# Meerkat: Pushing the Practical Limits of Dynamic Bisection with PoC Mutation

Joseph Bursey
*University of California, Irvine*

Christoph Sendner
*University of California, Irvine*

Ardalan Amiri Sani
*University of California, Irvine*

Zhiyun Qian
*University of California, Riverside*

## Abstract

Once a Linux kernel bug is identified, the developers perform the difficult task of patching it. Triage tools such as Syz-bisect provide a form of root cause analysis called bisection, which systematically checks historical kernel commits for the presence of the bug until it determines the Bug-Introducing Commit (BiC). Unfortunately, Syz-bisect correctly identifies the BiC for only 35% of bugs. In order to improve Syz-bisect while remaining within its time and resource constraints, we identify three areas where Syz-bisect could be improved: bug deduplication, the use of all available PoCs, and PoC mutation. Our solution Meerkat is the first dynamic bisection tool to apply scalable PoC mutation, improving over Syz-bisect by 64%. Through an in-depth manual analysis of Meerkat's bisection results, we find that dynamic bisection is critically limited by bug detectors changing with kernel versions, changes near the buggy code, and unrelated bugs that block dynamic analysis. Based on the nature of these issues, we argue that Meerkat is approaching the practical limits of what is possible with dynamic bisection in the real world. Furthermore, we correct the ground truth for 13 bugs, thus improving the dataset for future research.

## 1 Introduction

In the past several years, fuzz testing has grown to be one of the primary methods by which bugs are found in the Linux kernel [15, 27, 38, 41, 43, 45, 46, 52, 59, 66]. However, after a kernel bug is identified, the often more difficult process of patching the bug takes place. Google's state-of-the-art kernel fuzzer Syzbot [14], which is responsible for finding many of these bugs, provides developers with several artifacts, including the kernel commit the bug was found on, the kernel configuration, and, where possible, a **Proof-of-Concept** (PoC) program thought to reproduce the bug. As testament to its prominence, Syzbot has identified over 8,400 open and confirmed bugs since 2017, 7,100 of which have been patched. However, as many as 12,800 bugs found by Syzbot have been automatically marked as invalid by the fuzzer due to developer inactivity [11]. In order to make debugging easier, Syzbot provides a form of root cause analysis called *bisection*. Through this process, Syzbot systematically tests historical kernel commits for the presence of a bug until it identifies the commit which introduced it – the **Bug-Introducing Commit** (BiC). This commit likely has only a few changes that a developer can quickly look through to find the root cause.

Syzbot's bisection analysis is carried out by Syz-bisect [13], a dynamic analysis-based bisection tool. Syz-bisect reproduces the bug through historical kernel commits using the PoC until it identifies the BiC, which it reports to the kernel developers. Since Syz-bisect is fully integrated into Syzbot, the analysis can happen quite soon after the bug is found, providing developers with key information they can use to patch the bug. Furthermore, recent studies [35] find that the correctness of a bisection result is directly linkable to its usefulness in recommending the best developer to patch the bug, directing the developer to the buggy code, as well as speeding up time-to-patch. However, our analysis shows that Syz-bisect correctly identifies the BiC a mere 35% of the time, and suffers from both false positives and negatives. As a result, Syz-bisect hinders bug patching by incorrectly reporting the BiC, leading to both longer patch times and incorrect patches [1].

We identify 3 challenges which could help to improve Syz-bisect's accuracy: **(1)** Syz-bisect lacks any form of deduplication, **(2)** Syz-bisect depends on a single, arbitrarily chosen, PoC for the entirety of bisection, and **(3)** Syz-bisect does not adapt the PoC as the kernel changes.

Motivated by these, we set out to push the practical limits of dynamic bisection (*i.e.,* bisection that leverages dynamic analysis), improving its accuracy as much as possible while maintaining the time and resource constraints that make Syz-bisect useful. We present our solution, Meerkat, which addresses these challenges by implementing bug deduplication, incorporating all known PoCs, and utilizing PoC mutation to adapt the PoCs to the changing kernel. We evaluate Meerkat's performance on a dataset of 150 bugs over the course of

90,700 CPU-hours, and find that it is able to reach the BiC 58% of the time, improving over Syz-bisect by 64% (1.64×). Importantly, Meerkat is able to accomplish this with the same amount of computing resources and a median runtime within 8 minutes of Syz-bisect, thus it maintains usability and scalability in the context of Syzbot. Further, we perform an ablation study in order to highlight the improvement gained by each of the three design aspects of Meerkat, and demonstrate that the majority of the improvement comes from the aforementioned mutation of the PoCs.

Despite these improvements, Meerkat is still unable to reach the BiC in all cases. In light of this, we carry out an in-depth manual analysis of Meerkat's bisection results in order to understand the open challenges facing dynamic bisection tools. Across 300 person-hours, we identify key factors that limit dynamic bisection, which include bug detectors changing with kernel versions, code changes around the buggy code, blocking bugs, and other crashes preventing dynamic analysis. Based on the nature of these issues, we argue that Meerkat is approaching the practical limits of what is possible with dynamic bisection.

Through considerable amount of manual analysis of each bug's developer-reported BiC, we identify 13 cases where the BiC is incorrect. Commonly, an incorrectly labeled BiC is just the commit that introduced the assertion used to detect the bug or the commit that introduced the buggy feature. However, in neither case is the buggy logic introduced by the commit. In other cases, the BiC is automatically mislabeled by Syzbot. We rectify the ground truth of the dataset for future research by correcting the BiCs.

We list the contributions of this paper below:

- We present Meerkat, the first dynamic bisection tool that utilizes PoC mutation to reach the BiC in 64% more cases than the baseline Syz-bisect, while maintaining the same time and resource constraints.
- We perform an ablation study on a large set of 150 bugs, showing the importance of each design aspect of Meerkat.
- We carry out an in-depth manual analysis of our results in order to better understand the practical limitations facing dynamic bisection.
- We correct the developer-reported BiC in 13 cases, thus improving the dataset for future work.
- We open source Meerkat, our analysis results, and our dataset to promote future research[1].

[1]Our open source repositories can be found at https://doi.org/10.5281/zenodo.20317981 and https://github.com/trusslab/meerkat.

## 2 Background

Here, we provide some background in the field of bug triage and bisection for the purpose of this work.

### 2.1 Syzbot and Syzkaller

Syzbot is Google's continuous fuzzer that has been fuzzing the Linux kernel since 2017. Syzbot manages a large number of instances of the coverage-guided fuzzer Syzkaller [15] (29 as of writing), each of which fuzzes the Linux kernel with different configurations. Each Syzkaller instance works with 10 virtual machines (VMs), each with 2 CPU cores to fuzz a recent Linux kernel commit. Syzkaller randomly generates test cases from a set of syscall descriptions with the goal of covering as much of the kernel code as possible. Each test case is a series of syscalls with complex arguments to be run in the VM. As a test case is run, it hits instrumentations called coverage points in the kernel. Syzkaller tries to maximize the number of coverage points hit by collecting interesting test cases in its corpus and using those to mutate into new test cases. Whenever it finds a bug, Syzbot reports it to the kernel developers and tracks the bug until it is patched. Syzbot also provides a number of artifacts such as the commit the bug was found on, the kernel configuration, and the PoC [12]. Since its inception, Syzbot has become closely integrated with the Linux development cycle, with many bug patches directly referencing bug reports from Syzbot.

### 2.2 Bisection

Bisection is a form of root cause analysis that systematically sifts through historical commits to find the commit that introduced a bug, the BiC. It gets its name from its binary search-like method of dividing the remaining commits roughly in half at each step. Bisection proves useful to developers by directing them to the root cause of a bug as well as inferring vulnerable versions of the kernel.

Previous work has been done on improving the bisection algorithm itself [16, 51, 58], and how the bug is detected at each commit: either statically [24, 54, 68] or dynamically [13, 29]. Static analysis tools make use of the source code or symbolic execution to identify buggy logic in a given commit. However, it requires complex buggy state detectors, which has limited prior work to a few specific crash types such as use-after-frees [68]. In contrast, dynamic analysis tools build and boot a live kernel on which to run PoCs and observe the bug through a crash. This work focuses on utilizing dynamic analysis as we believe this strategy holds critical advantages over static analysis which make it better suited for bisection. By using the same bug detectors used by fuzzers to originally find the bugs, dynamic analysis tools are generalizable to any bug found through fuzzing. Moreover, the bug manifests as a kernel crash during dynamic analysis as a result of running the

PoC. This proves that the bug exists and that the kernel is vulnerable to that PoC, which static analysis cannot immediately do. We refer to bisection tools that leverage dynamic analysis as their detection method as **dynamic bisection** tools.

### 2.3 Syz-bisect

Syz-bisect is a dynamic bisection tool that has carried out bisection on the bugs found by Syzbot since March of 2019 [31].

Syz-bisect has two major components: a test component and a search component. The test component is responsible for determining, for any commit, if a particular bug is present on that commit. At each commit, Syz-bisect simply runs the most recently created PoC in a loop until a crash is found, or a timeout is hit. The search component is responsible for choosing which commit to test next in order to eventually find the BiC. Syz-bisect's search algorithm begins at the **anchor commit**, which is chosen because it is where the bug was originally found and is known to reproduce. Here, it runs the chosen PoC in order to confirm that the bug reproduces. If the bug is found, Syz-bisect traverses backwards on major Linux releases (*e.g.,* 6.8, 6.7) until the bug does not reproduce on one of them. This gives Syz-bisect two releases between which the BiC lies. Syz-bisect then uses Git Bisect [10] between the two releases until it finds a single commit before which the bug does not reproduce. This commit is reported as the BiC. It is important to note that during the entirety of bisection, Syz-bisect uses a single PoC and assumes that any crash observed (barring boot failures) is the same as the bug being bisected.

Much of the usefulness of Syz-bisect is derived from it begin fully automated and integrated into Syzbot, completing in under 12 hours, and only using as much compute resources as a single Syzkaller instance. It is important then, that any replacement for Syz-bisect also abide by these constraints.

## 3 Definitions

Here, we provide definitions for the terms used in this paper.

