RTRS module. In our experience, it is a common trend that Syzbot cannot find CVEs due to an incomplete syscall description set. We suggest taking advantage of this trend. CVEs point to areas in the kernel where vulnerabilities exist, and where Syzkaller needs more development. This is an easy way to identify incomplete descriptions that are key to fuzzing vulnerable areas.

Takeaway 2. We suggest using CVEs that are not found by Syzbot as an external trigger to develop syscall descriptions for vulnerable code.

2) Improving D_2: The goal of Syzbot is to identify, track, and assist in removing bugs from Linux. However, keeping up with kernel development is no small feat. Linux makes major releases such as 5.14 or 5.15 every 63 or 70 days – its release cycle. We studied 90 recent bugs and found that 76% of bugs are patched within 70 days of being found – their time-to-fix. To set an attainable goal, we focus only on D_2 as it is independent of D_1, and on shortening its long tail. Even with just D_2 and time-to-fix, bugs could exist in more than 3 releases before they are patched. We also note that due to the time-to-fix being as long as a release cycle, it is unreasonable to set a goal of removing all bugs from all releases. Instead, we set a goal of lowering the 75th percentile of D_2 to less than 60 days, ensuring most bugs exist in only 2 releases.

To reach this goal, we suggest adding more VMs to each manager under Syzbot. To find out how many VMs should be added, we look back to our study in § IX-C2. We note that each Syzbot manager uses 10 VMs. Based on Fig. 10, we find that doubling the compute power of Syzbot has the potential to halve its D_2. We estimate that doubling the number of VMs in each manager would be able to cut even the 80th percentile down to just 60 days from 119 days.

Takeaway 3. To best fit the release cycle of Linux, we set a goal of $D_2 \leq 60$ days, and estimate that doubling the VMs used by Syzbot will more than meet this goal.

XI. Threats to Validity

1) Bias 1: Infeasible Bugs: Despite our efforts, certain bugs cannot be completely retrospected and must be left out of our data set. The majority of these bugs are ones that fail to reproduce in a timely manner. Bugs that take longer than 30 minutes to reproduce could take several days to a week to complete retrospection, making a large scale study infeasible. We observe that these bugs are largely races or other non-deterministic bugs with tight windows such that it is near impossible for Syzkaller to find them quickly and consistently. Such bugs are not feasible to find, and thus cannot continue retrospection. It follows that hard-to-find bugs are left out of this study. However, we can generally reason that hard-to-find bugs have a longer D_2 in Syzbot. As we demonstrated in our findings (§IX-C2), D_1 and D_2 are independent of each other, meaning only D_2 is underestimated. This bias may affect our results presenting D_2, such that the true aggregate D_2 is higher than reported. For the subset of bugs that we did retrospect successfully, D_2 is correct.

Other bugs suffered from build errors in either Syzkaller or Linux that remain unpatched. However, we believe that the occurrences of such errors are distributed randomly enough to not introduce a bias in our results.

2) Bias 2: Crash Instance Choice: Since we only consider crash instances in upstream, the finding date used by SyzRetrospector may be inaccurate when a bug is found in another repository before it is found in mainline. However, we analyzed this difference and found that well over half of the bugs are found on the same day, and over 84% are found within 10 days. Compared to the relatively long lifespans of bugs, this difference is small. This may affect our results on D_2 for specific bugs, which may vary ± 10 days.

