Testing File System Implementations on Layered Models

Dongjie Chen
State Key Lab for Novel Software Technology and Department of Computer Science and Technology, Nanjing University, Nanjing, China
dongjie_cdj@163.com

Yanyan Jiang
State Key Lab for Novel Software Technology and Department of Computer Science and Technology, Nanjing University, Nanjing, China
jyy@nju.edu.cn

Chang Xu
State Key Lab for Novel Software Technology and Department of Computer Science and Technology, Nanjing University, Nanjing, China
changxu@nju.edu.cn

Xiaoxing Ma
State Key Lab for Novel Software Technology and Department of Computer Science and Technology, Nanjing University, Nanjing, China
xxm@nju.edu.cn

Jian Lu
State Key Lab for Novel Software Technology and Department of Computer Science and Technology, Nanjing University, Nanjing, China
lj@nju.edu.cn

ABSTRACT

Generating high-quality system call sequences is not only important to testing file system implementations, but also challenging due to the astronomically large input space. This paper introduces a new approach to the workload generation problem by building layered models and abstract workloads refinement. This approach is instantiated as a three-layer file system model for file system workload generation. In a short-period experiment run, sequential workloads (system call sequences) manifested over a thousand crashes in mainline Linux Kernel file systems, with 12 previously unknown bugs being reported. We also provide evidence that such workloads benefit other domain-specific testing techniques including crash consistency testing and concurrency testing.

KEYWORDS

Model-based testing, workload generation, file system

ACM Reference Format:
Dongjie Chen, Yanyan Jiang, Chang Xu, Xiaoxing Ma, and Jian Lu. 2020. Testing File System Implementations on Layered Models. In *42nd International Conference on Software Engineering (ICSE '20), May 23–29, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3377811.3380350

1 INTRODUCTION

The reliability of file systems is of critical importance because they are fundamental components in operating systems for persisting valuable application data. Bugs in file systems may result in serious consequences, such as kernel panic [39], data loss [1, 7, 51], or even security vulnerabilities [50, 57, 66]. Although file system implementations have been extensively validated by existing work like regression test suits [6], fuzzing [66], and model checking [69] (with hundreds of bugs being revealed), file system bugs are still threatening the operating system's reliability.

To validate such complex systems as file systems, we need to generate high-quality test inputs (a.k.a. workloads) to drive systems toward potentially buggy states. For file systems, we should generate *file system call sequences* to manifest bugs. However, effective workload generation is challenging because the input space of systems is astronomically large. For example, each file system call comprises various parameters, there are tens of such calls, and file system calls in a sequence influence each other.

Though there exist techniques to systematically generate workloads, they may be still ineffective to validate system implementations. For example, existing fuzzers are semantic-unaware and behave more or less like a random system call generator because coverage information [3] usually cannot effectively drive a gray-box fuzzer [5] to diverse file system states. On the other hand, building a precise and complete model for complex systems is unrealistic due to their complexities[1]. Therefore, model checking [69], model-based testing [61, 63], or formal verification [9, 53] are limited to a small subset of core functionalities.

This paper proposes a model-based approach to tackling the workload generation problem. The key idea is to *decompose* the system into *layered models* in consistent with the system's evolution history. Model construction starts from a high-level *core model* which captures the *simplified* semantics of the system but also reflects the fundamental functionalities designed in early versions of the system. We should systematically exercise diverse abstract workloads (e.g., using model checking or model-based testing) and incrementally *refine* them to obtain lower-level ones, which further manifest sophisticated system behaviors developed during the evaluation. Finally, we reach a syntactically complete model (theoretically covers all possible workloads) in which workload behaviors are impractical to specify.

ICSE '20, May 23–29, 2020, Seoul, Republic of Korea

ACM ISBN 978-1-4503-7121-6/20/05…$15.00
https://doi.org/10.1145/3377811.3380350

[1] File system is a *concurrent* data structure in which data can be organized (e.g., buffered, cached, or journaled) in various ways. Modern file system implementations contain tens of thousands of lines of code. Even the POSIX standard leaves many file system behaviors unspecified [53].

In the rounds of refinements, a high-level (abstract) workload w may correspond to multiple refined lower-level (abstract or concrete) workloads W'. Therefore, we should deliberately design the layered models such that $w' \in W'$ will behave mostly similar to w. Suppose that we have (and we later show such a design) a high-level model for generating high-quality *diverse* abstract workloads, in the rounds of refinements, (1) the *diversity* of high-level workloads is preserved; (2) *randomness* is added to the workloads for exercising system behaviors that are impractical to model. By striking a delicate trade-off between the controlled diversity and the randomness, this paper offers new opportunities to fully exploit the potential of incomplete models in effectively generating high-quality workloads.

Following such an intuition, this paper develops a three-layer file system model for high-quality (sequential) workload generation:

(1) The *core model* captures the common and predictable semantics of all existing UNIX file system variants. The core model provides a minimal abstraction of the directed acyclic graph structure of file systems, plus a set of *abstract file system calls* to manipulate the graph structure;
(2) The *core-ext model* refines workloads in the core model by adding a few extension abstract file system calls. Such extension calls manifest sophisticated functionalities developed during the system evolution, which are impractical to precisely specify. Examples of extension file system calls are reading/writing an open file, accessing extended attributes, and performing large structural changes to the file system;
(3) The *syntax model* further refines workloads in the core-ext model by syntactically modeling all other system call parameters: permissions, flags, lengths, etc. All possible file system call sequences can be generated by the syntax model.

Instead of directly generating workloads using the syntax model (as the fuzzers do [5]), the layered model separates the concerns in each stage of the workload generation. Particularly, the core and core-ext models are systematically *model-checked*, yielding high-quality "abstract" system call sequences, which are then filled with *random* parameters defined in the syntax model to finally obtain runnable workloads.

These ideas are implemented as a prototype tool—`Dogfood`. We evaluated it in three testing scenarios with six actively maintained mainline file systems in the 5.1.3 Linux Kernel, and the experimental results are encouraging: it has detected 12 previously unknown bugs among these file systems in 12 testing hours by generating sequential file system call sequences to test them; when combined with a crash consistency detector, it was able to detect hundreds of issues; and it also benefited concurrency testing by helping find concurrency bugs when executing workloads concurrently. In summary, our contributions are:

- A novel idea to model complex systems for workload generation, which achieves a balance between semantics controllability and randomness in workload generation;
- A three-layer model and checker for file system implementations, which generates effective file system call sequences; and
- A prototype tool and its evaluation, which detected 12 previously unknown bugs.

The rest of this paper is laid out as follows. Section 2 introduces the layered modeling approach to testing complex systems. Section 3 gives an overview of testing file systems on layered models. Section 4 comprehensively defines the abstract file system models and introduces the workload generation algorithm. Section 5 gives an overview of our prototype tool and its implementation, followed by the evaluation in Section 6. Section 7 describes related work and Section 8 concludes this paper.

2 LAYERED MODELING AND TESTING OF COMPLEX SYSTEMS

2.1 Background

It is a challenge to generate high-quality workloads to test complex software systems. Even if we clearly know the *syntax* of a system's inputs (workloads), it would be too difficult to completely specify the exact *semantics* of the system. For example, a file system provides a few well-structured system call APIs to the userspace. However, the semantics of a file system is far beyond one's understanding: Btrfs [21, 54] is ~100K LoC and extensively interacts with other components in the operating system. Another example is compilers which receive a program of unambiguously documented syntax (usually within a few pages). However, even a "tiny" C compiler is 47K LoC in size [11].

To test such systems, one can apply grammar-based fuzzing [10, 28, 30] that generates large amounts of random inputs under feedback (e.g., code coverage) guidance. However, fuzz testing is usually semantic-unaware and generates mostly useless trivial test inputs. For example, existing file system workload generators strive to pass early error checkings even with code coverage guidance [5]. Random programs are almost always trivial or failing-early, and thus, they have limited potential in revealing compiler bugs. Alternatively, one can provide a *model* of the system under test for further model checking or model-based testing [16, 23, 60, 69]. However, providing a precise model for adopting existing model-based testing techniques is also generally impractical for complex systems because the full precise specifications of complex systems such as file systems are usually infeasible [51, 53].

2.2 Generating Workloads for Complex Systems on Layered Models

Layered Modeling: An Evolutionary Perspective. To model complex systems for test input (which we call a *workload*) generation, our key observation is that *complex systems are always evolved from simpler ones* and the evolution process serves as a guidance to the model design. Therefore, instead of a direct syntax-guided fuzzing or model-based testing, we argue that workload generation should start from building a highest-level *core model* which is simple, abstract, and reflects the very basic functionalities designed in the early versions of the system. The model is a ordered set

$$M_0 = \{w_1^{(0)}, w_2^{(0)}, \ldots\}$$

and the implementation of M_0 should sequentially generate (abstract) workloads $w_1^{(0)}, w_2^{(0)}, \ldots$

Each lower-level model is a refinement of a higher-level model, which can be regarded as an *refinement* of higher-level model workloads. Given an i-th level abstract workload $w \in M_i$ $(0 \le i \le n)$, the $(i+1)$-th level model $M_{i+1}(w)$ is also an ordered set of workloads:

$$M_{i+1}(w^{(i)}) = \{w_1^{(i+1)}, w_2^{(i+1)}, \ldots\},$$

until the last-level *syntax model* M_n produces executable concrete workloads.

