SibylFuzzer: Stateful Fuzzing for File Systems

by

Catherine Zuo

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Bachelor of Science in Computer Science and Engineering

[Master of Engineering] at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2017

Author ..... Signature redacted .....
Department of Electrical Engineering and Computer Science
September 28, 2016

Certified by .. Signature redacted .....
Tim Leek
Technical Staff, Lincoln Laboratory Cyber Systems Assessment Group
Thesis Supervisor

Certified by .. Signature redacted .....
Stelios Sidiroglou-Douskos
Research Scientist, Computer Science and Artificial Intelligence Laboratory
Thesis Supervisor

Accepted by .. Signature redacted .....
Dr. Christopher Terman
Chairman, Masters of Engineering Thesis Committee

SibylFuzzer: Stateful Fuzzing for File Systems

by

Catherine Zuo

Submitted to the Department of Electrical Engineering and Computer Science
on September 28, 2016, in partial fulfillment of the
requirements for the degree of
Bachelor of Science in Computer Science and Engineering

Abstract

Correct file system behavior is vital to developing robust higher-level software and applications. However, correctly and efficiently investigating the wide range of file system behavior makes testing file systems a difficult task. In this thesis, I designed and implemented SibylFuzzer, a stateful fuzzer for testing file system behavior. SibylFuzzer is based on SibylFS, a third-party system comprised of a model for acceptable file system behavior and a procedure for comparing real-life file system implementation behavior against that model. SibylFuzzer uses SibylFS in two ways: first, as a source of file system knowledge to produce in-depth and meaningful tests; second, as a correctness standard such that any disagreement with a real-life file system's behavior indicates a potential bug within the real-life file system. I implemented SibylFuzzer in OCaml and performed all tests on a Linux file system.

Thesis Supervisor: Tim Leek
Title: Technical Staff, Lincoln Laboratory Cyber Systems Assessment Group

Thesis Supervisor: Stelios Sidiroglou-Douskos
Title: Research Scientist, Computer Science and Artificial Intelligence Laboratory

Acknowledgments

I would like express my gratitude to my advisor Tim Leek for his insight and support during my work on this thesis. He was a constant source of knowledge from which I could share and receive feedback on questions, ideas and results throughout this project. Not only that, but he also helped to provide me a workplace that I could use to work on this thesis.

I would also like to express thanks to my co-advisor Stelios Sidiroglou-Douskos who also helped advise me, particularly early on. Stelios was a great source of ideas and feedback, and always challenged me to better understand and explain the details and nuances of my project.

Lastly, I want to thank the original authors of SibylFS, particularly Dr. Tom Ridge and Mr. David Sheets, who were very responsive and helpful whenever I asked questions about the SibylFS system.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Fuzzing

Software systems have bugs, and bugs indicate potential security vulnerabilities that could lead to disastrous effects, from leaking personal information to compromising national defense. These bugs can arise from faults in a system's design, or mistakes in individual implementations of that system. Bugs are often associated with unexpected inputs that are either ill-formed, or unaccounted for by the system's developers. It is therefore imperative to robustly test systems for their ability to handle malicious deviant inputs.

Fuzzing is a widely used strategy to find such bugs in software. It is a security testing approach that tests robustness by stressing the system with a wide range of inputs. Fuzzing is attractive because it is relatively straightforward, and can be automated. Given a **target** system or system implementation that one wishes to test, the fuzzing process is as follows:

1. *Generate* a set of system **inputs** using a **test generation strategy**. As each represents a test of the target system, these inputs are often referred to as fuzzed test inputs or test cases, and can be valid or invalid (ill-formed).

2. *Run* these inputs on the system, and observe the system behavior.

3. *Detect* incorrect observed behavior, which indicates a potential bug in the tar-

get. Detection can be very tricky; one good strategy is to have a specification or model of what the expected correct behavior should be. Any disagreements with observed behavior can then be assumed to be the fault of the target system and a potential bug.

Coverage is an important concept in fuzzer evaluation and discussion. Code-based coverage is a measure of how much of the implementation code was tested. Another measure of coverage is how much of the total possible input space has been fuzzed and tested. In general, the quality and complexity of the fuzzed test inputs determine how well each element in the targeted system is tested. Therefore, a fuzzer's test generation strategy is key to its effectiveness and success.

1.1.1 Dumb Fuzzing

Dumb fuzzing, also known as *mutation-based fuzzing* and *random fuzzing*, consists of blindly mutating valid inputs at random. One of the earliest fuzzers, created by Barton Miller in 1990, employed this dumb fuzzing strategy to test basic kernel and utility programs [1]. Miller had noticed that dial-up noise could scramble user-inputted commands and cause UNIX utilities to crash. He therefore used a dumb fuzzing strategy that generated completely random strings of characters to use as fuzzed test inputs. His experiments found crash-inducing bugs in over 20 different UNIX utility programs. In more recent years, Miller has used this same dumb fuzzing technique to fuzz tests inputs that successfully find bugs in Windows [2] and MacOS [3] applications.

Dumb fuzzing is relatively simple to implement and automate, and requires little to no knowledge of the system being tested. Its key weakness is that the random generator usually generates invalid inputs. Therefore if the system has ways of validating its input beforehand, many of the fuzzed test inputs would be immediately rejected. In addition, it is extremely unlikely for dumb fuzzing to find bugs that are only revealed after an extended, complex sequence of valid inputs to the system [4].

1.1.2 Smart Fuzzing

Smart Fuzzing is also known as *generation-based fuzzing* and incorporates knowledge of the target system, such as input format, in its test generation. Although smart fuzzers require more work to design and implement than dumb fuzzers, they can produce inputs that are only invalid in small parts; furthermore, such invalid inputs may be specifically crafted to test for "interesting" cases where input constraints are intentionally violated. Consider the Trinity fuzzer, which fuzzes Linux system calls by tapping into hard-coded specification knowledge of what kinds of input a Linux system call can take, then generates an input that is either valid, or "interesting" [5]. For example, suppose Trinity decided to fuzz a *read* system call for a file `foo.txt` and needs to generate the byte count argument. It can choose to use a valid value by using a number less than `foo.txt`'s current size. Alternatively, Trinity can use an "interesting" value such as 0 or a negative integer that corresponds to a *read* corner case.

The following are some specific types of smart fuzzing, characterized by the type of system knowledge used:

- *Grammar-based fuzzing* is used when all of the target system's valid inputs can be generated using a set of constraints, formally defined as a grammar. A fuzzer that has knowledge of this grammar would also have knowledge of how to generate a valid input versus an invalid input. In 1999 the PROTOS project used grammar-based fuzzing to test protocols in a black-box fashion, and produced test suites that successfully found bugs [6]. Grammar-based fuzzing's main drawback is the effort required to develop the grammar necessary for fuzzing [7].

- *Coverage-based fuzzing* takes place over several iterations; each iteration tries to fuzz test inputs that test sections of code that have not yet been tested. A famous example is American Fuzzy Lop (AFL) [8], which tries to increase code coverage by instrumenting the target system's source code so that it can detect when test cases reach previously-untested code elements. Another example is

the COMET (COverage Maximization using Taint) system [9]; COMET uses dynamic taint tracing to identify from which inputs the variables in conditional code derive, therefore narrowing the search over inputs necessary to explore new code branches.

- *Stateful fuzzing* requires the system to exhibit stateful behavior; the set of possible generated inputs is then constrained according to the current state. For example, SNOOZE is a stateful fuzzer targeting applications that use Session Initiation Protocol (SIP) [10], which involves exchanging several messages between two end-points. SIP's behavior is stateful, and can easily be modeled by having messages serve as state transitions, with each individual state representing a step in the evolution of a SIP "conversation" (with legal state transitions dictated by the SIP protocol specification). As long as it keeps track of the current "conversation" state and what messages are currently legal state transitions, SNOOZE can easily calculate long sequences of valid messages for the SIP protocol. There is one caveat to stateful fuzzing: the *state-space explosion problem.* As the number of target system variables and processes increase, the state space quickly becomes extremely large, making it difficult to reason about and use for fuzzing.

This thesis presents SibylFuzzer, an automated stateful fuzzer for file systems. It can generate over a dozen different system calls and includes a procedure to fuzz for multiple iterations. For its state model, SibylFuzzer uses the third-party SibylFS [11] file system state model. For POSIX, Linux, Mac OS X and FreeBSD file systems, the SibylFS file system model specifies the range of correct behavior for a given sequence of system calls. It can also be used as a test oracle to deem whether or not observed real-life file system behavior conforms to the model. SibylFuzzer therefore uses the SibylFS model both as a specification for generating system call inputs and as a standard for correctness to compare real-life file system behavior against.

1.2 Motivation for Fuzzing File Systems

I chose to use stateful fuzzing to test file systems for two reasons. Firstly, file systems control how data is stored and retrieved, and therefore play a critical role in today's computing world. File system bugs can have serious consequences, from loss of data [12] to severe interruptions and security vulnerabilities [13]; therefore one would expect file systems to be extremely reliable and robust. Because file system code is so extensive and the range of file system behavior is so large, file system tests should be extensive, systematic and automatically generated for efficiency. Fuzzing can generate tests which have these desired qualities.

Secondly, file systems are stateful, which makes them good targets for stateful fuzzing. One can think of the file system and its contents as a state, with system calls acting as state transitions. Incorrect file system implementation behavior corresponds to bad, or illegal states. Test inputs would then consist of sequences of system calls which transform the initial file system from one state to the next. Longer sequences correspond to more state transitions, resulting in a more complex file system state where in-depth bugs might be revealed.

1.3 Approach

I implemented SibylFuzzer in OCaml. First I created a preliminary dummy version of SibylFuzzer that could safely run alongside and correctly interact with the SibylFS model. Given a sequence of system calls, SibylFuzzer must be able to insert a (initially dummied-out) fuzzed system call at a chosen point in the sequence, then output the resulting sequence. I defined a fuzzing parameter to dictate the probability of where this chosen point may be.

Next I wrote methods to expose and extract file system content from the SibylFS state model to SibylFuzzer for use in system call generation. To do this I had to add code to the original SibylFS model implementation; however, my additions do not affect SibylFS's original functionality.

