SibylFS: formal specification and oracle-based testing for POSIX and real-world file systems

Tom Ridge[1] David Sheets[2] Thomas Tuerk[3] Andrea Giugliano[1] Anil Madhavapeddy[2] Peter Sewell[2]

[1]University of Leicester [2]University of Cambridge [3]FireEye

http://sibylfs.io/

Abstract

Systems depend critically on the behaviour of file systems, but that behaviour differs in many details, both between implementations and between each implementation and the POSIX (and other) prose specifications. Building robust and portable software requires understanding these details and differences, but there is currently no good way to systematically describe, investigate, or test file system behaviour across this complex multi-platform interface.

In this paper we show how to characterise the envelope of allowed behaviour of file systems in a form that enables practical and highly discriminating testing. We give a mathematically rigorous model of file system behaviour, SibylFS, that specifies the range of allowed behaviours of a file system for any sequence of the system calls within our scope, and that can be used as a *test oracle* to decide whether an observed trace is allowed by the model, both for validating the model and for testing file systems against it. SibylFS is modular enough to not only describe POSIX, but also specific Linux, OS X and FreeBSD behaviours. We complement the model with an extensive test suite of over 21 000 tests; this can be run on a target file system and checked in less than 5 minutes, making it usable in practice. Finally, we report experimental results for around 40 configurations of many file systems, identifying many differences and some serious flaws.

1. Introduction

Problem File systems, in common with several other key systems components, have some well-known but challenging properties:

SOSP'15, October 4–7, 2015, Monterey, CA.

ACM 978-1–4503-3834-9/15/10. . . $15.00.
http://dx.doi.org/10.1145/2815400.2815411

- they provide behaviourally complex abstractions;
- there are many important file system implementations, each with its own internal complexities;
- different file systems, while broadly similar, nevertheless behave quite differently in some cases; and
- other system software and applications often must be written to be portable between file systems, and file systems themselves are sometimes ported from one OS to another, or written to support application portability.

File system behaviour, and especially these variations in behaviour, thus must be understood by those developing file systems, by those aiming to write robust and secure software above them, and by those porting file systems or applications. But at present *there is no good way to systematically describe, investigate, or test that behaviour*: either to describe what the envelope of allowed behaviour of a file system (or group of file systems) is, to investigate experimentally what behaviour occurs, or to test whether file system implementations conform to some particular envelope.

Normal practice has for decades relied on prose standards and documentation (the POSIX standard [33], Linux Standard Base (LSB) [21], man pages) and on test suites [22, 34]. Indeed, this is so well established that many practitioners would not imagine that any alternative can exist. But normal practice does not support any of the above: prose documents generally cannot be made complete and unambiguous; they cannot be used as a test oracle to automatically determine whether some observed behaviour is allowed or not; and building test suites without a test oracle requires manual curation of the intended outcome of each test. As we shall see from our test results, behavioural differences between file systems have proliferated, some intentional and many clearly bugs.

Contributions In this paper we show how to characterise the envelope of allowed behaviour of file systems and to use that for practical and highly discriminating testing. Our first contribution is a rigorous specification of file system behaviour, SibylFS: a model that specifies the range of allowed behaviours of a file system for any sequence of API calls. This model has several important and unusual properties:

1. It is *executable as a test oracle*: given an observed trace of API calls and returns, SibylFS can efficiently compute whether it is allowed by the model or not. In conjunction with our extensive test suite (see below), this lets us validate the model, to ensure it does not overly constrain implementation behaviour, and lets us test implementations, to ensure they do not exhibit behaviour not allowed by the model.
2. To characterise the behaviour of a particular implementation, SibylFS is parameterised in various ways. It currently supports four primary modes: POSIX, Linux, OS X and FreeBSD behaviour. This variation is essential when exploring file system behaviour, as otherwise a single difference (e.g. in path resolution) might give rise to thousands of individual test-result discrepancies; we have to be able to analyse and factor out such differences to make progress.
3. Within our scope, SibylFS aims to be *realistic and comprehensive*: it is a model of actual file system behaviour, not of idealised or simplified file systems, and it gives a functional-correctness criterion for arbitrary sequences of API calls.
4. To make it completely *precise and unambiguous*, while still admitting the loose specification needed, the model is written in a mathematically rigorous language, the typed higher-order logic of the Lem tool [28]. We use Lem to translate this into the theorem

provers HOL4 and Isabelle/HOL (which we have used to prove theorems about file system behaviour), and into the OCaml source code used by SibylFS.

5. We take care also to make the model *readable*, for use as an informal reference. The Lem language is in many ways similar to a conventional functional programming language; the model has a modular structure that isolates the conceptually distinct aspects of file system behaviour, and the model is expressed over abstract structures rather than the performance-oriented details of file system implementations. Key choices are linked to the experimental trace data that they relate to. All this is important also for *maintainability*, which has been essential in developing the model over the last two years.
6. Finally, SibylFS is *fast enough to run and easy enough to set up* to make it easily usable in practice. Pre-compiled binaries are available on Linux and OS X (and compile-from-source is additionally supported on these platforms and FreeBSD), and a test-and-check run for a single file system takes less than 5 minutes (§7.1). We envision SibylFS being used during file system development, quality assurance, and continuous integration.

Our second contribution is an extensive test suite, consisting of 21 070 automatically generated and hand-written test scripts. The fact that we have an executable test oracle greatly simplifies test suite development: we do not have to manually determine the intended outcome for each test, but rather can focus on generating tests with good coverage. Our present test suite achieves 98% coverage of the model (§7.2).

Our third contribution is the results of this testing and modelling process. We have run our test suite on over 40 system configurations, to simultaneously develop the model and to identify bugs, platform conventions, and deviations. We present an overview of some of the most interesting findings, including deviations of Linux HFS+ from OS X HFS+, the effect of mount options on SSHFS/tmpfs's behaviour, a storage leak in posixovl/VFAT, and an infinite busy loop on OpenZFS for OS X (§7.3).

Technical challenges There are two main challenges we have had to overcome. The first is nondeterminism (§3): file system implementations have considerable internal nondeterminism and our model must be loose enough to accommodate that, but the model must still be clear (not complicated with implementation detail), and, crucially, trace-checking must be efficient. Previous related work modelling and checking TCP [3] required a sophisticated higher-order logic constraint-based backtracking search, and checking around 1000 traces took 2500 CPU-hours; here, by carefully isolating nondeterminism we check 20 000 traces in about a minute on a four-core machine. These tasks are not directly comparable, but the specifications and traces have similar sizes and characters; this per-trace performance difference (around 6 orders of magnitude) makes the difference between something on the edge of practicality and something that can be done routinely during development.

The second main challenge is that of managing complexity, in constructing an accurate, readable and concise model that synthesises the many existing sources of information and our thousands of observed real-world traces of behaviour (§4). The sheer variety of behaviours and number of test results is potentially overwhelming; the challenge was to distil all this complexity down into a concise, structured, comprehensible document (rather than a collection of thousands of special cases). When we started the work, it was not obvious that this was feasible, but a range of model-structuring choices and our analysis tools have made it so.

Use cases SibylFS provides a turnkey black-box test setup that can be used routinely (with low effort for the user) to identify behavioural differences between file system implementations and between those and our specifications. It should be useful for:

- file system authors, kernel maintainers, and distribution teams, to identify POSIX violations (§7.3.2) and platform convention violations (§7.3.3);
- those porting file systems (e.g. OpenZFS to Linux, OS X, and FreeBSD; HFS+ to Linux; or ext2 to OS X), to ensure that their efforts strike the right balance between expected behaviour and platform convention (§7.3.3); and
- system administrators who, before deploying a file system to users, want to understand their deployment configuration. (§7.3.4)

The specification also serves as a precise reference document, both for file system developers and application authors, to understand what behaviour can be relied on and where file systems differ.

It also supports machine-checked and machine-assisted formal proofs about the specification. As a demonstration that this is feasible, for a previous version of the model we have proved two sanity properties: that libc calls that result in an error do not change the abstract file system state (for POSIX, Linux and OS X), and that (in the absence of resource-limit failures) whether a libc call succeeds or fails is deterministic.

SibylFS also opens up many possibilities for future work, as we touch on in §9. Our model, tools, and results are available under the BSD-style ISC licence at `http://sibylfs.io/`.

1.1 Scope

POSIX describes many aspects of operating systems, but our model covers only the part that is relevant to file systems. We include `close`, `closedir`, `link`, `lseek`, `lstat`, `mkdir`, `open`, `opendir`, `pread`, `pwrite`, `read`, `readdir`, `readlink`, `rename`, `rewinddir`, `rmdir`, `stat`, `symlink`, `truncate`, `unlink`, and `write`. This covers the essential commands that are necessary to manipulate and interrogate the directory structure and file contents, and the functions dealing with symlinks (`readlink`, `symlink`). Together this is sufficient to cover a broad range of uses.