### 3.1 Bug Manifestation

We define a **bug** to be an unintended kernel state resulting from a singular root cause. This definition is well known and has been used in several works [49, 69, 71]. However, this is not the granularity at which dynamic bisection operates. For this reason, we define the more precise **bug manifestation** as an unintended kernel state resulting from a singular root cause that manifests as a kernel panic resulting from a particular bug detector at a particular crash site. This definition has two key constraints that make it more specific than the definition for a bug. First, the bug must manifest as a kernel panic (*i.e.,* a crash) resulting from a bug detector (*e.g.,* KASAN, `WARN_ON()`). Dynamic bisection depends on the resulting crash in order to observe the buggy state and trace it back through commits. Second, the bug manifestation occurs at a specific crash site. A single bug can have multiple crash sites or types depending on thread interleaving or memory dynamics. However, it is quite difficult to automatically compare these manifestations, and many require time-consuming analysis or domain expertise to deduplicate [49]. Syzbot itself performs bug triage based solely on crash type and crashing function, grouping similar crashes and artifacts under a **bug report**. To approach this problem, we define two types of bug deduplication in §3.2, and focus our solution on deduplicating specific bug manifestations.

### 3.2 Inter vs. Intra-report Deduplication

Bug deduplication is a critical part of dynamic bisection as it allows the tool to determine if a specific bug has been observed. However, as we have mentioned, bugs can have multiple manifestations with widely differing symptoms. Based on this, we define two types of bug deduplication.

First, **intra-report deduplication** determines if two crashes are the same manifestation of a bug. This can be decided based on the type and location of a crash, which can be derived from the bug detector and the crashing function or stack trace. This type of deduplication is useful as bug manifestations can vary slightly over time or between kernel builds, while the underlying type and location remain invariant.

Second, **inter-report deduplication** compares two bug manifestations to determine if they are the same underlying bug. For instance, the two bugs `WARNING in inet_csk_get_port` [5] and `KASAN: use-after-free Read in inet_bind2_bucket_find` [3] are known to be duplicates as determined by kernel developers. However, neither crash is clearly related to the other. In fact, the warning does nothing to signify that concurrency is involved in the bug as implied by the use-after-free. Furthermore, previous work by Mu *et al.* [49] has shown that performing accurate deduplication would require rebuilding the kernel several times on top of in-depth static analysis. Because of this, it is not only difficult but prohibitively expensive to automatically deduplicate bug reports from each other. These methods are not fit to be performed during dynamic bisection.

In order to determine how useful inter-report deduplication could be, we perform inter-report deduplication of our dataset based on the ground truth of previous work [49] in §9.1. Our results show that in 14 bug groups containing duplication, inter-report deduplication would only improve the bisection result in 3 groups. Based on the minimal gain and extra cost of inter-report deduplication, we argue that it should not be performed during dynamic bisection and leave it out of Meerkat's implementation.

To our knowledge, the differentiation between inter- and intra-report deduplication has not been made in prior work, yet we believe it to be an important aspect when understanding the capabilities of dynamic bisection.

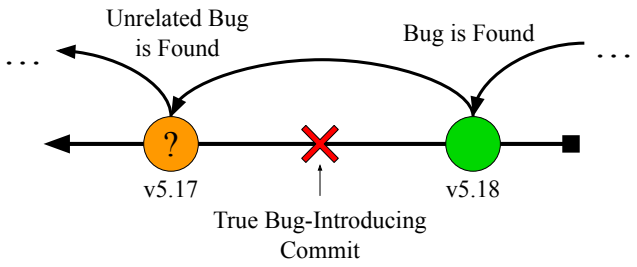


Figure 1: A visualization of how bisection can skip over the true BiC, leading to an incorrect result.

## 4 Challenges and Motivation

We observe an important issue: Syz-bisect is inaccurate and often fails to report the correct BiC. At best, the incorrect report is unhelpful to developers and a waste of fuzzer resources. But at worst, Syz-bisect may mislead the patch effort resulting in incomplete or entirely incorrect patches. In this section, we discuss three challenges through which we can improve bisection results while remaining within Syz-bisect's time and resource constraints.

**Challenge 1 (CH-1) - Intra-report deduplication.** First, Syz-bisect has no concept of deduplication. It assumes that a PoC reproduces a single, unique bug, therefore any crash resulting from running the PoC must be the same bug. However, this is not the case. Syz-bisect often finds unrelated bugs: either those that occur earlier in the execution of the PoC, or those that arise when the kernel is under stress, which we call **kernel instabilities**. As a result, Syz-bisect follows these false positives and eventually arrives at an incorrect result.

For example, when bisecting the bug `WARNING in nilfs_segctor_do_construct` [7] Syz-bisect finds unrelated bugs, leading to false positives. In this case, Syz-bisect begins by finding the bug on several releases, starting with Linux 6.3. However, at Linux 5.17, it finds an unrelated bug. Since it has no notion of deduplication, it assumes this bug must be the same as the original bug. Due to the kernel's instability as it gets older, Syz-bisect follows unrelated bugs all the way back to Linux 4.19, the oldest release it tests. Here, it says the bug reproduces on the "oldest tested release," which is clearly incorrect. Because of false positives like these, Syz-bisect can greatly overshoot the true BiC as illustrated in Figure 1.

**Challenge 2 (CH-2) - Using all available PoCs.** Not all PoCs provided by Syzbot are able to reliably reproduce their bug manifestation. This is often due to how PoCs are created. When Syzbot identifies a crash, the test case that caused the crash is not yet obvious. Therefore, Syzbot reruns the test cases that were executed leading up to the crash one at a time. However, there is once again no deduplication during this process. If an unrelated bug is found while searching for the PoC, Syzbot will publish the incorrect test case as a PoC for the manifestation in question. In this, Syzbot makes a critical assumption that each PoC uniquely reproduces a single bug. Any mistakes made when creating the PoC will have negative ramifications in any tools that depend on these PoCs, such as bisection. We have also observed cases in which one PoC triggers a concurrency bug more easily than others. During bisection, depending on a single PoC is a major source of inaccuracies.

```
1  int hci_register_dev(hdev) {
2      ...
3      alloc_ordered_workqueue();
4      dev_set_name(hdev->dev, ...);
5      device_add(hdev->dev);
6  }
7
8  int device_add(dev) {
9      ...
10     kobject_add(dev->kobj, ..., NULL);
11 }
12
13 int kobject_add(kobj, ..., fmt) {
14     ...
15     if (kobj->name && !fmt) return 0;
16     ...
17     if (!strchr(fmt, '%'))
18     ...
19 }
20
21 alloc_ordered_workqueue() {
22     ...
23     new_attrs = alloc_workqueue_attrs();
24 -   tmp_attrs = alloc_workqueue_attrs();
25     ...
26 }
```

Figure 2: Pseudo C code snippets for `hci_register_dev()`.

We observe that Syzbot naturally produces multiple PoCs for each bug, yet only one is used by Syz-bisect. It is worth noting that adding more PoCs to bisection will require some alteration of the search algorithm.

**Challenge 3 (CH-3) - PoC Adaptation.** The PoCs provided by Syzbot each represent a single method to reach the buggy code and trigger the bug. However, the code surrounding the bug may change as bisection goes back in time, leading some PoCs to break. Consider the bug `general protection fault in hci_register_dev` [6]. Syz-bisect is only able to find this bug as far back as September 1st, 2023. It reaches a merge commit that updates the kernel's workqueues' support for CPU affinity, which does not immediately appear related to the bug at all. We find that the bug is still present prior to this commit, but a mutated PoC is required to trigger it.

As the name implies, this bug is a null pointer dereference in the function `hci_register_dev()`, specifically in the kobject initialization portion. Figure 2 shows pseudo C code snippets involved in the crash. We note that before

we reach the buggy code, `hci_register_dev()` calls `alloc_ordered_workqueue()` on line 3, which allocates a workqueue for the device. `hci_register_dev()` then calls `dev_set_name()`, which is supposed to set `hdev->dev->kobj->name` to some string, and then calls `device_add()` on line 5. `device_add()` calls `kobject_add()`, passing `NULL` for the `fmt` parameter, and eventually performs the check seen on line 15. Past this check, `fmt` is dereferenced, which is where the general protection fault occurs.

The check on line 15 can be bypassed if both `kobj->name` and `fmt` are `NULL`. To accomplish this, the PoC uses a functionality called fault injection. Using this, Syzkaller can cause the *n*th call to `kmalloc` to fail. This is useful for testing how code reacts in rare situations, such as if the machine had just run out of memory. In this case, the crashing syscall of the PoC, `ioctl$KDADDIO`, is supposed to fail on the 20th `kmalloc` call. The 20th call is exactly the `kmalloc` used to allocate `kobj->name`. When it fails, the pointer remains `NULL`, allowing execution to bypass the check on line 15. It follows that if the number of `kmalloc` calls changes at any point, the PoC will not crash. Of course, the merge commit that Syz-bisect reaches is responsible for removing an allocation call in `alloc_ordered_workqueue()`, as shown in line 24. So, before this commit, the PoC should inject a fault on the 21st `kmalloc` call rather than the 20th. Otherwise, the PoC will inject its fault in the workqueue allocation, which fails without crashing.

Ironically, *the ever-changing kernel is the reason bisection is required, yet it is often completely ignored by bisection tools such as Syz-bisect*. Without any adaptation for older commits, Syz-bisect has no hope of consistently determining whether a bug is present.

# 5 Design

Based on the above challenges, we design our tool, Meerkat, which deduplicates bug manifestations, utilizes all available PoCs, and mutates PoCs to adapt them to older kernel commits while maintaining the same time and resource constraints placed upon Syz-bisect.

## 5.1 Overview

Meerkat's design, as shown in Figure 3, is *fully automated* and built to fit into Syzbot's workflow as a drop-in replacement for Syz-bisect. Its goal is to transform Syzbot-generated artifacts into precise *BiC*s to assist developers in patching bugs. Meerkat extracts artifacts from Syzbot and uses them to initiate a reliable, scalable bisection. The process is divided into two phases: the *PoC Mutation* phase and the *PoC-Only* phase, with *Intra-Report Deduplication* being used in both phases. Each phase is named for the test component used during that phase: either the *mutation engine* or *PoC loop*. Traversal through the commit history is defined by the *Bisection Search Algorithm*. In Figure 3, the search algorithm is shown as a timeline featuring the *Anchor Commit*, *Major Releases*, and *Git Bisect*. In the remainder of this section, we describe each aspect of Meerkat in detail. We first introduce functionalities shared across all components.

**Artifacts.** The artifacts generated by Syzbot include the PoCs thought to trigger the bug manifestation, the anchor commit on which the PoCs are thought to work, the Git branch on which to perform bisection, and the kernel configuration required to build the kernel.

