Improving Application Crash Consistency with Symbolic Execution and Fuzzing

Madeleine Barowsky, Milan Bhandari, and Mridu Nanda

1 Abstract

Data loss and corruption after a crash is a pernicious and often undetected problem. Prior work in finding crash consistency bugs exercises a system with idiomatic usage of the API and commands that fully succeed. We hypothesize that error handling paths and unexpectedly ordered commands have been under-explored for crash consistency vulnerabilities. We use a grammar-based fuzzer and symbolic execution with KLEE to search for diverse bugs missed by standard workloads. We evaluate our hypothesis on three applications: Git and two intentionally vulnerable simple utilities (copy and sort). We find that automatically generated workloads, combined with the crash consistency bug finder ALICE [10], perform well at finding vulnerabilities in each application. In particular, our custom grammar-based fuzzer finds more vulnerabilities than both KLEE and hand-crafted baseline developer workloads. Our success with finding bugs via a fuzzer emphasizes the importance of unusual API usage as a source crash consistency bugs. Our results also suggest that some tools are better suited than others at generating workloads depending on the type of applications. We advance the study of application crash-consistency bug finding by a novel approach to workload generation.

2 Introduction

When a device loses power, any information that has not been saved to persistent memory is immediately lost. After power is restored, the operating system and applications face the task of recovering internal state so they can run as seamlessly and similarly as possible to a point prior to the crash. At best, inconsistency can be repaired without user intervention, even if some data from immediately before the crash was lost. At worst, files or configuration may be deeply corrupted, perhaps even invisibly. This is caused when developers do not properly account for the atomicity and ordering guarantees of the underlying file system and disk. Such bugs are ideally fixed before the software is deployed, but they can be difficult to detect.

File systems use a variety of well-studied techniques to provide crash consistency for content and metadata. Some of these techniques include logging, copy-on-write, and soft updates. Many modern applications are built atop these file systems, and therefore get file-system-level crash consistency guarantees for free. However, an application must initiate its own application-level crash consistency protocol to ensure that user-level data structures are consistent post-crash. This sequence, known as an update protocol, invokes system calls that update the underlying files and directories in a recoverable way. For example, the default setting of SQLite uses an update protocol called rollback journaling to maintain transactional atomicity.

Unfortunately, applications often implement update protocols incorrectly. Incorrect update protocols are primarily a result of a mismatch of expectations: the application developer falsely assumes certain invariants about the underlying file system, causing the update protocol to behave in an unexpected manner. For example, an update protocol might assume writes are persisted in program order; however, most modern file systems re-order writes to avoid high latency seek times. The multitude of possible application states and the non-deterministic nature of application state post-crash makes it even more difficult to write a complete and correct update protocol.

Finding crash consistency bugs has been an active area of research for many years. Methods typically fall into one of two categories: formal verification, or the record-and-replay approach. While formal verification is exhaustive, it is often burdensome to instrument and run. In contrast the record-and-replay approach relies on heuristics, while still being able to uncover flaws in heavily used code.

At a high-level, the record-and-replay approach consists of three components: (1) an application's workload, (2) known good application state, and (3) a procedure to verify crash consistency. The record-and-replay tool will simulate the application's workload until an arbitrary point, which represents a crash. Then the tool tries to determine whether the application can recover to a known good state or whether it is left in an irreparable corrupted state. The procedure to verify crash-consistency varies based on the tool. For example, some tools use the application's update protocol [10], while others define a metric that captures the difference between the application's crash state and the application's known good state [4].

Most variants of record-and-replay tools require the application developer to provide the workload. As a result, the workloads are over simplified, and often trivial. Therefore, we hypothesize that record-and-replay tools are fundamentally limited by the diversity of application workloads. We predict that record-and-replay tools will be able to find a greater number of application-level crash consistency bugs if given a diverse and robust application workload.

In this project we generate high-coverage application workloads using fuzzing and symbolic execution. Fuzzing involves generating a set of random inputs for a program and

running the program with those random inputs to observe program crashes or misbehavior until some time budget is exhausted. The fuzzing technique is advantageous because it automatically generates test cases and does not require access to source code. Furthermore, fuzzing does not require control of the entire execution environment. For example, a program does not have to be run atop a special hypervisor, OS, or interpreter to be fuzzed [7].

Instead of running the code on randomly constructed inputs, as done in fuzzing, symbolic execution runs the program with symbolic values. These values replace the concrete inputs to the program, and are manipulated by the program's logic. For example, if a program has an `if` clause, the symbolic executor will maintain a set of constraints that capture the symbolic values for both branches. When a path terminates or hits a bug, the symbolic executor generates a test case by solving the current path condition for concrete values.