Then I defined fuzzing parameters that formed the backbone of the generation strategy for each system call SibylFuzzer had to fuzz. I studied the arguments for each system call to determine what the necessary fuzzing parameters were. I also defined a fuzzing parameter representing the probabilities of fuzzing each system call. Once all the fuzzing parameters were implemented into a generation strategy, SibylFuzzer was able to fuzz system calls.

Finally, I automated SibylFuzzer using a shell script so that it could run for multiple iterations, thereby increasing coverage. Since both the input and output to SibylFuzzer are sequences of system calls, I also used SibylFuzzer's output in one iteration as the input for the next. This feedback loop allows SibylFuzzer to potentially build up long subsequences of fuzzed system calls over several iterations, resulting in more complex file system states to explore.

1.4 Results

Testing revealed many model-implementation disagreements that were not covered by the original SibylFS test suite. After manually analyzing the test results and reaching out to the SibylFS authors, I discovered that most of the disagreements were caused by inaccuracies in the SibylFS model. Some of these inaccuracies were intentional and part of a SibylFS design trade-off for speed and reliability. Other model inaccuracies were unknown until now. Ultimately, I found evidence of only one real-life Linux file system bug.

1.5 Thesis Organization

Chapter two gives a simple overview of SibylFS, the third-party system SibylFuzzer uses to check file system behavior and generate tests. Chapter three describes the SibylFuzzer design and functionality. Chapter four goes over how I tested SibylFuzzer and what results were found. Chapter five concludes with what future improvements could be done to enhance SibylFuzzer coverage and analysis.

Chapter 2

Background

Understanding the variations in file system behavior is vital to developers at all levels; however, the large range of file system implementations available currently makes it difficult to standardize, describe, or test the behavior for any given file system. Testing file systems' behavior in particular faces the following challenges:

1. File system behavior is often written as complex abstractions,
2. There are multiple file systems in use by large sections of the population,
3. Different file systems can behave very differently in certain cases.

SibylFS was created by researchers from the University of Leicester, the University of Cambridge, and security company FireEye. Their goal was to develop a system that could characterize a broad envelope of behavior for multiple file systems that could easily be used to find bugs in file system implementations. To this end they developed the SibylFS model, a rigorous specification of file system behavior, which has the following important and relevant properties:

1. It can behave as a black-box test oracle: given an observed sequence of API system calls and return values from the implementation, SibylFS can compute whether it is allowed by the internal model or not.
2. SibylFS includes methods to easily procure afore-mentioned observed sequences of API system calls and return values.

3. It models the file system using a labelled state transition system where each label corresponds to either a system call or a return.

4. To cover a wider range of implementation standards, SibylFS's model is parameterized to check POSIX, Linux and OS X behavior.

5. Some system calls can give rise to multiple possible results, depending on the order in which checks are made in the file system implementation code. SibylFS incorporates this nondeterminism in its model by tracking multiple states.

6. SibylFS does *not* cover host crashes, race conditions, and timestamp bugs.

2.1 SibylFS Overview

To best understand SibylFS's architecture and the spirit of its functionality, think of the SibylFS project as a large system of components unified by one procedure, referred to throughout this thesis as the **SibylFS procedure**. The procedure takes as input lists of API system calls which it runs through an **executor** module that outputs the corresponding observed return values from running those calls on the real-life host file system. The **checker** module then compares the observed return values to the ones from the internal SibylFS model, and reveals any discrepancies. These discrepancies may indicate potential bugs with the real-life file system. However, another possibility is that the SibylFS model is incomplete in some way and therefore gave a false positive.

Figure 2-1 shows a flowchart of the SibylFS procedure, its different components and terminology. Two modules, the executor and the checker, form the core of SibylFS's functionality, and are mostly written in OCaml. The SibylFS model is written in Lem, a language used for writing semantic definitions, and automatically translated to OCaml. The executor and checker components are written in OCaml, and the entire SibylFS system is bound together with OCaml wrappers.

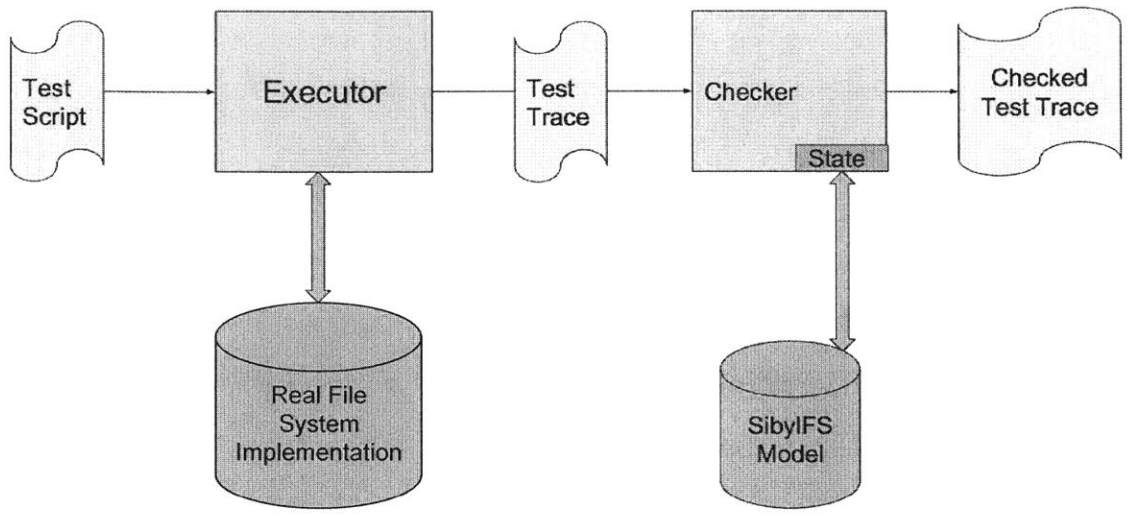

Figure 2-1: Overview of the SibylFS procedure. Input test scripts are given to the SibylFS executor, which runs them on the real-life file system implementation and records the return values on test traces. The SibylFS checker takes the test traces and checks the return values using the SibylFS file system model. The results of the check are written to the checked trace; model-implementation disagreements are written with detailed error messages.

2.2 Test Scripts

The input to the SibylFS process is a set of **test scripts**, which contain sequences of file system calls and a few SibylFS-invented shorthand commands. These shorthand commands include *dump*, which returns metadata such as timestamps and hard link count for every file and directory in the file systems, *open_close* which is analogous to the Linux *touch* command, and *add_user_to_group* which does exactly that. An example test script with two system calls is shown in Figure 2-2a.

The authors of SibylFS assembled a test suite consisting of 21,070 test scripts. The suite is organized by which system call or general topic a test script targets, resulting in fourteen sub-suites, as detailed in Table 2.1. The longest script contains 336 system calls, while the shortest has only three. Some test scripts were hand-written in order to test specific sequences; others were automatically generated using

```
@type script
mkdir "/newdir" 0o777
rename "/" "/"
```

(a) Test script.

```
@type trace
mkdir "/newdir" 0o777
RV_none

rename "/" "/"
EBUSY
```

(b) Test trace.

```
@type trace
mkdir "/newdir" 0o777
RV_none

rename "/" "/"
EBUSY
#
# Error:
#    !!! EXCEPT CHECKLIB DOES NOT AGREE !!!
#
# Error:
#    The spec permitted:
# RV_none
#
# Error: EBUSY
# Unexpected results: EBUSY
# Allowed are only: RV_none
#
# trace not accepted
```

(c) Checked trace.

Figure 2-2: Test script, test trace, and checked trace for a simple *mkdir* and *rename* sequence. Line numbers and other irrelevant artifacts have been edited out. The test script only contains the system calls. In the test trace, the real-life implementation's observed return values are below their corresponding system calls. In the checked trace the checker indicates that it has found a disagreement between the observed return value EBUSY from the real-life implementation, and the expected return value None from the SibylFS model, by printing out a detailed error message.

Sub-suite	Number of test scripts
chdir	1
file_descriptors	18
link	2503
lstat	50
mkdir	51
open	15361
permissions	165
readdir	2
rename	2506
rmdir	55
stat	50
symlink	10
truncate	250
unlink	52

Table 2.1: Number of test scripts per sub-suite in the original SibylFS test suite. Some sub-suites (*open, link, rename*) have many more test scripts than others. The sub-suites that only have a few test scripts may correspond to system calls that might have not been thoroughly tested.

random arguments.

The authors used this test suite to check the SibylFS model for completeness; any mistakes or holes in the model found were promptly fixed. However, this test suite does not guarantee completeness.

2.3 The SibylFS Executor

The executor is responsible for retrieving observed real-life file system behavior. It takes a set of test scripts and runs the system calls in a chroot jail, using OCaml's Unix interface and library. Running inside a chroot jail minimizes disruption to the host file system. This also ensures each test script looks like it begins executing in an empty file system, in accordance to the checker's initial state assumptions. However, as a side-effect certain system calls involving the root directory as one of its arguments behave oddly. For example, in a chroot jail the root directory link count is typically off-by-one compared to a non-chroot setup. This SibylFS-induced known deviancy

has significant implications for fuzzing results, as we will discuss in Section 4.1.

The output of the executor is a set of **test traces**, which interleave the system calls from the test scripts with corresponding observed real-life file system return values. As seen in Figure 2-2b, the real-life return values are inserted directly below their corresponding system call.

2.4 The SibylFS Checker

The core of SibylFS is the checking process, in which a set of test trace files from the executor is processed by the SibylFS checker for compliance with the internal model. The checker goes line by line through each trace file, which was written by the test executor to be checker-readable, "executing" the calls on its own internal state.

For each trace file, the checker initializes a *state set* that contains a state simulating an empty file system with only one process id. A state can have multiple ongoing process ids (PIDs), each with its own running status.

When the checker goes over a system call line in the trace file, it calculates a set of possible next states. Tracking a set of possible states instead of just one is necessary for many reasons. For example, there could be variance between file system behaviors based on storage layout which is too fine-grained to include in the SibylFS model. Or, a system call could throw several different errors; which error is thrown first could come down to concurrency or implementation code details, which the model does not expect to test but must tolerate.

When the checker encounters a return value line in the trace, it diffs the observed return value with each current possible state's saved expected return value. If there were no differences between observed and expected, the checker uses the observed return value as the transition for each of its possible states in the current state set. If there is a disagreement - signaling a potential bug in the file system - the checker chooses one model-acceptable return value at random to use as the transition, effectively ignoring the disagreement from the state set's perspective. This allows SibylFS to continue checking the trace even after reaching multiple potential errors. Using

only one of potentially many possible return values as a transition prevents the state set from exploding in size.