To make workload generation effective, we should design a *semantic-aware* core model with simple and precise specifications. Consequently, abstract workloads of higher-level models can be generated in a more controlled fashion: we expect that workloads generated in the core model are as diverse as possible in terms of core system behaviors, which existed in the very first stages of the system.

Lower-level models should gradually introduce complex system behaviors in an ascending order of difficulties in modeling semantics. Consequently, we (inevitably) lose control of the semantics of generated workloads. However, if we design the model refinement $M_{i+1}(w^{(i)})$ to be semantic-preserving as much as possible, low-level (concrete) workloads will behave similar to abstract ones even if the model refinement is not entirely sound.

Such a modeling methodology naturally applies to many complex systems because literally all man-made systems are eventually originated from scratch. Examples include:

- Programming languages evolve from simple core constructs for describing control-structure-based computations [38]. To generate workloads (programs) for testing a compiler, the simple constructs can be served as the core model, in which the control and data flow of the generated program is determined. Modern compiler extensions (syntactic sugars, function/variable attributes, inline assembly, etc.) can be loosely modeled in lower-level models for exercising diverse compiler behaviors.
- A multi-threaded program first works as a sequential program [49]. Therefore, for testing multi-threaded programs, we can build the core model on coarse-grained concurrency (e.g., function-level/component-level) and systematically generate workloads for diverse program states [18, 35]. Then, lower-level models may focus on fine-grained interleaving specific behaviors, e.g., race or deadlock directed testing [17].
- Protocols of distributed systems are designed before the actual implementation. Such protocols are extensively being model-checked in practice [37, 65, 67] and are natural candidates of core models. One can further validate distributed system implementations by extending abstract workloads with code-level complications, e.g., injecting API failures [31] or exercising thread-level concurrency.

This paper particularly addresses the problem of layered modeling and testing of file system implementations for effective workload generation.

Generating Effective Workloads on Layered Models. To generate effective workloads, we argue that *most human understandable defects can be explained mostly under a simplified system model, plus a few exceptional arguments.* This argument can be regarded as a variant (for complex systems) of the "small scope hypothesis" that a high proportion of bugs can be found by testing the program for all test inputs within some small scope [8, 32, 33]. Empirical evidences supported this claim to some extent. For example, most known (and fixed) concurrency bugs can be manifested within a few context switches [40]; crash consistency bugs can usually be triggered by a short workload and a single injected crash [45].

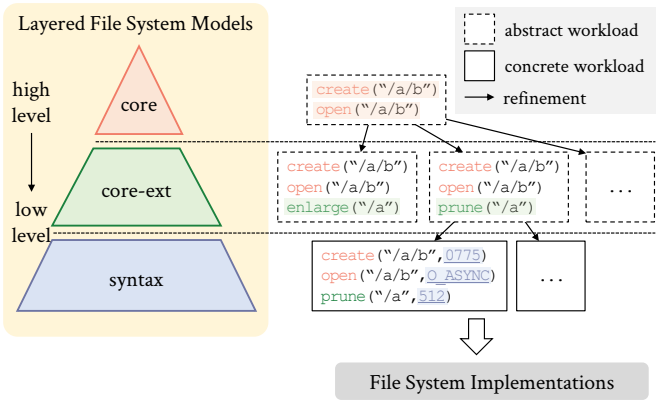

Figure 1: Testing file systems on layered models

Therefore, we recommend exercising abstract workloads systematically using exhaustive enumeration, model checking, or model-based testing for a maximized behavioral coverage (i.e., obtaining *diverse* abstract workloads). In lower-level models, where we gradually lose control of the semantics, we should cautiously generate mostly semantically-preserving workloads with controlled random perturbations.

As an abstract core model workload $w \in M_0$ may correspond to many concrete workloads $\mathcal{W}(w) = M_n(\ldots M_2(M_1(w)))$, we should carefully design the models such that each $w^{(n)} \in \mathcal{W}(w)$ behaves almost the same as w. Then, if abstract workloads w and w' behave differently, we guarantee that all workloads $W(w)$ and $W(w')$ are different ($\mathcal{W}(w) \cap \mathcal{W}(w') = \varnothing$). Therefore, maximizing the diversity of abstract workloads in M_0 will likely result in semantically diverse final concrete workloads.

In summary, we expect layered models to strike a balance between the *semantics controllability* (by semantically modeling in the high-level models) and the *syntax diversity* (by adding randomness to the low-level models). The rest of this paper focuses on realizing this methodology for the problem of file system workload generation.

3 TESTING FILE SYSTEM IMPLEMENTATIONS ON LAYERED MODELS: OVERVIEW

Following the intuitions in Section 2, it would be reasonable to build the *core* model on the modern file systems' common ancestor: the original UNIX file system. The UNIX file system is simple yet provides the fundamental abstractions of all file systems—maintaining a persistent data structure on storage devices.

The lower-level *core-ext* model is designed with caution to preserve the semantics of backbone abstract workloads[2], but exercises

[2]Otherwise a concrete workload may have a high chance to fail early.

```
⟨mode⟩      ::= ⟨mode⟩ ['|' ⟨mode⟩]*
              | S_IRWXU | S_IRGRP | ...
⟨dev-id⟩    ::= INTEGER
⟨flag⟩      ::= ⟨flag⟩ ['|' ⟨flag⟩]*
              | O_RDWR | O_SYNC | ...
⟨data⟩      ::= MEMORY BUFFER ID
⟨size⟩      ::= INTEGER
⟨mount-opt⟩ ::= ⟨mount-opt⟩ [',' ⟨mount-opt⟩]*
              | grpquot | acl | ...
⟨command⟩   ::= mkdir(p, ⟨mode⟩) | create(p, ⟨mode⟩)
              | mknod(p, ⟨mode⟩, ⟨dev-id⟩) | hardlink(p', p)
              | symlink(p', p) | rename(p', p) | remove(p)
              | open(f, p, ⟨flag⟩) | close(f) | chcwd(p)
              | read(f, ⟨data⟩, ⟨size⟩) | enlarge(p, ⟨size⟩)
              | write(f, ⟨data⟩, ⟨size⟩) | deepen(p, ⟨size⟩)
              | remount(⟨mount-opt⟩) | fsync(f) | statfs(p)
              | read_xattr(p, ⟨key⟩, ⟨data⟩) | sync()
              | write_xattr(p, ⟨key⟩, ⟨data⟩) | prune(p)
⟨sequence⟩  ::= ⟨command⟩ ⟨sequence⟩ | ⟨command⟩
```

Figure 2: Syntax of the three-layer model of file system implementations. "[]*" means that symbols in the bracket can repeat any times. p is a path and f is a file descriptor. Notations are defined in Section 4.

even more diverse file system behaviors. It almost does not change the file structures maintained by the core model, but manifests more other file system features. These feature supports are easy to extract and abstract from the evolution of systems, e.g., release notes of main versions provide descriptions about new APIs or parameters.

The lowest-level *syntax* model is a *complete* specification of all file system workloads[3]. It is also mostly semantics-preserving: for an abstract workload $w^{(1)}$ in the core-ext model, $M_2(w^{(1)})$ is a set of executable file system calls obtained by filling in missing system call parameters in $w^{(1)}$ without changing the actual system call sequence.

According to the small scope hypothesis, in case that we do not know what kind of "explanations" may trigger a bug in advance, we should systematically exercise abstract workloads in the core model. This resembles to "enumerating" the backbone of such an explanation.

We (intentionally) make the abstract file system model not a completely sound approximation of file system implementations. For example, if a file system does not support extended attributes or some device files cannot be created, executing a concrete file system call generated by our model will fail and it may influence its following calls. However, even if system call behaviors in a workload may not be captured by the abstract model, the workload still exercises diverse behaviors of the underlying file system implementation.

The three-layer model and the corresponding workload generation technique is shown Figure 1. The model primitives are summarized in Figure 2.

[3]All concrete workloads in $\mathcal{W}$ can be generated by M_2.

The Core Model. The *core model* M_0 captures the direct acyclic graph organization of file and directory structures in UNIX file systems. The heart of the core model is an abstract file system model, which models paths, inodes (file entities), links, and file descriptors (open files) but not the file contents. Ten abstract file system calls are included in the core model for creating/deleting/manipulating abstract files and directories (e.g., `symlink` and `remove`), with their formal semantics unambiguously defined.

Given two abstract workloads in the core model, we can precisely determine their differences in terms of file/directory structures. This property is crucial in our systematic enumeration of diverse abstract workloads.

The Core-Ext Model. In the years of evolution, sophisticated functionalities are added to modern file system implementations, e.g., files may be stored in different data structures when they are of different sizes; the background journaling daemon may periodically write data back to the storage device; and file system call APIs have undergone revisions over time to support more parameters. Generally, it is impractical to precisely specify these behaviors.

To model these behaviors to some extent, the *core-ext model* M_1 inherits the abstract file system in the core model and adds eleven more (abstract) file system calls to manifest diverse file system behaviors such as on-disk file system states and journaling. An example is `deepen` for creating a deep directory (of a random depth) to trigger lots of file system metadata updates.

Given an abstract workload w in the core model M_0, the core-ext model refines w by interleaving it with random core-ext abstract file system calls. We carefully designed the core-ext model such that workloads in $M_1(w)$, though manifesting lots of random file system behaviors, will behave mostly like w in a concrete file system implementation if one only concerns the files and directories in w.