We predict that diverse application workloads will play a critical role in drawing conclusions from the results of crash-consistency tools. In particular, these results may cast new insight on determining the crash consistency of a file system. These results may also help application developers write more robust, crash consistent applications, that are independent of the underlying file system. The rest of the paper is structured as follows. In Section 3, we recap related work, including existing crash consistency bug-finders, and high-test coverage techniques. In Section 4 we describe the architecture of our system, followed by details of the design in Section 5. Then, in Section 6 we give a brief overview of the implementation of our system. We present our evaluation and discussion in Sections 7 and 8. Finally, we conclude in Section 9.

3 Related Work

In this section we give an overview of existing crash-consistency bug finders and tools to generate high test coverage.

3.1 Crash-Consistency Bug Finders

Crash-consistency bug finders fall largely into two categories, formal verification and record-and-replay. Formal verification methods for crash consistency provide concrete guarantees under certain assumptions, but can be costly to implement. In contrast, the record-and-replay paradigm relies more on heuristics. As a high level, this model requires a user to define a workload of commands to exercise the system and, in some cases, a script to check the state after a real or simulated crash to determine whether the program state is consistent.

3.1.1 EXPLODE. An early work in this space, EXPLODE (2006), adapts model checking to find filesystem crash consistency vulnerabilities. EXPLODE exhaustively explores all crash states but needs the user to enumerate each consistency property and ways in which it might be tested by the application API. It also runs on a live kernel, thus requiring a VM [14].

The state exploration loop of EXPLODE, placed atop the stubbed out methods for each storage system layer, has a flavor of symbolic execution. This is unsurprising as one of the authors later co-created KLEE, a symbolic executor we discuss later.

3.1.2 Torturing Databases. A 2014 paper by Zheng et al. modifies a iSCSI driver to record disk I/O, simulates crashes, and attempts to restart the application on the resulting image [16]. They hand-design five workload and checker scripts that are tailored to their understanding of database weak points. This work effectively stresses database applications and provides trustworthy evidence and root cause for any resulting bugs; however, it is limited to database applications and requires significant programmer effort to port to new use cases or filesystems.

3.1.3 ALICE. One of the first general application-level crash consistency bug finders was introduced in 2014 by Pillai et al. [10]. Their system, ALICE, falls under the record-and-replay regime and requires that someone familiar with the application semantics write workload and checker scripts. ALICE has been successfully used in industry to find real crash consistency bugs [11]. For each filesystem they wish to test, the user must also implement an abstract persistent model (APM). During execution of the workload script, ALICE records a trace of the program's system calls and parses them into intermediate micro-operations and then, disk operations. Using the constrains defined by the APM, ALICE explores achievable crash states at various points in workload execution and checks for inconsistency with the user-supplied checker script. It is not feasible for ALICE to simulate all possible crash states, but a user can provide some direction for the granularity and depth of exploration.

ALICE reports discovered vulnerabilities as either dynamic or static:

- *A dynamic vulnerability* is a system call that results in an inconsistency. ALICE finds dynamic vulnerabilities by incrementally simulating each system call, and running the outputted state through the user-defined application checker.
- *A static vulnerability* is a source code line that results in an inconsistency. Multiple dynamic vulnerabilities can correspond to a single static vulnerability, e.g. if they originated from the same line that was executed multiple times.

Note that the utility of ALICE relies on the correctness and thoroughness of the user-supplied scripts. The workload must exercise many code paths in the application and the checker script must understand which filesystem states after

a crash are recoverable. We believe that the need for hand-written workload and checker scripts is a major limitation of ALICE in its current state and attempt to address the task of automatic workload generation.

3.1.4 C³. The 2016 paper "Crash Consistency Validation Made Easy" from Jiang et al. shared our goal of further automating record-and-replay tools. Instead of artificial workloads developed for the purpose of exercising the program, this work proposes *test amplification* to leverage existing functional tests for the software. To expose faulty developer assumptions, test amplification runs tests and inserts `sync` points after certain system calls whose consistency properties may have been misunderstood (e.g., `ftruncate` and `open`). These `sync` calls are benign during execution but offer a variety of snapshots of application state after a simulated crash. For automated checking, C^3 uses (estimated) edit distance as a proxy for whether the crash state is "close enough" to a consistent state as to be recoverable.