**Corpus.** The corpus is a collection of PoCs and test cases that are used by the mutation engine to exercise the buggy code. During the mutation phase, we add newly found unique PoCs to the corpus.

## 5.2 Bisection Search Algorithm

Here, we describe Meerkat's search algorithm, which is used in both phases of Meerkat's workflow. The algorithm itself shares the same strategy used by Syz-bisect, which is agnostic to the test component being used. The goal of the algorithm is to traverse the kernel's commit history, which contains over 1.4 million commits, while minimizing the number of commits visited before identifying the BiC. We structure the algorithm into three steps:

**Step ①:** The search algorithm begins at the *Anchor Commit* where Meerkat runs the test component to ensure the bug can be reproduced.

**Step ②:** If the bug is found, Meerkat continues to test on *Major Releases* roughly linearly back in time. We note that this process is roughly linear because both Meerkat and Syz-bisect begin skipping releases as they go back in time. The first two releases are traversed consecutively, then a single release is skipped for the next 6 releases, then 2 are skipped. If a predetermined oldest release is reached, bisection is terminated with the notice that the bug reproduces on the oldest tested release. This release has changed over time for Syz-bisect. As of 2025, the stopping point is Linux 4.19 [8], which Meerkat also uses.

**Step ③:** If a bug is not found in a given release, Meerkat performs *Git Bisect* between the oldest release on which the bug reproduces and the release where the bug does not reproduce. Meerkat iteratively narrows this interval of commits based on Git bisect [10] until it identifies the BiC.

## 5.3 Phase 1: PoC Mutation Phase

In the first of Meerkat's two phases, it tests for the presence of the bug by mutating the available PoCs to adapt them to older kernel commits, with the primary goal of arriving at a suspected BiC for the second phase.

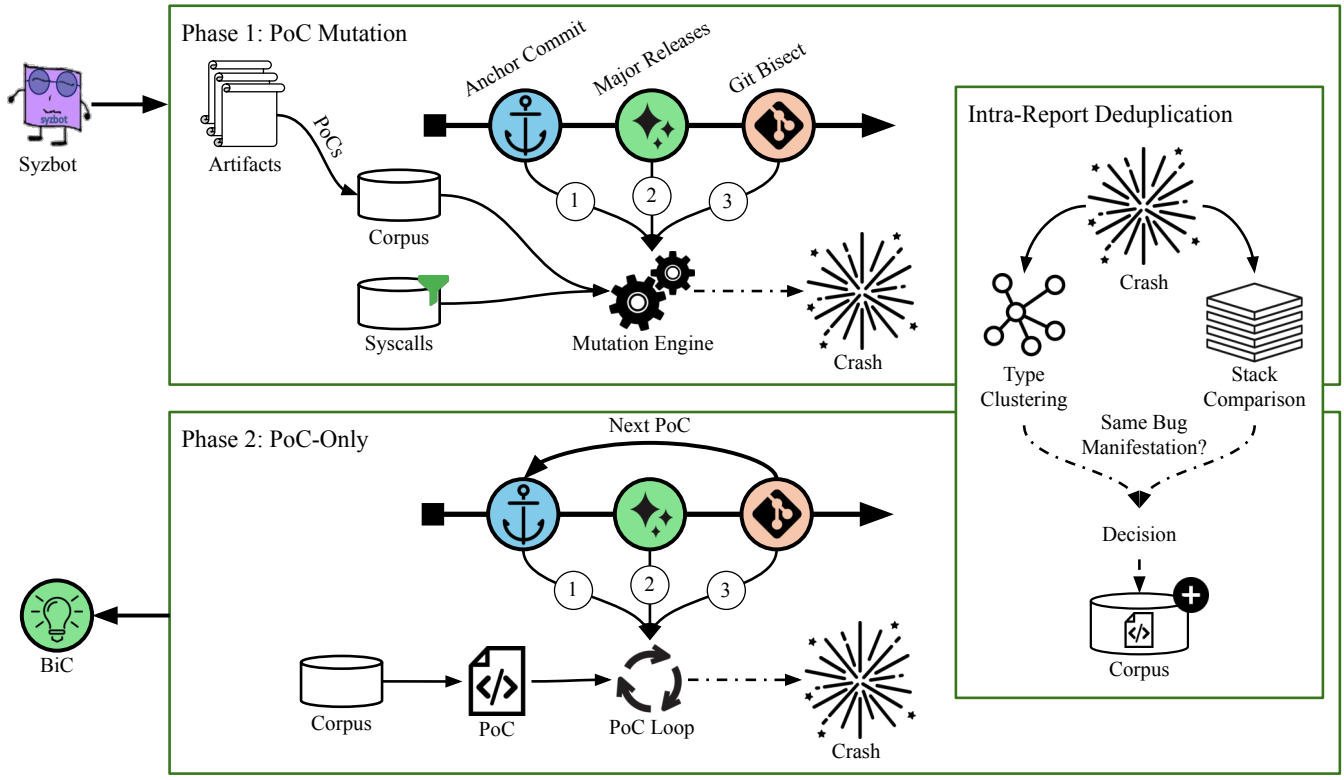


Figure 3: Meerkat's workflow.

**PoCs and Corpus:** Meerkat addresses CH-2 by incorporating all available PoCs for the bug into its starting corpus. We extract the PoCs from the artifacts provided by Syzbot. However, these PoCs are created on and for specific kernel versions, and may break on older commits. We observe that PoCs contain inherent information describing how to reach the buggy code, and that relatively small changes are required to continue reproducing the same bug manifestation on older commits. That information is encoded in the syscalls used in the PoCs as well as their arguments. Even if a PoC is broken on older kernel commits, we can use the information to create a new PoC that works. So, we enable Meerkat to mutate the available PoCs during bisection, allowing it to adapt them to older commits on the fly.

**Mutation Engine:** Addressing CH-3, Meerkat mutates the PoCs in order to find new ways to trigger the bug manifestation. As a precursor to mutation, the mutation engine runs each PoC in quick succession. If the PoCs do not directly trigger the manifestation, it mutates more test cases from them.

Each mutation is a small, incremental change to the test case. Meerkat can choose to either perturb an argument, or add or remove a syscall, but the goal is for resulting test cases to exercise as close to the buggy code as possible. Meerkat has access to the nearly 7,800 syscall variations from Syzkaller, but very few of these are related to the bug at hand. So, we restrict the syscalls to the union of those that appear in the PoCs. By filtering the syscalls in this way, we ensure the mutated test cases remain local to the buggy code.

As new test cases are mutated and run, they are added back into the corpus, which Meerkat can choose from for future mutations. So, across each commit, Meerkat builds up a corpus of test cases derived from the PoCs that exercise the area near the buggy code. By preserving this corpus between commits, we can jump-start future mutation tests on other commits and improve Meerkat's ability to reproduce the bug. Even if a new PoC is neither identified nor added to the corpus, the test cases from which it was mutated are already in the corpus. So, the corpus as a whole gets closer to and can more easily mutate the same PoC on other commits. Importantly, Meerkat's test case throughput and ability to reproduce bugs does not degrade as the corpus becomes larger and the kernel becomes older.

The test at each commit is terminated if either a 10 minute timeout is reached or the bug is found. For every crash observed, intra-report deduplication determines whether the correct bug manifestation has been found. If it is the same manifestation, Meerkat must then isolate the PoC. Since many test cases were run leading up to the crash, Meerkat does not immediately know which one caused it. Meerkat reruns the test cases in order to determine the correct PoC, adds it back into the corpus, and tracks it for use during the remainder of bisection.

Iterating once through the *Bisection Search Algorithm*, Meerkat arrives at a suspected BiC. At this point, the bug does not reproduce past the suspected BiC through mutation, but may still reproduce if the PoC is run in a loop. In order to catch these cases where the bug is difficult to reproduce, Meerkat advances to phase 2. Before continuing, Meerkat updates the anchor commit to be the first commit on which the bug did not reproduce. We note that if the bug failed to reproduce at the anchor commit, the original anchor commit is used.

We optimistically designed Meerkat to start with PoC mutation as it is able to reproduce most bugs, and allows Meerkat to find new PoCs that can be used in phase 2 if needed. On the way to the BiC, Meerkat has a limited amount of time with which to find new PoCs, so starting with mutation gives Meerkat a better advantage.

## 5.4 Phase 2: PoC-Only Phase

With the new *Anchor Commit*, Meerkat begins anew with one of the PoCs. It is likely that the bug does not reproduce, owing to phase 1 reaching the true BiC. However, if the bug does happen to reproduce, Meerkat again follows the *Bisection Search Algorithm*: traversing *Major Releases* roughly linearly, then using *Git Bisect* to identify a new, older suspected BiC. The only difference is how Meerkat tests for the bug: running the PoC in a loop for 10 minutes. When either the PoC fails to reproduce the bug at the anchor commit or a new suspected BiC is found, Meerkat chooses the next PoC and repeats the process. In theory, Meerkat attempts bisection on each PoC, but in practice usually only one PoC is needed. In the vast majority of cases, Meerkat must find one PoC that works and use it. The rest will likely fail to reproduce the bug either because they are broken or because the correct BiC has already been found. As a result, the runtime for bisection does not increase drastically, but merely by tens of minutes depending on the number of available PoCs.

## 5.5 Intra-Report Deduplication

Following from CH-1, Meerkat performs intra-report deduplication, comparing each crash that is observed to the original bug manifestation. Two manifestations are the same if both their crash types and crash sites match.

**Type Clustering:** In order to determine if two crash types are the same, Meerkat clusters the crash types and then confirms that the two crash types are in the same cluster. For instance, "`BUG: unable to handle kernel NULL pointer dereference`" and "`KASAN: null-ptr-deref`" both represent the same underlying failure, a null pointer dereference that was caught by different detectors. So, these crash types are put in the null pointer dereference cluster and treated as the same.

**Stack Comparison:** Here, Meerkat determines if the same crash site was observed. In some cases, crash sites can be compared using the crashing functions. However, across kernel versions and builds, the crashing function may be refactored or misidentified in some way. If the crashing functions do not match, Meerkat compares the stack traces recovered from the crashes.