The Syntax Model. The lowest-level *syntax model* M_2 is responsible for generating executable workloads on actual systems. The syntax model is a complete model in the sense that it contains all possible workloads, yet it is impractical to model what happens to the system when workloads are executed, e.g., opening a file with the `O_NONBLOCK` flag. The syntax model is simple: it translates 21 abstract system calls (in the core and core-ext models) to system call sequences, and fills required parameters with random values.

Workload Generation. Generating effective workloads directly using the syntax model (black-box testing) is challenging. Even though a workload is syntactically correct, it may be semantically meaningless, e.g., opening a nonexistent file. This is exactly why syntax-directed fuzzers [5] are not very effective in generating high-quality workloads: most workloads will fail early. Layered models mitigates this problem because any workload $w^{(2)} \in M_2$ has its correspondence $w^{(0)} \in M_0$ in the core model and sampling diverse abstract workloads in M_0 and M_1 should benefit the diversity of concrete workloads.

The workload generation is illustrated using a previously uknown bug in F2FS shown in Figure 3. The basis of manifesting the bug is finding the backbone abstract workload (Figure 3 (a)), which is generated in a systematic *model checking* for workloads of diverse file system states.

```
mkdir("/a")
rename("/a", "/b1")
mkdir("/b1/b2")
mkdir("/b1/b2/b3")
mkdir("/b1/b2/b3/b4")
mkdir("/b1/b2/b3/b4/b5")
```

(a) core

→

```
mkdir("/a")
deepen("/a")
rename("/a", "/b1")
mkdir("/b1/b2")
mkdir("/b1/b2/b3")
mkdir("/b1/b2/b3/b4")
mkdir("/b1/b2/b3/b4/b5")
sync()
enlarge("/b1/b2/b3/b4/b5")
```

(b) core-ext

→

```
mkdir("/a",0775)
deepen("/a",4096)
rename("/a", "/b1")
mkdir("/b1/b2",0775)
mkdir("/b1/b2/b3",0775)
mkdir("/b1/b2/b3/b4",0775)
mkdir("/b1/b2/b3/b4/b5",0775)
sync()
enlarge("/b1/b2/b3/b4/b5",512)
```

(c) syntax

Figure 3: A previously unknown bug (null pointer dereference) that crashes F2FS. The workload is "irreducible" in the sense that the crash will not be triggered if any line in the workload is removed.

The core model checker maintains a queue of workloads (the queue initially contains a single empty workload), picks up a workload from the queue in each iteration, and generates a new workload by *appending* a new abstract file system call. The search is exhaustive (eventually all abstract workloads will be generated). To improve efficiency, we leverage the semantics interpretation of workloads to prioritize "distinctive" workloads and prune equivalent ones.

In the example, when `"/b1/b2/b3"` has been created, we prefer creating a new file structure with path `"/b1/b2/b3/b4"` where a deep subdirectory is exercised, or `"/b1/b4;/b1/b2/b3"` where a larger top directory is exercised, rather than `"b1/b2/b3;/b1/b2/b4"` where two paths seems similar. Repeating the generation, the basic but nontrivial file structure with path `"/b1/b2/b3/b4/b5"` is finally constructed, shown in Figure 3 (a).

During the generation, abstract calls in the ext-model are also exercised, e.g., after the directory with path `"/a"` is created, it will `deepen`ed, and a `sync` operation will also be issued after the basic file structure is constructed. Similar to preferring creating deep directories, operating `enlarge` on the directory with path `"/b1/b2/b3/b4/b5"` will be preferred because it results in lots of file metadata updates under a deep directory, depicted in Figure 3 (b).

Finally, necessary parameters in an abstract workload are filled with random values drawn from a parameter model to obtain runnable workloads (system call sequences). Such a model will mostly return a default value but sometimes return random values to manifest diverse file system behaviors. Such a bug-triggering concrete workload is shown in Figure 3 (c).

Though the final triggering script seems succinct, it is difficult to generate an equivalent file system sequence directly by existing techniques [5, 41] because of that (1) the basic file and directory structure is nontrivial, which is hard to generate; (2) `deepen` and `enlarge` are expressive which represent creating a large number of files under a directory; (3) to trigger the bug, a F2FS file system has to be mounted with the specific options: `noinline_dentry`, `checkpoint=disable`, and `quota`.

4 TESTING FILE SYSTEM IMPLEMENTATIONS ON LAYERED MODELS

This section introduces an abstract file system first (Section 4.1) and describes the syntax and semantics of the core, core-ext and syntax model (Section 4.2). Workload generation is elaborated in Section 4.3.

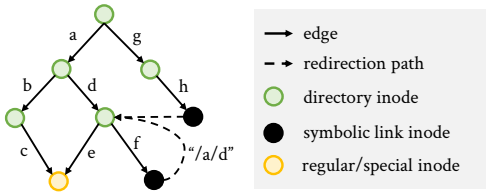

Figure 4: An example of the abstract file system

4.1 Abstract File System Model

Notations and Definitions. As mentioned earlier, the core and core-ext models share an abstract file system model, which is a simplification of the UNIX file system. A file system is a data structure for persisting files and directories. A file is indexed by a file path, a list of strings $p = [s_0, s_1, \ldots, s_{n-1}]$, also written as $p = \text{"}/s_0/s_1/\ldots/s_{n-1}\text{"}$, where each s_i is a file name. A path of $n = 0$ ("/") is the *root* of the file system. If $n > 0$, the *parent* of p is defined as $p^{\uparrow} = [s_0, s_1, \ldots, s_{n-2}]$. We call n the *depth* of p (denoted by $|p| = n$).

At runtime, the abstract file system state is determined by a four-tuple $\mathcal{S} = \langle \mathcal{N}, E, \mathcal{F}, \gamma \rangle$, where $\langle \mathcal{N}, E \rangle$ comprises an acyclic graph:

$\mathcal{N}$ is the set of all *inodes* (file or directory entities). Figure 4 shows an abstract file system example in which each vertex is an inode. There are four different types of inodes: directory, symlink (symbolic link) file, regular file, or device (special) file. Each inode $N \in \mathcal{N}$ is a key-value mapping, and we use $N\restriction_x$ to represent the value given by key x in N. For a symlink file, we use $N\restriction_{symlink}$ to store its redirection path. N_{root} is the root inode in the file system.

$E = \langle \mathcal{N} \times string \rangle \mapsto \mathcal{N}$ is the set of all *edges* in the graph, which means that given an inode N as a directory and a *string* as a file name, E maps them to an inode under this directory[4]. Since there may be hard links, an inode may be mapped to by more than one edge, i.e., the indegree of the inode is larger than one (`"/a/b/c"` and `"/a/d/e"` in Figure 4). Given any path $p = [s_0, s_1, \ldots, s_{n-1}]$,

[4]It may map to $\perp$ if the input N or file name is invalid.

we can resolve it to an inode by the following function:

$$\Upsilon(p) = \begin{cases} N_{root}, & p = \texttt{"/"}, \\ \Upsilon(\Upsilon(p^{\nwarrow})\restriction_{symlink} :: s_{n-1}), & \Upsilon(p^{\nwarrow})\restriction_{type} = \text{symlink}, \\ E[\langle \Upsilon(p^{\nwarrow}), s_{n-1}\rangle], & E[\langle \Upsilon(p^{\nwarrow}), s_{n-1}\rangle]\restriction_{type} \neq \text{symlink}, \\ \Upsilon(E[\langle \Upsilon(p^{\nwarrow}), s_{n-1}\rangle]\restriction_{symlink}), & E[\langle \Upsilon(p^{\nwarrow}), s_{n-1}\rangle]\restriction_{type} = \text{symlink}, \\ \bot & \text{otherwise}, \end{cases}$$

where "::" means list concatenation. In the example, $\Upsilon(\texttt{"/g/h/e"})$ will resolve $\Upsilon(\texttt{"/g/h"})$ first, which becomes $\Upsilon(\texttt{"/a/d"})$ by the fourth case; then, $\Upsilon(\texttt{"/g/h/e"}) = E[\langle \Upsilon(\texttt{"/a/d"}), \texttt{"e"}\rangle]$ by the third case, which maps to the final regular file inode.

$\mathcal{F}$ is the set of all *open files* in the system. System calls usually resolve a path p and return a file descriptor $f \in \mathcal{F}$ that is associated with p's corresponding inode.

$\gamma : \gamma \in \mathcal{N}$ is the current working directory.

The initial state $\mathcal{S}_0 = \langle \{N_{root}\}, \emptyset, \emptyset, N_{root}\rangle$ contains only a root directory.

File and Directory Attributes. Based on the file system state $\mathcal{S}$, we can derive a series of useful attributes of a file (or an inode, as each inode corresponds to a unique file). The attributes of an inode N, in the vector form of

$$[type, \#symlinked, atime, \#depth, \#hardlink, \#children, \#open]$$

are later used in measuring the diversity of inodes when generating command sequences (in Section 5)[5]. In the attribute vector, *type* is N's file type; *#symlinked* represents the number of symbolic link files targeting N; *atime* means the last access time of N; *#depth* is the depth of N (given a path p, as there may be multiple direct paths to the same inode); *#hardlink* is the hard link count of N, i.e., its indegree; *#children* is the number of children under directory N (a non-directory file has no child); *#open* is the number of open file descriptors targeting an inode (a non-regular file has a zero open count).