XII. Discussion

Our study carried out by SyzRetrospector focuses on the fuzzer-target pair, Syzbot and Linux. While SyzRetrospector is specifically designed to identify D_1 and D_2 for this pair, and thus is not directly usable with other fuzzers or targets, our key metrics and methodology are transferrable. Our two delays arise from the idea that bugs are not always findable by fuzzers due to factors in the fuzzer's algorithm, its interface, or the target code. For example, OSS-Fuzz [21] is a framework provided by google that allows developers to submit their projects as well as fuzzers to perform fuzz testing over time. Our study method could apply to individual fuzzer-target pairs to identify D_1 and D_2, assuming both the target and fuzzer are updated over time. In our example, OSS-Fuzz targets user-space programs, so there is no syscall interface, and input can vary widely between projects. This input space, the interface with which the fuzzer interacts with the target, can possibly hide bugs from the fuzzer if it is not complete. This is similar to, but not an exact match to our revealing factor relating to the syscall description set. So, while the exact revealing factors would likely differ, D_1 and D_2 are generalizable to other fuzzer-target pairs in continuous settings. We chose to focus on Syzbot and Linux since both projects receive daily or almost daily commits, and the bugs found are well documented.

XIII. Related Work

Despite the prominence of continuous fuzzers in project development, we have seen few empirical studies of their performance. The study from Rouhonen *et al.* [22] analyzes bugs found by Syzbot to understand their time-to-fix with regards to operating system and bug type. Ding and Goues [23] perform a more in-depth study of OSS-Fuzz [21]. The authors explore the idea of "fuzz blocker", similar to our idea of blocking bugs (§IV-2). They also analyze the flakiness of bugs, whether the bug is patched, and whether it has a CVE. Both of these studies rely on statistics and static analysis. By contrast, our work with SyzRetrospector is the first large-scale, dynamic analysis of bug lifetimes with regard to Syzbot, or any continuous fuzzer. We not only demonstrate that certain metrics like bug finding delay are inaccurate, but develop our own metrics to clearly analyze Syzbot's performance over 7 years.

Mu *et al.* [9] provide a comprehensive study of duplicate Linux kernel bugs found by Syzbot. In their paper, they organize Linux experts to identify duplicate bug reports through great manual effort. Then, building on the reasons for duplication, they prototype tools capable of testing whether two bug reports are duplicates of each other. The paper finds that 47% of the studied bug reports are duplicated with at least one other report. SyzRetrospector uses similar ideas to identify duplicate bugs during retrospection such as grouping bugs with the same patch. Such an improvement allows SyzRetrospector to get more accurate results by reducing false negatives.

Magma [24], LAVA [25], and more [26]–[28], are evaluation tools for a wide array of fuzzers. They use test suites of bugs and measure how quickly a fuzzer can find bugs as well as which bugs it can find. This information can then be used to compare fuzzing strategies and high-level capabilities such as directed fuzzing strategies or the capability to interleave threads. So, these tools benchmark fuzzers against each other. Our study compares Syzbot to itself as its fuzzing environment improves over time. SyzRetrospector leverages the great lengths of time involved in continuous fuzzing, which is not captured by benchmarking tools. Because of this, current benchmarking tools are not a good fit for evaluating continuous fuzzers like Syzbot.

Additional studies have evaluated specific types of bugs or their security impacts. A number of studies, e.g., SyzScope [29], KOOBE [30], and others [31]–[33], analyze the impact of Linux kernel bugs, looking for high-risk consequences. These works give a key reason to uncover and patch bugs quickly. Other studies [34]–[37] use post processing on information like core dumps, stack traces, and thread execution logs to better understand bugs. Still others [38]–[41] focus on record and replay to debug. Execution Synthesis [42] and Star [43] study concurrency bugs by reproducing them. Our study is not concerned with a bug's impacts or its inner workings, rather we study its lifetime with relation to Syzbot.

Bisection is well known throughout the community and closely related to our strategies in SyzRetrospector. It has been implemented in git [44] and is already used in Syzbot to assist developers [45]. Other works [46]–[51] have built on bisection and explored new methods of root cause analysis. Our tool builds on bisection to include Syzkaller and syscall description commits to determine when a bug was revealed, not its root cause.