To our knowledge, C^3 is the first approach towards automating both parts of the pipeline. However, the test amplification process may not find bugs that are related to long chains of system calls or other ways in which an application might be stressed (heavy workload or unusual API usage). Although C^3 reports finding fourteen bugs, upon investigating further, we found that only about five were considered real and fixable bugs; the others mostly resulted in updated documentation or were closed as being too theoretical and unlikely to occur in practice. Thus, we are skeptical that test amplification is as effective as promised. Because C^3 is modular, our contribution of more complex workloads can be evaluated under their crash simulation and automatic checking.

3.1.5 CrashMonkey. Another automated record-and-replay tool came in 2018 with a system called CrashMonkey [8]. Tasked with generating scripts to test file system consistency, CrashMonkey uses heuristic bounds on length, parameters, and depth to generate workload code. Although they note that "there seem to be no fuzzing techniques to expose crash-consistency bugs," the approach they developed of combining random system calls and inserting necessary dependencies is essentially a fuzzer.

After replaying recorded I/O calls and truncating at a persistence point to simulate a crash, the program attempts to mount the resulting disk image. If successful, and if all files from the workload can be properly read and written, CrashMonkey considers the state to be consistent.

We are heavily inspired by the workload fuzzer from CrashMonkey and borrow it to generate application-level workloads. This requires additional insights and customization because dependencies and proper API usage are specific to each application.

3.2 Diverse Workloads

There are many existing automated testing frameworks that help ensure the correctness of code. These testing tools come in three flavors: linting, fuzzing, and symbolic execution. The primary goal of these frameworks is to create a set of test cases with high code coverage, so that almost all of a program's execution paths are tested. Linting is a static analysis technique that involves inspecting source code for common bugs and stylistic errors. For example, a linter might detect an uninitialized variable [7]. However, linting is not used to find more complex logic bugs, so we will not utilize this technique in our project. Instead, we harness fuzzing and symbolic execution to generate high-coverage, diverse test cases that uncover application-level crash consistency bugs. For the former, we create our own grammar-based fuzzer. For the latter, we will make use of KLEE.

3.3 Grammar-Based Fuzzer

The basic idea behind a fuzzer is to generate random inputs for a program, and to observe program behavior after running on these random inputs. Often, random input exposes weaknesses in error handling or parsing and the program will throw an exception or display incorrect output. Fuzzers are advantageous because (1) they automatically generate test cases, (2) they do not require the developer to control the entire execution environment, and (3) they do not require a special hypervisor, OS, or interpreter [7]. Unfortunately, truly random fuzzing will have poor code coverage since most programs expect input in a certain order and format and fail quickly if these constraints are not satisfied. For example, arbitrary strings created to fuzz a web server will very rarely correspond to valid HTTP requests. Furthermore, randomly generating inputs can result in repeatedly exploring the same code paths, leaving other paths unexplored.

Syntax-aware fuzzers can help increase code coverage. This class of fuzzers utilizes application-specific knowledge to generate realistic commands that exercise the application sufficiently. A syntax-aware fuzzing engine can be driven by either input templates or input grammars [7]. In this project, we utilize a grammar to generate application workloads.

In general, a grammar defines the structure of the language and describes how to iteratively construct valid statements. It consists of a *start symbol*, *nonterminal symbols*, *terminal symbols*, and rules mapping nonterminals onto other symbols. A terminal symbol cannot be further expanded; it can be thought of as a concrete value. A nonterminal symbol is an intermediate state within an expression and must eventually be expanded into terminal symbols.

There are many types of grammars for both natural and artificial languages. For example, the Backus-Naur Form (BNF) grammar for arithmetic expressions looks like [3]:

$$e ::= x | n | e_1 + e_2 | e_1 \times e_2 | x := e_1; e_2$$

where e, e_1, and e_2 are expressions, n is an integer, and x is a variable. The BNF grammar concisely gives a recursive definition of simple mathematical statements using only a few terms. A valid expression expansion of the above grammar could look like: 1 + 2 × 3 + x. Of course, the order of operations in the above grammar is ambiguous. One way to solve this is to modify the grammar by including parenthesis.

We use a BNF grammar for our grammar-based fuzzer as described in Section 5.

3.4 KLEE

In contrast to fuzzing, symbolic execution is a more exhaustive test generation technique. In symbolic execution, all concrete program inputs are replaced with symbolic values. As the program runs, the symbolic executor accumulates a set of constraints that manipulate the symbolic values, and represent the various execution paths of the program. Once the program terminates, the symbolic executor "solves" the constraints to concretize the program inputs and generate a test case.