The abstract file system captures the graphical structure of files and directories shared by all file systems but not the file contents. The core model avoids modeling complex mechanisms (impractical to specify) under the apparently simple system call interface: quotas, crash consistency, extended attributes, etc.

4.2 Layered Models

The Core Model. A file system's workload is a sequence of system calls. Analogously, to manipulate the abstract file system, we designed 21 abstract file system calls (10 and 11 in the core and core-ext models, respectively), namely *commands*[6]. Each core command is an abstract version of a real system call for manipulating the file system structure or file descriptors. Take open as an example, it is an abstract form of the system call

```
int open(const char *pathname, int flags, mode_t mode);[7]
```

[5]The attribute designed is extensible. New attributes can be easily added in other file system testing scenarios.
[6]We use "abstract file system call" and "command" interchangeably when without ambiguity.
[7]http://man7.org/linux/man-pages/man2/open.2.html

$$\frac{N^{\nwarrow} = \Upsilon(p^{\nwarrow}) \quad \texttt{dir}(N^{\nwarrow}) \quad \langle N^{\nwarrow}, s_{n-1}\rangle \notin E \quad N = N^{dir}_{def}}{\mathcal{N} \leftarrow \mathcal{N} \cup \{N\} \quad E \leftarrow E \oplus [\langle N^{\nwarrow}, s_{n-1}\rangle \mapsto N] \quad \mathcal{F} \quad \gamma} \ \texttt{mkdir}(p)$$

$$\frac{N^{\nwarrow} = \Upsilon(p^{\nwarrow}) \quad \texttt{dir}(N^{\nwarrow}) \quad \langle N^{\nwarrow}, s_{n-1}\rangle \notin E \quad N = N^{dev}_{def}}{\mathcal{N} \leftarrow \mathcal{N} \cup \{N\} \quad E \leftarrow E \oplus [\langle N^{\nwarrow}, s_{n-1}\rangle \mapsto N] \quad \mathcal{F} \quad \gamma} \ \texttt{mknod}(p)$$

$$\frac{N^{\nwarrow} = \Upsilon(p^{\nwarrow}) \quad \texttt{dir}(N^{\nwarrow}) \quad \langle N^{\nwarrow}, s_{n-1}\rangle \notin E \quad N = N^{reg}_{def}}{\mathcal{N} \leftarrow \mathcal{N} \cup \{N\} \quad E \leftarrow E \oplus [\langle N^{\nwarrow}, s_{n-1}\rangle \mapsto N] \quad \mathcal{F} \quad \gamma} \ \texttt{create}(p)$$

$$\frac{N^{\nwarrow} = \Upsilon(p^{\nwarrow}) \quad \texttt{dir}(N^{\nwarrow}) \quad \langle N^{\nwarrow}, s_{n-1}\rangle \notin E \quad M = \Upsilon(p') \quad M\restriction_{type} = \text{reg}}{\mathcal{N} \quad E \leftarrow E \oplus [\langle N^{\nwarrow}, s_{n-1}\rangle \mapsto M] \quad \mathcal{F} \quad \gamma} \ \texttt{hardlink}(p', p)$$

$$\frac{N^{\nwarrow} = \Upsilon(p^{\nwarrow}) \quad \texttt{dir}(N^{\nwarrow}) \quad \langle N^{\nwarrow}, s_{n-1}\rangle \notin E \quad N = N^{sym}_{def}[symlink \mapsto p']}{\mathcal{N} \leftarrow \mathcal{N} \cup \{N\} \quad E \leftarrow E \oplus [\langle N^{\nwarrow}, s_{n-1}\rangle \mapsto N] \quad \mathcal{F} \quad \gamma} \ \texttt{symlink}(p', p)$$

$$\frac{N^{\nwarrow} = \Upsilon(p^{\nwarrow}) \quad \texttt{dir}(N^{\nwarrow}) \quad \langle N^{\nwarrow}, s_{n-1}\rangle \notin E \quad M = \Upsilon(p') \quad M^{\nwarrow} = \Upsilon(p'^{\nwarrow}) \quad N^{\nwarrow} \not\preceq M \wedge M \not\preceq N^{\nwarrow}}{\mathcal{N} \quad E \leftarrow E \ominus [\langle M^{\nwarrow}, s'_{n-1}\rangle \mapsto M] \oplus [\langle N^{\nwarrow}, s_{n-1}\rangle \mapsto M] \quad \mathcal{F} \quad \gamma} \ \texttt{rename}(p', p)$$

$$\frac{N = \Upsilon(p) \quad N \neq \gamma \quad \Phi(N)}{\mathcal{N} \leftarrow \mathcal{N}\mathring{\setminus}\Phi(N) \quad E \leftarrow E \ominus \{[\langle M, *\rangle \mapsto *] \mid M \in \Phi(N)\} \quad \mathcal{F} \quad \gamma} \ \texttt{remove}(p)$$

$$\frac{N = \Upsilon(p) \quad N\restriction_{type} = \text{reg}}{\mathcal{N} \quad E \quad \mathcal{F} \leftarrow \mathcal{F} \cup \{f\} \quad \gamma} \ \texttt{open}(f, p)$$

$$\frac{f \in \mathcal{F}}{\mathcal{N} \quad E \quad \mathcal{F} \leftarrow \mathcal{F} \setminus \{f\} \quad \gamma} \ \texttt{close}(f) \qquad \frac{N = \Upsilon(p) \quad N\restriction_{type} = \text{dir}}{\mathcal{N} \quad E \quad \mathcal{F} \quad \gamma \leftarrow N} \ \texttt{chcwd}(p)$$

Figure 5: Core model: semantics. $p = [s_0, s_1, \ldots, s_{n-1}]$. N^{dir}_{def}, N^{dev}_{def}, N^{reg}_{def}, and N^{sym}_{def} denote default values of four different inodes. The symbol $\oplus$ and $\ominus$ represent mapping operations, i.e., adding and deleting. $N \preceq N'$ means N an ancestor of N'. $\Phi(N)$ denotes all reachable inodes from an inode N including itself, and $\mathcal{N}\mathring{\setminus}\Phi(N)$ indicates that for each inode in $\Phi(N)$, it is removed from $\mathcal{N}$ if its indegree is zero.

The semantics of core commands are listed in Figure 5, in which the auxiliary function

$$\texttt{dir}(N) = \begin{cases} true, & N\restriction_{type} = \text{dir}, \\ false, & \text{otherwise}, \end{cases}$$

checks whether N is a directory inode. The semantics of core commands are explained below:

- mkdir creates a new directory, create builds a new regular file, and mknod constructs a new device file named p. [8] These three commands first resolve the parent inode $N^{\nwarrow}$ from path $p^{\nwarrow}$ and require that $N^{\nwarrow}$ must be a directory and does not contain an children inode named s_{n-1}. After that, they add a

[8]Creating a file means creating a new inode (and a path) of that specific file type.

new inode N and the corresponding edge between $N^{\nwarrow}$ and N.
- `hardlink` creates a new path p targeting the same regular file inode as p'. It is similar to `create` except that it does not create a new inode but adds an edge between two existing inodes: $N^{\nwarrow}$ (targeted by $p^{\nwarrow}$) and M (targeted by p').
- `symlink` creates a symlink file redirecting to p'. It is similar to `create` except that it creates a new symlink inode that stores a redirection path.
- `rename` deletes the existing inode M (targeted by p') from its current parent directory $M^{\nwarrow}$ (targeted by $p'^{\nwarrow}$) and moves it to another directory inode $N^{\nwarrow}$ (targeted by $p^{\nwarrow}$). Note that it requires that M and $N^{\nwarrow}$ are not an ancestor of each other.
- `remove` removes the inode N (targeted by p). If N is a directory, its all children are recursively removed. It requires that N is not the current working directory.
- `open` adds a new file descriptor f to $\mathcal{F}$ if path p targets a regular file.
- `close` deletes an existing file descriptor f from $\mathcal{F}$.
- `chcwd` changes the current working directory γ.

Each inode's attribute vector is also maintained according to their definitions. For brevity, we did not list them in Figure 5.

The Core-Ext Model. The core-ext model also operates on the abstract file system. We designed 11 extension commands (syntax in Figure 2 and semantics in Table 1), most of which do not change the abstract file system state. The extension commands are designed for driving the file system's underlying storage (e.g., a disk) to diverse states.

Core-ext commands may incur large amounts of file metadata operations, fragment the storage, or access the file contents. Given a core model workload $w \in M_0$, suppose that we insert an `enlarge`(p) command in the middle of w for a directory p yielding w'. Executing w' creates thousands of (random) sub-directories, which incurs lots of disk block I/O operations and complicates the directory's internal data structure. Furthermore, the file system structure is identical for w and w' if we ignore the sub-directories created by the `enlarge`. In other words, the basic file system structure in $w \in M_0$ is preserved in $w' \in M_1(w)$. Core-ext commands may also change file attributes, e.g., `enlarge` increases the #*children* attribute. `prune` is similar to `remove` except that it keeps the concerned inode and may change the #*children* attribute of concerned directories.

The Syntax Model. The syntax model captures all syntactically correct file system call APIs in C [36] as shown in Figure 2 where p is a path and $f \in \mathcal{F}$ is a file descriptor . Given a workload $w \in M_1$, it can be easily *refined* to obtain a syntactically valid concrete workload in M_2. Most abstract commands have its system call correspondence. Filling such an abstract command with syntactically valid parameters yields a concrete system call. For those abstract commands that do not have a system call correspondence (e.g., `prune`), we provide an implementation using file system calls (e.g., `unlink`). Due to the complexity of file system implementations, it is impractical to specify the semantics of the syntax model. Later, we will see that layered modeling is effective in generating high quality workloads without such semantics.

