Scalable Testing of File System Checkers

João Carreira[†§], Rodrigo Rodrigues[†], George Candea[‡], Rupak Majumdar[†]

[†]Max Planck Institute for Software Systems (MPI-SWS),
[§]Instituto Superior Técnico, and [‡]École Polytechnique Fédérale de Lausanne (EPFL)

Abstract

File system checkers (like e2fsck) are critical, complex, and hard to develop, and developers today rely on hand-written tests to exercise this intricate code. Test suites for file system checkers take a lot of effort to develop and require careful reasoning to cover a sufficiently comprehensive set of inputs and recovery mechanisms. We present a tool and methodology for testing file system checkers that reduces the need for a specification of the recovery process and the development of a test suite. Our methodology splits the correctness of the checker into two objectives: consistency and completeness of recovery. For each objective, we leverage either the file system checker code itself or a comparison among the outputs of multiple checkers to extract an implicit specification of correct behavior. Our methodology is embodied in a testing tool called SWIFT, which uses a mix of symbolic and concrete execution; it introduces two new techniques: a specific concretization strategy and a corruption model that leverages test suites of file system checkers. We used SWIFT to test the file system checkers of ext2, ext3, ext4, ReiserFS, and Minix; we found bugs in all checkers, including cases leading to data loss. Additionally, we automatically generated test suites achieving code coverage on par with manually constructed test suites shipped with the checkers.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification — *reliability, validation*; D.4.5 [*Operating Systems*]: Reliability — *verification*

Keywords testing, file system checker, symbolic execution

1. Introduction

Modern storage systems are complex and diverse, and can fail in multiple ways, such as physical failures or software or firmware bugs [10, 12, 13, 15, 16]. For example, Bairavasundaram et al. have observed $400,000$ instances of checksum mismatches (i.e., instances when reads returned incorrect data) in a period of 41 months, in a set of 1.53 million disks [2]. While many failures manifest as system crashes, the more pernicious bugs in the storage stack can lead to the corruption of file system structures, where there is no immediate observable "crash," but key system invariants are violated, leading to an eventual crash or data loss many operations later.

Owing to the reality of storage failures, developers of file systems ship a recovery tool, known as a file system checker, with each file system. The file system checker is responsible for checking the integrity and consistency of the on-disk data structures of a specific file system and, in case of data corruption, for recovering the disk data to a "consistent" state usable by the file system code.

Although file system checkers are an essential tool, their recovery behaviors are complex, hard to get correct, and hard to test [8, 9]. Developers have to consider potentially every possible corruption, every disk state, and low-level language subtleties. Moreover, the correct behavior of a file system checker is usually poorly specified. While, in principle, one can formulate the file system invariants in a declarative language [9], in practice, the correctness of file system checkers is checked with respect to a test suite provided by the developer, which checks for specific violations of the intended behavior. Developing good test suites requires significant manual effort, yet may still end up testing only a small fraction of the possible behaviors. In particular, there may be bugs in the recovery code for corruption and recovery scenarios that are not tested (and perhaps not envisioned when designing the test suite).

We present an automatic tool-supported methodology for the systematic testing of file system checkers without manual specifications or system invariants or test suites. Our methodology checks for two properties of the checker: *consistency* and *completeness*. Consistency is the property that the output of the checker and the final disk state are coherent, i.e., an indication of success or failure to recover a disk should leave the disk respectively in a consistent or irrecoverable state. Completeness is the property that the checker recovers a maximal amount of information. The key technical problem is that determining the notion of consistent state and complete recovery without explicit specifications of the behavior of the file system and the file system checker is hard. Our methodology addresses this problem based on two insights.

First, even though we do not have explicit invariants, we can use the checking logic in the file system checker as a proxy for correctness. This is because the logic that verifies whether a disk is consistent is simpler and more mature than the recovery logic, and therefore we can use this verification logic to determine whether the outcome of a recovery matches the actions and the degree of success that the recovery reported. To give a simple example, when a first run of the checker returns that the disk has been recovered to a consistent state, we can then use the file system checker's own consistency verification logic to test whether this state is indeed consistent. Thus, we avoid the problem of inferring a specification of consistency by focusing on the *internal* consistency of the checker.

Second, in order to assess the completeness of a checker, we run checkers for different file systems on semantically equivalent file system images and compare results for the same corruption of this state. Different checkers share a common specification for the recovery of file systems. Even though specific on-disk representations of the file system can vary, checkers operate on semantically equivalent data structures and have access to the same information, and so they should be able to perform the same recoveries. Consequently, differences in the recoveries performed by different file system checkers on the same semantic file system under the same corruption scenario are likely due to incompleteness in one of the checkers. Thus, we avoid the problem of inferring a specification of completeness by focusing on *relative* completeness between different checkers.

In summary, both parts of our methodology reduce the problems of checker consistency and checker completeness to the problem of state space exploration of checker *compositions*. In the first case, we compose the same checker sequentially and compare results; in the second case, we compose two or more different checker implementations in parallel and compare results.

We developed SWIFT (Scalable and Wide Fsck Testing), a system that implements the aforementioned methodology and allows developers to test file system checkers. SWIFT uses S^{2}E [6], a full-system symbolic execution engine for binaries, as a basic path exploration tool to explore the execution of file system checkers under different data corruption instances. SWIFT is able to systematically explore the recovery code of file system checkers and find bugs that are hard to detect with classic testing methodologies.

We used SWIFT to test e2fsck (ext2–4), reiserfsck (ReiserFS) and fsck.minix (Minix). SWIFT found bugs in all three checkers, including cases that lead to loss of data and incorrect recoveries. Additionally, SWIFT achieves code coverage on par with manual tests, but with much less effort.

The rest of the paper is structured as follows: §2 surveys the related work, §3 provides background on symbolic execution and S^{2}E, §4 presents our methodology for testing file system checkers, §5 describes SWIFT, the system that instantiates this methodology, §6 presents our evaluation of SWIFT, and §7 concludes.

2. Related Work

File System Testers. Model checking and symbolic execution have already been used in the past to systematically test file systems. EXPLODE [17] is a tool for end-to-end testing of real storage stack systems based on enumerative model checking. EXPLODE exhaustively explores all states of a storage system by systematically invoking file system operations, injecting crashes in the system, exploring all choices of a program at specific points, and checking each state against specific properties of the correctness of the system. The developer provides a *checker* program, which is responsible for systematically invoking system specific operations, injecting crashes and checking assertions on the system state. FiSC [19] applies this idea to file systems, and checks disk data against a model describing the data that is expected to be stored. In contrast to SWIFT, these tools are not symbolic, do not explore the recovery behavior of file system checkers, and require a human-provided specification to check the correctness of the code.

Yang et al. proposed a system for testing disk mount mechanisms [18]. During the mount of a disk, file system code must check if the disk data is consistent and if all the invariants of the file system hold. Since these invariants can be complex and depend on a high number of values, mount code is hard to test and exercise. In order to test this code, the authors proposed using symbolic execution to systematically explore the code of the *mount* function of different file systems. As multiple paths are explored, pluggable pieces of code (property checkers) check for specific bugs, such as null pointer accesses. However, bugs that lead to latent file system data or metadata errors may not be caught.