To compare stack traces, Meerkat uses a normalized Levenshtein distance [67]. This metric counts the number of additions, removals, and swaps required to transform one list into another, resulting in a positive integer. We then normalize the score based on the length of the stack trace to get a number in the range $[0,1]$ using the following formula.

$$Score = 1 - \frac{\text{LevenshteinDist}(S1,S2)}{max(len(S1),len(S2))} \quad (1)$$

A score of 1 means the stack traces are the same. Previous work [17,49] has found a threshold of 0.8 to be robust enough to handle typical changes in the stack trace such as refactoring, renaming, or inlining. If two stack traces score at least the threshold, Meerkat considers them to be the same.

**Decision:** If *Type Clustering* and *Stack Comparison* agree, Meerkat concludes that the same bug manifestation has been observed. During phase 1, the crash may have resulted from a newly mutated, unique PoC. In this case, Meerkat isolates the PoC and adds it back into the *Corpus*. The PoC is then reused throughout phases 1 and 2 as a known PoC.

# 6 Implementation

In this section, we discuss the finer details of Meerkat's implementation. We implemented Meerkat in 8,052 lines of C/C++ code as counted by cloc [9].

## 6.1 Syzkaller as a Mutation Engine

We build Meerkat's mutation engine on top of the already available Syzkaller, modifying it in the following ways.

First, we disallow the generation of entirely new test cases. When creating the next test case to run, Syzkaller can typically either mutate an existing test case or generate a new one from scratch. Test cases created through "generation" are initialized with default arguments that do not penetrate deep into call stacks and are thus not helpful for finding a specific bug. Instead, we want our mutation engine to find alternate ways of triggering the bug using the PoCs' intrinsic information about how to get close to the buggy code. To enforce this behavior, we modify the function `genFuzz()` in `pkg/fuzzer/fuzzer.go`, changing the mutation rate to make it impossible for Syzkaller to choose to generate new test cases. This way, all test cases are derived directly from the PoCs, forcing localized changes to explore close to the buggy code.

```
1  LevensteinDist(S1, S2)
2      if (min(len(S1), len(S2)) == 0)
3          return max(len(S1), len(S2))
4
5      if (S1[0] == S2[0])
6          return LevensteinDist(S1[1,], S2[1,])
7
8      return 1 + min(
9          LevensteinDist(S1[1,], S2[1,]),
10         LevensteinDist(S1[1,], S2),
11         LevensteinDist(S1, S2[1,]))
```

Figure 4: Pseudo-code for calculating the Levenshtein distance of two stacks.

Second, we restrict which syscalls Syzkaller is allowed to add to test cases to only those that appear in the PoCs. To do so, we enumerate the union of syscalls used in the PoCs, and pass them to Syzkaller using the "`enable_syscalls`" configuration. By performing this restriction, we ensure that the mutation engine does not get distracted by code in other parts of the kernel and instead focuses mutation towards the buggy code.

Choosing Syzkaller as our mutation engine comes with several advantages. Syzkaller already has the desired workflow, starting by running the PoCs and then mutating new test cases from them. Since Syzkaller is the underlying fuzzer for Syzbot, it is naturally compatible with the PoCs generated by Syzbot and is able to identify crashes with the same bug detectors. This also makes reintegration with Syzbot much easier, as we only have to modify the existing code base. Lastly, Syzkaller is scalable and can be configured to use the same amount of resources as Syz-bisect: 10 VMs each with 2 CPU cores.

## 6.2 Intra-Report Deduplication Method

Meerkat's intra-report deduplication method is built up in two parts: crash type clustering and stack trace comparison, based on the invariants of a bug manifestation.

The crash type clustering is meant to handle inconsistencies in which bug detectors cause the kernel to panic, as well as slight inaccuracies within those detectors. The clustering is hard-coded as there is a finite set of possible crash types, and used primarily for KASAN bugs and null pointer dereferences. As mentioned previously, a null pointer dereference could be detected by KASAN or the kernel's built-in null pointer handler. Whether a buggy state is caught by one detector or another makes no difference since it is still the same underlying issue. Similarly, sanitizers like KASAN may have some inaccuracies while triaging bugs. For instance, when KASAN observes a use-after-free, it tries to determine if the use was a read or a write, and appends this to the crash type information. However, KASAN cannot always correctly determine what type of use occurred, resulting in different crash types between crashes (*i.e.,* "`KASAN use-after-free Read`" vs. "`KASAN use-after-free`").

In order to carry out stack trace comparison, Meerkat extracts the dumped trace from the crash. While parsing the stack trace, Meerkat filters out certain debug functions such as those belonging to KASAN. This removes unhelpful functions and improves the accuracy of comparisons.

Meerkat's stack trace comparison is implemented using a normalized Levenshtein distance. The algorithm can be seen in Figure 4. In our implementation, we also memoized the recursion for efficiency.

Meerkat's deduplication method is notably different from Syzkaller's. Syzkaller sorts crashes based on their exact crash type plus crashing function, often seen as the "bug name." However, Syzkaller fails to perform deduplication any time after the initial discovery, such as when creating the PoC or during bisection. Meerkat deduplicates reports based on their clustered crash types and stack traces whenever a crash is observed, which is much more resilient to false negatives.

## 6.3 Compatibility With Older Kernels

As bisection progresses back in time, the kernel commits become less compatible with modern compiler toolchains and virtual machines. If a kernel commit does not compile or boot, we cannot perform dynamic analysis on it. However, the alternative of skipping commits is a major source of slowdown in the overall bisection runtime. If one commit cannot be built or booted, the commits around it usually have the same issue. Bisection tools including Meerkat can spend a lot of time attempting to build and boot these commits only to fail and have to skip them. Additionally, if the BiC happens to be one of the broken commits, the bisection tool will be unable to find it. On the other hand, patching arbitrary kernel commits is not the focus of this work, and would require a great amount of engineering effort. As such, we take the list of compilers and curated patches that Syz-bisect already uses. This way, Meerkat is able to build and boot many more kernel commits while not gaining an unfair advantage over Syz-bisect.

# 7 Dataset

## 7.1 Dataset Requirements

We initially sampled and tested 200 bugs found by Syzbot that meet the following requirements.

**Requirement 1.** The bug must have at least one PoC. This follows directly from Syz-bisect's PoC requirement, and is necessary for any dynamic bisection attempt.

**Requirement 2.** We require that the bug have been found in mainline Linux since this is where we will perform bisection. Auxiliary repositories such as Net and Linux-next are often

```
net/smc: Fix slab-out-of-bounds issue in
    fallback

syzbot reported a slab-out-of-bounds...

Reported-by: syzbot...
Fixes: 341adeec9ada ("net/smc: ...")
Link: ...
Signed-off-by: ...
```

Figure 5: An example of a `Fixes` tag in a patch [4].

reset or rebased, leaving many dangling commits. Syz-bisect has failed to find anchor commits on auxilliary repositories in the past. To avoid this issue, we perform our bisection evaluation on mainline only for both Meerkat and Syz-bisect.

**Requirement 3.** We require that the developers have marked a BiC for use as our ground truth and to compare Syz-bisect and Meerkat against. We discuss the ground truth in §7.2.

**Requirement 4.** The bug must have been found in 2022 or later. Syz-bisect has undergone major changes since its inception in 2019. In order to ensure our results are consistent with Syz-bisect in its current form, we consider bugs that have been found and tested recently.

We find about 400 bugs from Syzbot that meet all these requirements, and *randomly selected 200 of them to perform our evaluation on*. We chose to evaluate only a subset of the bugs due to time/resource limitations. Our ablation study requires that we perform bisection 5 times on each bug, each with different tools, and further manually analyze the results. Even evaluating 200 bugs combines for over 90,700 CPU hours of work, often running 4 to 5 instances of bisection in parallel.

After analyzing the 200 bugs, we discover that 50 do not reproduce for any of the tested tools. Since we are unable to gather any concrete data points for these bugs, we remove them from the study and continue with the remaining 150 that reproduced for at least one of our tools.

## 7.2 Ground Truth

In order to compare Meerkat and Syz-bisect, we put in considerable effort to ensure the ground truth is correct. We first consider the developer-marked BiC in each of our bugs' patches. In the patch for a bug, developers can use the `Fixes` tag to name a commit hash that the patch intends to fix, as shown in Figure 5. We take this hash, but do not immediately trust it without investigation. For each of the initial 200 bugs in our dataset, we determined whether the reported BiC was correct, and then replaced the BiC with the commit that actually introduced the bug. Indeed, we find the labeled BiC to be incorrect for 13 bugs as shown in Table 1. Our full corrections to the ground truth will be included as we open-source Meerkat to support future work.

| Verdict | Count |
|---|---|
| Incorrect | 13 |
| Correct | 187 |

Table 1: Results of the analysis of each bug's BiC

We find several reasons that BiCs were incorrectly reported. One reason is that the reported BiC only added the assertion that detects the bug. Logically, adding only the assertion cannot itself introduce the bug, so the ground truth must be incorrect in these cases. Often, we find that Syz-bisect misleads developers by reaching the commit which added the assertion. Syz-bisect is unable to reproduce the bug farther back, and the developer may not fully check the suspected BiC.

Another reason is that the developer attributes the introduction of a bug to the commit that added the related feature (*i.e.,* the initial commit for a module). In these cases, the buggy code may not have existed yet, thus the ground truth is incorrect. Finally, "fix bisection" sometimes mislabels bug patches. Fix bisection is an automatic process by which Syzbot finds unlabeled patches for bugs that no longer reproduce on mainline. However, fix bisection uses the same infrastructure as, and therefore suffers from the same issues as, Syz-bisect. If the result is not checked properly, the fixes tag in the suspected patch can be incorrectly marked as the BiC for a bug that it has nothing to do with.

## 7.3 Choice of 10 Minute Timeout

In our evaluation, we tested for each bug for 10 minutes, which we justify through a brief experiment. Meerkat reproduced and reached some bisection result, correct or not, for 149 bugs using the 10-minute timeout. If Meerkat is only allocated 5 minutes, we see that only 129 bugs are bisected, which is a substantial downgrade. To see if longer testing would improve Meerkat's results, we also ran Meerkat with a 20 minutes timeout. However, no additional bugs were successfully bisected during this test. Additionally, the extra time could increase the total runtime by as much as 40%. As a result, we reason that 10 minutes is a reasonable timeout for our evaluation.

# 8 Evaluation

We evaluate Meerkat in comparison to the baseline Syz-bisect based on the number of times each tool successfully reached the BiC. Additionally, we carry out an ablation study to better understand which design aspects contributed the most to Meerkat's performance increase. We then perform an in-depth manual analysis on each of the bugs in our dataset to determine why Meerkat stopped short of the BiC in some cases.