Our model also includes a model of processes and the operating system, again focusing on those aspects that are relevant to file systems. Processes can be created and destroyed. Each process has a working directory which is mainly used when resolving relative paths. For this reason we include `chdir`. Additional per-process structures that we model include the file-descriptor table and the process run state. We also model permissions, including `chmod`, `chown`, and `umask`, and a model of users, groups, and which users belong to which groups.

POSIX includes notions of *undefined*, *unspecified* and *implementation-defined* behaviour. Undefined behaviour results from using a libc function with arguments that are "invalid" according to POSIX. Unspecified behaviour results from a libc function call with arguments that are valid, but for which POSIX leaves the behaviour unspecified. Implementation-defined behaviour is similar to unspecified behaviour, but it is expected that conforming implementations document their behaviour in such cases. Our model for the POSIX platform covers all these cases. The variants of our model for real-world platforms describe the actual real-world behaviour, even where POSIX declares the behaviour to be undefined.

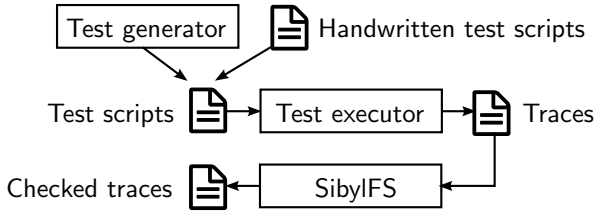

Figure 1. File system testing and trace checking

1.2 Limitations

Currently we do not model host crash-failure. We model concurrent file system API calls (the call occurs before the return, and there is an intermediate point in time where the effect of the call occurs), and our checking infrastructure supports them. Our test harness executes against the libc interface, and as a result it is not possible to force libc calls from different processes to execute in-kernel simultaneously, see §6.3. As a consequence, our test suite does not contain tests involving in-kernel racy behaviours; systematically testing such interactions would require significant additional effort to extend the test executor, but the model and the checker should be largely unchanged. However, our test harness and test suite do cover interleaved calls from multiple processes, which is important when modelling and testing permissions. We also model timestamps, but periodic timestamp updating results in extremely non-deterministic behaviour, which currently makes trace checking take excessive runtime, so this is largely untested at present. These are all important aspects of file system behaviour, and we believe our approach can be extended to cover them in the future, but each would require substantial additional work; we exclude them for the time being simply to keep the project manageable.

We do not model unusual file types (such as FIFO special files), or asynchronous I/O, signals and the associated `EINTR` error. We exclude errors such as `EIO` ("a physical I/O error has occurred") and `ENOMEM`; from a modelling perspective such errors could potentially occur at any time. We do not model many resource exhaustion behaviours, such as exceeding the maximum number of entries in a directory or using all available inodes. We do not currently model the `*at` forms of functions such as `openat`, although it should be straightforward to adapt our model to include them.

We do not model free space or storage media behaviour in general. One can imagine future work developing, for a particular file system of interest, an executable abstraction function that reads a concrete volume state (perhaps after a host crash and recovery) and calculates the corresponding abstract state of the model. Testing the correspondence between implementation and model at each step, analogously to [30], would likely be extremely discriminating.

Our model parameterisation (Point 2 of the first contribution above), while desirable and necessary, also has a cost: running SibylFS is low-cost, but adapting SibylFS to model a file system with significantly different behaviour can involve substantial work (though with a big pay-off: characterising that behaviour in detail).

2. Overview

The process of testing a file system and checking the resulting traces with SibylFS is depicted in Fig. 1. The process starts with a set of *test scripts*, organised into groups

```
@type script
# Test rename___rename_emptydir___nonemptydir
mkdir "emptydir" 0o777
mkdir "nonemptydir" 0o777
open "nonemptydir/f" [O_CREAT;O_WRONLY] 0o666
rename "emptydir" "nonemptydir"
```

Figure 2. Excerpt of a `rename` test script

```
@type trace
# Test rename___rename_emptydir___nonemptydir
3: mkdir "emptydir" 0o777
   RV_none
...[further calls and return values]
6: rename "emptydir" "nonemptydir"
   EPERM
```

Figure 3. Excerpt of a `rename` trace; `RV_none` indicates the call completed successfully.

according to the libc functions they target. The bulk of these are generated automatically by the *test generator*, supplemented by hand-written test scripts.

Test scripts contain sequences of file system commands that are used by the *test executor* to drive the real-world file system under test, via the libc interface. An example excerpt from a test script is given in Fig. 2; after the header, each line is the data for a single libc call (more complex test scripts can involve multiple processes). Each script sets up whatever file system state it needs, starting from an empty file system; they involve up to several hundred libc function calls. The resulting behaviour is recorded in a *trace* file, as in Fig. 3, interleaving the commands from the script with the responses received from the real-world system.

These trace files are processed by SibylFS to check for conformance with the model. The main part of the SibylFS checker is the model itself, automatically translated from Lem to OCaml and then linked together with a small OCaml wrapper. Checking is done with respect to a particular variant of the model (POSIX, Linux, OS X or FreeBSD); in addition, various flags control further checking parameters, such as whether the initial process runs with root privileges or not. The output from this checking phase is a set of *checked traces*, as in Fig. 4. For steps in the trace that conform to the model, the checked trace resembles the original. For steps that are non-conformant, the checked trace includes an error message and (if possible) diagnostic information to help identify why the behaviour is non-conformant. In Fig. 4 the error message indicates that at line 6 in the trace file the real-world file system returned `EPERM`, but the specification allowed only `EEXIST` or `ENOTEMPTY`.

Individual trace files may contain multiple test calls, and so it is important that the checker try to continue even when an individual step fails. In Fig. 4 SibylFS continues checking the trace under the assumption that `EEXIST` or `ENOTEMPTY` was returned rather than `EPERM`.

Analysis of the results also requires automation to assist with the volume of data, as each run produces tens of thousands of checked traces per platform, and the results must be compared between file systems and (during model development) between model versions. For example, it is common to compare different file systems (or different versions of the same file system) on a single operating system; we have also compared versions of a

```
# Error:   6: EPERM
#   unexpected results: EPERM
#   allowed are only: EEXIST, ENOTEMPTY
#   continuing with EEXIST, ENOTEMPTY
```

Figure 4. Excerpt of a checked `rename` trace from SSHFS/tmpfs 2.5 on Linux 3.19.1

single file system on several different operating systems, see §7.3. Checked traces can be rendered to HTML, along with autogenerated indexes and summaries of check results. To analyse the results of multiple runs, the system can intelligently combine the results across many different platforms, merging behaviours common to many runs and highlighting the differences.

In addition, a model-debugging tool allows model developers to analyse the checking process itself, taking a trace and producing a description of the real-world states that were being tracked by SibylFS at every step of trace. This has been extremely useful for developing the model, but we do not expect end users of SibylFS to need it.

The process of constructing the model has been intimately entwined with testing: testing (particularly on new operating systems and file systems) uncovers new real-world behaviours, which are then incorporated into the model; new tests are added and the updated model is then used for another round of testing, with those behaviours now not generating discrepancies. This represents a virtuous circle: at each stage the model becomes more accurate and comprehensive, and the test suite accumulates more and more tests.

3. Technical challenge: nondeterminism

In writing a model to be used as a test oracle (i.e., to compute whether observed traces are allowed by the model or not), the treatment of nondeterminism is a key issue. If both the model and the implementations are entirely deterministic, at the abstraction level at which they are being observed, then one could just run the two on the same input and check they have equal output. But for real-world software that is rarely the case: implementation behaviour typically varies, both between implementations and depending on implementation-internal runtime choices, and specifications are often deliberately loose. For example, for file systems:

- Some API calls could give rise to several distinct errors, e.g. `EISDIR`, `EEXIST` or `ENOTEMPTY` for a `rename` of a file to a non-empty directory; which is actually returned is determined by the order of the checks in the file system implementation code.
- The number of bytes returned by a `read` may be less than the number requested, determined by the implementation internal state.
- The order in which `readdir` returns entries from a directory with multiple entries will depend on the implementation and on details of the storage layout of the directory data (neither of which belong in an abstract specification).
- The behaviour of concurrent API calls may be determined by scheduling.