KLEE is one example of a symbolic execution engine. At a high level, KLEE functions as a hybrid between an operating system for symbolic processes and an interpreter [2]. KLEE refers tot eh representation of the symbolic process a state. KLEE compiles an application's source code to LLVM, which is a high-level assembly language. As the program executes, KLEE instruments the LLVM bytecode to add constraints. For example, to instrument the LLVM add instruction

```
%dst = add i32 %src0, %src1
```

KLEE will retrieve the addends from the `%src0` and `%src1`, and write a new expression `Add(%src0, %src1)` to the `%dst` register.

Unfortunately, even a simple program can generate thousands of concurrent states. To ensure an efficient symbolic execution engine, KLEE includes optimizations to compactly represent states and to efficiently complete queries. For example, to reduce per-state memory requirements, KLEE implements copy-on-write on an object-level granularity, instead of using the traditional page-level granularity [2]. An example of a query optimization includes *implied value concretization.* This optimization decreases the number of constraints that KLEE must solve for by substituting a constant for the symbolic value, whenever the constant value can be inferred. For example, KLEE will omit a constraint such as `x + 1 = 10`, since it can instead store the concrete value of `x` (in this case, $x = 9$). Users can also limit the number of states that KLEE explores by setting a time limit for KLEE's exploration, or by setting the `-only-output-states-covering-new` flag. This flag ensures that KLEE only generates test cases that cover distinct parts of code.

In order to test real-world programs, KLEE enables users to pass in symbolic arguments and files with the `-sym-arg` and `-sym-files` flags, respectively. Using these building blocks, previous work showed that KLEE successfully generated in-depth coverage all 89 for COREUTILS utilities [2].

4 Architecture

In this section we give a high-level overview of our architecture. We hypothesized that increased workload diversity would result in crash-consistency bugs missed by standard workloads. We classify diversity by three metrics: code coverage, length of commands, and exploring error handling paths/unexpected API usage. To this end, our architecture is comprised of three main components:

- *Applications:* We aim to find crash-consistency bugs in these programs. We will use open source programs, like Git, and hand-rolled programs like copy and sort.
- *Workload generators:* We will use three techniques for generating workloads. The baseline comparison will be a handwritten workload, written by an average developer. We will compare the bugs found in this workload against workloads generated by KLEE, and by our custom grammar-based fuzzer. We chose KLEE because extensive previous work has shown that this tool can successfully generate high coverage tests for applications. We chose to make a custom grammar-based fuzzer because this technique enabled easy stress-testing for unexpected API usage. The fuzzer also allowed us to test the effect of chaining commands together, and allowed us to test complex open source programs (e.g., Git), which were difficult to instrument in KLEE. We describe the KLEE implementation challenges in 6.
- *Crash Consistency tools:* We will feed the generated workloads into record-and-replay crash consistency bug finders. In this paper, we focus on the ALICE tool; however, we believe that these results could be replicated on different modular crash-consistency bug finding tools, like C3.

5 Design

We designed several components of the architecture from scratch. In particular, we designed hand-rolled applications and the grammar-based fuzzer. In the following sections, we give the design details for both.

5.1 Hand-rolled Applications

We hypothesized that more diverse workloads would result in a greater number of application-level crash consistency bugs. A key challenge in testing this hypothesis involved picking the right set of applications to generate workloads for. We found that "real world" applications, like Git, can be difficult to instrument with KLEE (as described in Section 6). Therefore, to provide a fair comparison among all our workload generation techniques, we wrote hand-rolled applications.

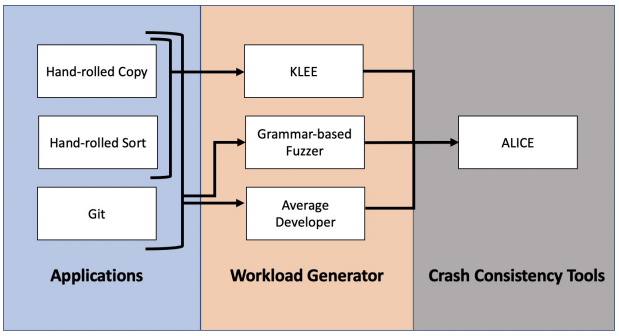

Figure 1. The workflow for our experiments. We test three applications: a hand-rolled copy program, a hand-rolled sort program, and Git. We generate workloads three ways: with KLEE, with the custom grammar-based fuzzer, and with an average developer. The fuzzer and devleoper generate workloads for all three programs; however, KLEE only generates workloads for the hand-rolled programs. We describe the implementation difficulties for KLEE instrumentation in Section 6. Finally, all workloads are input to ALICE, which produces crash consistency bugs.