In contrast to all of the above, we focus on testing file system checkers, and we look for problems that manifest in subtle ways, such as latent bugs, without requiring hand-written specifications of correctness properties. This is achieved by looking for inconsistencies that are internal to the checker or incompleteness relative to other checkers.

Leveraging code diversity. The idea of using different implementations of similar systems has been proposed in the context of replicated file systems for tolerating correlated faults [1, 14], and also for verifying the equivalence of coreutils tools [4]. In our work, we leverage this idea to test one aspect of the specification of file system checkers, namely the completeness property.

Corruption model Bairavasundaram et al. proposed *type aware pointer corruption* (TAPC) to explore how file systems react to corrupt block pointers [3]. The proposed methodology consists of creating different instances of block pointer corruptions and observing how the file system behaves in those cases. The systematic introduction of data corruption faces an explosion problem: it is not possible to explore all possible cases of corruption. TAPC addresses this problem by assuming that the behavior of the system being tested only depends on the type of the pointer that has been corrupted and on the type of the block where the pointer points to after corruption. SWIFT makes use of a corruption model, and TAPC could have been used for this purpose. However, we decided to use different models that enable a larger corruption space. Some of the bugs that SWIFT uncovered cannot be found using TAPC.

Declarative Specifications Gunawi et al. have proposed SQCK [9], a file system checker based on a declarative query language. The reason for using a declarative language is that declarative queries are a good match for file system cross-checks and repairs. While this is a promising approach to improve the reliability of file system checkers, it requires rewriting the checker code itself (and moves inconsistency and incompleteness issues to the declarative specification). In contrast, we aim to improve the reliability of the large code base of existing file system checkers.

3. Symbolic Execution and S^2E

SWIFT relies on symbolic execution to systematically explore the recovery behavior of file system checkers. Symbolic execution [7, 11] is a technique that allows the exploration of non-redundant paths of code. To achieve this, input data is initially marked as "symbolic," that is, instead of assigning concrete values to inputs, the symbolic execution engine represents each input with a symbolic constant that represents all possible values allowed for that input.

The symbolic execution engine executes the program on the symbolic input, and maintains a symbolic store that maps program variables to symbolic expressions defined over the symbolic inputs. The symbolic store is updated on each assignment statement. For example, if the program has an input x that is represented using the symbolic constant α_x, and the program performs the assignment $y = x + 5$, the symbolic store maintains the mapping $[x \mapsto \alpha_x, y \mapsto \alpha_x + 5]$.

In addition to the symbolic store, the symbolic execution engine maintains a path constraint that represents constraints on the inputs in order for the current path to be executed. Initially, the path constraint is $true$, indicating that there are no constraints on the inputs. When a conditional instruction that depends on symbolic data is executed, the system execution may be forked in two feasible paths. In the then-branch, the condition is marked as true (by conjoining the conditional with the current path constraint), and in the else-branch, it is marked as false (by conjoining the negation of the conditional with the current path constraint). For example, if the conditional is $y > 0$ and the symbolic store is as above, the then-branch of the conditional adds the constraint $\alpha_x + 5 > 0$ to the path constraint and the else-branch adds the constraint $\alpha_x + 5 \leq 0$. This constrains the input values that can cause execution to go down the then- or else-branch. As the code in each path is explored and execution is forked, input data is progressively constrained along each individual path and paths through the system are explored. When a path is fully executed, a constraint solver is used to find concrete inputs that satisfy the path constraints that were collected along that path. Any concrete input satisfying the path constraint is guaranteed to execute the program along that path. Symbolic execution continues until all paths of the program are covered (or some user-provided coverage goal or resource bound is met).

SWIFT uses S^2E [6], an in-vivo symbolic execution engine. Testing using in-vivo symbolic execution enables us to use a real software stack, without having to model the environment in which the target code runs. For improved scalability, S^2E combines symbolic execution with the concrete execution of certain sections of the code, by converting symbolic data into concrete data upon entering those sections, and judiciously converting back to symbolic upon leaving, in such a way that execution consistency is preserved. S^2E operates directly on binaries, and therefore it does not require access to the source code of the software being analyzed.

4. Methodology

4.1 Overview

Our methodology checks for two aspects of the specification of a file system checker. The first aspect, which we call *consistency*, is the property that the output of the checker and the final disk state are coherent, namely that a successful recovery should not lead to a corrupt disk, and a run of the checker that ends with an error code should produce a disk that is consistent with the error code. The second part, which we call *completeness*, is the property that a checker should recover as much information as reasonably possible from a corrupt disk.

If there was a full specification of the layout of a file system, one could check consistency by exploring the set of possible behaviors of the checker and verify that, when run from an arbitrary disk state, the checker either returns success and a disk state satisfying the specification, or returns an error code with the disk state satisfying the conditions implied by the error code. Similarly, if there was a full specification of recovery behavior, one could check completeness by verifying that the checker conforms to the recovery specification. In practice, file system layouts and file system checkers do not come with full specifications, therefore the developers of the file system checker construct a partial specification in the form of a test suite.

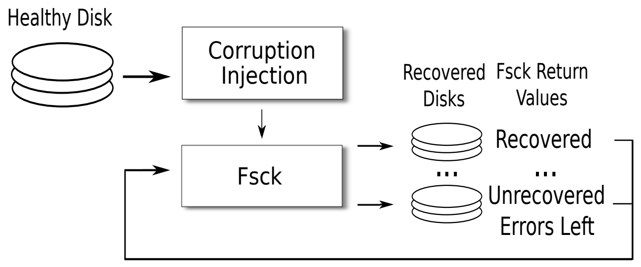

Figure 1. Finding fsck recovery inconsistencies.

Our goal is to provide a methodology and a tool to *systematically* test the implementations of file system checkers, *without* manually provided behavioral specifications and with minimal manual effort. Our methodology has three parts.

First, we expect a *corruption model* that systematically injects data corruption to disks. Our methodology is parametrized by a user-provided corruption model, and the depth and complexity of the corruption model influences the coverage of recovery behaviors. This is, in essence, a specification of what failures are of interest. In § 5 we present two methods for the systematic and symbolic injection of data corruption.

Second, we perform the systematic exploration of checker code (for each given corruption scenario) using symbolic execution [4–7]. This enables us to systematically explore a large number of paths through the checker code, and, when problems are found, to generate an input disk image that triggers the problematic path.

Third, we check the consistency and completeness of checkers using a *composition* of checkers, that is, by using the file system checker code itself as a specification, as follows.

We check consistency using sequential self-composition. This means that we take a corrupt disk and perform two consecutive runs of a checker on this disk. If the outcomes of the two runs are not coherent with each other, then we can be sure that some erroneous behavior occurred. This happens, for instance, if the second run needs to recover a disk that was successfully recovered in the first run. Thus, we can check for inconsistent behaviors by running the checker twice in succession and comparing the two results.

We check completeness using parallel composition of multiple file system checkers. That is, we run different implementations of file system checkers (possibly for different file systems) and compare their results when started with the same semantic state (i.e., a logical representation of the file system state that abstracts away the differences in the data and metadata physical layout) and the same data corruption. While different file systems may be represented differently on-disk, we use the fact that one can relatively simply specify the common data structures used in a file system (inodes, data blocks, etc.) and describe a corruption purely as mutations on these data structures. Assuming that correlated bugs are less likely, this detects sources of incompleteness where one checker cannot recover some information but the other can.