A sound specification must be loose enough to accommodate all such variation (looseness should not be confused with the question of whether a specification is precise: we want a mathematically precise model, but one that admits a range of allowable behaviours). But then checking a trace against such a specification poses an algorithmic problem, especially when there is internal nondeterminism that is not immediately observable: in general one must effectively track the set of all possible implementation states (abstracted to what can

affect external observation) at every step of the trace, or, equivalently, calculate the set of constraints on the specification state that arise from a trace of observations. The Netsem project of Bishop et al. [3] produced a specification and trace-checker in that form for TCP/IP and the Sockets API, but it required a sophisticated higher-order logic constraint solver and a backtracking search process, and as noted in §1, checking could take thousands of CPU-hours, at the limits of practicality.

At the same time, there is a tension between writing a model to be as clear as possible and one that supports efficient checking (both quite different from writing a file system implementation, of course); as far as possible we want to avoid polluting the model with algorithmic concerns.

Accordingly, for SibylFS we took great care up-front to write the model in a way that would remain clear and be efficiently checkable, without needing a backtracking search or sophisticated constraint solving. We used different strategies for different sources of nondeterminism, as follows.

Simple nondeterminism via possible next-state enumeration At the top level the model consists of a type of abstract file system states and a function that, given such a state and an API event (call, return, etc.), returns a finite set of possible next states. We go into more detail in §5. For the simple case of multiple possible API error return values, the model explicitly calculates the set of all allowed errors (using novel combinators, as described in §4, to do so concisely) and a next state for each, then when the real-system return value is observed we simply choose the corresponding state.

We use a similar approach to deal with the number of bytes processed by a `read` or `write`, just enumerating the possible immediate next states. This is attractively simple and suffices for our testing. It does involve some unnecessary cost for tests with large reads or writes, enumerating many next-states, but that blowup is resolved at the next step in the trace, when the actual number of bytes read by the real-world process becomes known. To test with very large reads and writes one could refactor the model slightly to produce continuations abstracted on the API return values, to check them and calculate a single next state. The downside of doing that uniformly is that it makes it hard for the checker to describe, for a failing step, the set of values that would have been allowed (as we showed in the example trace of the previous section).

Directory listing nondeterminism by hand-crafted specification The `readdir` command is more challenging to specify. A process can request a directory handle using `opendir` and then use `readdir` to return the directory entries. These can be returned in any order, so this command gives rise to significant nondeterminism. However, the real challenge in specifying this command is to deal with modifications of the directory (either by the same process or a different process) while the directory handle is open. If the directory is not modified at any point, then `readdir` returns all the entries in the directory, and each entry is returned exactly once. The POSIX intent is to provide a similar guarantee when the directory is modified, and real-world file systems also provide this guarantee, as far as we can observe: for any entry, if that entry is not modified from the time the directory handle is opened, then that entry will be returned by `readdir` exactly once. If an entry is deleted, and if it has not already been returned by `readdir`, then it *may* be returned by subsequent calls to `readdir` (if it has already been returned, then it is not returned again if it is deleted). Similarly, if an entry is added, then it may be returned by subsequent calls.

So far, the semantics could be modelled by taking a snapshot of the entries when `opendir` is called, and recording which entries have already been returned by `readdir`. On

subsequent calls to `readdir`, the entries in the directory at that point could be examined, and the possible entries that could be returned at that point could be determined. The problematic case arises for entries that are initially in the directory, then deleted, then added again (or, vice versa, those that are added then deleted). According to POSIX, these entries may (but need not) be returned. In order to model this behaviour, we are forced to *track all changes to a directory* from the point that `opendir` is called, as well as the entries that have already been returned by `readdir`. With this information, it is possible to determine the set of entries that must be returned, and those that may be returned, and thus to give a semantics to the whole command.

In fact, we maintain (rather than compute) sets of "must" and "may" entries in a directory. Whenever a directory handle is read from, it accesses the changes since the last time it was read from, and updates the must and may sets, before nondeterministically splitting to allow any of the entries in "must" or "may" to be read. This nondeterminism is resolved at the next step, when the label reveals the entry actually read.

It is worth noting that this is an area where a good specification is conceptually more complex than any particular implementation: the latter just returns some list of names, while the model has to capture all allowable sequences, ruling out all those that are not possible.

Concurrency nondeterminism via state sets Multiple user processes executing file system API calls concurrently also results in nondeterministic behaviour, e.g. if one process renames a file while another removes it. The SibylFS model and trace checker cope with multiple, concurrent API calls by maintaining explicit sets of possible (model) file system states.

We note also that another way to avoid internal nondeterminism is to instrument the implementation, to expose all the internal choices that affect external behaviour as trace events. For a single implementation that might be viable (and indeed desirable, as it would permit checking of internal invariants). But for checking many file systems, the black-box approach that we follow here is more tractable.

4. Technical challenge: managing complexity

To give an idea of the challenges in identifying and describing complex real-world behaviours, we describe the process of updating the model to support OS X. At that point, we already had variants for POSIX and Linux. We ran the tests on OS X with the default HFS+ file system, and checked the traces against the POSIX variant of the model. We were confronted with thousands of failing traces (around 5 000 for `open` alone).

We manually analysed the failing OS X traces to identify why they were not allowed according to our understanding of POSIX. This was painstaking work, taking roughly four to six weeks (though still small compared with the effort required to implement a production file system). The next step was to rework the model to incorporate these new OS X behaviours, while remaining concise, structured, and readable. The process is one of inferring, from thousands of observed behaviours, a compact description of those behaviours (as a higher-order logic specification).

To make it feasible to write the model and to extend it in this way, we have found it essential to structure the model in various ways. Different mechanisms have been useful to address different kinds of complexity.

Modules Lem provides a notion of module: a collection of type, pure function, and inductive relation definitions (analogous to the modules of OCaml and other ML-like

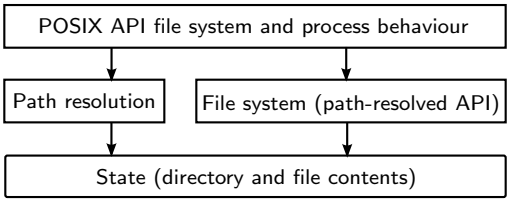

Figure 5. Modular structure of the model

languages). We used these to structure the model as a set of independent modules, with clearly defined interfaces, as shown in Fig. 5. A file system has to maintain the directory structure and the contents of files, typically using references. This is managed by the *state module*. The *path resolution module* defines how paths reference particular files and directories. The *file system module* represents the bulk of the model. It describes how each command (`link`, `rename` etc.) behaves, including how they modify the state, and the many possible error cases, but working over fully resolved paths. Finally, the *POSIX API* module glues those together and includes the behaviour of libc and the operating system, introducing the notion of a process, and per-process datastructures. This is the top-level module which exposes the interface used for trace checking.

Our module structure does not represent the structure of the existing POSIX specification or the internal structure of any file system implementation; rather, it is the result of an attempt to identify the conceptually key components and their interfaces, while simultaneously minimising the overall complexity of the model.

An important decision was to separate path resolution from the semantics of each command. When processing a command such as `rename p1 p2`, the POSIX API module first resolves the paths `p1` and `p2` to obtain two resolved paths. These are then used when invoking the file system module equivalent of the `rename` function. Thus, internally to the model, the file system module API is expressed in terms of resolved paths, not raw strings. This means that the file system model is clean, and unpolluted by the tricky details of path resolution which have been confined in a separate module.

Because they have pure value-passing interfaces, modules can be considered in isolation, allowing important invariants to be established. Modularization also allows unit testing of individual modules, which has been useful particularly to get the details of path resolution correct.

Traits Aspects of the model that cut across the modular structure but which are conceptually distinct have been isolated using a trait-like mechanism: there is a *core* model on top of which the user can "mix in" further traits for particular functionality. The *permissions* trait defines the behaviour of file permissions including functions such as `umask`. The *timestamps* trait defines how the timestamp information on files is updated, in both immediate mode and periodic mode. For example, "core without permissions" specifies a model where permission information is ignored, and all files are accessible by all users.

Monads and combinators Higher-order logic is based on the notion of (pure) functions. On top of this we have introduced various functional programming structuring techniques, including *monads* and associated *combinators*, to allow us to structure our definitions.