For each test trace the checker goes through it outputs a corresponding **checked trace** file. System calls and return values that agreed with the SibylFS model are copied from the original test trace. Non-conforming return values are marked with error messages containing the return values expected by SibylFS. For example, in Figure 2-2c the *rename* system call had an observed return value that did not match the SibylFS model's expected return value, as detailed in the highly visible error message.

Chapter 3

Design and Implementation

I implemented the SibylFuzzer's main body in the SibylFS checker, which was written in OCaml. Since my goal was to use the logical SibylFS model as a black box reference, I modified the original spec Makefile to directly recompile existing OCaml code rather than auto-generate from Lem. This change allowed me to maintain continuity between checker-level and model-level work, but also compromised the ease of integrating SibylFuzzer with future versions of SibylFS.

3.1 Fuzzer Overview

Broadly, SibylFuzzer's goal is to extract file system information from the ongoing saved state set as the SibylFS checker goes through the trace, then utilize this information to generate a new system call. To this end, SibylFuzzer acts as a add-on to the checker, whose core code is untouched. It will generate a new system call *if and only if* the checker does not find any disagreements. As the purpose of fuzzing is to find bugs, SibylFuzzer does not need to do any extra work trying to induce potential bugs on a test script that already has a potential bug.

Before the checker starts going through a test trace, SibylFuzzer first extracts the sequence of system calls. It then picks a location to insert the fuzzed system call, which could either be at the end of the sequence, or somewhere in the middle. In the latter case, SibylFuzzer truncates the trace at the corresponding location and hands it

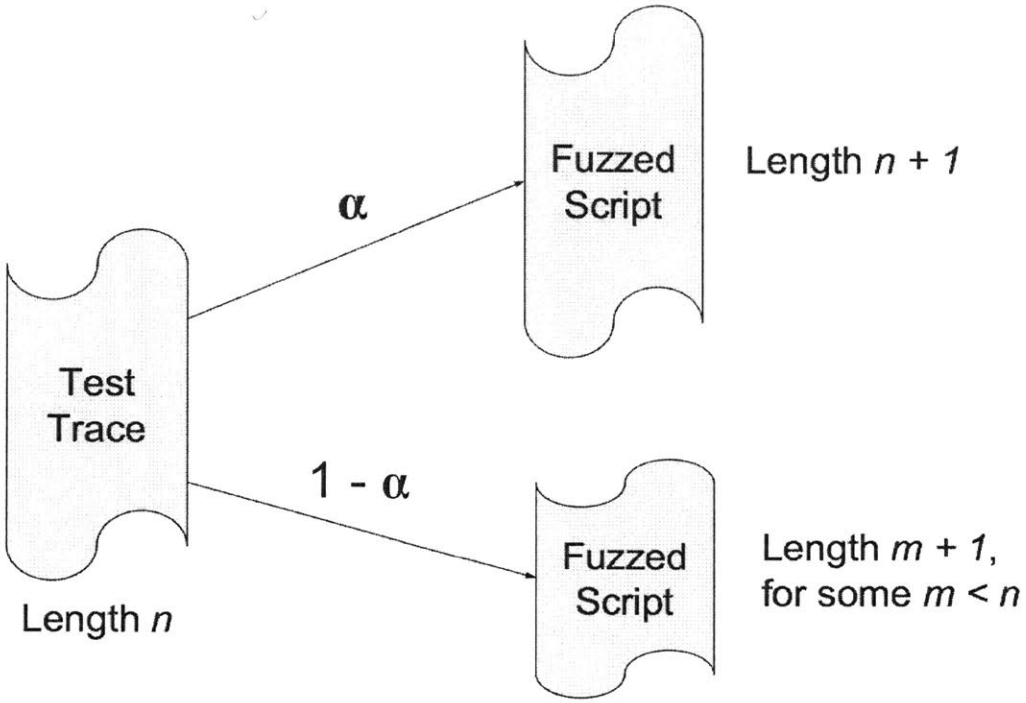

Figure 3-1: The effects of truncating a test trace on the subsequent checked trace and fuzzed test script; length is measured in the number of system calls present. The probability of SibylFuzzer inserting the fuzzed system call at the end of the original sequence is α.

off to the checker. The effect of this potential truncation on the length of subsequent outputs is illustrated in Figure 3-1. This ensures the fuzzed system call will always become the final call in the newly fuzzed script. Logically, if a sequence of system calls does not produce an erroneous trace, then a subsequence must not either. Therefore if the fuzzed test script is later passed through the SibylFS procedure and the checker detects a model-implementation disagreement, then the discrepancy must have been triggered by the final call in its sequence: the most recently fuzzed system call.

SibylFuzzer hands off the potentially-truncated trace to the checker, which proceeds as usual. For each system call the checker verifies, SibylFuzzer extracts file system state information from the resulting state set and stores it in local data structures. Recall that the checker updates its states according to the SibylFS model's specifications. As a result, by the time the checker finishes with the trace SibylFuzzer

will have amassed an accumulated history of all the files, directories, file descriptors, and directory handles that have ever appeared in the SibylFS model's view of the file system.

When the checker finishes going through the potentially-truncated test trace, SibylFuzzer checks to see if it has found any disagreements with the observed behavior (i.e. potential bugs). If the test trace passed the checker's inspection, SibylFuzzer generates a system call by randomly selecting a system call and populating the arguments using its accumulated history of file system contents. Next it takes the sequence of system calls from the truncated test trace and appends the newly fuzzed system call, then finally writes this extended sequence to a test script file.

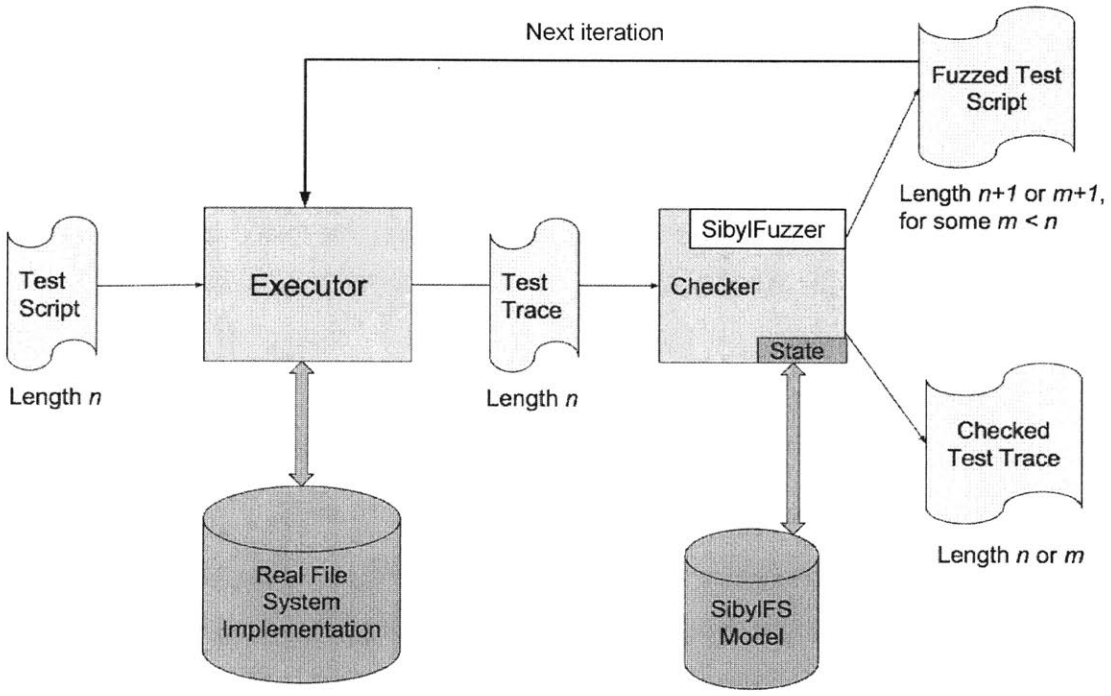

Figure 3-2: SibylFuzzer included in the SibylFS procedure. A set of fuzzed test scripts is outputted at the end, and can be used as input test scripts for the next iteration of fuzzing. The potential lengths of the test traces, checked traces and fuzzed test scripts as a result of truncation are also labelled.

Figure 3-2 illustrates SibylFuzzer's place in the SibylFS procedure. This diagram makes it clear that the fuzzed test scripts can also be used as input to the SibylFS procedure in a sort of feedback loop. Since fuzzing introduces an additional, randomly

generated system call, the fuzzed test script is guaranteed to be distinct from the original input script. Therefore fuzzing can and should be run for multiple iterations to increase coverage, using one iteration's generated fuzzed scripts as input for the next iteration. Since by design SibylFuzzer only runs if the input script does not have an error, as more and more potential bugs are found with each iteration, there will be fewer and fewer fuzzed scripts outputted.

Running the SibylFS procedure once consists of two commands: one to run the executor, another to run the checker. It is straightforward to write a shell script that runs the SibylFS procedure with SibylFuzzer for however many iterations desired.

3.2 Extracting From the File System State

Recall that in order for SibylFuzzer to be an effective stateful fuzzer, it must be able to generate inputs that are valid or interesting, and incorporate knowledge of the current file system state. By tracking the checker's file system state as the checker goes through the test trace, SibylFuzzer can accumulate "master lists" of all entries that have ever appeared in the state. By comparing the current contents of the SibylFS model state with these master lists, SibylFuzzer can identify entries that used to exist but no longer do according. Both of these types of entries make for "valid" or "interesting" inputs.

Retrieving and tracking relevant file system state information requires exposing some parts of the SibylFS model that were originally hidden via utility methods at the model-level. Utility methods were implemented for the following tasks:

1. **List pathnames for all files and directories currently in the file system state.**

2. **Retrieve users, groups, open file descriptors and directory handles for all ongoing processes.**

3. **Lookup file size from either a given file descriptor or a given file pathname.**

This allows SibylFuzzer to track and maintain separate "master lists" for each of the following entry types: files, directories, process IDs, users, groups, open file descriptors and directory handles. Throughout the rest of this thesis I use *relevant object* as a general term for the various entries that SibylFuzzer tracks.