First Execution Result	Second Execution Result
Disk is consistent	Disk is consistent
Uncorrected errors left	Uncorrected errors left
Recovered but uncorrected errors left	Uncorrected errors left
All errors corrected	Disk is consistent
Operational Error	Operational Error

Table 1. List of correct results fsck should output when run consecutively on the same disk.

4.2 Checking Consistency

Our approach to check consistency, depicted in Figure 1, works as follows.

A healthy disk is fed to our testing system. Using a corruption model, we systematically introduce corruption (in the form of unconstrained symbolic values) in this disk. We call this a symbolic corrupt disk, i.e., a disk that contains symbolic values, thus encoding any possible values for the bytes that are marked as symbolic.

Next, we symbolically execute the checker under test to recover from this corruption, if possible. This leads to multiple recovered disks and, for each recovery, the checker returns a numeric value indicating whether it was able to recover the disk and, if so, whether the recovery was partial or complete. Afterwards, we run the checker a second time on the recovered disk to determine whether the consistency property is met.

To exemplify how the second run checks the correctness of the first one, if the first checker execution completely recovers the disk data, the second run must indicate a fully consistent disk. If not, then it is likely that the recovery performed in the first execution is not correct, e.g., a set of recoveries performed in the second execution were missed by the first one, or the first recovery has introduced inconsistencies. A complete list of the expected outcomes of the sequential composition of checker executions is shown in Table 1. A deviation from these outcomes signals the existence of a bug in the checker; this process does not generate any false positives. Once a positive is flagged, the developer obtains a bug-triggering input that (s)he can use to manually inspect the cause of the bug. Note that we cannot exclude that the bug can be in the checking logic to verify if recovery is needed (that implies the specification we use).

The effectiveness of this phase in terms of not having any false negatives, relies on two assumptions: first, that the code for checking whether recovery is needed is correct, and, second, that an execution that outputs that the disk is initially consistent does not modify the file system state. We expect these assumptions to be commonly met in practice because of the following intuition: Since the code that checks

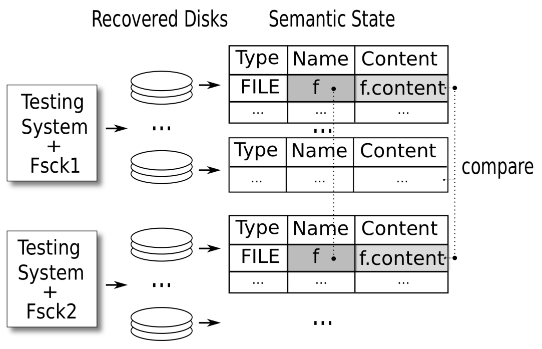

Figure 2. Assessing the completeness of recoveries.

Field	Description
Number of blocks	Number of used blocks
Number of free blocks	Number of unused blocks
Block size	Size of the block used by the file system
State of the file system	Value indicating if the file system is valid or not
Magic file system value	Value indicating which file system and version are being used
Symbolic link data	Path to the file that the symbolic link points to
Directory entries	< Name, Inode number> pairs
Data block pointers	Pointers stored in data blocks and inodes
VFS file attributes	File attributes exposed by the VFS layer (see Table 3)

Table 2. List of fields shared by modern file systems. The term "symbolic" is used above in the sense of symbolic (soft) links in file systems, which differs somewhat from the use of this term in the rest of the paper.

whether on-disk file system data structures need recovery is frequently executed, and also given its relative simplicity, we expect this checking logic to be mature and to be a good specification of a consistent disk. In contrast, recovery code often depends on intricate failure conditions and complex correction decisions, and thus this code is more likely to be fragile.

4.3 Checking Completeness

Even though the disk recovered by a file system checker may be consistent, a checker may not have done as good a job as it could have, i.e., may not have recovered the disk to a state that is as close to the pre-corruption state as possible. For instance, data or metadata may have been lost in the process, files or directories may have been relocated, or the checker may have not used all the available data redundancy during the recovery process and thus missed opportunities for recovery.

In order to assess the completeness of file system checker recoveries, we compare the contents from different disks resulting from the execution of different file system checkers. The idea is that file system checkers should be able to arrive at the same logical contents for the file system after recovering from corruption in equivalent fields. Note that it is only possible to use this strategy if file systems have common data structures that can be used to produce equivalent corruption scenarios. In practice, we expect this to be feasible for a large class of file systems that have similar organization strategies and use similar data structures, e.g., inode-based designs.

The second part of the methodology is depicted in more detail in Figure 2. We start with two or more healthy disks, formatted with different file systems and containing the same logical data (i.e., having identical file system entries). We use the corruption model to introduce semantically equivalent corruptions in each disk image. To ensure semantically equivalent corruptions, the corruption model introduces symbolic values in fields which are common to both disk data structures (see Table 2).

For each corrupted disk, we execute the corresponding file system checker symbolically to recover from the injected corruption. Then, for each recovery path and the corresponding recovered disk, we assemble a logical representation of the data stored in the disk (i.e., the set of file system entries it contains) using file system operations. Finally, we perform an all-to-all comparison among the logical representations of the recovered data in the various file systems and various recovery paths (but for the same corruption). Mismatches in the logical data contained in different recovered disks indicate that the different file system checkers performed different recoveries, and might be caused by a bug.

Unlike checking consistency, this approach is subject to false positives. Since different file systems may have different levels of on-disk redundancy, some file system checkers may be able to recover more information than others. Thus, the developer needs to manually inspect the bugs found by this part of our methodology in order to separate mismatches caused by a bug in the checker from those resulting from the file system's design.

Moreover, our approach does not pinpoint which of the file system checkers whose recoveries are being compared has a bug. However, we have found it easy to discover through manual inspection which of the file system checkers has performed an incorrect recovery. It is possible that a majority voting mechanism could be used to isolate the buggy file system checker from a set of three or more file system checkers being compared (akin to BASE [14] and EnvyFS [1]), but we did not explore this approach.

5. SWIFT

We now describe SWIFT (Scalable and Wide Fsck Testing), a testing tool that implements our methodology. As illustrated in Figure 3, SWIFT has three phases: processing a description of which fields are considered to be sources of corruption, systematically exploring paths in the checker code, and using these path traces to find inconsistent recoveries and instances of data loss or missed recovery opportunities.

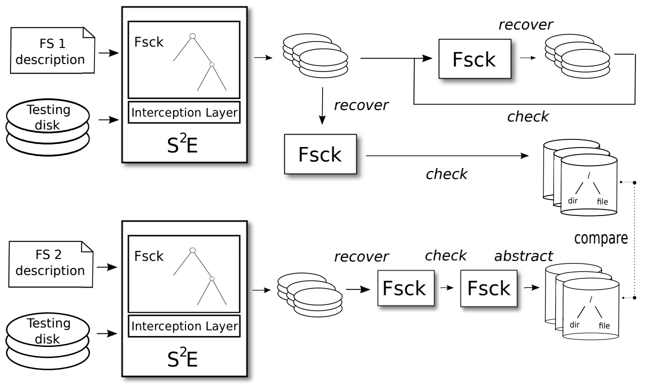

Figure 3. Overview of SWIFT's testing process.