In Fig. 6 we illustrate the use of the "parallel" combinator ||| to specify the checks that the `rename` function must perform. The conditional first checks whether the source and destination are the same, in which case the `rename` is a no-op, and the checks do nothing.

```
let fsop_rename_checks ... = ...
  if (fsop_rename_same_rsrc_rdst env rsrc rdst s0) then
   fsm_do_nothing
  else
   ( fsop_rename_checks_rsrc_rdst env rsrc rdst
     ||| fsop_rename_checks_root env rsrc
     ||| fsop_rename_checks_subdir env rsrc rdst
     ||| fsop_rename_checks_parentdirs env rsrc rdst
     ||| fsop_rename_checks_perms env rsrc rdst )
```

Figure 6. Structuring the model using combinators

Main modules	
State	502
Path resolution	291
File system	1388
POSIX API	818

Others	
Prelude	156
Types	888
Monads	130
Permissions	208
Formal properties	1103
Support files	497

Total 5981

Figure 7. The model, non-comment lines of specification

Otherwise the `rename` function needs to check various conditions: *fsop_rename_rsrc_rdst* checks various combinations of the source and destination that result in errors (for example, `ENOENT` may be raised if the source is missing); *fsop_rename_checks_root* checks attempts to rename the root directory; *fsop_rename_checks_subdir* checks attempts to rename a directory to a subdirectory of itself; *fsop_rename_checks_parentdirs* checks that the parent of the source and destination directory can be found (this check should always succeed; it is included to cover the case that a disconnected file or directory is involved in the rename); *fsop_rename_checks_perms* checks the permissions involved in the `rename`.

Each of these checks may raise many different errors. Moreover, as discussed in §3, and unlike an implementation, we have to loosely specify the behaviour: any error that arises from any of the checks is valid behaviour. The parallel combinator conceptually allows these checks to be carried out in parallel, and the resulting error may be from any of the individual checks. The excerpt in Fig. 6 concisely and readably expresses *all* the checks involved, and the use of the parallel combinator emphasizes that none of the errors arising from the individual checks has priority over any of the others. We would strongly argue that the precision and clarity of our model makes it a useful complement to the existing POSIX standard.

5. The model

Our model is about 6 000 lines of higher-order logic. In Fig. 7 we give details of the line count for each part of the model. We cannot hope to give full details in the space available. Instead, we give the main types involved, and representative excerpts from the model. We

start by discussing the notion of a labelled transition system: a mathematical way to specify complex real-world systems. We then discuss the main modules that make up our model, following Fig. 5.

Labelled transition systems Conceptually, SibylFS simply defines a labelled transition system: a nondeterministic infinite-state automaton where the states are abstract (model) file system states and the transitions (mostly) correspond to libc API calls and returns, labelled with the call parameter values and return values. Each model state can correspond to potentially infinitely-many real-world states. As a result, SibylFS only needs to track a finite number of model states to accurately represent all real-world possibilities.

Formally, an LTS can be thought of as a tuple (S, L, S_0, R), where S is a set of states, L is a set of labels, $S_0 \subseteq S$ is a set of start states, and $R \subseteq S \times L \times S$ is a set of triples (known as the transition relation): a triple (s, lbl, s') indicates that, from state s, a transition labelled with *lbl* to state s' is possible.

POSIX API module The POSIX API module defines a labelled transition system. Labels correspond to relevant events: those for a process *calling* a libc function, a value being *returned* to a process from a call, process *creation* and *destruction*, and τ events used to model an "internal" system transition (perhaps corresponding to asynchronous execution of a kernel thread). These are modelled using the Lem type os_label.

```
type os_label =
  | OS_CALL of (ty_pid × ty_os_command)
  | OS_RETURN of (ty_pid × error_or_value ret_value)
  | OS_CREATE of (ty_pid × uid × gid)
  | OS_DESTROY of ty_pid
  | OS_TAU
```

This defines a new datatype (similar to a tagged union or variant type) where values may be one of the five possible variants, distinguished by constructors such as OS_CALL, and each holding an immutable tuple of the associated type. For example, if the value *pid* is of type ty_pid (representing a process id) and *c* is of type ty_os_command (representing a particular instance of a libc function call such as `link`), then OS_CALL(*pid*, *c*) is a value of type os_label (representing the event where process *pid* makes a libc call *c*). The type ty_os_command (used in the OS_CALL constructor) models the various libc functions and their arguments:

```
type ty_os_command =
  | OS_CLOSE of ty_fd
  | OS_LINK of (cstring × cstring)
  | ...
```

The states of the model must represent real-world system states, including processes, open file descriptors, file descriptions and so on. The key type of model states, ty_os_state, is a Lem record type:

```
type ty_os_state ... = ⟨
  oss_fid_table : fmap ty_fid (fid_state 'dir_ref 'file_ref);
  oss_group_table : fmap gid (finset uid);
  oss_pid_table : fmap ty_pid (per_process_state 'dir_ref);
  ...⟩
```

The field *oss_fid_table* is a finite map (fmap) from open file description references (ty_fid) to the state of the file description (fid_state *'dir_ref 'file_ref*); here the pre-primed identifiers are generic type variables, and fid_state is actually a type constructor parameterised on arbitrary *'dir_ref* and *'file_ref* types. The field *oss_group_table* is the mapping from group ids to (sets of) user ids. The field *oss_pid_table* holds the per-process information tracked by the operating system. This includes the current working directory, file descriptors and directory handles, process run state, and various permissions-related state, such as the file creation mask, and the real and effective user ids.

We have now defined the states and the labels of our LTS. For the transition relation one might expect a relational definition, specifying a set of triples (s, lbl, s'), but a mathematically equivalent and more computationally convenient form of definition is as a function that takes a state and a label, and returns a finite set of states; we have a top-level function os_trans with that type:

```
val os_trans : ty_os_state → os_label →
  finset os_state_or_special
```

There is a subtlety here: the type finset os_state_or_special represents a finite set of elements, which are *either* normal states, *or* "special states" which correspond to POSIX undefined, unspecified and implementation-defined behaviours, as described in §1.1. If we ignore special states, the result type indeed represents a set of file system states.

The remainder of the model defines the transition relation: given a state, and a label corresponding to a libc function call, the definition of *os_trans* uses the path resolution module to resolve paths, and then calls the file system module to process the function itself. In addition to this, *os_trans* must deal with processes and concurrency, open file descriptors, file descriptions and so on.

A trace such as that in Fig. 3 is a *sequence of labels*. SibylFS checks a trace step by step. At each step i of the trace, SibylFS maintains a finite set S_i of values of type ty_os_state, which represents all the states that the real-world file system might be in. For each label lbl_i, SibylFS applies *os_trans* to each element of S_i, and takes a union of the resulting sets to form the set of values S_{i+1} at the next step. The initial set S_0 consists of a single state s_0 representing an empty file system. In effect, given S_0 and the sequence of labels, SibylFS computes a sequence $S_0 \overset{lbl_1}{\rightarrow} S_1 \overset{lbl_2}{\rightarrow} S_2 \ldots$. If the end of the trace is reached at lbl_n and the set S_n is non-empty, then the trace is *accepted* by the model. If the set S_i of possible file system states at step i is ever the empty set, then this indicates that the trace is *not accepted* by the model.

Path resolution module Path resolution is complicated for several reasons. The resolution of even simple paths (no symlinks, no permissions) can be counter-intuitive on real-world systems, particularly when the path ends in a trailing slash e.g. the path `/tmp/f.txt/` is sometimes resolved successfully under Linux, even when `f.txt` is a non-directory file. Symlinks introduce much additional complexity. For example, symlinks that occur as the last component of a path are sometimes followed and sometimes not, depending on the libc function involved and flags such as those for `open`; this "follow last symlink" behaviour is further complicated by trailing slashes on the path (a trailing slash makes it more likely the symlink is followed). Permissions further complicate matters. For example, there is the question of how permissions interact with path resolution, and what permissions should be assigned to symlinks.

Our model clearly describes the behaviour of path resolution in terms of the inputs to path resolution, and the output resolved path. The result of path resolution is captured by the *resolved name* type res_name:

```
type res_name 'dir_ref 'file_ref =
  | RN_dir of ('dir_ref × ...)
  | RN_file of ('dir_ref × name × 'file_ref × ...)
  | RN_none of ('dir_ref × name × ...)
  | RN_error of (error × ...)
```

Intuitively path resolution can give four possible results: the path can resolve to a directory (constructor RN_dir), a non-directory file (RN_file), or an error can occur during resolution (RN_error), or the path might resolve to "none" (RN_none), representing a non-existent entry in a directory. This last possibility occurs, for example, for functions such as `mkdir`, where the given path is intended to reference a non-existing entry that will be created by the function.

File system module The file system module defines the behaviour of individual functions such as `link` and `rename`. Internal to the model, its API is expressed using resolved names. In Fig. 6 we gave an excerpt from the file system module: the checks that the `rename` command must make.

State module The state module provides a simple model of directory and file contents. The main type is a record type which includes a field *dhs_dirs* (a finite map from directory references to directories) and a field *dhs_files* (a finite map from file references to files):

```
type dir_heap_state_fs = ⟨
  dhs_dirs  : fmap dh_dir_ref dh_dir;
  dhs_files : fmap dh_file_ref dh_file;
  ...⟩
```

The interface to the state model is expressed in terms of *references* to files and directories (types dh_dir_ref and dh_file_ref). The state-model API permits arbitrary linking and unlinking, in particular, our model can handle directory links, and disconnected files and directories can also be modelled (a disconnected file is one that does not appear in the directory tree, but is still accessible).