We created two hand-rolled applications as a proof of concept: copy and sort. Each program includes a trapdoor, which if triggered, will result in a crash consistency bug. It would be difficult for an average developer to find such trapdoors because they are omitted from the program's documentation. However, we hope that the high coverage, diverse workloads generated by KLEE and the fuzzer, will exercise the application enough to reveal the crash inconsistencies. We give details about each of these applications in the following sections.

5.1.1 Copy. The copy program we wrote has a similar API to GNU Coreutil's `cp`. At a high level, the implementation will read N bytes from the input file into a buffer and write those bytes into the output file. After each write, the program will call `sync` the output file and print the number of bytes written. This routine runs in a loop until the entire input file has been copied. Thus, our copy guarantees that if the user sees a number x printed to the console, at least x bytes from the input file have been copied to the output. However, given certain byte sequences in the input file, our copy program will miss a call to sync. This breaks the above agreement where at least x bytes might not be in the output file.

While clearly a contrived example, motivation for this program came from KLEE's symbolic execution guarantees to generate test cases that explore all possible code-paths in a program; given regular testing, one might not encounter such a bug.

The hand-written checker script for `copy` codifies these properties. It will read the number of bytes the program says were copied until the crash from `stdout` and assert that the output file is at least x bytes long, and that the first x bytes of the output file match the input file.

5.1.2 Sort. We used inspiration from [4] while writing this program. Our sort program takes in one or two files as input. If only one file is passed as input, then `sort` does the sort in-place. Otherwise, the second file contains the results of the sort operation. At a high-level, `sort` converts the input file into a stream, and breaks the input text into lines. Then the program sorts the lines file's lines in in non-descending order. Finally `sort` writes the sorted output to a file.

The key behind `sort`'s trapdoor is parsing files as streams. By converting the input file to a stream, the file is truncated to empty. A crash at this point would would result in the files contents being lost permanently. The `sort` checker script verifies if the application is recoverable by checking that the lines in the output are written in sorted order. If the `sort` program was provided the same input and output file, then `sort` also asserts that no data was lost.

5.2 Grammar-Based Fuzzer

For each of the applications we evaluate, the grammar is slightly different. At a high-level, they look like

```
GRAMMAR = {
"<start>": ["<program>"],
"<program>": ["<command>;", "<command>; <program>"],
"<command>": # see below,
"<name>": ["<letter>", "<letter><name>"],
"<file>": # see below,
"<number>": ["<digit>", "<digit><number>"]
}
```

Note that we construct programs, numbers, and names (i.e., alphanumeric strings) recursively. The maximum number of nonterminals that we expand can be optionally specified but defaults to 100. We use the `simple_grammar_fuzzer` from [15] for the actual expansion.

5.2.1 <command>. Each `<command>` nonterminal corresponds to a single command given to a program. Thus, we must describe valid ways to use the API. We wrote a parser that takes docstrings for the program (in the format of output from `–help` or the like) and converts them into commands for the grammar. Our parser understands positional, optional, and mutually exclusive switches and exhaustively chooses combinations thereof. As an example, parsing the string

```
git reset [-q] [<commit>] [--] <name>
```

yields the following list

```
["git reset <name>",
 "git reset -- <name>",
 "git reset <commit> <name>",
 "git reset <commit> -- <name>",
 "git reset -q <name>",
 "git reset -q -- <name>",
 "git reset -q <commit> <name>",
```

"git reset -q <commit> -- <name>"].

Our hand-rolled sort and copy programs have simple APIs: each takes in an existing filename and a second string which may or may not be an existing filename, e.g., `sort <file> <name>` and `sort <file> <file>`.

For git commands, we must employ more involved strategies. Although we did not specify the entire git API, we also included some additional options (via nonterminals) and, when the fuzzer is run in an existing git repository, added all previous commit hashes to the `<commit>` nonterminal. We were then also able to add relative commit pointers such as `HEAD^n` for each of the n previous commits.

5.2.2 `<file>`. It was important for each of our applications that some real input files existed and had nonempty contents. As part of the grammar-based fuzzer, we created a random number of files with random filenames and contents. The names of these inputs could be included in the `<file>` nonterminal. We also added the absolute and relative filenames for all files recursively found in the directory from which the fuzzer was run. Including both absolute and relative paths was a strategy to improve code coverage and expose application bugs.

6 Implementation

To best compare with the results from ALICE, we used git 1.9.2 [1] because it came out earlier in the same year the ALICE paper was published. This required compiling git from source and linking against OpenSSL 1.0.1g [12] and zlib 1.2.8 [5]. Our test programs use a library called `argparse`, which requires `g++`$\geq$ 8.0.