5.1 System Overview

Checking consistency. For this phase, the input to SWIFT is a file system and file system checker under test, an initial healthy disk, and a description of the corruption model. An example of a corruption model is a list of file system fields that can be mutated. SWIFT introduces a symbolic input according to each entry in the corruption model (e.g., in one file system field at a time), and runs S^{2}E to explore file system checker executions for each symbolic input. The symbolic execution of these paths will lead to checking and possibly recovering from the corruption that was injected. Moreover, each of these paths leads to S^{2}E generating a disk trace — a concrete corrupt disk that can be used as an input to exercise that path in the checker code. Once corruption in all fields has been explored, the disk traces generated by the path exploration phase are used to test the consistency of the checker. To this end, SWIFT uses the file system checker to recover each concrete disk in the set of disk traces two consecutive times. On each recovery, SWIFT records the value returned by the fsck. At the end, SWIFT checks these values, looking for incorrect recoveries. Sequences of executions whose return values deviate from the set of "good" cases that are listed in Table 1 are marked as buggy cases.

Checking completeness. For checking completeness, the input to SWIFT is two or more file systems and their corresponding checkers, a semantic description of the initial file system data, and a description of the corruption model. The corruption model marks fields shared by the file systems as symbolic. For each symbolic input, SWIFT runs S^{2}E to generate disk traces for each file system checker. For each field that is analyzed (as described in Section 4.3), SWIFT runs the recovery code on each disk in the set of disk traces (corresponding to the various recovery paths) and builds a logical representation of the data contained in it using the data and metadata accessible through VFS operations (see Table 3). Then, SWIFT compares all the logical disk contents across all the file system checkers being compared and all recovery paths, but for the same corrupted field. Finally, mismatches between these representations are flagged. In order to make the logical comparison agnostic to the order in which these entries were returned by the VFS calls, SWIFT sorts the directory entries that are read.

Field	Description
Path name	Path to the file
Type	Type of the file (e.g., directory)
Permissions	Read, write and execute permissions of the file
Number of hard links	Number of links to this file
User ID of owner	Identification number for the file's user owner
Group ID of owner	Identification number for the file's group owner
Total size	Size of the file
File content	Content of the file (in case of regular files)

Table 3. List of files attributes accessible with VFS.

Output. Since SWIFT starts from a real disk to exercise recovery code, and the output of S^{2}E is a set of concrete disks that lead to different execution paths, SWIFT can output, upon finding a potential bug, the corresponding concrete disk that triggers the identified problem. Developers can then use this disk image to replay the bug and manually find its root cause.

5.2 Initial Disk

To explore diverse recovery behaviors, we have to create an initial disk that uses many features of the file system. Otherwise, the execution of the checker under test will be confined to a small subset of recoveries. We developed a generic file system content for the testing disk, taking into account the data structures that specific file systems use to organize their data and metadata. This disk not only contains all types of entries, but also exercises specific layouts that stem from certain file system design features. For instance, our testing disk contains 100 small files in a directory to force specific file systems (e.g., ReiserFS) to use a multi-level tree with internal and leaf nodes. The resulting testing disk layout is depicted in Figure 4 and described in Table 4.

5.3 Corruption Model

The set of possible disk corruptions, file system structures, and file system fields is prohibitively large. Fortunately, many of the possible corruptions exercise the same fsck recovery code mechanisms (e.g., the same corrupt field in inodes of the same type is recovered in the same way).

To systematically explore corruption instances, and given that we are not aware of any existing studies for real disk corruption, we experimented with two simple methods for injecting data corruption in a disk. In the first method, the corruption that is injected in the disk is based on the fields of the data structures used by file systems. In the second method, the injection of corruption is guided by existing file system checker test suites. While we believe that more so-

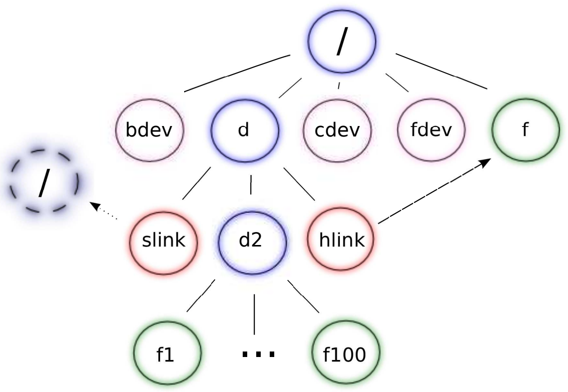

Figure 4. Semantic representation of the data stored in SWIFT's generic test disk. The meaning of the nodes is explained in Table 4.

Item	Type	Description
/	Directory	
/bdev	Block device	
/d	Directory	
/cdev	Char device	
/fdev	FIFO Device	
/f	Regular File	File with 278 1Kb blocks
/d/slink	Symbolic link	Link to root directory
/d/hlink	Hard link	Link to /f
/d/d2	Directory	
/d/f1...f100	Regular file	100 empty files

Table 4. Description of the contents of the test disk.

Disk Data Structure	Fields
Superblock	All fields
Group Descriptor	All fields
Inode Bitmap	All bits of used blocks and one bit of an unused block
Block Bitmap	All bits of used blocks and one bit of an unused block
Metadata	For each inode type: 1. all non-pointer fields 2. one direct block pointer 3. one indirect block pointer 4. one double indirect block pointer
	All fields of reserved inodes
Data	Directory entries (root and one other directory): 1. Directory entry (e.g., inode number, file name) to itself 2. Directory entry to parent 3. Directory entry to another
	Data pointers: 1. One direct pointer stored in an indirect block 2. One indirect pointer stored in a double indirect block
	Symbolic links: 1. One symbolic link path name
Journal Superblock	All fields
Journal Commit Record	All fields
Journal Revoke Record	All fields

Table 5. Corruption model. This table shows the fields and respective data structures considered to be possible sources of corruption.

phisticated models could help the productivity of our testing tool, we leave the task of finding such models to future work. We also note that even with these simple models our methodology is effective.

5.3.1 Corruption of Fields

Most file system checker recovery mechanisms target specific file system fields and data structures. Thus, we have developed and implemented a corruption model that is aware of the position of file system fields in the disk and that can selectively corrupt them. While developing this model we tried to exhaustively consider the different types of structures used by modern file systems. Moreover, we analyzed the list of recoveries performed by different file system checkers, namely e2fsck and xfs_fsck, whose description is provided by developers along with the source code, and asked the question "Is our corruption model able to exercise these recovery mechanisms?" We eventually converged onto a model that encompasses a wide set of corruption cases.

Table 5 shows the fields our model considers as possible sources of corruption. The first column describes the on-disk file system data structure, while the second column identifies the specific field(s) in that structure. This corruption model targets inode-based file systems. Thus, this model considers the most common structures in this family of file systems, such as superblock, inode bitmap, block bitmap, metadata (inodes) and data (data blocks). We additionally considered some more modern features (e.g., related to journaling).

We consider all the fields in each one of these structures as possible sources of corruption. However, in order to achieve scalability, we apply some simplifications. In the case of bitmaps, our model only considers one bit from the set of bits indicating unused blocks or inodes. Our model considers only one pointer of each type (direct, indirect, and double indirect), and inode fields are considered once for each type of inode. Finally, we only consider corruption in one field at a time, similar to single-event upset models in VLSI-CAD. While silent data corruption may affect several fields of the file system disk data structures, we believe that considering only one field at a time is a good compromise between the large number of possible corruptions and the efficiency of our testing methodology. Additionally, in this way, the developer is able to direct the execution of the file system checker towards specific recovery mechanisms.