Algorithm 1: File system workload generation

```
1  Procedure workload_generation()
2      Q ← {⟨S_0, []⟩}
3      while Q ≠ ∅ ∧ not_terminate() do
4          ⟨S, ŵ⟩ ← prio_pick(Q) // Rule P1 applies
5          w ← []
6          for α ∈ ŵ do
7              // α is an abstract command with type ty
8              w ← w :: fill_parameter(α, p ~ P_ty^param)
9          yield(w) // Generate a workload and execute it
10         if |ŵ| > L_max then
11             pop(Q, ⟨S, ŵ⟩) // Length is bounded as L_max
12             continue
13         ty ← select_type(S, ty ~ P^cmd) // Rule P2 applies
14         for α ∈ build_commands(S, ty) do
15             S' ← new_state(S, α)
16             ŵ' ← ŵ :: α
17             if ⟨S', ∗⟩ ∉ Q then
18                 Q ← Q ∪ {⟨S', ŵ'⟩}
19 Procedure build_commands(S, ty)
20     cmds ← {}
21     // Rule P3 applies
22     for N ← typical_inodes_by_kmeans(Φ(γ), ty) do
23         cmds ← cmds ∪ {new_command(ty, N)}
24     return cmds
```

4.3 Generating Workloads

We can apply bounded model checking (as did in ACE [45]) to generate file system workloads (of a length limit) that cover all reachable non-equivalent abstract file system states $\mathcal{S}$. Unfortunately, bounded model checking scales poorly to long workloads. On the other hand, such long workloads are essential to reaching deep file system states and manifesting file system bugs. Therefore, we add *prioritization* to the framework of bounded model checking [27], such that we can quickly generate a large amount of workloads, but also eventually achieve the effect of covering all non-equivalent abstract file system states (though it is not achievable practically). The prioritization rules are three-fold:

- P1: When picking up workloads, we favor longer ones because they may result in more IO operations, disk fragments, etc. for exercising diverse file system states.
- P2: When appending a command, the type of this new command is drawn from the probability distribution P^{cmd}.
- P3: When selecting an inode as the argument (the path or the file descriptor associated with an inode) for a new command, we choose the one that is mostly likely to drive the system to an *unexplored* state.

The testing procedure based on layered models is shown in Algorithm 1. It resembles a standard queue-based model checker [27]. The model checker starts with the initial abstract file system state $\mathcal{S}_0$ and an empty workload.

During each iteration, it picks up an abstract workload $\widehat{w}$ in the queue at Line 4 according to rule P1 ($\widehat{w}$ resides in the queue and may

Table 1: Extension calls

Precondition	Extension Calls	Description
$f \in \mathcal{F}$	`read` | `write` | `fsync`(f)	Read/write/synchronize the open file descriptor f.
$\top$	`sync`()	Synchronize the whole file system.
$\top$	`remount`()	Remount the file system.
$\Upsilon(p) \neq \perp$	`statfs`(p)	Retrieve the file system information with path p.
$\Upsilon(p) \neq \perp$	`read_xattr` | `write_xattr`(p)	Read extended attributes of the file with path p.
$\Upsilon(p) \wedge \Upsilon(p)\restriction_{type}$=dir	`deepen`(p)	Create subdirectories alongside a single path under the directory p.
$\Upsilon(p) \wedge \Upsilon(p)\restriction_{type} \in$ {dir, reg}	`enlarge`(p)	If p targets a directory inode, create subdirectories under the directory; otherwise, truncate the regular file to a larger size.
$\Upsilon(p) \wedge \Upsilon(p)\restriction_{type} \in$ {dir, reg}	`prune`(p)	If p targets a directory inode, remove all inodes under the directory; otherwise, truncate the regular file to be empty.

be picked up later). Then, $\widehat{w}$ is refined into concrete workload w and executed at Line 5—9. For each generated abstract workload in M_1 (the core-ext model), `fill_parameter` does the refinement job. For command type *ty*, values of its parameters p are drawn from the command's own distribution, i.e., $p \sim P^{param}_{ty}$. For example, we set the *mode* parameter mostly to a default value as its subtle changes may lead to early failures; *flag* and *mount-opt* can be encoded as a bit string, and we independently set each bit by a probability of 10%; for *size*, we uniform-randomly sample an integer in a predefined range; finally for *data* parameter, we uniform-randomly sample from a set of predefined memory regions.

If $\widehat{w}$ is longer than the limit, we discard it (Line 10—12); otherwise, it is appended a new command, yielding a new abstract file system state $\mathcal{S}'$ and a new workload $\widehat{w}'$ (Line 15—16). If $\mathcal{S}'$ has not been previously explored, it is added to the queue with its corresponding workload $\widehat{w}'$ (Line 17—18), where file system state comparisons are done by a Merkle tree alike hashing [42].

The new command generation first selects a command type to build according the probability distribution P^{cmd} (rule P2), where most commands are of an equal probability except that commands about deleting inodes, e.g., `remove` and `prune`, are of lower probabilities. This prevents inodes from being removed too frequently to manifest large and complex file structure. Commands about file system synchronization and remounting are also of lower probabilities because these commands do not modify the file structure directly but explore complicated file system behaviors when file structures are complex.

Second, the inode parameters for new commands are selected delicately according to rule P3. The merit of a candidate inode is measured by the "similarity" between inodes. An inode N, which is more "dissimilar" to others, should receive a higher probability of being chosen[9]. Such a similarity measurement is realized by a k-means clustering. In `build_commands`, we cluster all reachable inodes from the current working directory by their *attribute vectors* (described in Section 4). After that, we select typical inodes and create new commands for each of them. Note that inodes to be clustered should be related to the command type, e.g., only regular file inodes are considered for `open`, and if an inode has multiple associated paths or file descriptors, we choose one of them randomly.

5 SYSTEM IMPLEMENTATION

We implemented the models and workload generation algorithm in Section 4 as an open-source prototype tool—`Dogfood`[10] upon QEMU [12]. `Dogfood` consists of three main components: workload generator, sequence dumper, and testing backend. The model checker implements Algorithm 1: it generates file system call sequences and fills each system call with random parameters. These sequences are then dumped (by the dumper) to C programs, which are compiled, loaded, and executed by the testing backend in a QEMU virtual machine. The testing backend also collects and analyzes the kernel log of the virtual machine.

Sequence Dumper and Executor. The dumper serializes file system call sequences into interface function calls, which is then compiled and linked with different runtime libraries into various executors. The executors do the precondition checks (e.g., probing whether the target file exists or confirming whether a file descriptor is valid) and execute the system calls. Our `Dogfood` executor directly executes the system calls; the CrashMonkey [41][11] executor simulates a file system crash and checks for consistency.

Testing Backend. We use debootstrap [2] to build a rootfs disk image and run Linux kernels with different file systems in a QEMU virtual machine [12]. A controller is implemented to manipulate the virtual machine and to communicate with the guest OS via SSH. The controller copies executors to the guest OS, runs them sequentially, and collects the results. When testing file systems, we should keep the testing environment clean as much as possible that all changes to the file system are from the current workload instead of others; we take advantage of the QEMU snapshot functionality where a virtual machine snapshot is saved after booting the guest OS, and we revert the OS state before each executor running.

Sequence Reducer. Given a failure-inducing workload $\widehat{w}$, it may be long and difficult to debug. Therefore, `Dogfood` also includes a sequence reducer. The reducer first binary searches for the shortest prefix of $\widehat{w}$ which triggers exactly the same type of failure. Then, the reducer tries to iteratively remove commands from the beginning

[9] Consider that we attempt to make a `remove` command under a state, where there are four empty directories and a non-empty directory; we tend to exercise `remove`ing an empty directory and `remove`ing a non-empty one under the state, instead of trying two commands `remove`ing empty directories.

[10] https://midwinter1993.github.io/dogfood/

[11] https://github.com/utsaslab/crashmonkey/blob/master/docs/workload.md

Table 2: The evaluated file system implementations

File System	Description
Ext4	The default file system for most Linux distributions
Btrfs	A modern copy on write (CoW) file system
F2FS	A modern flash storage devices friendly file system
ReiserFS	A general-purpose, journaled file system
GFS2	A shared-disk file system
JFS	A journaled file system

as far as the reduced sequence can still trigger the failure. When removing a command, we conduct a define-use analysis [46] of the workload and remove all following commands that are data-dependent on it. Though the reduced sequence may not be the shortest failure-inducing subsequence, the evaluation results show that it is effective in practice.

6 EXPERIMENTS

Since our workload generation technique is general, we applied Dogfood to generate workloads in three file system testing scenarios: (Sequential) directly feeding the workloads to a file system implementation; (Crash Consistency) injecting a crash in the middle and check the file system consistency; (Concurrency) exploring the interleaving space of parallel workloads. The experimental setup is described in Section 6.1, followed by the results in Section 6.2.