For the KLEE work, we initially used the provided KLEE Docker container. However, this container came with a version of the KLEE binaries that compiled to use LLVM 6, and with different compilation flags than what we required for our usage. Thus, we had to build KLEE to work with LLVM 9.

We used the tests generated by KLEE as our workload scripts to be run by ALICE. Running the `klee` tool creates a directory with an encoded version of the generated tests. We used the `klee-replay` tool to then generate temporary directories containing the materialized versions of the files we need, as well as the invocation of the test command. We then created a python script that creates a vulnerability report by running ALICE on all of the generated tests. We used an Ubuntu 20.04 Docker container running on a quad-core machine with 16GB of memory for these tests.

For ALICE and fuzzing, we used an Ubuntu 18.04 AWS t2.micro instance. We checked consistency against the default APM (the least constrained filesystem which is allowed to reorder arbitrarily between `sync` calls) and APMs for CCFS and ext4-ordered implemented by two of the authors in [9]. Unless otherwise specified, we discuss the reported vulnerabilities by the default APM. This is what was done in the original ALICE paper.

Our checker scripts for copy and sort were already described in Section 5.1. Because the git consistency checker used in the original ALICE work was not available online, we had to reimplement our own, albeit much simplified. The original ALICE authors wrote a checker script that was approximately 500 lines of code and invoked "all recovery procedures [the authors] were aware of" [10, Sec 4.1]. In contrast, our checker script simply removes a `.lock` file (which might be generated during a crash), runs `git fsck –full`, and attempts to perform a few standard git operations like status, add, and commit to ensure the repository is working as intended.

All our harness, workload, checker, and grammar-based fuzzer scripts are made open source [6]. Available also are the reports from ALICE, the KLEE test cases, and necessary initial workload directories for reproducing the given ALICE output.

7 Evaluation

In general, we found that ALICE reported the most static vulnerabilities when run on workloads generated by the fuzzer, followed by KLEE, followed by the developer. After some investigation, we believe the fuzzer produced the most bugs because (1) it was able to create tests that involved large files, (2) it was able to create slightly malformed inputs, which stress tested the program's error handling, and (3) it was able to chain multiple smaller commands, which could expose procedures that fail to properly persist output upon completion (instead relying on other layers of the system).

	Developer	KLEE	Fuzzer
Copy	0	1S, 1D	6S, 11D
Sort	0	1S, 2D	3S, 3D
Git	9S, 10D	NA	10S, 13D

Table 1. Number of static (S) and dynamic (D) vulnerabilities found by ALICE per each program per each workload generation tool. All workloads were run atop the default APM.

Our developer workloads were very simple, executing a couple commands on short files without unusual bytes. We believe these are representative of an end user's potential misunderstanding of the ALICE tool. Because ALICE's exploration of potential crash states is similar to fuzzing, a programmer might not introduce sufficient randomness or bulk into their workloads to uncover crash consistency vulnerabilities.

7.1 Copy

The developer generated workload did not result in any vulnerabilities produced by ALICE. This result is reasonable;

the developer's file did not contain the magic byte sequence, which would cause the program to miss a sync call. In contrast, the workload generated by KLEE reported exactly one static vulnerability. This vulnerability corresponded to the file that contained the malicious byte sequence. In particular the output generated by ALICE, when run on this workload looked like:

```
Logical operations:
0   creat("out", parent=31717, mode='0600',
          inode=31719)
1   append("out", offset=0, count=5, inode=31719)
2   stdout("'\nA,out:5,'")

(Dynamic vulnerability) Ordering: Operation 1
needs to be persisted before 2
(Static vulnerability) Ordering: Operation
copy.cpp:44[main(int, char**)] needed before
B-/home/ubuntu/mset-alice/test-programs/bin/copy
:0x4010de[_start]
```

After re-compiling with debugging flags, we confirmed that the reported static vulnerability corresponds with the reported dynamic vulnerability. The ALICE tool correctly finds that the file append should be persisted before it is written to standard out.

The workload generated by the fuzzer had the most vulnerabilities. However, ALICE's output was somewhat confusing. In particular, ALICE outputted several variants of this static vulnerability:

```
(Static vulnerability) Ordering: Operation
B-/home/ubuntu/alice/example/test/test-programs/
bin/copy:0x5623299027ca[_start] needed before
B-/home/ubuntu/alice/example/test/test-programs/
bin/copy:0x560faeee17ca[_start]
```

This vulnerability was confusing because the _start function is the entry point of a C program which makes a call to main() [13]. None of our scripts should be able to modify this code. We verified this hypothesis by looking at the disassembled the binary copy file; the start code dissembled output only showed that `_start` made a call to `main`, without any other mention of our specific scripts.