3.3 Fuzzing Parameters

SibylFuzzer can fuzz nearly all the system calls in the SibylFS model. It does not include *pwrite* nor *pread*, which are almost identical to *read* and *write* respectively but have an additional offset argument. Because of a lack of documentation, the SibylFS-specific commands *dump* and *add_user_to_group* are also not included (though *open_close* is). That leaves SibylFuzzer with twenty system calls to fuzz for. Each system call requires SibylFuzzer to generate one or more arguments. To enable experimentation, SibylFuzzer must have a parameterized methodology. The fuzzing parameters therefore each designate a probability distribution that guides SibylFuzzer's test generation strategy while fuzzing system calls. In the interest of efficiency SibylFuzzer uses shared parameters: one fuzzing parameter may influence the generation strategy for multiple system calls.

3.3.1 High-Level Parameters

The first major fuzzing choice is where to insert the fuzzed system call. The fuzzer has two placement choices: either append to the end of the original sequence or insert at a randomly chosen breakpoint (see Figure 3-1). I define α as the probability the fuzzer picks the former throughout the rest of the thesis.

The second major choice is which system call to fuzz. I define the *syscall_distribution* parameter as a normalized probability distribution containing the probabilities of fuzzing each of the twenty available system calls.

3.3.2 System Call Argument Generation

Once SibylFuzzer has decided where and which system call to fuzz, it must generate arguments. From the previous section, SibylFuzzer is able to keep track of the contents of the file system as the SibylFS checker steps through the trace, and therefore has access to two types of interesting inputs: relevant objects that are in the current file system state, and relevant objects that were formerly in the file system state. Depending on the system call being fuzzed, usually one type results in successful system calls while the other causes errors to be thrown. In order to check that the file system implementation properly handles both scenarios, both types should have a nonzero probability of being used.

The fuzzer has two main ways to choose argument values, which are abstracted in the following two fuzzing parameters:

- *use_former_or_current* (β) - For a given relevant object type ρ, the normalized probability distribution of picking each of the following two options:
  - the set of relevant objects of type ρ that used to exist in the file system but no longer do,
  - the set of relevant objects of type ρ which currently exist in the file system
- *use_former_or_current_or_random* (γ) - For a given relevant object type ρ, the normalized probability distribution of picking each of the following three options:
  - the set of relevant objects of type ρthat used to exist in the file system but no longer do,
  - the set of relevant objects of type ρwhich currently exist in the file system,
  - the set of randomly generated objects of type ρ with no regard for file system state

Below are some more system-call-specific fuzzing parameters:

- *perm_distribution* - The normalized probability distribution that the fuzzer fuzzes a certain permission. There are probabilities for 0o000, 0o700, 0o770, 0o111, 0o222, 0o333, 0o444, 0o555, 0o666, 0o777, as well as for generating a permission completely randomly.

- *open_flag_independent_probs* - The independent probabilities that the fuzzer includes a certain open flag when fuzzing a bitfield. The open flags consist of O_EXEC, O_RDONLY, O_RDWR, O_SEARCH, O_WRONLY, O_APPEND, O_CLOEXEC, O_CREAT, O_DICTIONARY, O_DSYNC, O_EXCL, O_NOCTTY, O_NOFOLLOW, O_NONBLOCK, O_RSYNC, O_SYNC, O_TRUNC, and O_TTY_INIT.

- *use_size_within_or_larger* - In *read* and *write* system calls which require size and file arguments, decide whether to use a size that is larger than the file's size or not.

- *use_dir_or_leaf* - When selecting a directory argument, the probability of choosing from the set of only empty directories over the set of all directories.

- *use_sticky_perm_or_not* - In *chmod*, whether or not to set the sticky bit. SibylFS's original test suite does not extensively feature sticky-bit permissions.

- *use_state_or_random_former_or_current* - The normalized probability distribution of picking each of the following four options:

  - the set of directories and files that used to exist in the file system but no longer does,
  - the set of directories and files which currently exist in the file system,
  - the set of directories that used to exist in the file system but no longer does, combined with something randomly generated
  - the set of directories which currently exist in the file system, combined with something randomly generated

The fuzzer's decision-making process when constructing system call arguments is summarized as follows:

- *mkdir* - Generate directory path with γ, generate permissions with *perm_distribution.*
- *rmdir* - Choose directory path with β and *use_dir_or_leaf.* Only current directories which are leaves should be successfully removed.
- *open* - Choose path name with γ, generate list of open flags with *open_flag_independent_probs*, generate permissions with *perm_distribution.*
- *open_close* - Same as *open.*
- *close* - Choose file descriptor with β.
- *read* - Choose file descriptor with β, choose number of bytes to read with *use_size_within_or_larger.*
- *write* - Choose file descriptor with β, choose number of bytes to write with *use_size_within_or_larger.* The buffer being written is always randomly generated.
- *rename* - Choose source pathname with β, choose destination pathname with *use_state_or_random_former_or_current.*
- *link* - Same as *rename.*
- *symlink* - Choose source pathname with γ (as *symlink* does not validate the source pathname), choose symlink pathname with *use_state_or_random_former_or_current.*
- *chdir* - Choose the new current working directory with β.
- *truncate* - Choose file name with β, choose truncation size with *use_size_within_or_larger.*
- *chmod* - Choose entry pathname with β, generate permissions with *perm_distribution* and *use_sticky_perm_or_not.*

- *opendir* - Choose directory path with β.
- *readdir* - Choose directory handle with β.
- *closedir* - Same as *readdir*.
- *rewinddir* - Same as *readdir*.
- *chown* - Choose entry pathname with β.
- *stat* - Choose entry pathname with β.
- *lstat* - Same as *stat*.

In order to easily adjust fuzzer parameters without having to recompile every time, a centralized JSON file contains all but one of the parameters used by SibylFuzzer to generate system calls. The black sheep is α, the probability that SibylFuzzer will append a new system call to the end of the file rather than insert it at a randomly chosen breakpoint, due to it being used in a separate module. Instead α is implemented as a command line argument which is set in the shell script; this also allows for easier debugging. When running SibylFS, the checker loads parameters from the JSON file upon initialization.

3.4 Fuzzing Effect on Test Script Length

Each successful system call run changes some aspect of the file system state. Therefore, longer sequences of system calls frequently result in more diverse and complex file system states, translating to wider in-depth coverage from a fuzzing perspective. Conversely, sequences that are only a few system calls long correspond to relatively empty and uninteresting states, which are unlikely to find bugs. Therefore it would be good for SibylFuzzer to increase, or at least maintain test script length throughout multiple iterations.

Consider the changes in length between a test script and its fuzzed descendant produced by running SibylFuzzer many times. In each fuzzer iteration, appending the fuzzed system call to the end increases the fuzzed script's length relative to

the original, while truncating the script before appending usually does the opposite. Mathematically the expected length of a script after iteration $i+1$ can be roughly approximated as follows:

$$l_{i+1} \approx \alpha \cdot (l_i + 1) + (1 - \alpha) \cdot (\frac{l_i}{2} + 1)$$

which can be rewritten as

$$\frac{l_{i+1}}{l_i} \approx \alpha \cdot (1 + \frac{1}{l_i}) + (1 - \alpha) \cdot (\frac{1}{2} + \frac{1}{l_i})$$

where l_i refers to the length of the script in iteration i, and α is the probability that SibylFuzzer inserts the fuzzed system call at the end of the original sequence. This is only a very rough approximation - there are several inaccuracies, most prominently that the equation assumes truncation will always halve the original test script. However, it suffices to highlight a certain relationship between script length and α.

Suppose I enforce that scripts must increase in length from one fuzzing iteration to the next. In order for the expected fuzzing length to increase, it must be that:

$$\frac{l_{i+1}}{l_i} > 1$$

Combining equations gives

$$\alpha \cdot (1 + \frac{1}{l_i}) + (1 - \alpha) \cdot (\frac{1}{2} + \frac{1}{l_i}) > 1$$

which simplifies to

$$\alpha > 1 - \frac{2}{l_i}$$

As the script length increases, α must increase in order to keep the expected final script's length greater than or equal to its original length.

Intuitively this makes sense - if SibylFuzzer makes a random break and insertion N lines before the end, in order to merely return to the original script length SibylFuzzer must append to the end for at least N further iterations. If SibylFuzzer only runs

for 100 iterations, then doing a random insertion more than 99 calls before the end makes it impossible for the test script to come out at the end of 100 iterations with a greater length than what it had originally. Therefore as the script length increases it becomes more and more likely that *any* instance of SibylFuzzer doing a random insertion immediately dooms the final script length to end up shorter than its original ancestor.

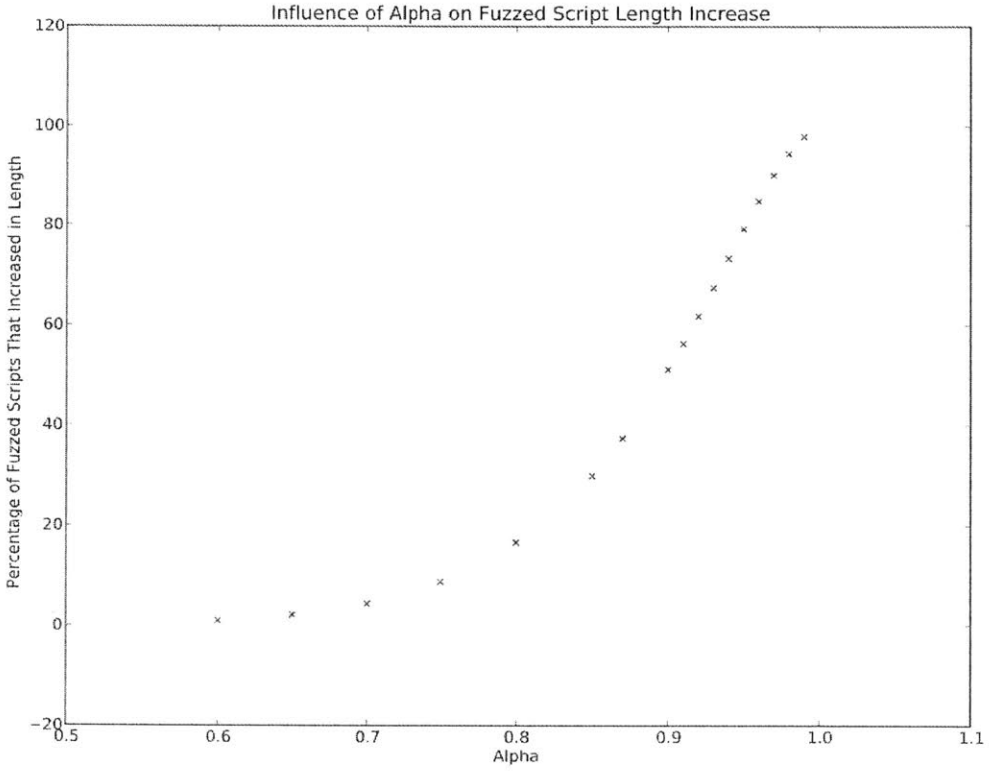

Figure 3-3: As α increases, so does the probability that the length of the fuzzed script ends up greater than the original.