Contrasting this to the block-structured storage state one might find in a typical file system implementation is instructive: the model can abstract from all that implementation detail while still correctly describing the envelope of allowed behaviour visible at the API we consider, and that abstraction is essential to make the model simple.

6. Test suite and harness

In §2 we gave an overview of the system, and described the virtuous circle formed by testing and revising the model. We now describe the tests and test execution in more detail.

6.1 The tests

Autogenerated scripts test commands such as `link` and `rename` where combinatorial testing is straightforward, feasible, and expected to cover all static real-world behaviour. The combinatorial nature of the tests means that functions such as `link` and `rename` which take two arguments have many more tests than functions such as `rmdir` which take only one. The `open` function has an especially large number of tests because one argument is a bitfield of open flags.

To reduce the test cases to a finite number, we use equivalence partitioning, which requires identifying classes of inputs where a function is assumed to behave "the same", and testing only one member of each class. For example, in a given file system state where neither `f1` nor `f2` exist, the behaviour of `rename f1 f1` should be the same as `rename f2 f2`, so it suffices to test only one of these two possibilities: the assumption is that the exact name of a file is irrelevant. A potential weakness is that these assumptions might not actually hold for real-world file systems. For example, even if neither `f1` nor `.snapshot` exist, it could be that any reference to `.snapshot` triggers unusual file system behaviour so that `rename .snapshot .snapshot` behaves differently to `rename f1 f1`. Our tests would typically fail to establish this difference. This is an inherent weakness in equivalence partitioning, not specific to our use.

The equivalence classes are based on properties (of file system state, and the file system API calls) which we believe affect file system behaviour. For example, properties of paths used in API calls include: whether the path ends in a slash; whether it starts with 0, 1, 2, or $\geq$3 slashes; whether it is the empty string; whether it is a single slash; the type of the resolved path (file, directory, symlink, nonexistent, error); if the resolved path is a directory, then the number of entries in the directory; and whether the path has a symlink component or not. These properties are used to construct equivalence classes. We then make sure that we have *at least one test case* for each logically-possible combination of properties. For API calls involving two paths (such as `rename`) we consider all combinations of properties of each path individually, together with equivalence classes based on properties of two paths: whether they are equal or not; whether they are different paths to the same file (hard links); and whether one path is a proper prefix of the other. Again, we ensure we have at least one test case for each logically-possible combination of properties.

The construction of equivalence classes is carried out manually: extensive human involvement is necessary to determine which combinations of properties are logically possible. For example, it makes no sense to require that a path corresponds to an empty directory, and is at the same time a proper prefix of a path that corresponds to a file (or directory or symlink). Potentially the model itself could be used to determine that certain combinations are not possible. We suspect that this could not be done automatically, but would require significant proof effort for each combination, and there are many logically-impossible combinations. Even if we could automatically determine whether a combination was logically possible, constructing missing test cases requires human involvement (see below). Instead of using the model, we manually inspect each combination for which no test case is available, to certify that the combination is indeed impossible. This takes significant effort, and there is the danger that the human mistakenly labels some combination as impossible, and thereby omits an interesting test case. If a combination *is* possible, but no test case exists, we manually examine the combination, identify (at least one) missing test case, and extend our automatic test generation to include this case. As an example of the missing test cases our approach uncovered, for commands involving a single path, our test suite initially lacked test cases which resulted in a path resolution error, where the error was not due to a trailing slash on the end of a file. The fix was to include commands that attempt to resolve a nonexistent file *in a nonexistent directory*; a nonexistent file (in an existing directory) does not suffice since it resolves to `RN_none` rather than `RN_error`. We used OCaml to model properties and equivalence classes, and mechanically verify that all logically-possible combinations were matched by at least one test case.

For commands such as `read` and `write` we need to test sequences of calls, which is inherently hard to test combinatorially. We therefore wrote extensive manual tests,

attempting to cover all possible behaviours. We have done preliminary investigation of automated generation of tests for these calls but this is future work. An alternative is to use randomized testing.

The standard OpenGroup POSIX test suite includes hand-written code to check the results of calling libc functions. Our use of combinatorial testing, made possible by the SibylFS oracle, allows us to test many more cases: 2 500 autogenerated scripts for `rename` alone, supplemented by further hand-written scripts, whereas the OpenGroup test suite for `rename` includes around 50 tests. On the other hand, they test a wide range of POSIX functionality, whereas we test file system functions only.

6.2 Script execution

Test scripts may involve multiple processes making libc file system calls. Each test script execution forks an interpreter process from the controller process to provide signal and fault isolation. The interpreter process then reads, parses, and dispatches script commands over a high-`fd` UNIX socket to worker processes running in a `chroot` jail. Each worker runs with real user and group IDs and supplementary group IDs generated to match the permissions relations for the corresponding process in the script. Our use of `chroot` jails means that we can effectively test as if the file system namespace is empty. This design trades off complete accuracy regarding the behaviour of the root directory (e.g., in a `chroot` jail the root directory link count is typically off-by-one compared to a non-chroot setup), for fast, reliable execution. We have considered testing modes involving per-test virtualization but have not yet constructed such a test harness.

6.3 Testing, interleaving, concurrency and races

The SibylFS model allows interleaving and concurrent behaviours and many test scripts involve multiple processes making interleaved libc calls. However, in-kernel racy behaviours are inherently difficult to elicit from real-world systems, and such racy behaviour, although modelled, is not currently tested. In this section we clarify the nature of the interleaving and concurrent behaviours allowed by the SibylFS oracle, and the difficulty in testing racy behaviours.

A file system API function call and return is not modelled as an atomic event. Instead, there is an initial event corresponding to the call, a second internal τ event corresponding to the libc/OS/file system processing the call, and a final event corresponding to the return from the libc call. Additionally, the model satisfies a receptivity property: at any time, any running process can make a libc call, at which point the process blocks until the call returns. This model allows multiple processes to execute calls concurrently. Test scripts can involve multiple processes, each making calls to libc. For example, many of our hand-written test scripts involve multiple processes making interleaved calls to libc, in order to test file system features such as ownership and permissions. The test infrastructure will execute a script line-by-line and no attempt is made to execute calls from different processes at the same time: typically a libc call from one process will complete before the test infrastructure executes a call from another process.

It should be possible to extend the test infrastructure to initiate libc calls from different processes simultaneously, perhaps by assigning different processes to different cores. This would at least make it *possible* for concurrent calls to race in the kernel, but the probability of a race actually occurring would likely be very low (there is no way to force calls to race in the kernel). The next step would be to run such potentially-racy tests many times, to try to increase the chance of racy behaviour being observed. However, the time cost of

doing this for a large test suite such as ours is prohibitive, and such racy testing should probably be restricted to particular test scenarios where the racy behaviour is expected to be "interesting".

7. Evaluation and test results

Testing focused on the Linux, OS X and FreeBSD operating systems for which we have models. On Linux, we tested tmpfs, Btrfs, ext2, ext3, ext4, F2FS, XFS, HFS+, MINIX, NILFS2, NFSv3/tmpfs, NFSv4/tmpfs, fusexmp/tmpfs (the example FUSE pass-through backed by tmpfs), SSHFS/tmpfs, bind/tmpfs, posixovl/VFAT, posixovl/NTFS-3G, aufs/tmpfs/ext4, overlay/tmpfs/ext4, GlusterFS/XFS, and OpenZFS. On OS X, we tested HFS+, NFSv3/HFS+, fusexmp/HFS+, SSHFS/HFS+, fuse-ext2, Paragon ExtFS, and OpenZFS. On FreeBSD, we tested tmpfs and ufs. In addition, on Linux we compared the standard libc (glibc) and the lightweight libc musl, and kernels 3.13, 3.14, and 3.19.

An individual test run currently executes 21 070 tests and produces 46MB of trace data. Because manually analysing system traces of this volume is difficult, we index, filter, and highlight specification deviations in HTML. Our tools can also produce merged test runs comparing local specification deviations across multiple platforms with platform differences identified and highlighted. With appropriate experimental design, OS, file system, and libc defects are easy to find.

7.1 Performance

To use our specification during file system development or behavioural exploration, individual script execution and trace checking must run quickly. As described in §3, nondeterminism can, without careful management, lead to very long run times. In our checking system, we have both engineered the specification to ruthlessly control nondeterminism and taken advantage of trace independence for parallel speedup.