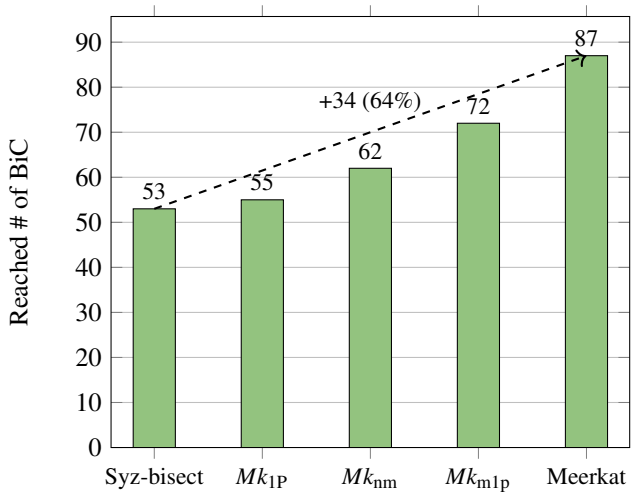


Figure 6: The number of times Syz-bisect and each Meerkat iteration reached the BiC.

## 8.1 Experimental Setup

All of our tests were run on two servers, each of which have two Intel Xeon E5-2697 v4 CPUs at 2.3 GHz, and 128 GB of memory. Each test uses 8 VMs with 2 cores and 4 GB memory, regardless of whether mutation or PoC testing was used.

## 8.2 Ablation Study

In order to better understand which of Meerkat's components contributed the most to its performance increase, we carried out an ablation study, which we lay out below.

First, we consider Syz-bisect to be the baseline for our test, which we run on the same hardware and with the same resource allocation as Meerkat. $Mk_{1P}$ (Meerkat 1 PoC) only uses a single PoC, the same one used by Syz-bisect, and only runs it in a loop to try to reproduce the bug. Its main advantage over Syz-bisect is that it has intra-report deduplication enabled. The next step up is $Mk_{nm}$ (Meerkat no-mutation), which uses all available PoCs found by Syzbot, cycling through each in turn. Essentially, $Mk_{nm}$ begins at the PoC-only phase as described in §5.4. $Mk_{m1p}$ (Meerkat mutate 1 PoC) is the first tool to utilize mutation during bisection, but only uses 1 PoC, same as $Mk_{1P}$. It will help to isolate the performance gain from mutation rather than multiple PoCs. Finally, $Mk$ is the fully realized form of Meerkat, which uses all of the above improvements plus PoC mutation. Figure 6 summarizes the results of our study, showing the number of times each tool reached the BiC.

$Mk_{1P}$ is able to reach the BiC 55 times, only twice more than Syz-bisect, despite Syz-bisect being misled by unrelated bugs in 27 cases. Interestingly, this means that deduplication alone is not enough to improve a bisection tool's ability to reach and stop at the BiC. Consider the cases where Syz-bisect would fail due to a lack of deduplication. Either it stops short of the BiC because no crash was observed, or it goes past the BiC because an unrelated bug was found. In the case where Syz-bisect stops short, no crash was observed so deduplication clearly cannot solve the issue. In the case where Syz-bisect traces another bug farther back, deduplication is still unlikely to help solve the issue. There are few cases where both an unrelated bug and the bug being bisected are observed on the same commits where deduplication can help identify the true BiC. Instead, $Mk_{1P}$ often stops short of the BiC in these cases as the unrelated bug prevents it from finding the bug of interest. We do note that using deduplication helps bring the bisection result closer to the BiC, but fails to precisely identify it. Despite this, deduplication becomes vital when mutating PoCs. The newly mutated test cases find unrelated bugs quite often, and these bugs need to be weeded out. So, while deduplication alone does not improve bisection results, it is still a critical part of Meerkat.

$Mk_{nm}$ adds the remaining PoCs. The bugs in our dataset have 3.2 PoCs on average, a median of 2, and a maximum of 43. Using these, $Mk_{nm}$ reaches the BiC 62 times, 7 more times than $Mk_{1P}$. In these 7 cases, it is clear that the chosen PoC was broken, yet Syzbot had already generated another that could be used to reach the BiC. In each of these cases, the PoC chosen by Syz-bisect and $Mk_{1P}$ is entirely unable to reproduce the bug, and simply using a different PoC allows bisection to reach the BiC.

$Mk_{m1p}$ is an alternate step up from $Mk_{1P}$, adding the PoC mutation, but still only using 1 PoC. Even with just the singular PoC, $Mk_{m1p}$ is able to outperform both of the previous iterations, reaching the BiC 72 times. This demonstrates that for some bugs, the number of PoCs Syzkaller can find on a single commit is irrelevant when they are broken by a changing kernel. We observe mutation helping in several ways. The first and most obvious is that it fixes broken PoCs, adapting them to older commits. A more subtle way is that mutation allows the bisection tool to detour around other bugs that may trigger earlier in the same execution path. In cases such as this, mutation may be able to find a different path to the same bug, something the PoC alone cannot do. Thus, PoC mutation is the dominant factor in Meerkat's performance increase.

Lastly, $Mk$ uses PoC mutation on all available PoCs to reach the BiC 87 times, improving $Mk_{m1p}$ in 15 cases, $Mk_{nm}$ in 25 cases, and Syz-bisect in 34 cases. While PoC mutation is the dominant contributor, no single component is all of Meerkat. The combination of all components allows Meerkat to greatly outperform any singular component.

> Out of 150 bugs, Meerkat was able to reach the BiC 87 times, giving it a success rate of 58%, and improving over Syz-bisect by 64% (1.64×).

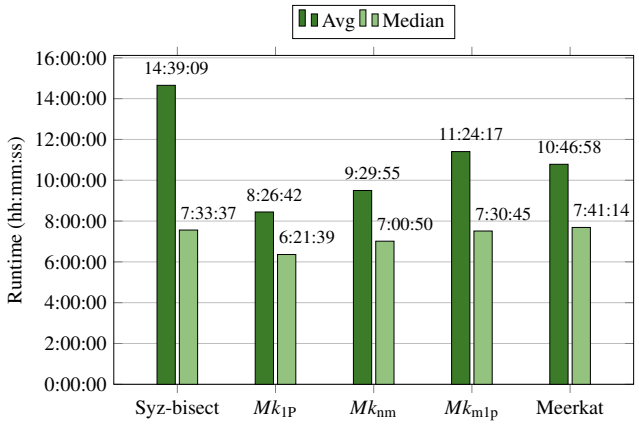


Figure 7: Average and median runtimes for Syz-bisect and each Meerkat iteration.

## 8.3 Runtime Evaluation

One of the key requirements of Meerkat is that it does not substantially increase the runtime of bisection. Syz-bisect is tightly bound by a 12 hour time limit, so Meerkat's performance increase is only significant if it remains largely within the same time limit. While analyzing the runtime, we only consider bisection attempts where the bug was found at the anchor commit, as failing to find the bug here terminates bisection and only takes about 25 minutes. Including these attempts would improperly favor tools that failed to find bugs more often. Additionally, we ran each tool for a maximum of 48 hours in order to gather more accurate data, and observe fewer timeouts.

Figure 7 shows a breakdown of the average and median runtimes for each bisection tool. In the average case, Meerkat vastly outperforms Syz-bisect, reducing the runtime from over 14 hours and 39 minutes to 10 hours and 46 minutes. However, the majority of this is due to the fact the Syz-bisect timed out in 19 cases, whereas Meerkat only timed out 7 times. Removing these cases for a more direct comparison of the tools gives Syz-bisect an average runtime of 8 hours and 22 minutes, and Meerkat an average of 8 hours and 56 minutes. Here, Meerkat does slightly worse, but we argue that this is well within the acceptable runtime. The median runtime for each tool tells a similar story. Here, Syz-bisect and Meerkat only differ by 8 minutes, with Syz-bisect performing marginally better.

Finally, we return to the runtime allotted to Syz-bisect by Syzbot, 12 hours, and confirm that Meerkat stays within this time limit. We find that Meerkat completes 117 bisection attempts that continued past the anchor commit within 12 hours. In fact, this is a vast improvement over Syz-bisect, which only completed 89 bisection attempts in the same time.

Notably, Meerkat's additional algorithm steps do not drastically increase its runtime. As mentioned in §5.4, the PoC mutation of phase 1 is usually enough to reach the BiC, and when the PoCs are required, Meerkat just needs to find the one that works. Cycling through the remaining PoCs only increases the runtime by tens of minutes depending on the number of PoCs available.

In a real-world deployment scenario, we envision Meerkat as a drop-in replacement for Syz-bisect. Based on the results shown here, Meerkat can be deployed with the same resource allocation as Syz-bisect, and outperforms it with little to no overhead.

## 8.4 Importance of Phase 2

Phase 2 (§5.4) is a key component of *Mk*, however it has a non-negligible impact on its runtime, seeing that each PoC must be considered at least once. If phase 2 were unnecessary, Meerkat might be able to shave tens of minutes off its runtime, but this is not the case. Out of the 150 bugs in our dataset, the phase 2 improved Meerkat's bisection result in 32 cases. Additionally, phase 2 was responsible for 12 out of the 87 cases where Meerkat reached the BiC, often using a PoC identified during phase 1. Thus, phase 2's importance cannot be overlooked, even when considering the runtime of Meerkat.

## 8.5 Manual Analysis of Meerkat's Results

Meerkat accurately identified the BiC in 87 cases, but this leaves many bugs for which the BiC was not reached. Here we perform an in-depth manual analysis of the cases where Meerkat stopped short of the BiC in order to better understand the issues faced by dynamic bisection tools.

### 8.5.1 Methodology

For each bug, we consider the complete bisection log and commit reported by Meerkat. By following the execution path and data flow leading up to the bug, we can determine how the reported commit affected the reproducibility of the bug. Additionally, we build and rerun either the PoCs or mutation engine at specific commits. This helps us to determine if the bug is truly unreproducible past a commit, or just difficult to reproduce. The full analysis took 300 person-hours.

### 8.5.2 Symptoms and Causes of Failed Bisection

We observe the following symptoms in bisection attempts that failed to identify the BiC and explore their underlying causes. Figure 8 shows the ratio of each of the results reached by Meerkat.