SyzDescribe [16], Syzgen [52], SyzSpec [53] DIFUZE [17], and more [18]–[20], [54], provide solutions for fuzzing kernel interfaces without syscall descriptions by automatically generating their own descriptions. Each of these works has the potential to be invaluable for fuzzing areas of code that lack syscall descriptions. Though Difuze was integrated into Syzkaller for testing on Android, Syzkaller still depends on manual description generation for Linux.

Bowknots [55] and Talos [56] both provide workarounds for found bugs that are not patched yet. These strategies represent possible solutions for blocking bugs until they are patched out, though they are not in use with Syzbot.

Much previous work has been done in kernel fuzzing in recent years. HFL [57] is a hybrid fuzzer that combines syzkaller with symbolic execution. SyzDirect [58] is a directed fuzzer that is capable of handling cross-syscall dependencies in the kernel [59]. HEALER [60] improves the quality of test cases and improves code coverage by learning the relationships between syscalls. SyzVegas [61] uses reinforcement learning to dynamically optimize the fuzzing strategies and maximize coverage. Other fuzzers focus on concurrency bugs [62]–[64], often controlling how threads are interleaved. Our contribution, focused fuzzing, is different in that we remove some syscall descriptions from Syzkaller without changing the fuzzing algorithm. Furthermore, focused fuzzing is not meant to compete with these fuzzers. Indeed directed fuzzers are faster and more reliable than focused fuzzing, but using them would nullify our ability to retrospect Syzbot.

XIV. Conclusion

In this paper, we undertook a large-scale study to provide a better understanding and quantification of Syzbot's bug-finding performance and improvements. We used our tool, SyzRetrospector, to analyze 695 bugs and found that 40% of bugs are hidden for more than 258 days before they are revealed. We presented findings on the behaviors of revealing factors, the effort required to induce these reveals, and the trends in delays over the past 7 years. Finally, we provide takeaways for improving Syzbot's delays based on our findings and experience.

XV. Acknowledgments

This work was supported in part by NSF Awards #1953932, #1953933, #2247880, #2247881, NSA Award #H98230-22-1-0308, and DARPA under Agreement No. FA8750-24-2-0002. The authors thank the anonymous reviewers and the shepherd for their insightful feedback.

References

[1] "syzbot," https://syzkaller.appspot.com/upstream, 2024.

[2] "The Kernel Address Sanitizer (KASAN)," https://docs.kernel.org/dev-tools/kasan.html, 2023.

[3] "WARNING in rtl28xxu_ctrl_msg/usb_submit_urb," https://syzkaller.appspot.com/bug?id=e98d2e8aa7283d11aa8e0b718d8afa1a058e6ae0, 2021.

[4] "WARNING in dlfb_submit_urb/usb_submit_urb," https://syzkaller.appspot.com/bug?id=9c2df342be9d102da75f9532e168a95b9c379ae4, 2022.

[5] "WARNING in exception_type," https://syzkaller.appspot.com/bug?id=dccafc201251e8dfa52f17986d33f7ecbd6747fc, 2021.

[6] "KASAN: slab-out-of-bounds Read in packet_recvmsg," https://syzkaller.appspot.com/bug?id=7c7245f9088053e9e49b97a341dee26c9ed40a2c, 2022.

[7] "memory leak in kobject_set_name_vargs (4)," https://syzkaller.appspot.com/bug?id=89c3ddb9936d3552995130298f1d2633ab9d3541, 2021.

[8] "WARNING in futex_requeue," https://syzkaller.appspot.com/bug?id=03f29b6252786a6f17661d727c03c83a7f70c86e, 2021.

[9] D. Mu, Y. Wu, Y. Chen, Z. Lin, C. Yu, X. Xing, and G. Wang, "An in-depth analysis of duplicated linux kernel bug reports," in *Network and Distributed Systems Security Symposium (NDSS)*, 2022.

[10] "Syzkaller: Syzcall Description Language," https://github.com/google/syzkaller/blob/master/docs/syscall_descriptions_syntax.md, 2022.