Trace checking the entire test suite with 4 processes on a machine running Linux 3.14-2 with an Intel Core i7-3520M 2.90GHz CPU with performance governor, Samsung 840 PRO SSD, and 12GB RAM takes about 79s, which is a mean rate of 266 test traces per second. With test suite execution on Linux tmpfs clocking in at 152s, it takes less time to check a trace set than it does to execute the test suite. Our naïve single-threaded HTML generator takes about 48s to process a single, unmerged test run. Thus, we believe the performance of SibylFS is suitable for use during development and continuous integration. The slowest phase of testing is due to user and group creation: we need to employ application-level locking to avoid race conditions on Linux, OS X, and FreeBSD; these race conditions have been reported upstream. We have not yet aggressively optimized either the test harness or the checker architecture. For example, we spawn a new process for each trace being checked.

7.2 Test results

Trace acceptance For the "standard" Linux platforms (Linux 3.19, with glibc and either ext2, ext3, or ext4), all but 9 of 21 070 traces are accepted by SibylFS. The 9 failures are mostly due to the use of a `chroot` jail for testing, i.e., they do not represent real deviations of the underlying file system from the SibylFS model. For other Linux platform variations, failing traces differ mostly in aspects that POSIX indicates are implementation-defined or unspecified. These include default permissions for symlinks, writing 0 bytes to bad file descriptors, and specific errors due to removal or renaming of the root directory.

On OS X 10.9.5 with the default HFS+ file system, the script which tests `pwrite` with a negative offset fails to execute to completion due to an integer underflow bug in OS X (§7.3.4). In total, 34 traces fail to check, due to a handful of issues similar to the Linux failures, and the resolution of symlinks with trailing slashes (which we are presently correcting). The FreeBSD results are similar.

Test coverage To understand the completeness of our test suite, we measured the proportion of lines in the model that are covered by a test run (statement coverage). The ideal target of 100% coverage is not possible for two reasons. First, some lines of the model correspond to situations that are impossible to reach. However, the fact that these situations are not reachable is often far from obvious, so we have explicitly included annotated lines covering these cases as a form of documentation, and as a guarantee of exhaustivity of match clauses. Second, clauses for particular platform behaviour will not be exercised when checking traces for another platform. This can be addressed by annotating each line of the specification with the relevant platforms. Taking these factors into account, our tests currently cover 98% of the model. The remaining 2% consist of lines that probably could be tested (for example, we do not currently test process destruction during a test), and lines for unused internal definitions generated by Lem (which should be excluded from our analysis).

The high level of coverage is partially attributable to our decision to use automatically generated test cases in an attempt to exhaustively explore all behaviours. Related work [14] has used randomized testing of the POSIX file system interface applied to a novel file system implementation, achieving 89.06% coverage of the implementation code. If one considers the model as a (non-deterministic) reference implementation, there is a sense in which these figures are comparable.

Our coverage figures come with a caveat: these figures show only that the test scripts produce traces whose checking exercises almost all of the model. As noted earlier, it may still be the case that the assumptions underlying equivalence partitioning are invalid or that some real-world behaviour, unrepresented in the model, is not being tested.

Our tests aim for complete implementation code coverage, but we use coverage of the model, rather than coverage of the implementations, for two main reasons. First, we believe that the model is detailed and accurate (although admittedly this belief partly depends on the testing itself), so that tests which cover the model should also exercise all interesting behaviours of the implementations. Second, attempting to measure implementation coverage is difficult. There are at least three distinct pieces of code which form an implementation (the libc library, the OS, and the file system implementation code), and only parts of each piece are in the domain of the SibylFS model. In order to measure implementation coverage, we would first need to determine, for each of the three pieces, which lines of code are relevant, and which are not, so that we can restrict our coverage checking to the relevant lines. This requires expert knowledge of libc, the OS and the file system code, but should be possible for a single test platform, and we believe this would provide further evidence that the tests provide high coverage. However, providing such implementation coverage for each of the many combinations of libc, OS and file system that we consider, would surely be infeasible.

7.3 Survey results

In our testing of over 40 different system configurations, we discovered numerous deviations from the specification ranging from mundane to critical. We classify by increasing severity the file system defects found during our survey.

7.3.1 Issues in the POSIX specification

POSIX specifies the behaviour of each libc function separately, with clauses for common errors duplicated between functions. Almost inevitably it is difficult to keep these clauses in sync when updating the POSIX text, and minor mistakes have crept in. Our formal model is in part a formal counterpart to the informal POSIX specification, and clauses in the formal model that were not uniform suggested underlying issues with the POSIX specification. For `link`, `mkdir` and `open` we queried the POSIX specification of the allowable errors on the Austin Group mailing list; new issues were recorded and subsequently resolved on the Austin Group bug tracker.

7.3.2 POSIX specification violation

Core behaviour If we restrict to *successful* invocations of libc functions, for file system states which do not contain symlinks, and paths that do not end in a trailing slash, and if we ignore permissions and work with a single process, then the behaviour across most system configurations is very similar. On some file systems, specific features such as directory link counts are not supported. Btrfs, SSHFS/tmpfs, and Linux HFS+ all exhibit this violation with SSHFS/tmpfs also not supporting link counts for regular files due to limitations in the SFTP protocol.

Error codes POSIX often allows different errors in a given circumstance, and this looseness is present in implementations: Linux is substantially different from OS X and, even on the same operating system, different file systems can return different errors in the same situation. There are also cases where error codes not allowed by POSIX are returned; for example, Linux follows the LSB for `unlink` of directories and returns `EISDIR`, where OS X follows POSIX and returns `EPERM`. On OS X, when attempting to rename the root directory, `EISDIR` is returned instead of `EBUSY` or `EINVAL`.

Path resolution, trailing slashes, and symlinks Trailing slashes on paths, even without symlinks, are treated in what appears to be an *ad hoc* manner. For example, if `f.txt` is a path to a file, then `f.txt/` intuitively should result in an error, but often such a path is resolved successfully. For example, on Linux `link /dir/ /f.txt/` can return `EEXIST` to indicate that the file `f.txt` exists (this is not allowed by POSIX), whereas one might expect `ENOTDIR` to indicate that the path `/f.txt/` cannot be resolved because `f.txt` is not a directory. Symlinks introduce further complications. For example, a path to a symlink followed by a trailing slash is often used to mean "resolve to the target of the symlink (even if a file)", but this is not universally followed on either Linux or OS X. The behaviour when symlinks to symlinks are involved can be confusing. For example, if `s1` is a symlink to a directory, and `s2` is a symlink to `s1`, then on OS X, `readlink s2/` will return the contents of the symlink `s1`, whereas one might expect that the trailing slash would force the path to be resolved to the directory, resulting in an `EINVAL` error returned by `readlink`. Creation of hard links to symlinks using `link` is permitted by Linux and support is specified as implementation-defined. Notably, HFS+ on Linux returns `EPERM` when this is attempted rather than either linking the symlink or following the symlink as OS X does. This behaviour is likely a portability compromise for removable volumes.

Invariants POSIX specifies that calling `open` with flags `O_CREAT`, `O_DIRECTORY` and `O_EXCL` on a symlink to an existing directory should fail with `EEXIST`. FreeBSD instead returns `ENOTDIR`. POSIX also mandates a strong invariant: a libc call which returns with an error should leave the underlying file system state unchanged. On Linux and OS X this

invariant holds for all our tests. However, in the above scenario, as well as returning the ENOTDIR error, FreeBSD deletes the symlink and replaces it with a newly created file. This breaks the POSIX invariant. If the symlink points to a non-existent target rather than a directory, and the flag O_EXCL is omitted, then the new file is created as the target of the symlink and ENOTDIR is returned, again violating the invariant.

7.3.3 Platform conventions

Some platforms, such as Linux, have well-known and longstanding defects in their POSIX compliance. For example, on Linux, calling pwrite on a file descriptor opened with O_APPEND will ignore the offset and instead append data to the file. It is crucial that any file system or application software ported to or from Linux ensure that it follows this convention on Linux and provides or expects POSIX compliance on operating systems that attempt POSIX compliance. Our specification and development process ensures that we explicitly express and check behaviour of this kind.

7.3.4 Defects likely to cause application failure

A comparison of SSHFS/tmpfs mount options An organization's system administrator might consider deploying a shared SSHFS/tmpfs mount to their users and wonder what mount options to use in the configuration scripts. With SibylFS, the administrator can easily compare, in under an hour, the behaviour of various mount configurations *in their specific deployment* of SSHFS/tmpfs and conclude that, using *only* allow_other is dangerous because it allows users to violate permissions, using allow_other *and* default_permissions is safer but still is not adequate for a shared mount deployment due to SSHFS/tmpfs's unconfigurable default creation ownership set to the mount owner (root). Additionally, without a mount option umask, a user process's umask is bitwise *ORed* with 0022 (regardless of the parent process's umask) but when setting a mount option umask of 0000, a user process's umask is ignored entirely. Using this empirical evidence, the system administrator is now informed enough to reject SSHFS/tmpfs for this deployment scenario.