**Symptom 1:** No crashes are observed before a particular commit. In these cases, the kernel does not crash and no other bugs are observed that could be interfering. The most basic cause of this is that the code surrounding the crash site has changed in such a way that the PoC can no longer be used to reach the buggy code either by executing or mutating it. However, this can also signify that the bug detector has

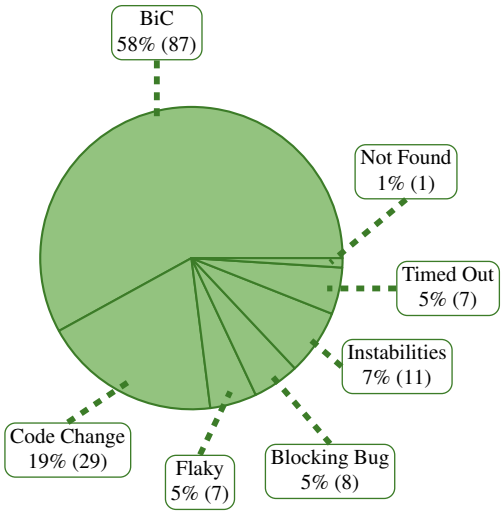


Figure 8: The results reached by Meerkat

changed. Several of Meerkat's bisection attempts traced bugs back to the commit that introduced their detectors, such as `WARN_ON()` assertions. This highlights an interesting dilemma for dynamic bisection: the bug detectors are built into the kernel and are thus directly tied to kernel versions. Based on our study, code changes are the single most prominent issue facing dynamic bisection, resulting in 29 failed bisection attempts.

**Symptom 2:** The bisection tool observably struggles to reproduce the bug. When the PoC does not reproduce the bug consistently, we refer to it as flaky. In these cases, Syz-bisect may only reproduce the bug in 1 out of 8 attempts, or in the case of Meerkat, the mutation engine takes longer to reproduce the bug. We note that a bug may be flaky only when testing commits that are older than a particular commit, making the flakiness more difficult to spot in the bisection log. When a bug is flaky, it becomes easy for Meerkat to not reproduce that bug on a commit, leading to false negatives. Meerkat was unable to reach the BiC 7 times due to flaky bugs.

For one severe case, Syz-bisect was the only tool able to reproduce the bug, and it did so exactly once. In subsequent tests, Syz-bisect was unable to reproduce the bug like it had the first time. In addition, none of our tools were able to reproduce the bug either. In this outstandingly lucky case, Syz-bisect was the only tool to reproduce the bug. This accounts for Meerkat's single "Not Found" result.

**Symptom 3:** Other crashes are observed that prevent the bisection tool from progressing. This can occur in two ways: either there is a bug earlier in the execution path that is preventing the PoC from reaching the true crash site, or there is another bug that triggers when the kernel is under stress. The former is known as a blocking bug and has been explored in recent work [26]. We refer to the latter as an instability: that is, the kernel is too unstable to reasonably test for the presence of a bug. In either case, the bisection tool reports a false negative as it is unable to reproduce the true bug. Meerkat was prevented from reaching the BiC 8 times due to blocking bugs and 11 times due to kernel instabilities.

**Symptom 4:** The kernel fails to boot such that dynamic analysis cannot take place on a commit. In these cases, the error can be detected as a failure to secure shell into the test VM.The bisection search algorithm has a provision for dealing with such commits by skipping them and choosing another commit. However, this can become problematic. Skipping too many commits greatly increases the runtime of bisection as each commit still needs to be built, and can potentially hide the BiC. We note that building the kernel takes the majority of the runtime already. Overall, Meerkat timed out at 48 hours 7 times as a result of boot errors.

### 8.5.3 Open Problems

The symptoms laid out in §8.5.2 present clear open problems which are not easily solved.

For symptom 1, code changes leading up to the crash site are difficult to work around. While it may be possible that another path exists to the same bug, finding such a path is difficult. Previous work [57] has already demonstrated the difficulty of identifying new entry points (*i.e.,* syscalls). Furthermore, backporting directed fuzzing targets becomes increasingly difficult as code is refactored.

For cases where the detector is changed, we are faced with the problem of reintroducing the detector. Code is refactored as the kernel gets older, creating the problem of where to put the detector. Additionally, the data flow and objects themselves may change. If the data changes, how does the assertion change? In the case of large-scale and precise detectors like KASAN, older kernels simply may not support such instrumentations and checks. These issues make it practically impossible to backport the bug detectors.

Symptom 2 is difficult to solve for dynamic analysis. Several prior works have focused on expanding race windows for concurrency bugs [42, 62]. However, these solutions focus on a bug as it exists in a single commit. They face the same problem of backporting complex targets and may need to change their thread interleavings as well. We note that a more basic approach of testing for longer would become prohibitively expensive in the real world.

Symptoms 3 and 4 share an underlying issue that there exists some other bug in the kernel that is preventing dynamic analysis. Unfortunately, patching these bugs is not always straight-forward. For known kernel instabilities, Syz-bisect and Meerkat already use a list of patches to help mitigate them, but adding more takes great engineering effort. Furthermore, blocking bugs can be difficult to identify especially without inter-report deduplication. Patching a blocking bug incorrectly may accidentally patch the bug being bisected, thus interfering with the bisection result.

Based on the nature of the problems presented above, we argue that Meerkat is approaching the practical limits of what is possible for dynamic bisection.

## 9 Discussion

Here we briefly discuss the effects inter-report deduplication would have had on Meerkat's bisection results.

### 9.1 Inter-Report Deduplication

Inter-report deduplication is an expensive process by which two bug manifestations are determined to be the same bug. Previous works [49, 68] have used patches as the ground truth for this deduplication, arguing that if two bugs share a patch, they must also share a root cause. Since this information is not available at the time of bisection, we cannot include it in our study. However, we do carry out an analysis of the initial 200 bugs in our dataset and find 14 groups of bugs with duplication comprised of 32 bug manifestations.

Interestingly, the lack of inter-report deduplication did not appear to be a hindrance to Meerkat. For all but 3 of the groups, Meerkat reached the same result for each manifestation within the group. This hints that inter-report deduplication provides an overall minimal gain to bisection accuracy. However, given the proper deduplication, a bisection tool could reduce the number of times bisection has to be run on bug manifestations of the same group.

Additionally, studies using patches as deduplication should be careful to ensure the patches are correct. We find among our 14 bug groups, 2 that are incorrectly labeled as duplicates.

## 10 Related Work

Our strategy of PoC mutation is similar to SyzRetrospector's strategy of Focused Fuzzing [26]. However, we have several key improvements. First, SyzRetrospector is focused primarily on determining whether Syzbot would have been able to reproduce a bug on a certain date. This means the tool builds Syzkaller based on the date in question. Since Meerkat only requires one version of Syzkaller, we are able to modify it to improve performance. Notably, we prevent Syzkaller from generating new test cases and only allow it to mutate the test cases already in the corpus. We also use a different method for restricting the available syscalls by taking advantage of a Syzkaller configuration rather than rebuilding the syscall descriptions for each test, thus improving reliability. We also maintain a stateful corpus between tests so the engine gets better at testing the buggy area as bisection proceeds. Finally, we identify and use new PoCs on the fly, whereas SyzRetrospector is not concerned with these.

VulScope [29] uses a directed fuzzer to migrate PoCs to different versions of user-space programs. It was tested on a large number of project releases and boasts over 90% accuracy in determining a bug's presence in a release. However, VulScope is intended for user-space programs and has not been tested on the Linux kernel, which is usually more difficult to fuzz as it often requires complex test cases and has bugs spanning multiple code paths, threads, and execution contexts.

SyzBridge [70] is another recent work that bridges the gap between the upstream Linux development repository and downstream OS releases such as Ubuntu. It adapts PoCs to the downstream environments by fixing environment preparation, loading modules, reducing background noise, *etc.* Meerkat differs by adapting PoCs through mutation to older kernel versions.

ARVO [48] is an evolving dataset that performs a form of bisection on its bugs. However, ARVO focuses on userspace programs, and only bisected a maximum of 14 commits compared to the tens of thousands that Meerkat parses. Additionally, ARVO only deduplicates bugs within its dataset based on patches, and not crash artifacts.

Dynamic analysis is not the only method available for determining the introducing commit of a bug. SymBisect [68] uses under-constrained symbolic execution to determine whether a bug is present in a kernel commit. Essentially, SymBisect checks for the presence of buggy logic at each commit rather than a crash. The main advantage here is that no execution is needed, and that the results are more accurate. While SymBisect greatly improves the occurrence of false negatives and positives, it is only applicable to a small subset of bugs, namely use-after-free and out-of-bounds bugs, it does not handle concurrency bugs, and it can only reason on the final syscall in a test case. Meerkat, in contrast, is generalizable to any bug found by Syzbot that has a reproducer, which can include concurrency bugs.

Popular bisection algorithms such as Git bisect [10] can be easily thrown off by a single mistake. One solution builds on Noisy Binary Search [22, 30, 40] to create a Probabilistic Bisection Algorithm [58]. Such algorithms assign confidence to each remaining commit as to whether the bug exists in that commit. While not deterministic, the algorithm is resilient against erroneous test results during bisection and gives it a chance to report the correct result despite the errors.

SZZ algorithms [18, 19, 21, 28] attempt to identify the BiC of a bug based on its patch. They operate under the assumption that the lines changed in order to fix a bug are the same ones that caused the bug to appear. Using the patch, it considers the commits that most recently changed those lines to be the bug inducing commits. However, this is not always the case. Moreover, since SZZ algorithms require a patch in order to work, this kind of analysis is not useful to developers as they work to patch a bug. Meerkat and other bisection algorithms identify the BiC based on a PoC and symptoms of the crash, making them much more useful during the patching process.

V0Finder [61] uses code-cloning techniques to find bugs

across projects that may share code. It is able to quickly identify affected projects in only 22 seconds, but requires a hefty setup period of up to 12 days. Similar to SZZ, this solution requires some knowledge of the root cause, often the patch. Thus, it will not be helpful to developers as they create a patch.

Fonte [16] is an interesting work that greatly reduces the search space for bisection. It uses static analysis to filter out unrelated commits based on the coverage gathered by running the PoC, style commits, and other no-op commits. Fonte is often able to reduce the search space down to only 40-80 commits, compared to normal bisection, which often searches through thousands to even tens of thousands. We see Fonte as a valuable engine that can be used to drive weighted bisection, perhaps under the hood of Meerkat.