Since our corruption model targets inode-based file systems, it can be used to test most of the currently used file system checkers. However, some file system checkers may not implement some of the structures outlined, or may use additional structures not captured by it. In particular, many modern file systems provide more advanced features, such as indexed directory entries. To allow for capturing these

```
<FileSystem name="Ext4">
  <AddStructure name="Superblock">
    <Field name="s_inodes_count" position="0h"
                                      size="4"/>
    <Field name="s_blocks_count_lo" position="4h"
                                      size="4"/>
    <Field name="s_r_blocks_count_lo" position="8h"
                                      size="4"/>
    <Field name="s_free_blocks_count_lo" position="Ch"
                                      size="4"/>
  </AddStructure>
  <AddStructure name="Inode">
    <Field name="i_mode" position="0h" size="2"/>
    <Field name="i_uid" position="2h" size="2"/>
    <Field name="i_size" position="4h" size="4"/>
    <Field name="i_atime" position="8h" size="4"/>
    <Field name="i_ctime" position="Ch" size="4"/>
  </AddStructure>
  <AddStructure name="DirEntry">
    <Field name="inode_nr" position="0" size="4"/>
    <Field name="rec_len" position="4" size="2"/>
    <Field name="name_len" position="6" size="2"/>
    <Field name="entry_name" position="8" size="1"/>
  </AddStructure>
  <AddTest>
    <TestStructure name="Superblock"
          startPosition="1024"/>
    <TestStructure name="Inode" startPosition="49300h"
                          description="journal inode"/>
  </AddTest>
</FileSystem>
```

Figure 5. Test description for an ext4 disk. Some structures and fields are omitted for clarity.

```
char* disk_data; // disk data
char* test_disk; // disk file path
int symb_pos; // position of the symbolic field
int symb_size; // size of the field
char buffer[4096];

int open(const char * pathname, int flags, mode_t mode) {
    int fd = _open(pathname, flags, mode);

    // not opening the test disk
    if (strcmp(pathname, test_disk) != 0)
        return fd;

    isTestDisk[fd] = 1;
    s2e_make_symbolic(disk_data + symb_pos,
                              symb_size, "disk");
    return fd;
}

ssize_t read(int fd, void *buf, size_t count) {
    // check if we are reading our test disk
    if (isTestDisk[fd] == 1) {
        // get seek position
        off64_t seek = _lseek64(fd, 0, SEEK_CUR);
        // get number of bytes read
        int readRet = _read(fd, buffer, count);
        memcpy(buf, disk_data + seek, readRet);
        return readRet;
    } else {
        // delegate operation to the 'real' file system
        return _read(fd, buf, count);
    }
}
```

Figure 6. Simplified interception layer for the `open` and `read` file system operations.

features, developers can easily extend the model by editing the corresponding XML-based description of the possible sources of corruption and their respective positions on the disk. An example of such a description is depicted in Figure 5. In this description, the developer defines disk data structures (e.g., an inode), using `AddStructure`. These structures contain file systems fields, their names, relative positions and sizes.

5.3.2 Corruption Using Fsck Test Suites

One limitation of the above corruption model is that it considers that corruptions do not span more than one file system field, which is important to keep the number of scenarios being tested within a reasonable bound. Ideally, we would like to additionally test more intricate cases of corruption that exercise more complex recovery mechanisms, but doing so in a scalable way is challenging.

To address this, we use the file system checker test suites to drive the exploration of recovery paths towards recoveries that are harder to exercise, while maintaining the scalability of our methodology. When available, test suites can be a good vehicle for exercising recoveries, because they make use of the knowledge of the developers about the file system, the checker and real data corruptions. In addition, test suites tend to be developed over a long period of time and, as they mature, they explore more intricate cases of data corruption.

In this work we focus on the test suite of e2fsck, since it was the only system we tested that provided a test suite along with the checker. In this suite, each test consists of one corrupt disk and the expected output from recovering it. Our corruption model uses the test suite in the following way. For each test, we run e2fsck on the test disk and record which bytes of the disk are changed during the recovery. Then, we run SWIFT, considering these bytes as possible sources of corruption, thus exploring multiple fault scenarios for the same set of bytes. As our results will highlight, this latter corruption model proved more effective than the individual field model in uncovering incorrect recoveries.

5.4 Generating Symbolic Input

To explore recoveries of on-disk data corruption, we invoke the file system checker inside S^{2}E on the testing disk. However, this execution must differ from a normal fsck execution, because the data that is read from the testing disk must be in memory and must contain a mix of concrete and symbolic values, the latter being marked through an invocation of S^{2}E's `s2e_make_symbolic` function.

To feed symbolic disk data to a file system checker in a transparent way, we have developed an interception layer, as depicted in Figure 6. This layer is compiled in Linux as a shared library and is used to intercept functions that handle file system operations provided by the *libc* library. The role of the layer is to load the test disk contents into memory and replace portions of the data read from disk with symbolic values by calling the `s2e_make_symbolic` function. Then, when the checker invokes a *read()* call, instead of accessing

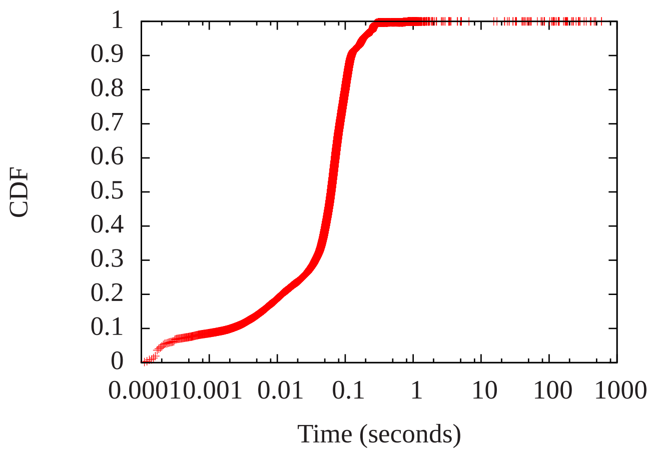

Figure 7. CDF of the solving time for a sample set of constraint expressions generated during our tests.

the disk contents, the mix of concrete and symbolic data is read from memory. The behavior of the *write()* method is not changed, since code running on S^2E can handle symbolic values transparently.

5.5 Improving Scalability: A Concretization Strategy

S^{2}E relies indirectly on a constraint solver to solve constraint expressions during the exploration of paths. Most of the constraint solver queries are issued in two situations: when a path is forked and when a symbolic value is concretized in order to simplify a symbolic memory address access.

Figure 7 shows the distribution of constraint solving times. While most of the queries can be solved by the constraint solver in less than 1 second, a small fraction of the queries takes a considerable amount of time. These queries can hurt the performance of the path exploration and slow down the rate of explored paths. In order to solve this problem, symbolic execution systems allow developers to set a timeout on the constraint solving process. This way, whenever a constraint solving query takes more than a specified amount of time, the constraint solving process is canceled and the respective path is discarded. This approach is far from ideal: discarding a path wastes the time that was spent executing the code that led to that path and solving previous constraint queries. Moreover, constraints that may be hard to solve by the constraint solver may also be harder to reason about by the developer, and thus can arguably be more likely to exercise corner cases of the file system checker.