[11] H.J. Lu, "x86/build/64: Force the linker to use 2MB page size," https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=e3d03598e8ae7d195af5d3d049596dec336f569f, 2018.

[12] "How to contribute to syzkaller," https://github.com/google/syzkaller/blob/master/docs/contributing.md, 2018.

[13] "KASAN: slab-out-of-bounds Read in thrustmaster_probe," https://syzkaller.appspot.com/bug?id=e1c3525a4f4e2e4b6c1f73611ceaf61ef462700c, 2022.

[14] Qu Wenruo, "Revert "btrfs: compression: don't try to compress if we don't have enough pages"," https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=4e9655763b82a91e4c341835bb504a2b1590f984, 2021.

[15] "Upstream Bugs Found per Month," https://syzkaller.appspot.com/upstream/graph/found-bugs, 2024.

[16] Y. Hao, G. Li, X. Zou, W. Chen, S. Zhu, Z. Qian, and A. A. Sani, "Syzdescribe: Principled, automated, static generation of syscall descriptions for kernel drivers," in *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 2023, pp. 3262–3278.

[17] J. Corina, A. Machiry, C. Salls, Y. Shoshitaishvili, S. Hao, C. Kruegel, and G. Vigna, "Difuze: Interface aware fuzzing for kernel drivers," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 2123–2138.

[18] Z. Shen, R. Roongta, and B. Dolan-Gavitt, "Drifuzz: Harvesting bugs in device drivers from golden seeds," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 1275–1290.

[19] A. Bulekov, B. Das, S. Hajnoczi, and M. Egele, "No grammar, no problem: Towards fuzzing the linux kernel without system-call descriptions." in *Proc. Internet Society NDSS*, 2023.

[20] H. Sun, Y. Shen, J. Liu, Y. Xu, and Y. Jiang, "KSG: Augmenting kernel fuzzing with system call specification generation," in *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, 2022, pp. 351–366.

[21] K. Serebryany, "Oss-fuzz-google's continuous fuzzing service for open source software," in *USENIX Security symposium*. USENIX Association, 2017.

[22] J. Ruohonen and K. Rindell, "Empirical notes on the interaction between continuous kernel fuzzing and development," in *2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. IEEE, 2019, pp. 276–281.

[23] Z. Y. Ding and C. Le Goues, "An empirical study of oss-fuzz bugs," in *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. IEEE, 2021, pp. 131–142.

[24] A. Hazimeh, A. Herrera, and M. Payer, "Magma: A ground-truth fuzzing benchmark," *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, vol. 4, no. 3, pp. 1–29, 2020.

[25] B. Dolan-Gavitt, P. Hulin, E. Kirda, T. Leek, A. Mambretti, W. Robertson, F. Ulrich, and R. Whelan, "Lava: Large-scale automated vulnerability addition," in *2016 IEEE symposium on security and privacy (SP)*. IEEE, 2016, pp. 110–121.

[26] S. Lu, Z. Li, F. Qin, L. Tan, P. Zhou, and Y. Zhou, "Bugbench: Benchmarks for evaluating bug detection tools," in *Workshop on the evaluation of software defect detection tools*, vol. 5. Chicago, Illinois, 2005.

[27] J. Metzman, L. Szekeres, L. Simon, R. Sprabery, and A. Arya, "Fuzzbench: an open fuzzer benchmarking platform and service," in *Proceedings of the 29th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering*, 2021, pp. 1393–1403.

[28] Y. Li, S. Ji, Y. Chen, S. Liang, W.-H. Lee, Y. Chen, C. Lyu, C. Wu, R. Beyah, P. Cheng *et al.*, "Unifuzz: A holistic and pragmatic metrics-driven platform for evaluating fuzzers." in *USENIX Security Symposium*, 2021, pp. 2777–2794.

[29] X. Zou, G. Li, W. Chen, H. Zhang, and Z. Qian, "SyzScope: Revealing High-Risk security impacts of Fuzzer-Exposed bugs in linux kernel," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 3201–3217.