OS X VFS pwrite **integer underflow and signal** POSIX specifies a negative offset to pwrite should return an EINVAL error. We believe the OS X VFS layer incorrectly uses an unsigned integer type for the offset argument to pwrite which causes negative values to be interpreted as extremely large positive values, and the operating system then sends a SIGXFSZ signal to the process which almost certainly does not handle it. This results in premature, potentially unclean process termination for what otherwise would be a simple error condition.

Various issues in deployed but older versions of Linux In Ubuntu "Trusty" Linux 3.13.0-34, HFS+ did not support chmod and would return EOPNOTSUPP for every chmod call. This was not the case in Debian "sid" Linux 3.14-2.

In OpenZFS 0.6.3-2~trusty, also on Ubuntu "Trusty" Linux 3.13.0-34, files opened with O_APPEND would not seek to the end of the file before *either* write or pwrite potentially resulting in application malfunction and data loss or corruption.

7.3.5 Defects causing system halt, data loss, or resource exhaustion

posixovl/VFAT 1.2 storage leak posixovl is an overlay file system which provides POSIX functionality on file systems such as VFAT. Our test suite revealed that posixovl/VFAT fails to decrement the hard link count correctly in certain rename scenarios. We wrote a simple

```
mkdir("deserted",0700);
chdir("deserted");
rmdir("../deserted");
open("party",O_CREAT | O_RDONLY,0600);
```

Figure 8. Function call sequence causing OpenZFS 1.3.0 on OS X 10.9.5 to unkillably spin processes

C program to repeatedly create 64MB files with hard links and delete them using `rename`. On Linux 3.14, this resulted in the process receiving a `SEGFAULT`. On Linux 3.19, we found that the `open` with `O_CREAT` libc call would fail with `ENOENT`. In both cases, the file system would have no remaining space despite being empty – even through an unmount cycle.

OpenZFS on OS X unkillably spins processes in a disconnected directory case OpenZFS 1.3.0 on OS X 10.9.5 has a defect which, after executing the sequence of function calls in Fig. 8, causes the calling process to consume 100% CPU and ignore all signals. The file system is still usable by other processes at this time but the OpenZFS volume cannot be unmounted and the machine cannot be shutdown. Force unmounting the OpenZFS volume may succeed and release the storage device or may cause the storage device to become unusable until the next restart.

8. Related work

Model checking Yang et al. [35] have used model checking to find serious file system errors. Their FiSC tool included a simple model of file system state (name, size and link count for files and directories), sufficient for finding errors, but not intended as a realistic model of file systems in the way that SibylFS is. FiSC requires intrusive access to file system internal state: "ReiserFS took between one and two weeks of effort to run in FiSC as it violated one of the larger assumptions we made". In contrast, we have opted to test file systems solely via the libc interface, making it trivial to test new file systems. FiSC is also focused on errors typically arising from host crashes. SibylFS does not currently model such scenarios at all.

Ad-hoc models FiSC includes a simplified, ad-hoc file system model. Such models are reasonably common. For example, the COMMUTER tool [6] includes a model expressed in a symbolic variant of Python. The model is simplified, e.g., filenames have no structure and can only be compared for equality, and there is no support for symlinks. Even these simplified models can take significant time to develop, and are typically not reused across projects. We believe the SibylFS model is more detailed and better validated, and we hope that it will be reused in place of such ad hoc models in future file system research projects.

The COMMUTER project is similar to our work in other respects. They use equivalence partitioning to ensure only a finite number of tests are generated, which nevertheless "cover all possible paths and data structure access patterns in the model". As with our work, they focus on coverage of the model, rather than implementation code. Moreover, their tests are somewhat simplified because, in addition to the model simplifications described above, their test cases do not deal with directories (other than the root directory).

Differential testing Differential testing compares the behaviour of multiple implementations to identify possible errors without a reference model [26]. In some cases it can be very effective, e.g. for C compilers [36]: by restricting the domain to C programs that (ac-

cording the C standard) should be deterministic, any behavioural difference in compiled programs identifies a compiler bug. File systems are more complicated to test because of nondeterminism, with a large envelope of allowable behaviours within which file systems are expected to behave differently, so one cannot simply compare runtime behaviours without a reference model that identifies when they are sufficiently similar. SibylFS instead allows differential testing of multiple file systems taking this allowable variability into account. In this sense it improves on differential testing, but the downside is the effort needed to construct the model.

Differential testing has also been applied to a novel file system implementation [14] to ensure it behaved the same as a reference implementation.That paper also applied randomized testing to file systems, a low-cost alternative (that SibylFS also supports) to the model checking approach described earlier. SibylFS can also be used as a reference implementation by determinizing the model (selecting one of the many possible states at each step) and we have mounted previous versions of SibylFS as prototype FUSE file systems under Linux. The good performance of the SibylFS test oracle should also make it feasible to integrate with dynamic verification engines such as EnvyFS [2] or Recon [12].

Formal methods Previous models of file systems [11, 27] do not aim to capture the full complexity of POSIX or real-world file systems. As a result they are usually much simpler than our model: symlinks, permissions and timestamps are ignored, and there is no model of concurrent processes and per-process datastructures. Recently Schierl et al. gave an abstract specification of a single file system: UBIFS [32]. However, this is not a general model of POSIX. No previous work that we are aware of has attempted to formally model the subtle behavioural quirks of real-world systems as we do for Linux, OS X and FreeBSD. Work on verified implementations is complementary to our work: it should be possible to prove that a verified implementation behaves according to our model. Implementations of file systems have previously been formally verified [7, 8, 10, 11, 16, 18], but these are highly idealized and do not represent realistic file system implementations. The seL4 team previously produced a verified operating system [20], and some of the researchers are now working on a formally verified file system implementation [19]. Another approach [4] uses a modified Hoare logic inside the Coq theorem prover to attempt to prove correctness of a novel file system implementation. The specification is based on POSIX, but does not attempt to deal with the full variability allowed by POSIX and real-world implementations, since the focus is on a single verified implementation. The authors note, "we found that significant care is needed when writing specifications [...] it is easy to write an incomplete specification that does not eliminate the possibility of some bugs". As with other ad hoc models, SibylFS could be used as an alternative, high-quality specification. Recently Ernst et al. [9, 29, 31] achieved a significant milestone by producing a verified implementation based on UBIFS that is actually usable as a flash file system.

Preliminary work on specifying the semantics of storage stacks in Isabelle/Isar has been carried out [1]. The researchers argue for expressive logics to capture specifications of each layer, and theorem prover support for proving that file system stacks satisfy the desired guarantees. The researchers list "obtaining specifications" as one of the main challenges. At least for the uppermost layer that is exposed to the application, SibylFS can provide such a specification.

For reasoning about POSIX file system behaviour, Gardner et al. [13] proposed a variation of separation logic. The SibylFS model could be used as a basis to prove soundness for this logic.

File system innovation Recent studies have shown that the workloads imposed on POSIX file systems now vary widely [15], and there are also many new FUSE-based file systems such as Ori [25], OptFS [5], and kernel-based ones such as Betrfs [17] and ReconFS [24] that optimise particular use cases. File system evolution in this style often results in subtle semantic and data corruption bugs [23], and SibylFS is the first rigorous specification that can be used, in a developer-friendly way, to test directly that these implementations remain POSIX compliant.

9. Conclusion and future work

SibylFS gives a detailed, precise, and usable characterisation of the behaviour of a range of file systems. As described in §1, it should be immediately usable for multiple purposes, and it also opens up many directions for future work.

With modest additional engineering, SibylFS could support analysis of API traces of applications, identifying when they rely on non-portable aspects of the model; by providing an executable test oracle it could support randomised testing; and it could support automatic test case reduction. It could and should also be extended with extensive testing of concurrent API calls and timestamps, with modelling and testing of behaviour in the presence of crash failure, and with testing that imposes more resource stress on file system implementations. On the theoretical side, SibylFS provides, in the HOL4 and Isabelle/HOL theorem prover definitions generated from our model by Lem, a basis for rigorous proof about file system properties and about software that uses file systems. Especially, it can serve as a specification of correctness for work on verified file systems.

Most generally, one can see SibylFS as a novel instrument for examining behavioural differences in a largely black-box fashion. We have done this for file systems, but much of our approach should also be applicable in other contexts.

Acknowledgements We acknowledge funding from EPSRC grants EP/K022741/1 (Future filesystems, Ridge and Tuerk), and EP/K008528/1 (REMS Programme Grant, Sewell). This work was supported by Microsoft Research through its PhD Scholarship Programme.