In order to decrease the number of paths that are discarded this way, we introduce the following optimization: Whenever a constraint query times out during a path fork, we partially concretize the constraint expression of the path, i.e., find a set of values for a subset of the variables in the constraint expression of the path that satisfy these constraints. The idea is that, by concretizing a subset of the symbolic bytes of the path's constraint expression, we can simplify the constraint expression, thus transforming it into an expression that can be solved more efficiently.

The subset of bytes concretized can be chosen according to different strategies. One possibility is to choose the bytes according to the order in which they appear in the constraint expression. Another possibility is to concretize the symbolic bytes that occur more often in a constraint expression. Finally, a third strategy can pick the symbolic bytes to concretize that appear in arithmetic expressions.

In this work we only evaluate the first strategy. Our strategy allows the developer to decide the fraction of the bytes that is concretized, i.e., the developer has, at compile time, the ability to decide if the constraint expression should be fully or only partially concretized. We leave the evaluation of the remaining strategies to future work.

6. Evaluation

We used SWIFT to check the consistency and completeness of three file system checkers: e2fsck 1.41.14, reiserfsck 3.6.21 and fsck.minix 2.17.2. These file system checkers are used to check the integrity of ext2 through ext4, ReiserFS and the MINIX file system, respectively. Both e2fsck and reiserfsck are modern and reasonably sized file system checkers: $18,046$ and $8,125$ lines of code, respectively. Fsck.minix is a small but mature file system checker with $1,156$ lines of code.

To implement our first corruption model that tests the corruption of individual fields, we developed a test description for each one of these file system checkers, as described in Section 5.3.1. Table 6 summarizes the complexity of these three descriptions. For the second corruption model, SWIFT collects the bytes changed during the recovery of the disk that is provided by each test, as described in Section 5.3.2.

Fsck	**# Data Structures**	**# Fields**	**LoC**
e2fsck	13	233	323
reiserfsck	10	63	128
fsck.minix (V1)	4	28	63
fsck.minix (V2)	4	31	72

Table 6. Size of the test descriptions developed for each file system checker.

In order to execute the file system checkers being tested, we had to provide command-line arguments they support (as summarized in Table 7). An interesting case was reiserfsck, which provides one argument to check the consistency and three other arguments that enable different recovery mechanisms. As a result, to test the reiserfsck file system checker, we invoke it three times in the first execution, one for each command-line argument, to test all recovery mechanisms.

When testing individual field corruption, we let SWIFT perform the code exploration for 1,000 seconds for each field

Fsck	Bug Type	Description
e2fsck	Wrong return type	e2fsck incorrectly flags a disk as recovered in the first execution
e2fsck	Unrecoverable field is recovered	Corrupt *s_wtime* field is recovered despite being considered unrecoverable
e2fsck	Cloning of blocks	Wrong recovery leads to unnecessary cloning of blocks
e2fsck	Recovery fails to recover	Recovery of inconsistent resize inode leaves inode in an inconsistent state
e2fsck	Journal backup mismatch	Recovery may create mismatches between the journal inode and its backups
reiserfsck	Infinite Loop	Corruption in entry key leads to an infinite loop
reiserfsck	Segmentation fault	Corruption in the size field of a child pointer leads to a segmentation fault
fsck.minix	Data Loss	Wrong inode number in . entry of root inode makes all data unreachable
fsck.minix	Segmentation fault	Bug detecting double indirect pointer leads to segmentation fault
fsck.minix	Segmentation fault	Loop in the file system leads to an infinite loop and segmentation fault

Table 8. List of bugs found when testing the consistency of recoveries.

Command-Line Invocation	Description
e2fsck -fy test_disk	Performs all the possible recoveries
reiserfsck -fy –rebuild-sb test_disk	Recovers the superblock
reiserfsck -fy –fix-fixable test_disk	Recovers cases of corruption which do not involve the file system tree
reiserfsck -f –rebuild-tree test_disk	Recovers the file system tree
reiserfsck –check test_disk	Checks the consistency of the file system
fsck.minix -fa test_disk	Performs all automatic repairs
fsck.minix -f test_disk	Checks the consistency of the file system

Table 7. List of command-line arguments used while testing e2fsck, reiserfsck and fsck.minix.

indicated in the test description. When using the test suite, we configure SWIFT to use 1,000 seconds for each test. All the experiments run on an Intel 2.66GHz Xeon CPU using 48GB of RAM, and we run S^{2}E with 8 threads.

To understand what was the cause for incorrect recoveries, we manually analyzed as many buggy paths as time permitted.

6.1 Checking the Consistency of Recoveries

We start by reporting several bugs we found using each of the corruption models. Our findings are summarized in Table 8.

6.1.1 Corruption of individual fields

Overall, using the individual field corruption model, our tests found buggy paths in all three file system checkers. We manually analyzed a subset of the paths that SWIFT flagged as buggy. This analysis revealed 5, 2 and 3 bugs in e2fsck, reiserfsck and fsck.minix, respectively. These include instances of infinite loops and segmentation faults, even though SWIFT does not directly target this type of bugs.

To give an example of an inconsistent recovery that our methodology detected, in one case, when the disk contains two duplicate files, e2fsck outputs a message indicating that this error was found and fixed. However, e2fsck fails to rename one of the duplicate files in the first execution of e2fsck. Only a second execution fixes this problem. The reason for this bug is that the check that detects this inconsistency and the code that recovers it have different assumptions about the order of files in the same directory. While the former is always able to detect the duplicate files by using a hash of file names in a dictionary, the latter assumes that the files are ordered alphabetically in a directory, and thus fails to detect some instances of duplicate files.

Bug	Description
Deleted inode is treated as an orphan inode	e2fsck uses *s_wtime* and *s_mtime* to determine if an orphan inode release has been missed. Corruption in these fields can make e2fsck consider an inode as regular in the first execution and as an orphan inode in the second execution.
salvage_directory recovers file	e2fsck is unable to detect corruption in ext4's i_mode field. When recovering a file as if it was a directory, the recovery is unable to leave the disk in a consistent state.

Table 9. Some of the bugs found by SWIFT using the corruption model based on e2fsck's test suite.

Another interesting finding is that e2fsck does not check the values stored in the main superblock against the values stored in the backup superblocks before using them. This makes e2fsck blindly trust corrupt values, which leads to incorrect recoveries.

6.1.2 Corruption Using Test Suites

Using SWIFT along with the test suite of e2fsck, we found more than 6,000 paths that lead to incorrect recoveries, compared to less than 200 with the previous corruption model (though this model leads to about twice the exploration time). While this is an encouragingly positive result, especially since this part of the methodology does not lead to false positives, the same bug may trigger incorrect recoveries in tens or hundreds of paths. We were unable to manually analyze *all* these recoveries to understand the bugs that lead to them, and so we show in Table 9 only a partial list of bugs that we found during our tests.

The first bug was found while exploring e2fsck recoveries on a disk containing orphan inodes. Orphan inodes are inodes that are deleted while the corresponding file is still open. In order to delete the file once it is closed, ext4 stores the inode in a special linked list, the orphan inode list. When e2fsck is invoked, it checks for the presence of inodes in this list. In order to identify inodes in the orphan inode list, e2fsck relies on the *s_wtime* and *s_mtime* fields. When one of these fields is corrupt, e2fsck may wrongly identify a normal inode as orphan. When e2fsck is invoked the first time, e2fsck recovers a disk containing a list of orphan inodes successfully. When e2fsck recovers the disk, it overwrites the previously mentioned time fields. As a result, when e2fsck is invoked again, an inode is identified as being orphan and the execution of e2fsck aborts.