Sleuth [60], SyzScope [71], and GREBE [44] share a common goal of uncovering unknown impacts of bugs, optimized for cases with known viable PoCs. Meerkat, on the other hand, is concerned with determining whether the bug is findable, and possibly generating a new PoC.

The Levenshtein distance used in this work has been used by previous work deduplicating stack traces [17, 49], as well as many other applications where deduplication of data is required [23, 64, 67].

Two works, Bowknots [56] and Talos [36], provide methods of working around bugs that have not been patched yet. Considering the instability of older kernel commits, it is likely that bisection would benefit from integration with these kinds of tools.

Fuzz testing has seen a lot of work in recent years [20, 37, 39, 47]. Directed fuzzers have found use in targeting suspected buggy code, allowing a fuzzer to probe deeper than other strategies [2, 25, 32, 43, 50, 69]. Hybrid fuzzers utilize a mix of symbolic and dynamic execution in order to solve for constraints leading to more code paths [27, 34, 53, 55, 59, 63, 65]. Some fuzzers have taken on the challenge of concurrency bugs by controlling how threads are interleaved [33, 38, 62]. Each of these fuzzers is fine-tuned to perform a specific task. We chose to optimize Syzkaller as it is readily available and would make integration back into Syzbot much easier.

## 11 Conclusion

In this work, we set out to push the practical limits of dynamic bisection within the time and resource constraints that make Syz-bisect useful. We identified three challenges through which we could improve bisection results and designed our solution Meerkat. Meerkat is the first dynamic bisection tool to apply scalable PoC mutation and measure its effects and limitations on a large dataset with an ablation study. In our analysis of 150 bugs, we found that Meerkat was able to reach the BiC in 87 cases, a 64% improvement over Syz-bisect. Furthermore, we performed an in-depth analysis across 300 person-hours of the cases where Meerkat failed to reach the BiC in order to better understand the challenges facing dynamic bisection tools. We additionally studied and repaired the ground truth for 13 bugs in our dataset through considerable manual effort, thus promoting future research.

## Acknowledgements

This work was supported in part by NSF Awards #2247880, #2247881, United States Air Force and DARPA under Agreement No. FA8750-24-2-0002. The authors thank the anonymous reviewers for their insightful feedback.

## Ethical Considerations

We structure our ethical considerations as a stakeholder-based ethical analysis, first identifying the groups impacted by our research and then weighing ethical principles with respect to those groups.

**Stakeholder Identification.** We identify several impacted groups which we list here. (1) Google and the Syzkaller Team: The Syzkaller Team is responsible for developing and maintaining Syzbot and by extension, Syz-bisect, which in turn is owned and by Google and run on Google's servers. (2) The Linux Foundation and Linux Developers: The Linux kernel is the target of both Syz-bisect and Meerkat. One of the goals of Meerkat is to improve the developer's ability to patch bugs quickly and effectively. (3) Linux End-Users: Meerkat intends to improve the security of Linux, which end-users rely on. (4) The Research Team: Those who carried out and contributed to this work. (5) The Fuzzing Community: Including all those who carry out research on and actively use fuzzing techniques.

**Beneficence.** Considering Google and the Syzkaller Team (1), should they choose to implement Meerkat, we expect no additional operational overhead. Meerkat is designed to use the same time and resource constraints placed upon Syz-bisect. We do expect some deployment overhead regarding internal review of the code base and fully incorporating Meerkat into Syzbot. Both the Syzkaller Team (1) and Linux Developers (2) stand to benefit from Meerkat as it is able to provide more accurate bisection results. However, (2) should be careful to inspect those results before implementing a patch based on them. We believe disclosure of bugs to The Linux Foundation (2) is not required as each bug in our dataset is already publicly available, known to the Linux developers, and patched. End-users (4) of Linux and Linux-based services stand to benefit from patched bugs, making the platforms they use more secure. We will discuss the possibility of the use of Meerkat by malicious actors in "Potential Harms" below. Finally, the Fuzzing Community (5) stands to benefit from our curated dataset and enumeration of open problems, in addition to our implementation of an improved bisection tool.

**Respect for Persons.** This work contains no human subjects or human-based experiments. No personal data was collected.

We respect the fuzzing community (5) by properly citing previous work. All contributors (4) to this work are properly listed as authors.

**Justice.** This work provides functional PoCs for vulnerabilities on historical kernel versions, which could potentially be used by attackers to exploit unpatched kernels in the wild. This is not necessarily a novel capability for attackers, but may make their objectives easier in some cases. We argue that the assistance provided to kernel developers (2) in patching bugs quickly, ideally leading to backports in downstream builds, outweighs the possible gain for attackers.

**Respect for Law and Public Interest.** This work and the experiments herein abide by the applicable research policies and laws. The dynamic analysis is carried out on quarantined VMs and are not capable of interfering with live networks. Our dataset consists of known and patched bugs, so disclosure is not applicable.

**Potential Harms.** Meerkat provides the ability to adapt PoCs for known vulnerabilities to historical, vulnerable versions of the Linux kernel. This ability could be used by adversaries to create attack chains for unpatched kernels in the wild. Such exploits could cause financial or data loss to server owners and clients (3). However, Meerkat's assistance in bug patching eventually leads to preventing attacks which use these vulnerabilities. As stated above (in "Justice"), we argue that the benefits gained from Meerkat far outweigh the potential use by attackers.

**Decision to Publish.** Weighing the ethical analysis above, we decided to move forward in pursuing and publishing this work. We believe that Meerkat provides a clear benefit in the field of bug triage and future research, and is unlikely to result in harm to the stakeholders.

## Open Science

Meerkat, its source code, dataset, and results of our evaluation are made available and can be found on Zenodo: https://doi.org/10.5281/zenodo.20317981, and Github: https://github.com/trusslab/meerkat.

## References


[1] memory leak in rds_send_probe. https://groups.google.com/g/syzkaller-bugs/c/RYROqSKy2qY/m/risLySvtEQAJ, 2019. Accessed: 2026-02-05.

[2] american fuzzy lop. https://github.com/google/AFL, 2021. Accessed: 2026-02-05.

[3] KASAN: use-after-free Read in inet_bind2_bucket_find. https://syzkaller.appspot.com/bug?id=190828444408403a087cb8fd424e338f69ef6de6, 2022. Accessed: 2026-02-05.

[4] net/smc: Fix slab-out-of-bounds issue in fallback. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=0558226cebee256aa3f8ec0cc5a800a10bf120a6, 2022. Accessed: 2026-02-05.

[5] WARNING in inet_csk_get_port. https://syzkaller.appspot.com/bug?id=ea15ed20f874608f0b5e8f90c9d1cda69f1e34be, 2022. Accessed: 2026-02-05.

[6] general protection fault in hci_register_dev. https://syzkaller.appspot.com/bug?extid=410a8e33c6a740b40d51, 2023. Accessed: 2026-02-05.

[7] WARNING in nilfs_segctor_do_construct. https://syzkaller.appspot.com/bug?extid=33494cd0df2ec2931851, 2023. Accessed: 2026-02-05.

[8] Bisection. https://github.com/google/syzkaller/blob/master/docs/syzbot.md#bisection, 2025. Accessed: 2026-02-05.

[9] cloc. https://github.com/AlDanial/cloc, 2025. Accessed: 2026-02-05.

[10] Git Bisect. https://git-scm.com/docs/git-bisect, 2025. Accessed: 2026-02-05.

[11] Invalid bugs. https://syzkaller.appspot.com/upstream/invalid, 2025. Accessed: 2026-02-05.

[12] Reproduce a bug with syzbot's downloadable assets. https://github.com/google/syzkaller/blob/master/docs/syzbot_assets.md, 2025. Accessed: 2026-02-05.

[13] Syz-bisect. https://github.com/google/syzkaller/blob/master/docs/bisect.md, 2025. Accessed: 2026-02-05.

[14] Syzbot. https://syzkaller.appspot.com/upstream, 2025. Accessed: 2026-02-05.

[15] Syzkaller. https://github.com/google/syzkaller, 2025. Accessed: 2026-02-05.

[16] Gabin An, Jingun Hong, Naryeong Kim, and Shin Yoo. Fonte: Finding Bug Inducing Commits from Failures. In *Proc. IEEE/ACM ICSE*, 2023.

[17] Abhishek Arya and Cris Neckar. Fuzzing for Security. https://blog.chromium.org/2012/04/fuzzing-for-security.html, 2012. Accessed: 2026-02-05.

[18] Muhammad Asaduzzaman, Michael C. Bullock, Chanchal K. Roy, and Kevin A. Schneider. Bug Introducing Changes: A Case Study with Android. In *Proc. IEEE MSR*, 2012.

[19] Lingfeng Bao, Xin Xia, Ahmed E. Hassan, and Xiaohu Yang. V-SZZ: Automatic Identification of Version Ranges Affected by CVE Vulnerabilities. In *Proc. IEEE ICSE*, 2022.

[20] Nils Bars, Moritz Schloegel, Tobias Scharnowski, Nico Schiller, and Thorsten Holz. Fuzztruction: Using Fault Injection-based Fuzzing to Leverage Implicit Domain Knowledge. In *Proc. USENIX Security*, 2023.

[21] Gabriele Bavota, Bernardino De Carluccio, Andrea De Lucia, Massimiliano Di Penta, Rocco Oliveto, and Orazio Strollo. When does a Refactoring Induce Bugs? An Empirical Study. In *Proc. IEEE International Working Conference on Source Code Analysis and Manipulation*, 2012.

[22] Michael Ben-Or and Avinatan Hassidim. The Bayesian Learner is Optimal for Noisy Binary Search (and Pretty Good for Quantum as Well). In *Proc. IEEE FOCS*, 2008.

[23] Bonnie Berger, Michael S. Waterman, and Yun William Yu. Levenshtein Distance, Sequence Comparison and Biological Database Search. *IEEE Transactions on Information Theory*, June 2021.

[24] Ranjita Bhagwan, Rahul Kumar, Chandra Sekhar Maddila, and Adithya Abraham Philip. Orca: Differential Bug Localization in Large-Scale Services. In *Proc. USENIX OSDI*, 2018.

[25] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed Greybox Fuzzing. In *Proc. ACM SIGSAC CCS*, 2017.

[26] Joseph Bursey, Ardalan Amiri Sani, and Zhiyun Qian. SyzRetrospector: A Large-Scale Retrospective Study of Syzbot. In *Proc. IEEE RAID*, 2025.