In contrast, the dynamic vulnerabilities produced by ALICE running on the fuzzer workloads, were more understandable. Multiple dynamic dynamic vulnerabilities correspond to a single static vulnerability; therefore the number of dynamic vulnerabilities reported in Table 1 is an overestimate of the true number of bugs. Regardless, the dynamic vulnerabilities still give some insight into the crash consistency behavior of the application. In particular, the dynamic vulnerabilities report that the series of sort commands must be persisted in a particular order. This finding is interesting because these dynamic vulnerabilities disappear if each individual sort command in the chained workload is run as an individual workload on ALICE. Therefore, we conclude that short commands that are chained together in the fuzzer help discover a greater number of application-level crash consistency bugs.

7.2 Sort

Sort, on the other-hand, will generate crash-consistency warnings for any workload that tries to sort a file in-place while using the default APM. Specifically, the vulnerability discovered to be very similar to the one described in [4], where a file that is sorted in place will be truncated before the results are written. If a crash happens just after this truncate, the contents of the original file are lost.

As indicated by [4], certain versions of ext4 may perform small-file overwrites atomically, even if not specified in the file-system guarantees. ALICE, assuming a correct specified APM, would capture these vulnerabilities even with small files. Since ALICE relies on replaying operations following the constraints of a file system model rather than intercepting actual block-level operations such as C3, it is not constrained to the choices of a particular implementation.

In our developer-written workload scripts, we assumed the output files would be different from the inputs. Thus, ALICE found no vulnerabilities.

In order to generate test cases for `sort` via KLEE, we made use of symbolic files and arguments. Overall the results from ALICE determined that the append operation to the output file (which in this case is the truncated input file) must be persisted before we write the results to the console. This condition arises when the input file is equal to the output file because our checker asserts that the original file must be present, either in the original state or the sorted state, at any crash point. This condition is relaxed when the output file is different from the input file because there is no potential for data-loss when this is the case.

The fuzzed scripts reliably produce workloads that sort the same file in place. While ALICE outputted a slightly different number of static and dynamic vulnerabilities in comparison to KLEE, we believe that these actually are the same bugs. In particular, the bugs involve durability and ordering, which were the same bugs generated in the KLEE workload. Unfortunately, the durability related bug involved some arcane C++ code, which we did not have enough time to dig into.

7.3 Git

Git is a piece of popular version-control software that stores its history in a tree and metadata about the repository in a log. Much of open source software development happens in git repositories, and it is important for historical and functional purposes that these repositories do not silently lose data or become corrupted. Because of the importance and ubiquity of git, and because both ALICE and C3 tested their tools on git, we chose to evaluate on it as well.

We were unable to successfully use KLEE to generate git workloads because of the complexity of git's input requirements and codebase. Instead, we focused our efforts on grammar-based fuzzing (described in Section 5.2).

The original ALICE paper reported a total of 9 static crash consistency bugs in git under the default APM [10, Table 3(a)]. These include things like `git-log` and `git-commit` being unable to complete after a crash and more subtle errors like unreachable references (which could be resolved by an advanced user). The ALICE authors state that git never intended to provide crash guarantees and thus they did not report the vulnerabilities to the developers. C3 reported none and, noting the disparity with ALICE, described their trade-off ease-of-use in place of effectiveness [4]. With our grammar-based fuzzer that parses documentation strings, we attempt to balance both.

Even our basic git workload with a single commit reports multiple static and dynamic vulnerabilities. In Table 1, we replicate the 9 unique static vulnerabilities found by ALICE and find 10 unique dynamic vulnerabilities (the ALICE paper did not report dynamic vulnerabilities). To arrive at these counts, we considered only distinct pairs of start and end functions (i.e. a reported bug in which an operation from one function should occur before one in the other) and ignored vulnerabilities originating in library code. We looked into the git 1.9.2. source and attempted to determine whether some of these vulnerabilities were legitimate or had been fixed in the newest version of git, but both proved difficult tasks.

For the fuzzed git workloads, we find slightly more static and dynamic vulnerabilities. There is some overlap in reported bugs, a strong signal that those lines should be investigated for critical flaws. All but one of the static vulnerabilities in the basic workload were also found by the fuzzed workload. The remaining bug occurs during a commit, which the fuzzed workload did not generate. The fuzzed workload points to a number of possible bugs in `wt-status.c` and `create_symref`, whereas the basic workload does not.