Finally, the second bug is an example of a recovery that fails to leave the file system in a consistent state. When a file is mistakenly identified as being a directory, its contents are blindly considered by e2fsck to be directory entries. In one of the cases found by SWIFT, e2fsck invokes the `salvage_directory` function in order to recover the contents of the directory. When this function finds a directory entry with a size of 8 bytes, the entry is considered to be a 'hole' in the directory data and the remaining contents are moved 8 bytes to the left, thus overwriting this space. However, e2fsck does not realize that the data moved to this space does not contain valid directory entries. Thus, repeated executions of e2fsck lead to multiple recoveries of this data.

Both bugs presented above support our idea that the test suites of file system checkers allow us to perform more complex tests that would not be possible with the corruption model presented in Section 5.3.1. For instance, test suites provided by e2fsck exercise specific features of the file system, such as orphan inodes, that require an advanced knowledge of the file system being tested.

6.2 Checking the Completeness of Recoveries

In order to check the completeness of the recoveries performed by e2fsck and reiserfsck, we first identified the fields that are used by both ext4 and ReiserFS (shown in Table 10). Next, we used the disk traces produced during the exploration of corruption of those fields in both file systems to compare the logical representations of the data after recovery, as described in Section 4.3.

SWIFT was able to reveal a situation in which a wrong file type (symbolic link type) makes e2fsck erase a regular file, thus losing data, whereas reiserfsck was able to restore the type of the file to the correct value. In this case, the e2fsck recovery works as follows: e2fsck looks at the size of the file data, wrongly considered as being a symbolic link, and is able to detect an inconsistency, since in this specific case the file data occupies more than one file system block and symbolic links can only use one block of data. Once the file system checker detects this inconsistency, it clears the inode and the pointers to the data of the file. In this situation, e2fsck ignores the file type value that is stored in the directory entry that leads to this file. Therefore, this is a case where e2fsck does not use all the information available in the disk.

ext4 field	ReiserFS field	Description
s_blocks_count_lo	s_block_count	Number of blocks
free_blocks_count_lo	s_free_blocks	Number of free blocks
s_log_block_size	s_blocksize	Size of the block
s_state	s_umount_state	State indicating whether the disk needs to be checked/recovered
magic_string	s_magic	Magic value identifying the file system
Block bitmap	Block bitmap	Block bitmap
i_mode	sd_mode	Type and permissions of a file
i_links_count	sd_nlink	Number of hard links pointing to file
i_uid	sd_uid	User ID
i_gid	sd_gid	Group ID
i_size	sd_size	Size of file
i_atime	sd_atime	Access time
i_mtime	sd_mtime	Modification time
i_ctime	sd_ctime	Creation time
i_dtime	sd_dtime	Deletion time
i_blocks_lo	sd_blocks	Number of blocks of the file
direct data pointer (i_block)	direct data pointer	Pointer to data
indirect data pointer (i_block)	indirect data pointer	Pointer pointing to data pointers
double indirect data pointer (i_block)	double indirect data pointer	Pointer pointing to pointers to data pointers

Table 10. Fields in common between Ext4 and ReiserFS.

6.3 Efficiency and Scalability

In this section we evaluate the efficiency and scalability of SWIFT. Efficiency refers to the execution time of the tests performed by the system, and scalability refers to the number and diversity of recovery behaviors of the file system checkers that are tested.

In Table 11 we show the execution time, number of paths, and paths flagged as executing inconsistent recoveries for the tests of the three file system checkers considered in this work. This table shows that SWIFT is able to find numerous instances of wrong recoveries. Even though the testing process can take up to several hours, we consider this a small price to pay given the significantly lower effort required of the tester, when compared to traditional approaches.

In order to evaluate the ability of SWIFT to explore a comprehensive set of recovery behaviors of a file system checker, we measured the statement coverage obtained during the path exploration phase of the tests we performed. In Figure 8 we show how the statement coverage of e2fsck and fsck.minix, using corruption on individual fields, varies over the course of the testing process. Moreover, in Figure 9 we show the statement coverage of reiserfsck, for the three different command-line arguments.

In the first graph, we can observe that applying SWIFT to the e2fsck file system checker led to a lower code coverage than that obtained with the e2fsck test suite. This has

Fsck	Exploration Time (hours)	# Paths Explored	# Paths with Bugs
e2fsck (individual field corruption model)	9.5	50064	193
e2fsck (test suite corruption model)	21	163288	6305
reiserfsck	37	27636	3
fsck.minix	11.5	22488	229

Table 11. Summary of the exploration phase of the different file system checkers.

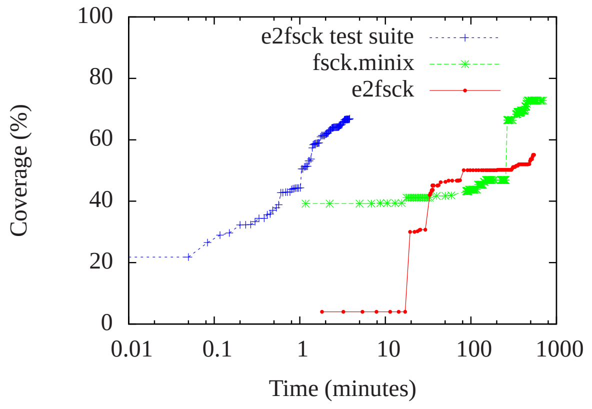

Figure 8. Comparison between the statement coverage of SWIFT applied to e2fsck and fsck.minix, and the coverage achieved by the original test suite of e2fsck.

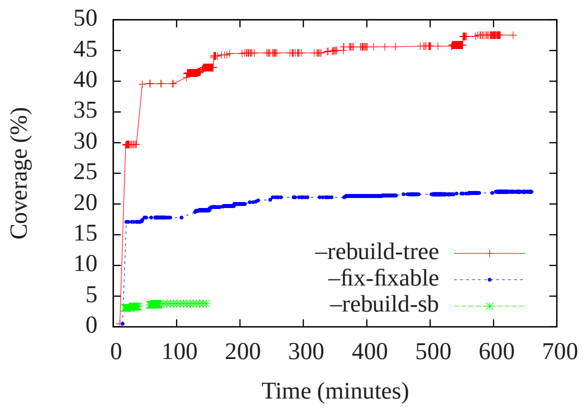

Figure 9. Statement coverage of the three tests of reiserfsck. In total, SWIFT achieved 62.3% code coverage.