A Monte Carlo simulation of the fuzzing process's effect on script length, displayed in Figure 3-3 confirms this intuition. The simulation ran for twenty trials on each test script from the original SibylFS suite, for values of $\alpha = 0.6, 0.65, 0.7, 0.75, 0.8, 0.85,$ and 0.9 to 0.99 incrementing by 0.01. Pseudocode for this simulation is as follows:

for $\alpha \in alpha\_values$ **do**

$successes \leftarrow 0$

$total \leftarrow 0$

for $script \in original\_sibylfs\_suite$ **do**

$trial \leftarrow 1$

while $trial < 21$ **do**

$iteration \leftarrow 1$

$current\_length \leftarrow len(script)$

while $iteration < 101$ **do**

if $random(0, 1) < \alpha$ **then**

$current\_length \leftarrow current\_length + 1$

else

$current\_length \leftarrow random(1, current\_length) + 1$

end if

$iteration \leftarrow iteration + 1$

end while

$trial \leftarrow trial + 1$

$total \leftarrow total + 1$

if $current\_length \geq len(script)$ **then**

$successes \leftarrow successes + 1$

end if

end while

end for

Print $\alpha, \frac{successes}{total}$

end for

Chapter 4

Testing and Results

SibylFuzzer is a file system fuzzer targeting a Linux file system; therefore its goal is to detect new Linux file system bugs. How many bugs SibylFuzzer finds is a measure of the effectiveness of its test generation strategy. Another benchmark for effectiveness is if SibylFuzzer can generate test scripts containing Linux bugs that were not also detected by the original SibylFS test suite. If it can, then SibylFuzzer's coverage is not inferior to that of the original SibylFS test suite.

In order to observe the effects of the two high-level fuzzing parameters (see Section 3.3.2), separate trials must be run for different values of these parameters. Therefore testing consists of running the automated fuzzing script four times, with SibylFuzzer running for 100 iterations each time, on a Linux ext4 file system. For each of these four trials, after the final fuzzing iteration all fuzzed test scripts are accumulated into one large corpus, which is then run through the original SibylFS procedure (sans SibylFuzzer) to check for potential bugs.

The two high-level fuzzing parameters serving as test variables are: α, the probability the fuzzer adds an extra system call to the end as opposed to inserting at a random point in the middle of the script, and *syscall_distribution*, the probability distribution for which system call to fuzz. All other fuzzing parameters are fixed to values shown in Table 4.1.

According to Figure 3-3, when α is close to 1, then fuzzing is expected to maintain or increase script length. However once α drops below 0.9, then fuzzing is more likely

Parameter	Value
perm_distribution	0.05 for "0o000"/"random", else 0.1
open_flag_independent_probs	0.5 for all
use_size_within_or_larger	0.5
use_dir_or_leaf	0.5
use_sticky_perm_or_not	0.05
use_state_or_random_former_or_current	0.25 for each option

Table 4.1: Fixed fuzzing parameter values for the four test trials.

	α	*syscall_distribution*
Trial 1	0.7	Uniform
Trial 2	0.95	Uniform
Trial 3	0.7	Original Suite-Based
Trial 4	0.95	Original Suite-Based

Table 4.2: The two high-level parameters' values for the four test trials.

to decrease script length. To test this paradigm, α is set to 0.7 in half of the trials and 0.95 in the other half.

Now consider how *syscall_distribution* might be configured. One candidate is the uniform distribution - each system call is equally likely to be fuzzed. This may induce fuzzed scripts that have more balanced coverage, but are dissimilar to real-life system call usage - for example, the *rewinddir* system call is much less common than *mkdir*.

The second candidate for *syscall_distribution* is a distribution based on the frequency of system calls in the original SibylFS test suite. However this skew towards system calls that have already been extensively tested may result in fuzzed scripts that are redundant with the original suite. For about half of the twenty fuzz-able system calls, their probabilities of being fuzzed becomes close to 0 under this distribution (see Figure 4-1). Table 4.2 displays what values are used for each test variable in the four trials.

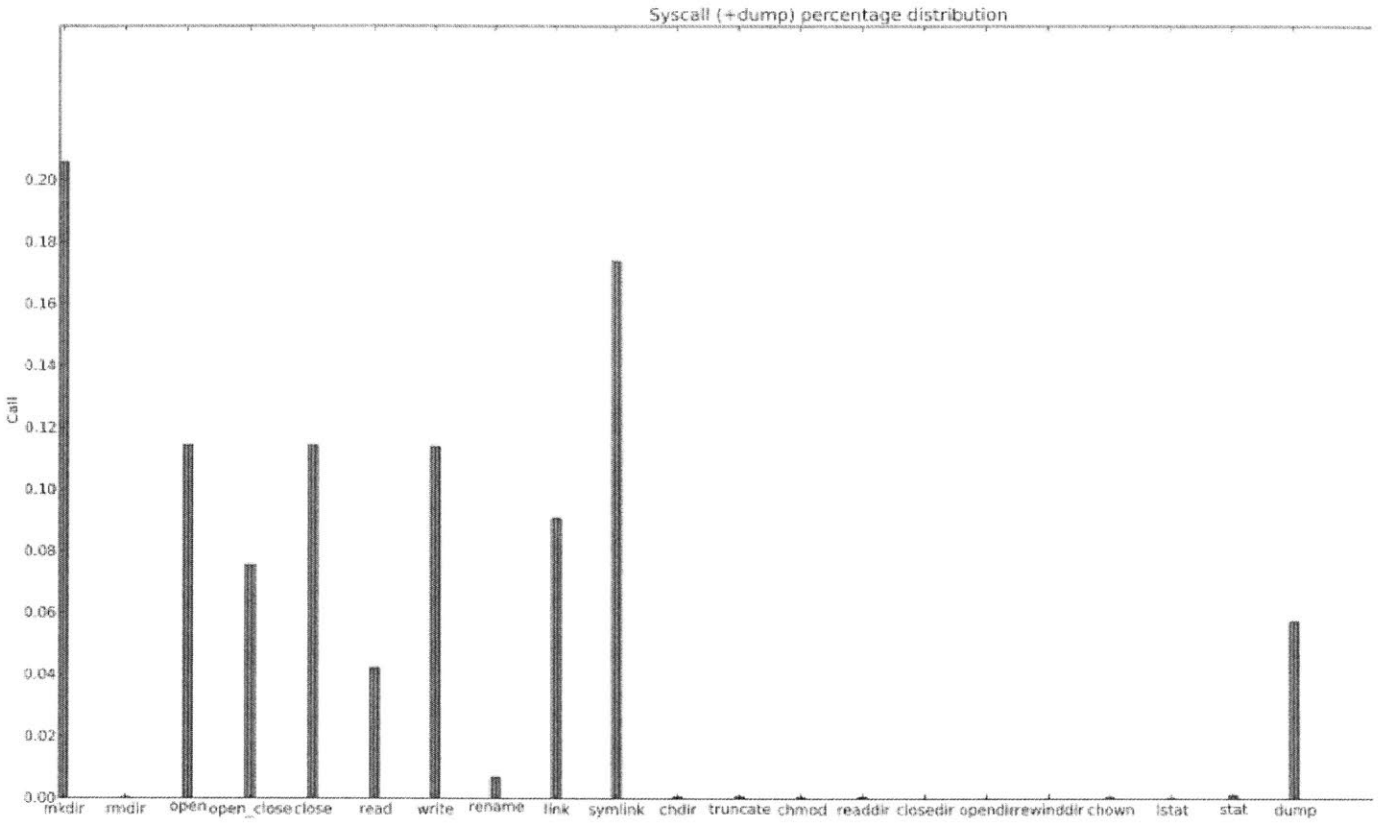

Figure 4-1: The distribution of system calls in the original SibylFS test suite. System calls that create objects such as *mkdir* and *open* dominate. About half of the system calls, including *rmdir* and *opendir*, almost never appear.

4.1 The Starting Test Script Set

The starting set of test scripts was formed using a sampling of the original test suite. Each sub-suite contributed up to 200 randomly sampled scripts. This resulted in a starting set of 1251 test scripts featuring all the relevant system calls which served as the basis for all four trials.

Ideally I would have been able to use all 21,070 test scripts as the fuzzing base, but this would have increased trial runtimes by at least a factor of 20. I did not use a purely random sampling because of two reasons. The first reason is that I wanted to always include tests from all suites, so that the initial suite was guaranteed to include every relevant system call at least once. As shown in Figure 4-1, the distribution

	Number of Raw Fuzzed Inputs	Number of Model-Implementation Disagreements	Caused by chroot jail	Caused by *rename X "/"*	Others
Trial 1	30787	1249	550	511	188
Trial 2	29736	1237	523	246	468
Trial 3	100629	617	93	272	252
Trial 4	61635	443	43	126	274

Table 4.3: The first column contains the number of fuzzed inputs that were generated over the entire trial. The second column displays the unfiltered number of model-implementation disagreements, calculated by searching for SibylFS checker keywords. The third and fourth columns display the number of model-implementation disagreements from the test trials that were caused by two major known SibylFS inaccuracies detailed in Section 4.2.1. The fifth column displays the number of model-implementation disagreements that were *not* caused by the two known SibylFS inaccuracies. I calculated these numbers using a Python filtering script that searches for specific keywords corresponding to the known SibylFS inaccuracies.

of system call appearances in the original suite varies greatly from call to call. A random sampling would have been unlikely to procure a script which contains *unlink*, for instance. However, sub-suites are guaranteed to feature their eponymous system call in their scripts. The second reason is that often in larger individual suites such as *open* and *mkdir*, the differences between two test scripts can be minuscule - for example, using a double-slash instead of a single slash in a single system call. The test scripts of smaller individual suites, on the other hand, tend to be more varied, and therefore should be prioritized.