6.1 Experimental Setup

We evaluated Dogfood on six actively maintained mainline file systems in the 5.1.3 Linux Kernel, the latest Kernel when the experiments were conducted. As listed in Table 2, these file systems are Ext4, Btrfs, F2FS, ReiserFS, GFS2, and JFS, representing diverse subjects. These file systems are widely-used and have been tested by existing work [5, 43, 51, 66]. The workloads are used in three testing scenarios:

The Sequential Scenario. The most straightforward way of testing is directly executing the generated workloads on an operating system. For each file system, we ran Dogfood for 12 hours, and the testing logs were collected. For an (informal) comparison, we also ran the state-of-the-art kernel fuzzer Syzkaller [5] for 12 hours[12], which generates workloads to corrupt the kernel directly. Syzkaller is an industrial testing tool that has been widely-used and found thousands of bugs. We restricted Syzkaller to generate only file system calls. For both evaluated techniques, a disk formatted with the designated file system format was mounted before the workloads were executed. Syzkaller used default parameters; Dogfood was capable of taking different parameters during the runtime.

The Crash Consistency Scenario. A system may crash at any time and the on-disk file system should be in a reasonable state after any crash. CrashMonkey [41] simulates a physical disk and can inject simulated crashes. Such simulated crash disk images are checked against consistency problems, which is a major source of file system bugs [14, 45, 51, 69].

CrashMonkey has a built-in workload generator ACE [45] that is dedicated to yield workloads for crash consistency checking, and we wonder if Dogfood can generate higher quality workloads. For both of Dogfood and ACE, we randomly sampled 1,000 workloads and the inconsistency reports were analyzed. We also tested file system crash consistency using 10,000 random workloads generated by Dogfood. Since CrashMonkey can only check the consistency of Ext4, Btrfs, and F2FS, the experiments were limited to these file systems for fairness.

The Concurrency Scenario. We generated 500 workloads that did not manifest any failure when executed sequentially for 1,000 times. To simulate the concurrent workloads, we ran in parallel a worker thread that periodically synchronizes the file system and remounts the disk to manifest potential race conditions on metadata operations. To manifest potential concurrency bugs, we use KProbes [58] to instrument the Kernel by inserting random delays before and after functions calls related to file systems and synchronizations [55]. Concurrency testing is time-consuming. As a proof-of-concept, we evaluated it on the Btrfs where Dogfood manifested fewer crashes than the others. Since concurrency bugs do not manifest deterministically [40], we repeated the concurrent executions of each workloads for 10 times.

All experiments were conducted on a desktop PC with a quad-core 3.40G Hz Intel Core i7 processor and 16 GiB memory running Ubuntu Linux 16.04. When conducting the crash consistency experiment, we used libvirt [52] to create a virtual machine with two virtual CPUs and 2 GiB memory running Linux Kernel 4.4.0 (the same setting as the the original paper of CrashMonkey [41]).

Table 3: #Crashes/#warns in the 12-hour testing. 1* means that crash occurred too frequently.

	File Systems					
	Ext4	Btrfs	F2FS	ReiserFS	GFS2	JFS
Dogfood	0/0	13/71	456/0	296/0	276/0	1*/0
Syzkaller	0/0	0/0	0/0	0/0	0/0	1*/0

6.2 Experimental Results

The Sequential Scenario. Excluding JFS (crashing too frequently), sequential workloads generated by Dogfood crashed the Kernel by 1,041 times (Table 3). Ext4, the most widely used file system, is considerably more reliable than the others. It is the only one that survived the 12-hour testing. We manually inspected the crashes and identified 12 distinct bugs that: (1) have a clear root cause, (2) have a severe consequence (kernel panic, use-after-free, or NULL pointer deference), and (3) can be stably reproduced. We have reported all of them, as listed in Table 4. Some have already been fixed by the developers. Dogfood also found 71 warnings in Btrfs, and three of them are unique warnings (likely to be a bug).

Such results are encouraging. File system implementations underwent extensive testing, fuzzing, and validation; some are even

[12]The comparison is informal because Syzkaller is so famous that many file system had already underwent thousands CPU hours of fuzzing.

Table 4: Previously unknown bugs found in the 12-hour testing. Len. is the length of a reduced sequence.

Bug ID	Consequence	Location	Len.
JFS-203737	NULL pointer deref. (kasan)	jfs_logmgr.c	1
Btrfs-203989	Use-after-free (kasan)	free-space-cache.c	100
Btrfs-204045	BUG_ON()	ctree.h	165
F2FS-204043	BUG_ON()	data.c	60
F2FS-204135	NULL pointer deref. (kasan)	data.c	2
F2FS-204137	BUG_ON()	segment.c	7
F2FS-204193	NULL pointer deref. (kasan)	data.c	9
F2FS-204197	BUG_ON()	inode.c	40
ReiserFS-204263	BUG_ON()	prints.c	2
ReiserFS-204265	BUG_ON()	journal.c	11
GFS2-204335	Use-after-free (kasan)	lops.c	130
GFS2-204337	BUG_ON()	glock.c	300

maintained by companies (e.g., developers in Huawei quickly respond to our bug reports). Finding sequential workloads that directly crash the kernel is considerably challenging. Furthermore, our workload generator is highly configurable and extendable. A massive parallel run of Dogfood under different configurations is promising in finding even more file system bugs.

Bug Cases. We take two more bug cases as examples to illustrate the bug detection capability of Dogfood. First, there is another bug-triggering workload for F2FS listed below:

```
mkdir("/a",0775) mkdir("/b",0775) mkdir("/a/c",0775)
mkdir("/a/c/d",0775)
enlarge("/", 4096) sync()
```

The workload creates several directories, enlarges the root directory, and synchronizes the file system; then, a bug manifest. Though the workload is simple, it is difficult to obtain without our file system model because our abstract file system calls are expressive, which may represent a bunch of underlying file system calls. When generating an equivalent workload directly, it should yield a sequence with thousands of system calls.

Second, a bug-triggering workload for ReiserFS is also listed below:

```
mkdir("/a",0775) mkdir("/a/b",0775) mkdir("/a/b/c",0775)
rename("/a/b", "/e")
("/e/c/f",0775) mkdir("/e/c/f/g",0775)
create("/e/c/f/g/h",0775) (fd, "/e/c/f/g/h",O_RDWR)
prune("/e/c/f/") sync() write(fd, 57344)
```

The workload drives the file system to prune a directory, i.e., removing its children. The subsequent sync and write to a dangling file descriptor triggers the bug. Even though how to react to a write to a file descriptor whose target file has been deleted is specified by the Linux Programmer's Manual [4], the tricky orderings of these operations and the certain file structure together result in this bug, i.e., it needs both a non-trivial file structure and complex file operations to manifest the bug. However, Dogfood expose this bug by separating the concerns of constructing file structures in the core model and exercising diverse file system states in the core-ext model.

Table 5: #Detected file system issues in 1,000 workloads. Left-/right number is Dogfood/ACE, respectively.

	File Systems		
Issue type	Ext4	Btrfs	F2FS
File size	0/0	32/0	0/0
#Hard links	10/0	3/0	6/0
File content	3/0	3/0	3/0
Failed stating	16/16	19/14	18/15

Table 6: #Detected file system issues from 10,000 workloads.

	File Size	#Hard links	File content	Failed stating	#Blocks
Ext4	3	64	35	100	2
Btrfs	289	25	46	124	7
F2FS	6	53	41	99	1

The Crash Consistency Scenario. The inconsistency reports from 1,000 workloads are listed in Table 5. Compared the ACE workload generator (a bounded model checker) [45], Dogfood can generate longer sequences and thus more effectively exposed the inconsistencies. The inconsistency reports from 10,000 workloads are shown in Table 6. A manual inspection shows that all reported bugs by ACE[13] are covered by our workloads. Furthermore, wrong hard link count is a strong indicator of a bug[14]. Therefore, the incorrect link count in Ext4, which was never manifested by existing work [41, 45], is a strong evidence of previously unknown crash consistency bug.

The Concurrency Scenario. We captured 20 crashes in the concurrency testing in the 5,000 executions. Since concurrency failures are difficult to diagnose, we examined the stack trace and found two unique crashes. We did not report the bugs due to the non-determinism and low probability of reproduction without random delays in the kernel. As a small-scale proof-of-concept experiment, such results also indicate that workloads generated by Dogfood are profitable in driving concurrency testing [25].

Discussions. The evaluation results show that Dogfood generated effective and high-quality workloads for testing file system implementations. At least, these workloads, whose sequential execution directly crashed the kernel, had never been able to be effectively generated before Dogfood. Such workloads benefit nearly *every* domain-specific testing technique: sanitizers [34, 56], trace miners [29, 48], crash consistency checkers [41, 45], interleaving space samplers [24, 47, 55], to name but a few. Furthermore, models introduced in this paper can be improved in various aspects: modeling more file attributes, adding more M_1 commands, exploring efficient prioritization policies.

7 RELATED WORK

A broad spectrum of techniques have been proposed to make file systems more reliable, from formal verification, model checking to fuzzing.

[13] https://github.com/utsaslab/crashmonkey/blob/master/newBugs.md
[14] When CrashMonkey reported such a case, it would be definitely a bug because the identity should be maintained atomically by a file system implementation.

Model Checking. Model checking file systems has been proposed and found serious file system errors [68, 69]. These techniques model file systems directly, change file system states by executing file system calls, and check disk image states. They need to modify operating systems to obtain the underlying I/O operations and to inject checkpoints or faults. They could model check behaviors of a file system under potential API failures given a fixed workload, but they do not address the workload generation problem. Though we have a similar model checking framework shown in Algorithm 1, we only model check *abstract states* instead of file system implementations directly. This key treatment alleviates the state space explosion problem by focusing on the systematic workload generation for a smaller core part of file system.