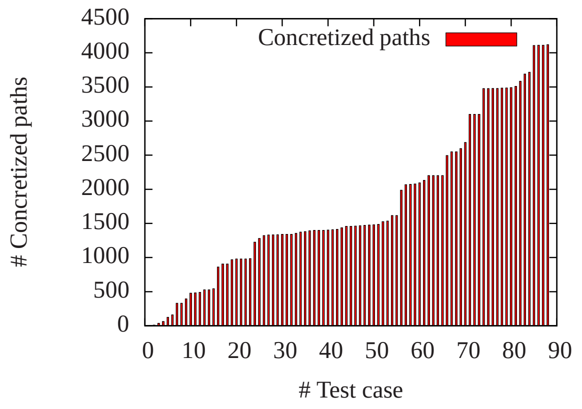

Figure 10. Number of paths concretized during the SWIFT tests using the e2fsck suite. SWIFT explored a total of 163288 paths.

to do with the fact that the test suite of e2fsck makes use of different disks, which have diverse configurations of the file system being used and make use of specific file systems data structures that go beyond the simple corruption model used by SWIFT. For instance, one of the tests contains a separate file with journal data in order to exercise the file system checker code that replays the journal. These results are nonetheless positive for SWIFT, given the relatively short testing period, and the fact that, unlike the test suite of e2fsck, our tests require little advanced knowledge about file system checkers and file systems.

Figure 9 shows three lines, one for each command-line argument used in each test of reiserfsck. The three arguments exercise considerably different amounts of code, which translates into different values for the statement coverage. In total, our tests of the reiserfsck file system checker exercised 62.3% of the code.

In both graphs we can observe long periods of time during which the statement coverage does not change. These periods of time exist because some of the corruptions injected by SWIFT are not checked and recovered by the file system checker, and thus do not lead to the exploration of new code. One can also observe moments during which the coverage of the file system checker abruptly increases. This usually occurs when SWIFT starts exploring corruption on a new field that is checked and recovered by the file system checker.

To conclude, it is worth noting that not only SWIFT does not require deep knowledge of the file system code, but once disk traces have been generated by SWIFT, they can be subsequently reused as regression tests.

6.4 Optimizations - Concretization Strategy

Next, we evaluate the concretization strategy described in Section 5.5, aimed at making SWIFT discard a smaller number of paths than using the original timeout-based strategy.

In Figure 10 we show the evolution of the cumulative number of paths that were not discarded due to the concretization of path constraints, and would have been discarded otherwise. During the execution of SWIFT using the test suite of e2fsck, SWIFT was able to continue the execution of around 4,000 paths that would have been discarded if we had not used our strategy. From this set, SWIFT completely explored 283 paths during the symbolic execution phase. From these, SWIFT marked 3 paths as buggy.

7. Conclusion

In this paper we presented a methodology for testing file system checkers. Our methodology builds upon two insights. First, we can use the file system checker itself to check the consistency of its recoveries. Second, we can use different file system checkers to check the completeness of recoveries. Based on these two ideas, we provide a testing methodology that requires minimal effort from the tester and no formally written specification of the file system checker.

We implemented SWIFT, a system that implements our methodology. Our experimental evaluation shows that SWIFT can find bugs in real, widely used, file system checkers. SWIFT found cases of bugs in which the file system checker fails to use all available redundancy to perform recovery, as well as cases that lead to the loss of data. SWIFT is able to achieve code coverage levels comparable to that obtained with manual tests.

Acknowledgements

We thank the anonymous reviewers, the members of the sysnets group at MPI-SWS, and our shepherd, Leendert van Doorn, for valuable feedback. We are indebted to Vitaly Chipounov for tirelessly helping us with our use of S^2E.

References

[1] BAIRAVASUNDARAM, L., SUNDARARAMAN, S., ARPACI-DUSSEAU, A., AND ARPACI-DUSSEAU, R. Tolerating File-System Mistakes with EnvyFS. In *USENIX Annual Technical Conference '09* (2009), USENIX.

[2] BAIRAVASUNDARAM, L. N., ARPACI-DUSSEAU, A. C., ARPACI-DUSSEAU, R. H., GOODSON, G. R., AND SCHROEDER, B. An analysis of data corruption in the storage stack. *ACM Transactions on Storage 4* (November 2008).

[3] BAIRAVASUNDARAM, L. N., RUNGTA, M., AGRAWA, N., ARPACI-DUSSEAU, A. C., ARPACI-DUSSEAU, R. H., AND SWIFT, M. M. Analyzing the effects of disk-pointer corruption. In *DSN '08: Dependable Systems and Networks* (2008), IEEE Press.

[4] CADAR, C., DUNBAR, D., AND ENGLER, D. KLEE : Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *OSDI '08: Operating systems design and implementation* (2008), USENIX.

[5] CADAR, C., AND ENGLER, D. Execution Generated Test Cases: How to Make Systems Code Crash Itself. In *SPIN '05: Model Checking Software* (2005), vol. 3639 of *Lecture Notes in Computer Science*, Springer.

[6] CHIPOUNOV, V., KUZNETSOV, V., AND CANDEA, G. S2E: A platform for in-vivo multi-path analysis of software systems. In *ASPLOS '11: Architectural Support for Programming Languages and Operating Systems* (2011), ACM.

[7] GODEFROID, P., KLARLUND, N., AND SEN, K. DART: directed automated random testing. In *PLDI '05: Programming language design and implementation* (2005), ACM.

[8] GUNAWI, H., DO, T., JOSHI, P., ALVARO, P., HELLERSTEIN, J., ARPACI-DUSSEAU, A., ARPACI-DUSSEAU, R., SEN, K., AND BORTHAKUR, D. FATE and DESTINI: A framework for cloud recovery testing. In *NSDI '11: Networked Systems Design and Implementation* (2011), USENIX.

[9] GUNAWI, H. S., RAJIMWALE, A., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. SQCK: a declarative file system checker. In *OSDI '08: Operating Systems Design and Implementation* (2008), USENIX.

[10] JIANG, W., HU, C., ZHOU, Y., AND KANEVSKY, A. Are disks the dominant contributor for storage failures?: a comprehensive study of storage subsystem failure characteristics. In *FAST '08: File and Storage Technologies* (2008), USENIX.

[11] KING, J. Symbolic execution and program testing. *Commun. ACM 19* (July 1976).

[12] PANZER-STEINDEL, B. Data integrity. `http://indico.cern.ch/getFile.py/access?contribId=3&sessionId=0&resId=1&materialId=paper&confId=13797`, 2007.

[13] PINHEIRO, E., WEBER, W.-D., AND BARROSO, L. A. Failure trends in a large disk drive population. In *FAST '07: File and Storage Technologies* (2007), USENIX.

[14] RODRIGUES, R., CASTRO, M., AND LISKOV, B. BASE: Using abstraction to improve fault tolerance. In *SOSP '01: Symposium on Operating Systems Principles* (2001), ACM.

[15] SCHROEDER, B., DAMOURAS, S., AND GILL, P. Understanding latent sector errors and how to protect against them. *ACM Transactions on Storage 6* (2010).

[16] TALAGALA, N., AND PATTERSON, D. An analysis of error behavior in a large storage system. Tech. Rep. UCB/CSD-99-1042, EECS Department, University of California, Berkeley, Feb 1999.

[17] YANG, J., SAR, C., AND ENGLER, D. EXPLODE: a lightweight, general system for finding serious storage system errors. In *OSDI '06: Operating systems design and implementation* (2006), USENIX Association.

[18] YANG, J., SAR, C., TWOHEY, P., CADAR, C., AND ENGLER, D. Automatically generating malicious disks using symbolic execution. In *S&P '06: IEEE Symposium on Security and Privacy* (2006).

[19] YANG, J., TWOHEY, P., ENGLER, D., AND MUSUVATHI, M. Using model checking to find serious file system errors. *ACM Transactions on Computer Systems 24*, 4 (2006).