4.2 Results

The results of the four trials are displayed in the first two columns of Table 4.3. SibylFuzzer detected over a thousand model-implementation disagreements each in Trials 1 and 2. Trials 3 and 4 had fewer disagreements, but still came up with several hundred apiece. The dramatic difference in disagreements-found is most likely due to the different *syscall_distribution* values used - the more popular system calls were more heavily tested, while buggy behavior lingered on in the less popular calls. For this rea-

son, I was not surprised that Trials 1 and 2, which used a uniform *syscall_ distribution* and were more likely to feature relatively untested fuzzed system calls, ended up detecting the most model-implementation mismatches.

At first glance it would appear that the fuzzer has found many potential Linux bugs. However, over the course of my analysis I found that many of the model-implementation disagreements were actually false positives caused by inaccuracies in the SibylFS system.

I investigated the detected model-implementation disagreements by comparing real-life system behavior to the Linux specification for correctness. Rudimentary C programs can effectively mimic shorter traces for additional sanity checking.

4.2.1 Known SibylFS System Inaccuracies

Through email exchanges with the authors and examining their notes I found that there are some known inaccuracies with the SibylFS system. These deficiencies were responsible for two major patterns of detected model-implementation disagreements found in testing. In certain cases the SibylFS model is left incomplete due to unclear specifications. Consequentially any scripts that fall under these cases will trigger false positives. One such case is when either one of the two directory arguments to *rename* is currently in use by the system or another process. According to the Linux and POSIX specification, the call should fail with an EBUSY error. As displayed in Figure 4-2, this edge case is triggered if the test script contains *rename* X *"/"*, where X is any directory pathname.

Also, recall in Section 2.1 when it was mentioned that running the SibylFS executor in a chroot jail causes the root directory to register an extra link. The extra link causes another frequently recurring disagreement detected by SibylFuzzer when checking metadata by running either *stat "/"* or *lstat "/"*. In addition, if the root directory has a symbolic link, checking the metadata for that symbolic link later in the test script also triggers this disagreement.

As shown in the rightmost column of Table 4.3, filtering out the disagreements caused by SibylFS's chroot jail and *rename* misbehavior leaves a much smaller body

```
@type trace
mkdir "empty_dir1" 0o777
RV_none

rename "/empty_dir1" "/"
EBUSY
#
# Error:
#     !!! EXCEPT CHECKLIB DOES NOT AGREE !!!
#
# Error:
#     The spec permitted:
# ENOTEMPTY, EEXIST
#
# Error:                  EBUSY
# Unexpected results: EBUSY
# Allowed are only: RV_none
# Continuing execution with RV_none
#
# trace not accepted
```

Figure 4-2: Model-implementation disagreement caused by known SibylFS *rename* misbehavior. This case of *rename* behavior was left incomplete in the model due to a hazy specification. Therefore, the model disagreeing with correct real-life behavior is actually expected.

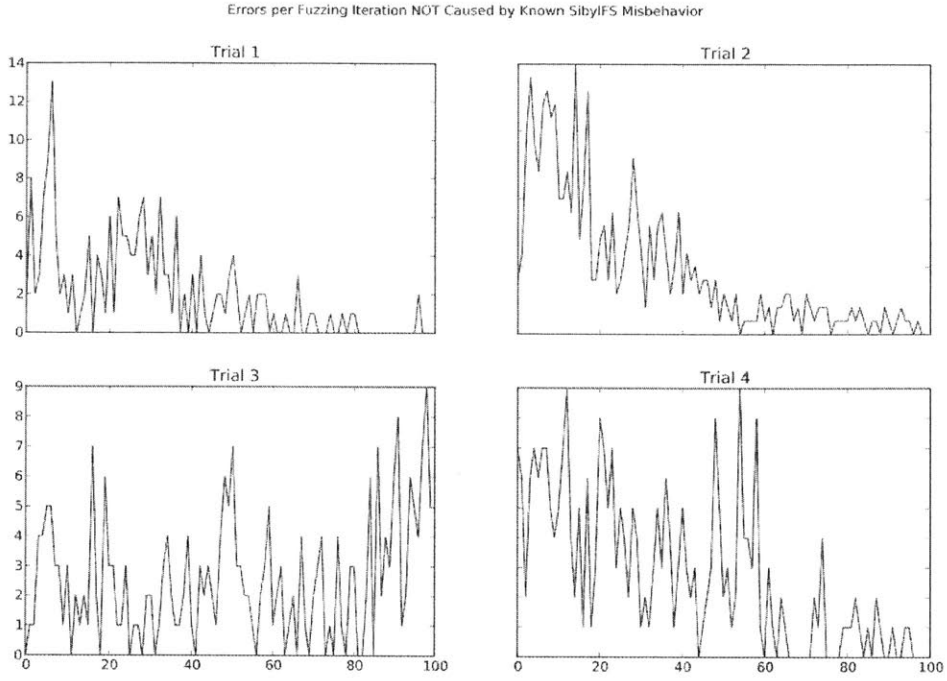

Figure 4-3: The number of disagreements (not caused by known SibylFS misbehavior) found per fuzzing iteration. For Trials 1 and 2, roughly speaking, there were more disagreements detected in early iterations. This makes sense because each disagreement found reduces the input script set for the next iteration, limiting how many disagreements may be found in the future.

of potentially-meaningful detected differences to investigate. Figure 4-3 displays how many of these meaningful disagreements came from each iteration of fuzzing. Trials 1 and 2 having many more of these disagreements was expected, since the uniform *syscall_distribution* has higher probabilities of fuzzing *rename*, *stat* and *lstat* compared to the original SibylFS test-suite-based *syscall_distribution*.

4.2.2 Discovered SibylFS Inaccuracies

The test trial results also revealed previously-unknown bugs in the SibylFS model. These are instances where the real-life system and specification agree but the model does not.

The most widespread model-implementation disagreement was due to how SibylFS

```
@type trace
mkdir "nonempty_dir1" 0o777
RV_none

mkdir "nonempty_dir1/d2" 0o777
RV_none

open "nonempty_dir1/d2/f3.txt" [O_CREAT; O_WRONLY] 0o666
RV_num(3)

write (FD 3) "Lorem ipsum dolor sit amet,
      consectetur adipisicing elit,
      sed do eiusmod tempor inc" 83
RV_num(83)

close (FD 3)
RV_none

opendir "/"
RV_num(1)

open "/nonempty_dir1/d2/LansicmiqTpEtxfBXl1SVwY"
      [O_TRUNC; O_NOCTTY; O_DSYNC; O_DIRECTORY;
          O_CREAT; O_RDWR] 0o555
RV_num(4)
#
# Error:
#    !!! EXCEPT CHECKLIB DOES NOT AGREE !!!
#
# Error:
#    The spec permitted:
# RV_num(3)
#
# Error:                  RV_num(4)
# Unexpected results: RV_num(4)
# Allowed are only: RV_num(3)
# Continuing execution with RV_num(3)
#
# trace not accepted
```

Figure 4-4: This disagreement, along with many others, is caused by the SibylFS model's incorrect portrayal of file descriptor and directory handle space. In this sequence, the directory handle returned by *opendir* is also supposed to take up the next available file descriptor 3. The SibylFS model thinks file descriptors and directory handles are completely separate, so it still believes that file descriptor 3 is available for the final *open* call. Note that this error could also be triggered by many other system calls.

models file descriptors and directory handles. The POSIX and Linux specifications imply that file descriptors and directory handles share the same space, but the SibylFS model uses two different spaces. Therefore, when a directory handle is opened with *opendir*, the model and real-life system have very different views of which file descriptors are currently open or available. In Figure 4-4, the disagreement is revealed with an attempted *open*, but it can also be triggered by many other system calls that result in different error messages. For example, if the final *open* were replaced by a *read* to file descriptor 3, the real-life system would throw an EISDIR error, while the SibylFS model would incorrectly throw an EBADF error.

I have notified the SibylFS authors of this SibylFS model bug. Their preliminary patch, which simply combines the model's directory handle and file descriptor space into one, failed because it was not compatible with how the SibylFS executor enumerates directory handles in observed real-life return values.

Another recurring disagreement lay in the SibylFS model's handling of symbolic links: the model acts as if symbolic links that were previously made do not exist. As Figure 4-5 illustrates, the model throws an ENOENT because it believes it must create the "/nonempty_dir1/d2/sl_dotdot_f1.txt" file but lacks an O_CREAT flag. However, the correct behavior is to throw an ELOOP because "/nonempty_dir1/d2/sl_dotdot_f1.txt" is a symbolic link. As was the case with the previous *opendir* issue, this disagreement may be triggered by many system calls in a variety of different contexts. I have alerted the SibylFS authors of this issue as well, but have not received a response yet.

4.2.3 Potential Linux File System Bug

Although the majority of model-implementation disagreements found turned out to be the fault of bugs in SibylFS, there were still checked traces that indicated a possible bug in the real-life Linux file system. This potential Linux bug happens when an a file is opened with both the O_WRONLY and O_RDWR flags. A file descriptor is returned but cannot be written to.