Furthermore, compared with traditional model checking refinement approaches [22, 59], our key trade-off is to partially abandon the soundness of model refinements, while still providing a good-enough approximation of workload semantics in layered models. Such a balance facilitates handling the aforementioned file system features which are generally impractical to soundly model.

Model Based Testing. Model based testing is a promising approach to testing systems, which builds a model from specifications and generate inputs automatically on top of the model [16, 23, 60–63]. The model can be UML [15], finite state automata [20], event sequence graph [13], Markov chains [64], etc.;

Since we used layered models in workload generation, this paper can also be regarded as a variant of model-based testing. However, existing model-based testing approaches require a sufficiently precise model to serve as a basis for the generation [62], which is infeasible for file systems because the precise semantics of file systems are usually unavailable [53]. Similarly, our approach has a precise model in the core model layer, and the abstract workload generation in it can be viewed as model based generation. However, we further refine abstract workloads loosely with more randomness layer by layer instead of building a full precise model. Thus, it does not need a sound complex model of file systems and obtains a balance between the semantics controllability and the syntax diversity.

There exists a model-based tool with refinement [26], which attempts to refine the model from execution traces. It is more close to model checking refinement [22] and needs developers to specify a mapping between the states in the model and the events in the traces. This specification is infeasible for complex systems, like file systems. However, our approach is different as far as refinement, which refines workloads instead of models and does not need other complicated specifications.

The success of this paper's layered modeling and adopting model checking in workload generation may potentially benefit future model-based testing of complex systems.

Fuzzing. Fuzzing is an effective and practical technique for test input generation. By generating a large amount of random or invalid workloads to explore corner cases in a file system implementation, fuzzers revealed hundreds bugs, which explored exceptional inputs for file systems randomly or by certain coverage criteria [50, 66]. However, fuzzers fall short on generating long, meaningful, and effective workloads that drive the file system to diverse states. Therefore, fuzzing benefits scenarios like security testing [5]; while our approach is more useful in reinforcing other testing techniques (e.g., a crash consistency checker [41] or an interleaving space explorer [25]). The workloads generated on layered models can also be useful workload prefixes for a fuzzer.

Grammar-based fuzzing [10, 28, 30] has been a promising approach to generating structured workloads. However, it is hard to generate meaningful workloads with complex dependencies, like file system call sequences, because it is easy to incur early failures, e.g., a failed `mkdir` affects all following system calls related to the directory. Our layered modeling, on the other hand, controllably generates diverse abstract workloads first, which are then refined to concrete ones with more randomness. The refinement in the syntax model can be regarded as fuzzing, but we do fuzzing on top abstract workloads instead of fuzzing from the grammar directly.

Crash Consistency Testing. Much work has been proposed to detect crash consistency bugs [45, 51]. To uncover these bugs, system call sequences are executed and underlying I/O operations are recorded, and then permutations of these I/O operations are replayed with simulated faults injected [45]. Crash consistency issues are found when damaged disk images are obtained. Most existing work uses random sequences and builds sequences manually [14], or explores sequences with a bounded length [45]. `Dogfood` is also orthogonal to these tools in that more diverse sequences can be exercised and more issues may be detected as shown in Section 6.

Formal Verification. Formal methods have been applied to file systems [19, 53] to provide strong guarantees of reliability. These techniques attempt to build a mathematical model for file systems and prove invariants; based on them, new file systems have been designed and implemented [19, 44]. Being a promising future direction, it is currently impractical to model existing file systems (their behaviors are even beyond the POSIX standard). Instead of a proof, we build an abstract model to generate sequences. Such a strategy trades off soundness guarantee for practical usefulness.

8 CONCLUSION

Generating high-quality workloads is a fundamental and challenging problem to complex systems such as file system implementations. This paper presents an approach to building layered models of these systems to generate workloads. It also demonstrates the potential of layered modeling by instantiating a three-layer file system model and a prototype tool `Dogfood`. In a 12-hour testing run, `Dogfood` produced 12 system call sequences whose sequential executions yielded numerous kernel crashes. Therefore, we hope that our approach would be a promising research direction for enhancing the reliability of other complex systems.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for comments and suggestions. This work is supported in part by National Key R&D Program (Grant #2018YFB1004805) of China, National Natural Science Foundation (Grants #61932021, #61690204, #61802165) of China. The authors would like to thank the support from the Collaborative Innovation Center of Novel Software Technology and Industrialization, Jiangsu, China. Yanyan Jiang (jyy@nju.edu.cn) and Xiaoxing Ma (xxm@nju.edu.cn) are the corresponding authors.