[30] W. Chen, X. Zou, G. Li, and Z. Qian, "Koobe: Towards facilitating exploit generation of kernel out-of-bounds write vulnerabilities," in *Proceedings of the 29th USENIX Conference on Security Symposium*, 2020, pp. 1093–1110.

[31] Z. Lin, Y. Chen, Y. Wu, D. Mu, C. Yu, X. Xing, and K. li, "Unveiling exploitation potential for linux kernel bugs," in *2022 IEEE Symposium on Security and Privacy (SP)*, 2022, pp. 3201–3217.

[32] X. Zou, Y. Hao, Z. Zhang, J. Pu, W. Chen, and Z. Qian, "Syzbridge: Bridging the gap in exploitability assessment of linux kernel bugs in the linux ecosystem," in *Proc. Internet Society NDSS*, 2024.

[33] Z. Liang, X. Zou, C. Song, and Z. Qian, "K-leak: Towards automating the generation of multi-step infoleak exploits against the linux kernel," in *Proc. Internet Society NDSS*, 2024.

[34] D. Weeratunge, X. Zhang, and S. Jagannathan, "Analyzing multicore dumps to facilitate concurrency bug reproduction," in *Proceedings of the fifteenth International Conference on Architectural support for programming languages and operating systems*, 2010, pp. 155–166.

[35] F. A. Bianchi, M. Pezzè, and V. Terragni, "Reproducing concurrency failures from crash stacks," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 2017, pp. 705–716.

[36] J. Huang, C. Zhang, and J. Dolby, "Clap: Recording local executions to reproduce concurrency failures," *Acm Sigplan Notices*, vol. 48, no. 6, pp. 141–152, 2013.

[37] N. Machado, B. Lucia, and L. Rodrigues, "Production-guided concurrency debugging," in *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2016, pp. 1–12.

[38] S. T. King, G. W. Dunlap, and P. M. Chen, "Debugging operating systems with time-traveling virtual machines," in *Proceedings of the 2005 USENIX Technical Conference*, 2005, pp. 1–15.

[39] S. M. Srinivasan, S. Kandula, C. R. Andrews, Y. Zhou *et al.*, "Flashback: A lightweight extension for rollback and deterministic replay for software debugging," in *USENIX Annual Technical Conference, General Track*. Boston, MA, USA, 2004, pp. 29–44.

[40] J. Huang, P. Liu, and C. Zhang, "Leap: Lightweight deterministic multi-processor replay of concurrent java programs," in *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, 2010, pp. 207–216.

[41] D. Lee, B. Wester, K. Veeraraghavan, S. Narayanasamy, P. M. Chen, and J. Flinn, "Respec: efficient online multiprocessor replayvia speculation and external determinism," *ACM Sigplan Notices*, vol. 45, no. 3, pp. 77–90, 2010.

[42] C. Zamfir and G. Candea, "Execution synthesis: a technique for automated software debugging," in *Proceedings of the 5th European conference on Computer systems*, 2010, pp. 321–334.

[43] N. Chen and S. Kim, "Star: Stack trace based automatic crash reproduction via symbolic execution," *IEEE transactions on software engineering*, vol. 41, no. 2, pp. 198–220, 2014.

[44] "Git Bisect," https://git-scm.com/docs/git-bisect, 2020.

[45] D. Vyukov, "syzbot Bisection," https://github.com/google/syzkaller/blob/master/docs/syzbot.md\#bisection, 2018.

[46] R. Saha and M. Gligoric, "Selective bisection debugging," in *Fundamental Approaches to Software Engineering: 20th International Conference, FASE 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings 20*. Springer, 2017, pp. 60–77.

[47] G. An, J. Hong, N. Kim, and S. Yoo, "Fonte: Finding bug inducing commits from failures," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 589–601.