Figure 4-6 illustrates this bug. The specifications for *open* state that at least one

```
@type trace
mkdir "nonempty_dir1" 0o777
RV_none

mkdir "nonempty_dir1/d2" 0o777
RV_none

symlink "../f1.txt" "nonempty_dir1/d2/s1_dotdot_f1.txt"
RV_none

open_close "/nonempty_dir1/d2/s1_dotdot_f1.txt"
      [O_TRUNC; O_NOFOLLOW; O_DIRECTORY;
          O_WRONLY; O_RDWR] 0o666
ELOOP
#
# Error:
#    !!! EXCEPT CHECKLIB DOES NOT AGREE !!!
#
# Error:
#    The spec permitted:
# ENOENT
#
# Error:                  ELOOP
# Unexpected results: ELOOP
# Allowed are only: ENOENT
# Continuing execution with ENOENT
#
# trace not accepted
```

Figure 4-5: A disagreement caused by the SibylFS model's incorrect handling of symbolic links. The model behaves as if the symbolic link "/nonempty_dir1/d2/sl_dotdot_f1.txt" does not exist, when according to the specification the presence of an O_NOFOLLOW flag should throw an ELOOP.

```
@type trace
mkdir "empty_dir1" 0o777
RV_none

mkdir "nonempty_dir1" 0o777
RV_none

mkdir "nonempty_dir1/d2" 0o777
RV_none

open "nonempty_dir1/d2/d3.txt" [O_CREAT; O_WRONLY] 0o666
RV_num(3)

open "/empty_dir1/D5bTdUAsviX8AnGCLuzSLiQut
      DCNCJXg7lKHbjuyfU" [O_NOFOLLOW;
          O_NOCTTY; O_DSYNC; O_CREAT; O_RDWR] 0o555
RV_num(4)

open "/empty_dir1/xUHL1Uuogf6Gh3MlFSRbzp3hWNxQ
      d0dj9D7US" [O_NOFOLLOW; O_NOCTTY; O_CREAT;
          O_CLOEXEC; O_WRONLY; O_RDWR] 0o700

write (FD 5) "blahblahblahblahblahblah" 10
EBADF
#
# Error:
#     !!! EXCEPT CHECKLIB DOES NOT AGREE !!!
#
# Error:
#     The spec permitted:
# RV_num(10), RV_num(9), RV_num(8), RV_num(7),
# RV_num(6), RV_num(5), RV_num(4), RV_num(3),
# RV_num(2), RV_num(1), RV_num(0)
#
# Error:                  EBADF
# Unexpected results: EBADF
# Allowed are only: RV_num(10), RV_num(9), RV_num(8), RV_num(7),
# RV_num(6), RV_num(5), RV_num(4), RV_num(3),
# RV_num(2), RV_num(1), RV_num(0)
#
# trace not accepted
```

Figure 4-6: An example of the single potential Linux bug SibylFuzzer found. Despite being opened with both O_WRONLY and O_RDWR flags, file descriptor 5 is considered unwritable by the real-world Linux file system. The SibylFS model disagrees and believes that a write is perfectly legal.

Disagreements Caused By...	SibylFS's bad *opendir*	SibylFS's bad symbolic link	Unwritable Linux file descriptor	Miscellaneous
Trial 1	168	10	6	4
Trial 2	426	23	7	12
Trial 3	7	110	94	41
Trial 4	7	135	100	32

Table 4.4: The model-implementation disagreements from the test trials that were not caused by known SibylFS inaccuracies. Again, I used a Python script based on keyword search for filtering. Miscellaneous disagreements are not easily grouped, but they mostly comprise disagreements that the original SibylFS test suite were also able to find and are therefore irrelevant.

of the O_RDONLY, O_WRONLY and O_RDWR flags should be set, corresponding to read-only, write-only, and read-write modes. It is not stated what happens if two or more of these flags are set, but it seems counterintuitive that setting the two flags which allow for writing returns a file descriptor that cannot be written to.

Table 4.4 displays the breakdown of model-implementation disagreements not caused by known SibylFS misbehavior. The number of disagreements that were caused by *opendir* was relatively very high for Trials 1 and 2. This is because Trial 3 and Trial 4's *syscall_distribution* has a minuscule probability of picking *opendir* to fuzz; indeed, the lack of *opendir* in the original SibylFS suite explains why this SibylFS bug went undetected.

4.2.4 Fuzzed Script Lengths

The previous chapter discussed the benefits of ensuring that fuzzed script length does not decrease. For Trials 1 and 3, α was set to 0.7 which corresponds to an expected overall decrease in fuzzed script length. For Trials 2 and 4, α was set to 0.95, which is expected to result in a constant or slightly increased fuzzed script length. Figures 4-7 to 4-10 show the script length distributions; in general the overall test script length changed as expected. However, there does not seem to be a strong correlation between higher α and the number of disagreements caused by newly-discovered SibylFS issues and the Linux bug. Therefore the relationship between

script length and the probability of discovering a bug remains unclear.