References

[1] Alagappan, R., Chidambaram, V., Pillai, T. S., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. Beyond storage APIs: Provable semantics for storage stacks. In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)* (Kartause Ittingen, Switzerland, May 2015), USENIX Association.

[2] Bairavasundaram, L. N., Sundararaman, S., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. Tolerating file-system mistakes with EnvyFS. In *Proceedings of the 2009 Conference on USENIX Annual Technical Conference* (Berkeley, CA, USA, 2009), USENIX'09, USENIX Association, pp. 7–7.

[3] Bishop, S., Fairbairn, M., Norrish, M., Sewell, P., Smith, M., and Wansbrough, K. Rigorous specification and conformance testing techniques for network protocols, as applied to TCP, UDP, and Sockets. In *Proceedings of SIGCOMM 2005: the ACM Conference on Computer Communications (Philadelphia),* published as Vol. 35, No. 4 of *Computer Communication Review* (Aug. 2005), pp. 265–276.

[4] Chen, H., Ziegler, D., Chlipala, A., Kaashoek, M. F., Kohler, E., and Zeldovich, N. Specifying crash safety for storage systems. In *Proceedings of the 15th Workshop on Hot Topics in Operating Systems (HotOS XV)* (Kartause Ittingen, Switzerland, May 2015).

[5] CHIDAMBARAM, V., PILLAI, T. S., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Optimistic crash consistency. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2013), SOSP '13, ACM, pp. 228–243.

[6] CLEMENTS, A. T., KAASHOEK, M. F., ZELDOVICH, N., MORRIS, R. T., AND KOHLER, E. The scalable commutativity rule: Designing scalable software for multicore processors. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP 2013)* (Farmington, Pennsylvania, November 2013).

[7] DAMCHOOM, K., AND BUTLER, M. Applying event and machine decomposition to a flash-based filestore in Event-B. In *Formal Methods: Foundations and Applications*. Springer, 2009, pp. 134–152.

[8] DAMCHOOM, K., BUTLER, M., AND ABRIAL, J. Modelling and proof of a tree-structured file system in Event-B and Rodin. *Formal Methods and Software Engineering* (2008), 25–44.

[9] ERNST, G., SCHELLHORN, G., HANEBERG, D., PFÄHLER, J., AND REIF, W. Verification of a virtual filesystem switch. In *Verified Software: Theories, Tools, Experiments VSTTE 2013*. Springer, 2014, pp. 242–261.

[10] FERREIRA, M., AND OLIVEIRA, J. An integrated formal methods tool-chain and its application to verifying a file system model. *Formal Methods: Foundations and Applications* (2009), 153–169.

[11] FREITAS, L., FU, Z., AND WOOCOCK, J. POSIX file store in Z/Eves: an experiment in the verified software repository. In *12th IEEE International Conference on Engineering Complex Computer Systems, 2007* (2007), IEEE, pp. 3–14.

[12] FRYER, D., SUN, K., MAHMOOD, R., CHENG, T., BENJAMIN, S., GOEL, A., AND BROWN, A. D. Recon: Verifying file system consistency at runtime. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies* (Berkeley, CA, USA, 2012), FAST'12, USENIX Association, pp. 7–7.

[13] GARDNER, P., NTZIK, G., AND WRIGHT, A. Local reasoning for the POSIX file system. In *Programming Languages and Systems*. Springer, 2014, pp. 169–188.

[14] GROCE, A., HOLZMANN, G. J., AND JOSHI, R. Randomized differential testing as a prelude to formal verification. In *29th International Conference on Software Engineering (ICSE 2007), Minneapolis, MN, USA, May 20-26, 2007* (2007), IEEE Computer Society, pp. 621–631.

[15] HARTER, T., DRAGGA, C., VAUGHN, M., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. A file is not a file: Understanding the I/O behavior of Apple desktop applications. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2011), SOSP '11, ACM, pp. 71–83.

[16] HESSELINK, W., AND LALI, M. Formalizing a hierarchical file system. *Formal Asp. Comput. 24*, 1 (2012), 27–44.

[17] JANNEN, W., YUAN, J., ZHAN, Y., AKSHINTALA, A., ESMET, J., JIAO, Y., MITTAL, A., PANDEY, P., REDDY, P., WALSH, L., BENDER, M., FARACH-COLTON, M., JOHNSON, R., KUSZMAUL, B. C., AND PORTER, D. E. BetrFS: A right-optimized write-optimized file system. In *13th USENIX Conference on File and Storage Technologies (FAST 15)* (Santa Clara, CA, Feb. 2015), USENIX Association, pp. 301–315.

[18] KANG, E., AND JACKSON, D. Formal modeling and analysis of a flash filesystem in Alloy. *Abstract state machines, B and Z* (2008), 294–308.

[19] KELLER, G., MURRAY, T., AMANI, S., O'CONNOR, L., CHEN, Z., RYZHYK, L., KLEIN, G., AND HEISER, G. File systems deserve verification too! *ACM SIGOPS Operating Systems Review 48*, 1 (2014), 58–64.

[20] KLEIN, G., ELPHINSTONE, K., HEISER, G., ANDRONICK, J., COCK, D., DERRIN, P., ELKADUWE, D., ENGELHARDT, K., KOLANSKI, R., NORRISH, M., ET AL. seL4: Formal

verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles* (2009), ACM, pp. 207–220.

[21] LINUX FOUNDATION. Linux Standard Base (LSB). http://www.linuxfoundation.org/collaborate/workgroups/lsb.

[22] LINUX TEST PROJECT. Linux Test Project testsuite. http://linux-test-project.github.io/. Accessed 2015.03.26.

[23] LU, L., ARPACI-DUSSEAU, A. C., ARPACI-DUSSEAU, R. H., AND LU, S. A study of Linux file system evolution. In *Presented as part of the 11th USENIX Conference on File and Storage Technologies (FAST 13)* (San Jose, CA, 2013), USENIX, pp. 31–44.

[24] LU, Y., SHU, J., AND WANG, W. ReconFS: A reconstructable file system on flash storage. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST 14)* (Santa Clara, CA, 2014), USENIX, pp. 75–88.

[25] MASHTIZADEH, A. J., BITTAU, A., HUANG, Y. F., AND MAZIÈRES, D. Replication, history, and grafting in the Ori file system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2013), SOSP '13, ACM, pp. 151–166.

[26] MCKEEMAN, W. M. Differential testing for software. *Digital Technical Journal 10*, 1 (1998), 100–107.

[27] MORGAN, C., AND SUFRIN, B. Specification of the Unix filing system. *Software Engineering, IEEE Transactions on 10*, 2 (1984), 128–142.

[28] MULLIGAN, D. P., OWENS, S., GRAY, K. E., RIDGE, T., AND SEWELL, P. Lem: reusable engineering of real-world semantics. In *Proceedings of ICFP 2014: the 19th ACM SIGPLAN International Conference on Functional Programming* (2014), pp. 175–188.

[29] PFÄHLER, J., ERNST, G., SCHELLHORN, G., HANEBERG, D., AND REIF, W. Crash-safe refinement for a verified flash file system. Tech. rep., University of Augsburg, 2014.

[30] RIDGE, T., NORRISH, M., AND SEWELL, P. A rigorous approach to networking: TCP, from implementation to protocol to service. In *Proceedings of FM 2008: the 15th International Symposium on Formal Methods (Turku, Finland), LNCS 5014* (May 2008), pp. 294–309.

[31] SCHELLHORN, G., ERNST, G., PFHLER, J., HANEBERG, D., AND REIF, W. Development of a verified flash file system. In *Abstract State Machines, Alloy, B, TLA, VDM, and Z*, Y. Ait Ameur and K.-D. Schewe, Eds., vol. 8477 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2014, pp. 9–24.

[32] SCHIERL, A., SCHELLHORN, G., HANEBERG, D., AND REIF, W. Abstract specification of the UBIFS file system for flash memory. *FM 2009: Formal Methods* (2009), 190–206.

[33] THE IEEE AND THE OPEN GROUP. *The Open Group Base Specifications Issue 7 – IEEE Std 1003.1, 2008 Edition*. IEEE, New York, NY, USA, 2008.

[34] THE OPEN GROUP. POSIX Conformance Test Suite. http://www.opengroup.org/testing/downloads.html. Accessed 2015.03.26.

[35] 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 (Nov. 2006), 393–423.

[36] YANG, X., CHEN, Y., EIDE, E., AND REGEHR, J. Finding and understanding bugs in C compilers. *SIGPLAN Not. 46*, 6 (June 2011), 283–294.