[48] L. Bao, X. Xia, A. E. Hassan, and X. Yang, "V-szz: automatic identification of version ranges affected by cve vulnerabilities," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 2352–2364.

[49] S. Woo, D. Lee, S. Park, H. Lee, and S. Dietrich, "V0Finder: Discovering the correct origin of publicly reported software vulnerabilities," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 3041–3058.

[50] J. Dai, Y. Zhang, H. Xu, H. Lyu, Z. Wu, X. Xing, and M. Yang, "Facilitating vulnerability assessment through poc migration," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021, pp. 3300–3317.

[51] Z. Zhang, Y. Hao, W. Chen, X. Zou, X. Li, H. Li, Y. Zhai, and B. Lau, "SymBisect: Accurate bisection for Fuzzer-Exposed vulnerabilities," in *33rd USENIX Security Symposium (USENIX Security 24)*, 2024, pp. 2493–2510.

[52] W. Chen, Y. Wang, Z. Zhang, and Z. Qian, "Syzgen: Automated generation of syscall specification of closed-source macos drivers," in *Proc. ACM CCS*, 2021.

[53] Y. Hao, J. Pu, X. Li, Z. Qian, and A. A. Sani, "Syzspec: Specification generation for linux kernel fuzzing via under-constrained symbolic execution," in *Proc. ACM CCS*, 2025.

[54] B. Liu, C. Zhang, G. Gong, Y. Zeng, H. Ruan, and J. Zhuge, "Fans: Fuzzing android native system services via automated interface analysis." in *USENIX Security Symposium*, 2020, pp. 307–323.

[55] S. M. S. Talebi, Z. Yao, A. A. Sani, Z. Qian, and D. Austin, "Undo workarounds for kernel bugs," in *USENIX Security Symposium*, 2021.

[56] Z. Huang, M. DAngelo, D. Miyani, and D. Lie, "Talos: Neutralizing vulnerabilities with security workarounds for rapid response," in *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2016, pp. 618–635.

[57] K. Kim, D. R. Jeong, C. H. Kim, Y. Jang, I. Shin, and B. Lee, "HFL: Hybrid Fuzzing on the Linux Kernel," in *Proc. Internet Society NDSS*, 2020.

[58] X. Tan, Y. Zhang, J. Lu, X. Xiong, Z. Liu, and M. Yang, "Syzdirect: Directed greybox fuzzing for linux kernel," in *ACM CCS*, 2023.

[59] Y. Hao, H. Zhang, G. Li, X. Du, Z. Qian, and A. Amiri Sani, "Demystifying the Dependency Challenge in Kernel Fuzzing," in *Proc. IEEE/ACM ICSE*, 2022.

[60] H. Sun, Y. Shen, C. Wang, J. Liu, Y. Jiang, T. Chen, and A. Cui, "HEALER: relation learning guided kernel fuzzing," in *SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event / Koblenz, Germany, October 26-29, 2021*, R. van Renesse and N. Zeldovich, Eds. ACM, 2021, pp. 344–358. [Online]. Available: https://doi.org/10.1145/3477132.3483547

[61] D. Wang, Z. Zhang, H. Zhang, Z. Qian, S. V. Krishnamurthy, and N. B. Abu-Ghazaleh, "Syzvegas: Beating kernel fuzzing odds with reinforcement learning." in *USENIX Security Symposium*, 2021, pp. 2741–2758.

[62] D. R. Jeong, K. Kim, B. Shivakumar, B. Lee, and I. Shin, "Razzer: Finding kernel race bugs through fuzzing," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 754–768.

[63] M. Xu, S. Kashyap, H. Zhao, and T. Kim, "Krace: Data race fuzzing for kernel file systems," in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 1643–1660.

[64] P. Fonseca, R. Rodrigues, and B. B. Brandenburg, "SKI: Exposing kernel concurrency bugs through systematic schedule exploration," in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, 2014, pp. 415–431.