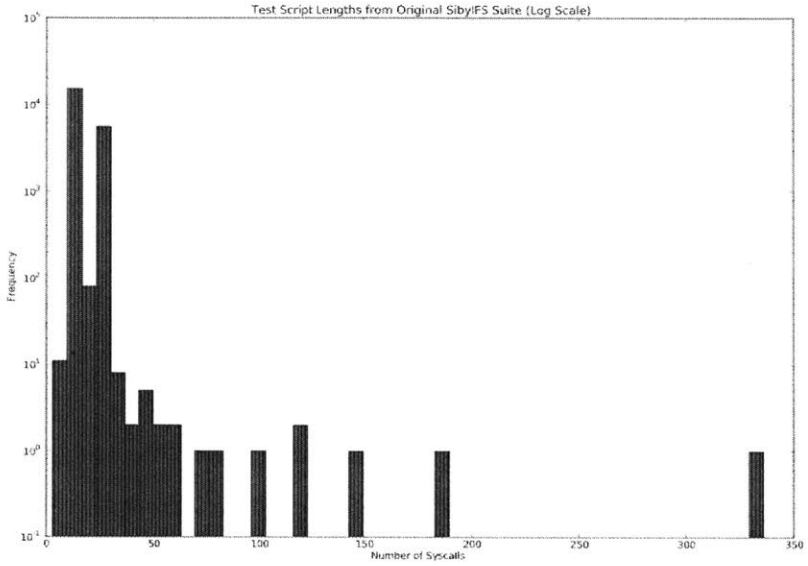

Figure 4-7: Frequency of Test Scripts by Length in Original SibylFS Test Suite

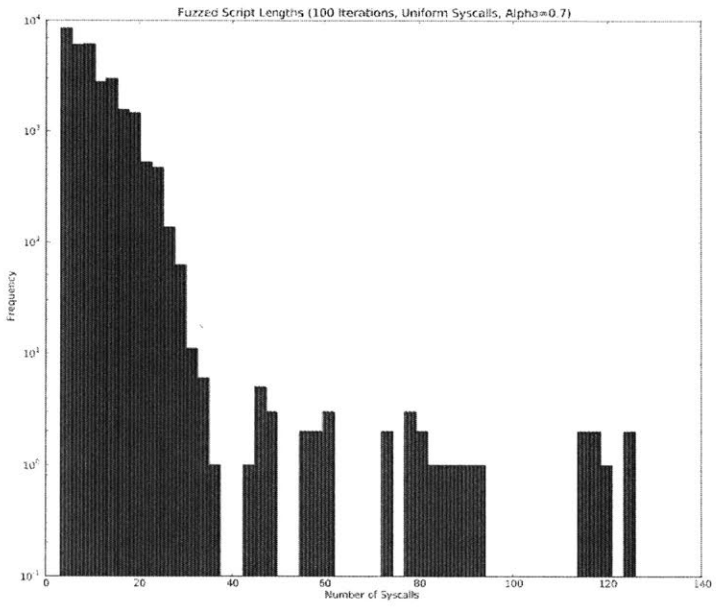

Figure 4-8: Frequency of Test Scripts by Length in Trial 1's Fuzzed Corpus

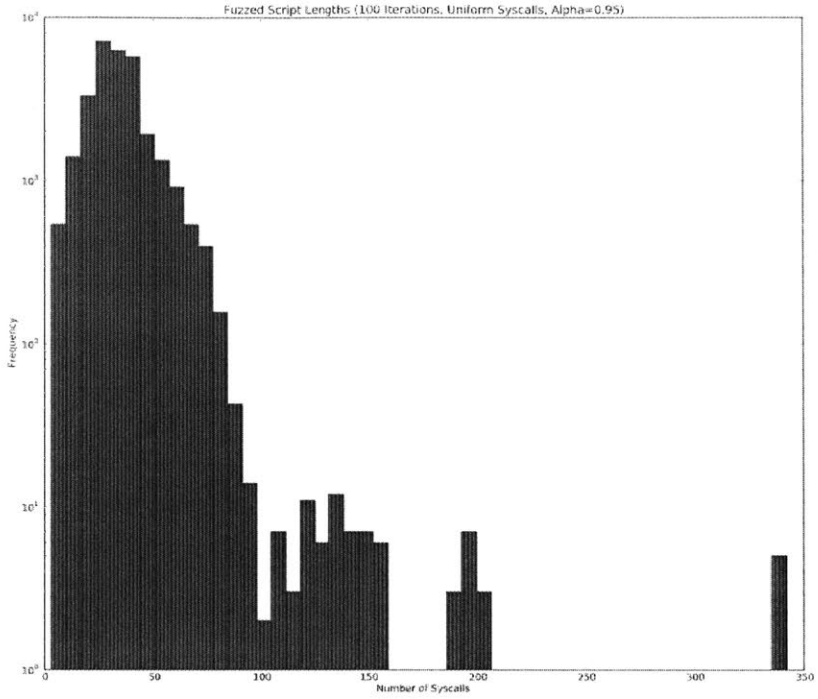

Figure 4-9: Frequency of Test Scripts by Length in Trial 2's Fuzzed Corpus

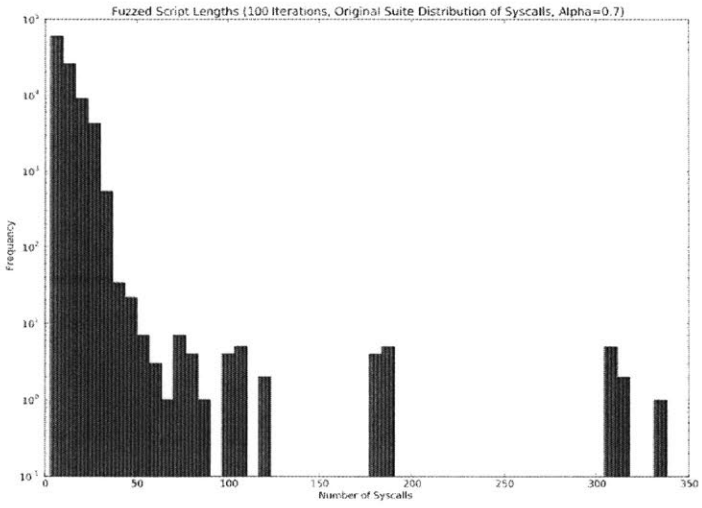

Figure 4-10: Frequency of Test Scripts by Length in Trial 3's Fuzzed Corpus

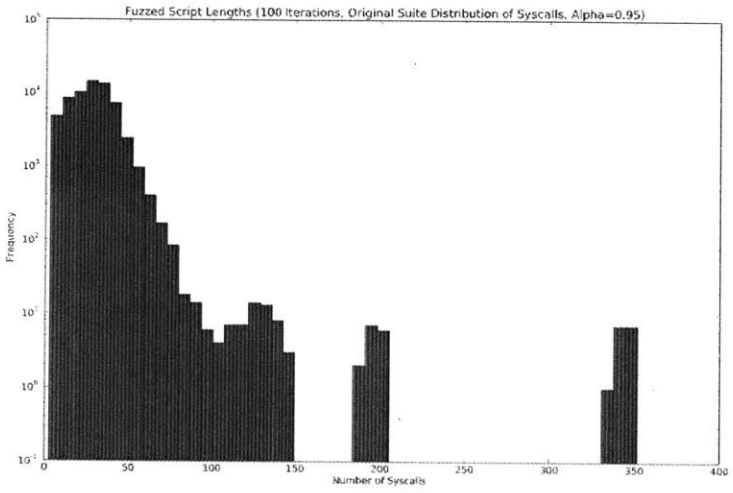

Figure 4-11: Frequency of Test Scripts by Length in Trial 4's Fuzzed Corpus

Chapter 5

Conclusion

This thesis presented the design and implementation of SibylFuzzer, a stateful fuzzer to test file system behavior. In the four test trials, SibylFuzzer was able to discover previously-unknown errors in the SibylFS model, proving its coverage to be at least as good as the original SibylFS test suite. SibylFuzzer also demonstrated that it was also able to find potentially real bugs in the host file system.

5.1 Related Work

John Regehr wrote a small stateful file system fuzzer that generates system calls as test inputs, similar to SibylFuzzer. In Regehr's fuzzer the state model consists of which files were currently opened which were currently closed. Even though this state is extremely small and simplistic, it grants the fuzzer knowledge of how to fuzz valid arguments for *close*, *open*, *read*, and *write*. In comparison to the state used by SibylFuzzer, however, Regehr's fuzzer's state does not contain enough knowledge for fuzzing other system calls such as *mkdir*. In addition, Regehr's fuzzer cannot detect incorrect file system behavior; it can only print out results for its human user to check by hand.

Yang et al [14] have used model checking to find file system errors. Their FiSC tool provides a simpler model of the file system that currently only consists of the name, size, and link count for directories. This suffices to find errors, but is not as complete

or realistic as the state that SibylFuzzer uses. In addition, FiSC often requires the user to modify the implementation's code; SibylFuzzer on the other hand is much less intrusive. Finally, FiSC focuses on bugs that cause crashes, while SibylFuzzer identifies bugs that result in incorrect return values.

The JUXTA tool [15] also finds file system errors, but it focuses on finding semantic file system bugs which violate high-level specification rules or invariants. JUXTA relies on the fact that different file system implementations should all implicitly follow certain high-level semantics. It directly derives semantics from many file system implementations' source code all at once, then identifies bugs as deviant behavior using common semantics shared by multiple implementations. JUXTA's semantic bug coverage includes those that cause concurrency issues and crashes; in comparison SibylFuzzer can only detect a smaller, lower-level set of bugs.

Stateful fuzzing is not limited to only checking file systems. Ruiter and Poll [16] implemented a stateful fuzzer for testing TLS protocol implementations. They use the state machine learning tool LearnLib, which queries the target implementation through fuzzed test inputs. From the results of running these fuzzed test inputs, LearnLib can infer a state machine describing the implementation which is then checked for correctness. Checking state machines is currently only done manually, but the authors note that model checking could be added in the future.

5.2 Future Work

There are many improvements and extensions that could be done to increase SibylFuzzer's performance, control and coverage. More tests may be run in the future to better understand how varying the fuzzing parameters affects performance. On the analysis side, the results set presents many avenues of improved post-processing.

5.2.1 Additional System Calls

SibylFuzzer's system call coverage could be expanded to include *pwrite* and *pread*. Being able to fuzz calls which affect current users and groups (such as *add_user_to_group*)

would also open a new dimension of the state space for exploration.

5.2.2 Additional Syscall Distributions

Neither of the two system call distributions I tested is truly representative of real-life usage. Monitoring the daily system call usage for an average user's Linux file system would produce a proper real-life distribution for testing. Using smoothed or sharpened versions of previously-tested distributions could reveal correlations between specific system calls and model-implementation disagreements.

5.2.3 Inserting Fuzzed System Calls Without Truncation

SibylFuzzer currently truncates input test scripts so that the fuzzed system call always ends up as the last in the sequence. This was initially done so that model-implementation disagreements would always be triggered by a fuzzed system call. However, truncation limits the probability of fuzzing longer test scripts that correspond to more complex states. Therefore it would be a good idea to try running SibylFuzzer without truncation, so that fuzzed system calls can be inserted in the middle of the input test script.

5.2.4 Reusing Input Test Scripts

Right now SibylFuzzer uses the output fuzzed test scripts of the previous iteration as input to the current one. If SibylFuzzer also reused some or all of the previous input test scripts, it could dramatically increase its coverage per iteration. This would also allow SibylFuzzer to run for more iterations without fear of having the set of input test scripts eventually shrink to zero.

5.2.5 Expanded Fuzzing Parameters

In general, fuzzing parameters should be expanded and tested. Introducing more specific fuzzing parameters instead of shared parameters would allow for more fine-grained control and variation across experiments - for example, one might not want

the same distribution of directory and file permissions, therefore *mkdir* and *open* should have separate probability distributions for generating permissions rather than the single *perm_distribution* parameter. Performing more test trials with varying *syscall_distribution* values would also reveal which distributions are most efficient for finding potential bugs.

5.2.6 Dynamic Fuzzing Parameters

A useful extension to SibylFuzzer would be adding the option to dynamically calculate α depending on the current test script length. Then SibylFuzzer may have a greater probability of doing random insertion when the script length is small; as more and more fuzzed system calls are added, SibylFuzzer becomes more and more likely to append to the end in order to preserve the increasingly complex file system state that has been built up.

5.2.7 Excluding Known SibylFS Misbehavior

A sizable majority of errors initially found by SibylFuzzer turned out to be caused by known SibylFS misbehavior. Since these errors are triggered by only two types of system calls (*stat "/"* and *rename X "/"*), SibylFuzzer could be improved by intentionally never fuzzing any calls which fit one of the two types. This would save time when analyzing the checked traces for true errors.

5.2.8 Grouping Similar Erroneous Traces

In the previous chapter it was briefly mentioned how erroneous traces with different final system calls and error messages were actually triggering the same error. An automated tool that could group traces by error would be immensely useful. This tool would require knowledge of how to identify previously-encountered errors, and be able to detect patterns among ungrouped traces. Identifying similarities across different traces is not obvious, but sometimes there are hints such as presence of a common system call (as in the case of the *opendir* error).

5.2.9 Modularity and Compatability

Implementing SibylFuzzer required adding code to the SibylFS checker and model, as well as editing the model Makefile to ignore its original Lem source code. These changes compromise modularity and compatibility with future versions of SibylFS. There are a couple potential fixes. Firstly, spinning off SibylFuzzer as a completely separate module independent of the checker would greatly increase modularity. Secondly, incorporating the model-level utility methods into the original Lem code removes the need for Makefile edits.

Bibliography

[1] Barton P. Miller. Fuzz testing of application reliability. `http://pages.cs.wisc.edu/~bart/fuzz/`. Accessed: 2016-09-01.

[2] J.E. Forrester and Barton P Miller. An empirical study of the robustness of windows nt applications using random testing. *Proceedings of the 4th conference on USENIX Windows Systems Symposium - Volume 4*, 2000.

[3] Barton P. Miller et. al. An empirical study of the robustness of macos applications using random testing. *Proceedings of the 1st international workshop on Random testing*, 2006.

[4] A. Takanen, J. DeMott, and C. Miller. *Fuzzing for software security testing and quality assurance.* Artech House, 2008.

[5] Dave Jones. Trinity. `http://codemonkey.org.uk/projects/trinity/`. Accessed: 2016-09-01.

[6] R. Kaksonen. A functional method for assessing protocol implementation security. Technical Research Centre of Finland. Licentiate thesis.

[7] R. McNally, K. Yiu, D. Grove, and D. Gerhardy. Fuzzing: The state of the art. DSTO Defence Science and Technology Organisation. Accessed: 2016-09-01.

[8] American fuzzy lop. `http://lcamtuf.coredump.cx/afl/`. Accessed: 2016-09-01.

[9] T.R. Leek, G.Z. Baker, R.E. Brown, M.A. Zhivich, and R.P. Lippmann. Coverage maximization using dynamic taint tracing. Technical Report 112 for Lincoln Laboratory.

[10] G. Banks, M. Cova, V. Felmetsger, K. Almeroth, R. Kemmerer, and G. Vigna. Snooze: Toward a stateful network protocol fuzzer. *Proceedings of the 9th international conference on Information Security*, 2006.

[11] T. Ridge, D. Sheets, T. Tuerk, A. Giugliano, A. Madhavapeddy, and P. Sewell. Sibylfs: formal specification and oracle-based testing for posix and real-world file systems. Symposium on Operating Systems Principles 2015.

[12] H. Chen, D. Ziegler, T. Chajed, A. Chlipala, M. F. Kaashoek, and N. Zeldovich. Using crash hoare logic for certifying the fscq file system. *Proceedings of the 25th ACM Symposium on Operating Systems Principles*, 2015.

[13] J. Corbert. A story of three kernel vulnerabilities. `https://lwn.net/Articles/538898/`. Accessed: 2016-09-01.

[14] T. Yang et al. Using model checking to find serious file system errors. *ACM Transactions on Computer Systems 24*, 2006.

[15] C. Min, S. Kashyap, B. Lee, C. Song, and T. Kim. Cross-checking semantic correctness: The case of finding file system bugs. Symposium on Operating Systems Principles 2015.

[16] Joeri de Ruiter and Erik Poll. Protocol state fuzzing of tls implementations. 24th USENIX Security Symposium, 2015.