REFERENCES

[1] [n.d.]. Bug #317781: Ext4 Data Loss. https://bugs.launchpad.net/ubuntu/+source/linux/+bug/317781?comments=all.
[2] [n.d.]. Debootstrap - Debian Wiki. https://wiki.debian.org/Debootstrap.
[3] [n.d.]. Kernel: Add Kcov Code Coverage. https://lwn.net/Articles/671640/.
[4] [n.d.]. Linux Programmer's Manual UNLINK(2). http://man7.org/linux/man-pages/man2/unlink.2.html.
[5] [n.d.]. Syzkaller. https://github.com/google/syzkaller.
[6] [n.d.]. (x)fstests Is A Filesystem Testing Suite. https://github.com/kdave/xfstests.
[7] 2015. Btrfs: Add Missing Inode Update When Punching Hole. https://patchwork.kernel.org/patch/5830801/.
[8] Alexandr Andoni, Dumitru Daniliuc, Sarfraz Khurshid, and Darko Marinov. 2003. Evaluating the "Small Scope Hypothesis". Citeseer.
[9] Konstantine Arkoudas, Karen Zee, Viktor Kuncak, and Martin Rinard. 2004. Verifying a File System Implementation. In *Formal Methods and Software Engineering*, Jim Davies, Wolfram Schulte, and Mike Barnett (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 373–390.
[10] Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. 2019. NAUTILUS: Fishing for Deep Bugs with Grammars. In *NDSS*.
[11] Farbrice Bellard. [n.d.]. Tiny C Compiler. https://bellard.org/tcc/.
[12] Fabrice Bellard. 2005. QEMU, a Fast and Portable Dynamic Translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference (ATEC '05)*. USENIX Association, Berkeley, CA, USA, 41–41.
[13] Fevzi Belli, Andre Takeshi Endo, Michael Linschulte, and Adenilso Simao. 2014. A Holistic Approach to Model-based Testing of Web Service Compositions. *Software: Practice and Experience* 44, 2 (2014), 201–234.
[14] James Bornholt, Antoine Kaufmann, Jialin Li, Arvind Krishnamurthy, Emina Torlak, and Xi Wang. 2016. Specifying and Checking File System Crash-consistency Models. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '16)*. ACM, New York, NY, USA, 83–98.
[15] Lionel Briand and Yvan Labiche. 2002. A UML-based Approach to System Testing. *Software and Systems Modeling* 1, 1 (2002), 10–42.
[16] Manfred Broy, Bengt Jonsson, J-P Katoen, Martin Leucker, and Alexander Pretschner. 2005. Model-based Testing of Reactive Systems. In *Volume 3472 of Springer LNCS*. Springer.
[17] Yan Cai, Shangru Wu, and W. K. Chan. 2014. ConLock: A Constraint-based Approach to Dynamic Checking on Deadlocks in Multithreaded Programs. In *Proceedings of the 36th International Conference on Software Engineering (ICSE '14)*. ACM, New York, NY, USA, 491–502.
[18] Dongjie Chen, Yanyan Jiang, Chang Xu, Xiaoxing Ma, and Jian Lu. 2018. Testing Multithreaded Programs via Thread Speed Control. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2018)*. ACM, New York, NY, USA, 15–25.
[19] Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. 2015. Using Crash Hoare Logic for Certifying the FSCQ File System. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP '15)*. ACM, New York, NY, USA, 18–37.
[20] Tsun S. Chow. 1978. Testing Software Design Modeled by Finite-state Machines. *IEEE transactions on software engineering* 3 (1978), 178–187.
[21] Mason Chris. 2007. The Btrfs Filesystem. http://oss.oracle.com/projects/btrfs/dist/documentation/btrfs-ukuug.pdf.
[22] Edmund Clarke, Ansgar Fehnker, Zhi Han, Bruce Krogh, Joël Ouaknine, Olaf Stursberg, and Michael Theobald. 2003. Abstraction and Counterexample-guided Refinement in Model Checking of Hybrid Systems. *International journal of foundations of computer science* 14, 04 (2003), 583–604.
[23] Arilo C Dias Neto, Rajesh Subramanyan, Marlon Vieira, and Guilherme H Travassos. 2007. A Survey on Model-based Testing Approaches: A Systematic Review. In *Proceedings of the 1st ACM international workshop on Empirical assessment of software engineering languages and technologies: held in conjunction with the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE '07)*. ACM, 31–36.
[24] Michael Emmi, Shaz Qadeer, and Zvonimir Rakamarić. 2011. Delay-bounded Scheduling. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '11)*. ACM, New York, NY, USA, 411–422.
[25] Pedro Fonseca, Rodrigo Rodrigues, and Björn B. Brandenburg. 2014. SKI: Exposing Kernel Concurrency Bugs Through Systematic Schedule Exploration. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI '14)*. USENIX Association, Berkeley, CA, USA, 415–431.
[26] Ceren Şahin Gebizli and Hasan Sözer. 2017. Automated Refinement of Models for Model-based Testing Using Exploratory Testing. *Software Quality Journal* 25, 3 (2017), 979–1005.
[27] Patrice Godefroid. 1997. Model Checking for Programming Languages Using VeriSoft. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '97)*. ACM, New York, NY, USA, 174–186.
[28] Nikolas Havrikov and Andreas Zeller. 2019. Systematically Covering Input Structure. (2019).
[29] Shilin He, Qingwei Lin, Jian-Guang Lou, Hongyu Zhang, Michael R. Lyu, and Dongmei Zhang. 2018. Identifying Impactful Service System Problems via Log Analysis. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2018)*. ACM, New York, NY, USA, 60–70.
[30] Renáta Hodován, Ákos Kiss, and Tibor Gyimóthy. 2018. Grammarinator: A Grammar-based Open Source Fuzzer. In *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation (A-TEST '18)*. ACM, New York, NY, USA, 45–48.
[31] Mei-Chen Hsueh, Timothy K Tsai, and Ravishankar K Iyer. 1997. Fault Injection Techniques and Tools. *Computer* 30, 4 (1997), 75–82.
[32] Daniel Jackson. 2003. Alloy: A Logical Modelling Language. *ZB* 2651 (2003), 1.
[33] Daniel Jackson and Craig A. Damon. 1996. Elements of Style: Analyzing a Software Design Feature with a Counterexample Detector. In *Proceedings of the 1996 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '96)*. ACM, New York, NY, USA, 239–249.
[34] Edge Jake. 2014. The Kernel Address Sanitizer. https://lwn.net/Articles/612153/.
[35] Baris Kasikci, Weidong Cui, Xinyang Ge, and Ben Niu. 2017. Lazy Diagnosis of In-production Concurrency Bugs. In *Proceedings of the 26th Symposium on Operating Systems Principles*. ACM, 582–598.
[36] Brian W Kernighan and Dennis M Ritchie. 2006. *The C programming language*.
[37] Charles Edwin Killian, James W. Anderson, Ryan Braud, Ranjit Jhala, and Amin M. Vahdat. 2007. Mace: Language Support for Building Distributed Systems. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07)*. ACM, New York, NY, USA, 179–188.
[38] Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler Validation via Equivalence Modulo Inputs. In *ACM SIGPLAN Notices*, Vol. 49. ACM, 216–226.
[39] Lanyue Lu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Shan Lu. 2014. A Study of Linux File System Evolution. *Trans. Storage* 10, 1, Article 3 (Jan. 2014), 32 pages.
[40] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. 2008. Learning from Mistakes: A Comprehensive Study on Real World Concurrency Bug Characteristics. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIII)*. ACM, New York, NY, USA, 329–339.
[41] Ashlie Martinez and Vijay Chidambaram. 2017. Crashmonkey: A Framework to Systematically Test File-system Crash Consistency. In *Proceedings of the 9th USENIX Conference on Hot Topics in Storage and File Systems (HotStorage '17)*. USENIX Association, Berkeley, CA, USA, 6–6.
[42] Ralph C. Merkle. 1988. A Digital Signature Based on a Conventional Encryption Function. In *A Conference on the Theory and Applications of Cryptographic Techniques on Advances in Cryptology (CRYPTO '87)*. Springer-Verlag, London, UK, UK, 369–378.
[43] Changwoo Min, Sanidhya Kashyap, Byoungyoung Lee, Chengyu Song, and Taesoo Kim. 2015. Cross-checking Semantic Correctness: The Case of Finding File System Bugs. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP '15)*. ACM, New York, NY, USA, 361–377.
[44] Zou Mo, Ding Haoran, Du Dong, Fu Ming, Ronghui Gu, and Chen Haibo. 2019. Using Concurrent Relational Logic with Helper for Verifying the AtomFS File System. In *Proceedings of the 27th Symposium on Operating Systems Principles (SOSP '19)*. ACM, New York, NY, USA.
[45] Jayashree Mohan, Ashlie Martinez, Soujanya Ponnapalli, Pandian Raju, and Vijay Chidambaram. 2018. Finding Crash-consistency Bugs with Bounded Black-box Crash Testing. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI '18)*. USENIX Association, Berkeley, CA, USA, 33–50.
[46] Steven Muchnick et al. 1997. *Advanced compiler design implementation*. Morgan kaufmann.
[47] Madanlal Musuvathi and Shaz Qadeer. 2007. Iterative Context Bounding for Systematic Testing of Multithreaded Programs. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07)*. ACM, New York, NY, USA, 446–455.
[48] Karthik Nagaraj, Charles Killian, and Jennifer Neville. 2012. Structured Comparative Analysis of Systems Logs to Diagnose Performance Problems. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI '12)*. USENIX Association, Berkeley, CA, USA, 26–26.
[49] Adrian Nistor, Qingzhou Luo, Michael Pradel, Thomas R Gross, and Darko Marinov. 2012. Ballerina: Automatic Generation and Clustering of Efficient Random Unit Tests for Multithreaded Code. In *Proceedings of the 34th International Conference on Software Engineering*. IEEE Press, 727–737.
[50] Vegard Nossum and Quentin Casasnovas. 2016. Filesystem Fuzzing with American Fuzzy Lop. *Vault 2016* (2016).

[51] Thanumalayan Sankaranarayana Pillai, Vijay Chidambaram, Ramnatthan Alagappan, Samer Al-Kiswany, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2014. All File Systems Are Not Created Equal: On the Complexity of Crafting Crash-consistent Applications. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI '14)*. USENIX Association, Berkeley, CA, USA, 433–448.

[52] Hat Red. 2005. libvirt: The virtualization API. https://libvirt.org/.

[53] Tom Ridge, David Sheets, Thomas Tuerk, Andrea Giugliano, Anil Madhavapeddy, and Peter Sewell. 2015. SibylFS: Formal Specification and Oracle-based Testing for POSIX and Real-world File Systems. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP '15)*. ACM, New York, NY, USA, 38–53.

[54] Ohad Rodeh, Josef Bacik, and Chris Mason. 2013. BTRFS: The Linux B-Tree Filesystem. *Trans. Storage* 9, 3, Article 9 (Aug. 2013), 32 pages.

[55] Koushik Sen. 2008. Race Directed Random Testing of Concurrent Programs. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '08)*. ACM, New York, NY, USA, 11–21.

[56] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker. In *Presented as part of the 2012 USENIX Annual Technical Conference (USENIX ATC '12)*. USENIX, Boston, MA, 309–318.

[57] J. D. Strunk, G. R. Goodson, M. L. Scheinholtz, C. A. N. Soules, and G. R. Ganger. 2003. Self-securing Storage: Protecting Data in Compromised Systems. In *Foundations of Intrusion Tolerant Systems, 2003 [Organically Assured and Survivable Information Systems]*. 195–209.

[58] Goswami Sudhanshu. 2005. An introduction to KProbes. https://lwn.net/Articles/132196/.

[59] Cong Tian, Zhenhua Duan, and Nan Zhang. 2012. An Efficient Approach for Abstraction-refinement in Model Checking. *Theoretical Computer Science* 461 (2012), 76–85.

[60] GJ Tretmans and Hendrik Brinksma. 2003. Torx: Automated Model-based Testing. In *First European Conference on Model-Driven Software Engineering*. 31–43.

[61] Jan Tretmans. 2008. Model Based Testing with Labelled Transition Systems. In *Formal methods and testing*. Springer, 1–38.

[62] Mark Utting, Alexander Pretschner, and Bruno Legeard. 2012. A Taxonomy of Model-based Testing Approaches. *Software Testing, Verification and Reliability* 22, 5 (2012), 297–312.

[63] Margus Veanes, Colin Campbell, Wolfgang Grieskamp, Wolfram Schulte, Nikolai Tillmann, and Lev Nachmanson. 2008. Model-based Testing of Object-oriented Reactive Systems with Spec Explorer. In *Formal methods and testing*. Springer, 39–76.

[64] James A Whittaker and Michael G Thomason. 1994. A Markov Chain Model for Statistical Software Testing. *IEEE Transactions on Software engineering* 20, 10 (1994), 812–824.

[65] James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas Anderson. 2015. Verdi: A Framework for Implementing and Formally Verifying Distributed Systems. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '15)*. ACM, New York, NY, USA, 357–368.

[66] Wen Xu, Hyungon Moon, Sanidhya Kashyap, Po-Ning Tseng, and Taesoo Kim. 2019. Fuzzing file systems via two-dimensional input space exploration. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 818–834.

[67] Junfeng Yang, Tisheng Chen, Ming Wu, Zhilei Xu, Xuezheng Liu, Haoxiang Lin, Mao Yang, Fan Long, Lintao Zhang, and Lidong Zhou. 2009. MODIST: Transparent Model Checking of Unmodified Distributed Systems. (2009).

[68] Junfeng Yang, Can Sar, and Dawson Engler. 2006. EXPLODE: A Lightweight, General System for Finding Serious Storage System Errors. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7 (OSDI '06)*. USENIX Association, Berkeley, CA, USA, 10–10.

[69] Junfeng Yang, Paul Twohey, Dawson Engler, and Madanlal Musuvathi. 2006. Using Model Checking to Find Serious File System Errors. *ACM Trans. Comput. Syst.* 24, 4 (Nov. 2006), 393–423.