8 Discussion

8.1 KLEE

KLEE uses symbolic execution to generate tests cases that explore all possible execution paths for a given program. However, KLEE operates at the program instruction level. This implies that KLEE will not be able to explore different symbolic inputs that may result in different underlying disk operation but flow through the same system-call. This is concerning in the context of varying write sizes. Previous work [10] has shown that file systems will guarantee different atomicity semantics when dealing with writes of different sizes. A further restriction of KLEE is that it requires the user to specify the maximum file size of each symbolic file.

An extension of KLEE to better support file-system related operations could include modeling the behavior of I/O related system calls such that different ranges of file-sizes would produce different execution paths.

9 Conclusion

We emphasize that unexpected API usage must not be overlooked as a source of crash consistency bugs. Although the application developer must understand how to use the program under test in order to write a thorough workload, there is significant value in including intentional errors, unusual command ordering and options, and unnatural input bytes. This work has shown the potential of grammar-based fuzzing and symbolic execution for automatically finding application crash consistency vulnerabilities. Grammar-based fuzzing has particular utility for more complex programs, and symbolic execution is best suited for environments with more narrow scope and additional computation power. We evaluated fuzzed workloads with ALICE on three applications and were able to uncover more bugs than prior work. We evaluated KLEE-generated workloads on two proof-of-concept binaries and found the vulnerabilities we expected. Our contributions are both modular (to different crash consistency tooling) and generalizable (to different programs) and we hope the community will build upon them for use in practice.

References

[1] 2014. git. https://github.com/git/git/releases/tag/v1.9.2. Commit: 0bc85ab.

[2] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. [n. d.]. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs.

[3] Steve Chong. 2013. Intro to semantics; Small-step semantics: CS152 Lecture 1.

[4] Yanyan Jiang, Haicheng Chen, Feng Qin, Chang Xu, Xiaoxing Ma, and Jian Lu. 2016. Crash Consistency Validation Made Easy. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Seattle, WA, USA) *(FSE 2016)*. Association for Computing Machinery, New York, NY, USA, 133–143. https://doi.org/10.1145/2950290.2950327

[5] Jean loup Gailly and Mark Adler. 2013. zlib. https://zlib.net/. Version 1.2.8.

[6] Mridu Nanda Madeleine Barowsky and Milan Bhandari. 2021. mset-alice. https://github.com/madeleine-b/mset-alice/.

[7] James Mickens. 2020. Testing: CS263 Lecture 5.

[8] Jayashree Mohan, Ashlie Martinez, Soujanya Ponnapalli, Pandian Raju, and Vijay Chidambaram. 2018. Finding Crash-Consistency Bugs with Bounded Black-Box Crash Testing. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 33–50. https://www.usenix.org/conference/osdi18/presentation/mohan

[9] Mridu Nanda and Milan Bhandari. 2021. Reproducing Crash Consistency Experiments with ALICE. Unpublished.

[10] Thanumalayan Sankaranarayana Pillai, Vijay Chidambaram, Ramnatthan Alagappan, Samer Al-Kiswany, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2014. All File Systems Are Not Created Equal: On the Complexity of Crafting Crash-Consistent Applications. In *Proceedings of the 11th USENIX Conference on Operating Systems*

Design and Implementation (Broomfield, CO) *(OSDI'14)*. USENIX Association, USA, 433–448.

[11] Peter Stace. 2017. Making Badger Crash Resilient with ALICE. https://dgraph.io/blog/post/alice/.

[12] The OpenSSL Project. 2014. OpenSSL. https://github.com/openssl/openssl/releases/tag/OpenSSL_1_0_1g. Commit: b2d951e.

[13] ulidtko. [n. d.]. https://stackoverflow.com/questions/15919356/c-program-start.

[14] Junfeng Yang, Can Sar, and Dawson Engler. 2006. EXPLODE: A Lightweight, General System for Finding Serious Storage System Errors. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation* (Seattle, Washington) *(OSDI '06)*. USENIX Association, USA, 131–146.

[15] Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler. 2019. Fuzzing with Grammars. In *The Fuzzing Book*. Saarland University. https://www.fuzzingbook.org/html/Grammars.html Retrieved 2019-12-21 16:38:57+01:00.

[16] Mai Zheng, Joseph Tucek, Dachuan Huang, Feng Qin, Mark Lillibridge, Elizabeth S. Yang, Bill W Zhao, and Shashank Singh. 2014. Torturing Databases for Fun and Profit. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. USENIX Association, Broomfield, CO, 449–464. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/zheng_mai