[27] Peng Chen and Hao Chen. Angora: Efficient Fuzzing by Principled Search. In *Proc. IEEE S&P*, 2018.

[28] Daniel Alencar Da Costa, Shane McIntosh, Weiyi Shang, Uirá Kulesza, Roberta Coelho, and Ahmed E. Hassan. A Framework for Evaluating the Results of the SZZ Approach for Identifying Bug-Introducing Changes. *IEEE Transactions on Software Engineering*, October 2016.

[29] Jiarun Dai, Yuan Zhang, Hailong Xu, Haiming Lyu, Zicheng Wu, Xinyu Xing, and Min Yang. Facilitating Vulnerability Assessment through PoC Migration. In *Proc. ACM SIGSAC CCS*, 2021.

[30] Dariusz Dereniowski, Aleksander Łukasiewicz, and Przemysław Uznański. An Efficient Noisy Binary Search in Graphs via Median Approximation. In *Proc. Springer IWOCA*, 2021.

[31] Dmitry Vyukov. Syzbot Bisection Analysis. https://lore.kernel.org/all/CACT4Y+Y3nN=nLEkHXLFcX7vxp_vs1JrD=8auJ3cX9we6TQHO+w@mail.gmail.com/T/#u, 2019. Accessed: 2026-02-05.

[32] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. AFL++: Combining Incremental Steps of Fuzzing Research. In *Proc. USENIX WOOT*, 2020.

[33] Pedro Fonseca, Rodrigo Rodrigues, and Björn B. Brandenburg. SKI: Exposing Kernel Concurrency Bugs through Systematic Schedule Exploration. In *Proc. USENIX OSDI*, 2014.

[34] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed Automated Random Testing. In *Proc. ACM SIGPLAN PLDI*, 2005.

[35] Kangzheng Gu, Yuan Zhang, Jiajun Cao, Xin Tan, and Min Yang. How Well Industry-Level Cause Bisection Works in Real-World: A Study on Linux Kernel. In *Proc. ACM FSE*, 2024.

[36] Zhen Huang, Mariana DAngelo, Dhaval Miyani, and David Lie. Talos: Neutralizing Vulnerabilities with Security Workarounds for Rapid Response. In *Proc. IEEE S&P*, 2016.

[37] Patrick Jauernig, Domagoj Jakobovic, Stjepan Picek, Emmanuel Stapf, and Ahmad-Reza Sadeghi. DARWIN: Survival of the Fittest Fuzzing Mutators. In *Proc. Internet Society NDSS*, 2023.

[38] Dae R Jeong, Kyungtae Kim, Basavesh Shivakumar, Byoungyoung Lee, and Insik Shin. Razzer: Finding Kernel Race Bugs through Fuzzing. In *Proc. IEEE S&P*, 2019.

[39] Dae R. Jeong, Byoungyoung Lee, Insik Shin, and Youngjin Kwon. SegFuzz: Segmentizing Thread Interleaving to Discover Kernel Concurrency Bugs through Fuzzing. In *Proc. IEEE S&P*, 2023.

[40] Richard M Karp and Robert Kleinberg. Noisy binary search and its applications. In *Proc. ACM-SIAM SODA*, 2007.

[41] Kyungtae Kim, Dae R Jeong, Chung Hwan Kim, Yeongjin Jang, Insik Shin, and Byoungyoung Lee. HFL: Hybrid Fuzzing on the Linux Kernel. In *Proc. Internet Society NDSS*, 2020.

[42] Yoochan Lee, Changwoo Min, and Byoungyoung Lee. ExpRace: Exploiting Kernel Races through Raising Interrupts. In *Proc. USENIX Security*, 2021.

[43] Caroline Lemieux and Koushik Sen. FairFuzz: A Targeted Mutation Strategy for Increasing Greybox Fuzz Testing Coverage. In *Proc. ACM/IEEE ASE*, 2018.

[44] Zhenpeng Lin, Yueqi Chen, Yuhang Wu, Dongliang Mu, Chensheng Yu, Xinyu Xing, and Kang Li. GREBE: Unveiling Exploitation Potential for Linux Kernel Bugs. In *Proc. IEEE S&P*, 2022.

[45] Chenyang Lyu, Shouling Ji, Chao Zhang, Yuwei Li, Wei-Han Lee, Yu Song, and Raheem Beyah. MOPT: Optimized Mutation Scheduling for Fuzzers. In *Proc. USENIX Security*, 2019.

[46] Dominik Maier, Benedikt Radtke, and Bastian Harren. Unicorefuzz: On the Viability of Emulation for Kernelspace Fuzzing. In *Proc. USENIX WOOT*, 2019.

[47] Alessandro Mantovani, Andrea Fioraldi, and Davide Balzarotti. Fuzzing with Data Dependency Information. In *Proc. IEEE EuroS&P*, 2022.

[48] Xiang Mei, Pulkit Singh Singaria, Jordi Del Castillo, Haoran Xi, Abdelouahab, Benchikh, Tiffany Bao, Ruoyu Wang, Yan Shoshitaishvili, Adam Doupé, Hammond Pearce, and Brendan Dolan-Gavitt. ARVO: Atlas of Reproducible Vulnerabilities for Open Source Software. *arXiv preprint arXiv:2408.02153*, 2024.

[49] Dongliang Mu, Yuhang Wu, Yueqi Chen, Zhenpeng Lin, Chensheng Yu, Xinyu Xing, and Gang Wang. An In-depth Analysis of Duplicated Linux Kernel Bug Reports. In *Proc. Internet Society NDSS*, 2022.

[50] Sebastian Österlund, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. ParmeSan: Sanitizer-guided Greybox Fuzzing. In *Proc. USENIX Security*, 2020.

[51] Ripon Saha and Milos Gligoric. Selective Bisection Debugging. In *Proc. Springer ETAPS/FASE*, 2017.

[52] Kostya Serebryany. OSS-Fuzz - Google's continuous fuzzing service for open source software. In *Proc. USENIX Security*, 2017.

[53] Dongdong She, Kexin Pei, Dave Epstein, Junfeng Yang, Baishakhi Ray, and Suman Jana. NEUZZ: Efficient Fuzzing with Neural Program Smoothing. In *Proc. IEEE S&P*, 2019.

[54] Youkun Shi, Yuan Zhang, Tianhan Luo, Xiangyu Mao, and Min Yang. Precise (Un)Affected Version Analysis for Web Vulnerabilities. In *Proc. IEEE/ACM ASE*, 2022.

[55] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting Fuzzing Through Selective Symbolic Execution. In *Proc. Internet Society NDSS*, 2016.

[56] Seyed Mohammadjavad Seyed Talebi, Zhihao Yao, Ardalan Amiri Sani, Zhiyun Qian, and Daniel Austin. Undo Workarounds for Kernel Bugs. In *Proc. USENIX Security*, 2021.

[57] Xin Tan, Yuan Zhang, Jiadong Lu, Xin Xiong, Zhuang Liu, and Min Yang. SyzDirect: Directed Greybox Fuzzing for Linux Kernel. In *Proc. ACM SIGSAC CCS*, 2023.

[58] Rolf Waeber, Peter I. Frazier, and Shane G. Henderson. Bisection Search with Noisy Responses. *SIAM Journal on Control and Optimization*, January 2013.

[59] Daimeng Wang, Zheng Zhang, Hang Zhang, Zhiyun Qian, Srikanth V. Krishnamurthy, and Nael Abu-Ghazaleh. SyzVegas: Beating Kernel Fuzzing Odds with Reinforcement Learning. In *Proc. USENIX Security*, 2021.

[60] Haolai Wei, Liwei Chen, Zhijie Zhang, Gang Shi, and Dan Meng. Sleuth: A Switchable Dual-Mode Fuzzer to Investigate Bug Impacts Following a Single PoC. In *Proc. ACM SIGSOFT ISSTA*, 2024.

[61] Seunghoon Woo, Dongwook Lee, Sunghan Park, Heejo Lee, and Sven Dietrich. V0Finder: Discovering the Correct Origin of Publicly Reported Software Vulnerabilities. In *Proc. USENIX Security*, 2021.

[62] Meng Xu, Sanidhya Kashyap, Hanqing Zhao, and Taesoo Kim. Krace: Data Race Fuzzing for Kernel File Systems. In *Proc. IEEE S&P*, 2020.

[63] Tai Yue, Pengfei Wang, Yong Tang, Enze Wang, Bo Yu, Kai Lu, and Xu Zhou. EcoFuzz: Adaptive Energy-Saving Greybox Fuzzing as a Variant of the Adversarial Multi-Armed Bandit. In *Proc. USENIX Security*, 2020.

[64] Li Yujian and Liu Bo. A Normalized Levenshtein Distance Metric. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, June 2007.

[65] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. QSYM : A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing. In *Proc. USENIX Security*, 2018.

[66] Jia-Ming Zhang, Zhan-Qi Cui, Xiang Chen, Huan-Huan Wu, Li-Wei Zheng, and Jian-Bin Liu. DeltaFuzz: Historical Version Information Guided Fuzz Testing. *Springer J. of Computer Science and Technology*, February 2022.

[67] Shengnan Zhang, Yan Hu, and Guangrong Bian. Research on String Similarity Algorithm based on Levenshtein Distance. In *Proc. IEEE IAEAC*, 2017.

[68] Zheng Zhang, Yu Hao, Weiteng Chen, Xiaochen Zou, Xingyu Li, Haonan Li, Yizhuo Zhai, and Billy Lau. SymBisect: Accurate Bisection for Fuzzer-Exposed Vulnerabilities. In *Proc. USENIX Security*, 2024.

[69] Xiaogang Zhu and Marcel Böhme. Regression Greybox Fuzzing. In *Proc. ACM SIGSAC CCS*, 2021.

[70] Xiaochen Zou, Yu Hao, Zheng Zhang, Juefei Pu, Weiteng Chen, and Zhiyun Qian. SyzBridge: Bridging the Gap in Exploitability Assessment of Linux Kernel Bugs in the Linux Ecosystem. In *Proc. Internet Society NDSS*, 2024.

[71] Xiaochen Zou, Guoren Li, Weiteng Chen, Hang Zhang, and Zhiyun Qian. SyzScope: Revealing High-Risk Security Impacts of Fuzzer-Exposed Bugs in Linux kernel. In *Proc. USENIX Security*, 2022.