Cross-checking Semantic Correctness: The Case of Finding File System Bugs

Changwoo Min Sanidhya Kashyap Byoungyoung Lee Chengyu Song Taesoo Kim
Georgia Institute of Technology

Abstract

Today, systems software is too complex to be bug-free. To find bugs in systems software, developers often rely on code checkers, like Linux's Sparse. However, the capability of existing tools used in commodity, large-scale systems is limited to finding only *shallow bugs* that tend to be introduced by simple programmer mistakes, and so do not require a deep understanding of code to find them. Unfortunately, the majority of bugs as well as those that are difficult to find are *semantic ones*, which violate high-level rules or invariants (e.g., missing a permission check). Thus, it is difficult for code checkers lacking the understanding of a programmer's true intention to reason about semantic correctness.

To solve this problem, we present JUXTA, a tool that automatically infers high-level semantics directly from source code. The key idea in JUXTA is to compare and contrast multiple existing implementations that obey latent yet implicit high-level semantics. For example, the implementation of `open()` at the file system layer expects to handle an out-of-space error from the disk in all file systems. We applied JUXTA to 54 file systems in the stock Linux kernel (680K LoC), found 118 previously unknown semantic bugs (one bug per 5.8K LoC), and provided corresponding patches to 39 different file systems, including mature, popular ones like ext4, btrfs, XFS, and NFS. These semantic bugs are not easy to locate, as all the ones found by JUXTA have existed for over 6.2 years on average. Not only do our empirical results look promising, but the design of JUXTA is generic enough to be extended easily beyond file systems to any software that has multiple implementations, like Web browsers or protocols at the same layer of a network stack.

SOSP'15, October 4–7, 2015, Monterey, CA.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-3834-9/15/10...$15.00.
http://dx.doi.org/10.1145/2815400.2815422

1. Introduction

Systems software is buggy. On one hand, it is often implemented in unsafe, low-level languages (e.g., `C`) for achieving better performance or directly accessing the hardware, thereby facilitating the introduction of tedious bugs. On the other hand, it is too complex. For example, Linux consists of almost 19 million lines of pure code and accepts around 7.7 patches per hour [18].

To help this situation, especially for memory corruption bugs, researchers often use memory-safe languages in the first place. For example, Singularity [34] and Unikernel [43] are implemented in `C#` and `OCaml`, respectively. However, in practice, developers largely rely on code checkers. For example, Linux has integrated static code analysis tools (e.g., Sparse) in its build process to detect common coding errors (e.g., checking whether system calls validate arguments that come from userspace). Other tools such as Coverity [8] and KINT [60] can find memory corruption and integer overflow bugs, respectively. Besides these tools, a large number of dynamic checkers are also available, such as `kmemleak` for detecting memory leaks and AddressSanitizer [51] for finding use-after-free bugs in Linux.

Unfortunately, lacking a deep understanding of a programmer's intentions or execution context, these tools tend to discover *shallow bugs*. The majority of bugs, however, are *semantic ones* that violate high-level rules or invariants [15, 42]. According to recent surveys of software bugs and patches, over 50% of bugs in Linux file systems are semantic bugs [42], such as incorrectly updating a file's timestamps or missing a permission check. Without domain-specific knowledge, it is extremely difficult for a tool to reason about the correctness or incorrectness of the code and discover such bugs. Thus, many tools used in practice are ineffective in detecting semantic vulnerabilities [15].

In this regard, a large body of research has been proposed to check and enforce semantic or system rules, which we broadly classify into three categories: model checking, formal proof, and automatic testing. A common requirement for these techniques is that developers should manually provide the correct semantics of code for checking: models to check and proofs of program properties. Unfortunately, creating such semantics is difficult, error-prone, and virtually infeasible for commodity systems like Linux.

To solve this problem, we present JUXTA, a tool that automatically infers high-level semantics from source code. The key intuition of our approach is that different implementations of the *same* functionality should obey the same system rules or semantics. Therefore, we can derive *latent* semantics by comparing and contrasting these implementations. In particular, we applied JUXTA to 54 file system implementations in stock Linux, which consists of 680K LoC in total. We found 118 previously unknown semantic bugs (one bug per 5.8K) and provided corresponding patches to 39 different file systems, including mature and widely adopted file systems like ext4, btrfs, XFS, and NFS. We would like to emphasize that these semantic bugs JUXTA found are difficult to find, as they have existed for over 6.2 years on average; over 30 bugs were introduced more than 10 years ago.

Challenges. The main challenge in comparing multiple file system implementations arises because, although all of them implicitly follow certain high-level semantics (e.g., expect to check file system permissions when opening a file), the logic of each (e.g., features and disk layout) is dramatically different from that of the others. More importantly, these high-level semantics are deeply embedded in their code in one way or another without any explicit, common specifications. To overcome these challenges, instead of directly comparing all of the file system implementations, we devised two statistical models that properly capture common semantics, yet tolerate the specific implementation of each file system; in other words, JUXTA identifies deviant behavior [29] derived from common semantics shared among multiple different software implementations.

Contributions. This paper makes the following contributions:

- We found 118 previously unknown semantic bugs in 39 different file systems in the stock Linux kernel. We made and submitted corresponding patches to fix the bugs that we found. Currently, patches for 65 bugs are already in the mainline Linux kernel.
- Our idea and design for inferring latent semantics by comparing and contrasting multiple implementations. In particular, we devise two statistical comparison schemes that can compare multiple seemingly different implementations at the code level.
- The development of an open source tool, JUXTA, and its pre-processed database to facilitate easy building of different checkers on top of it. We made eight checkers including a semantic comparator, a specification/interface generator, an external APIs checker, and a lock pattern checker.

The rest of this paper is organized as follows. §2 provides the motivation for JUXTA's approach with a case study. §3 gives an overview of its workflow. §4 describes JUXTA's design. §5 shows various checkers built on top of JUXTA. §6 shows JUXTA's implementation. §7 explains the bugs we found. §8 discusses our potential applications. §9 compares JUXTA with previous research and §10 provides the conclusion.

2. Case Study

Linux provides an abstraction layer called the virtual file system (VFS). The Linux VFS defines an interface between a file system and Linux, which can be viewed as an implicit specification that all file systems should obey to be interoperable with Linux. To derive this latent specification in existing file systems, JUXTA compares the source code of each file system originating from these VFS interfaces. However, the VFS interface is complex: it consists of 15 common operations (e.g., `super_operations`, `inode_operations`) that comprise over 170 functions. In this section, we highlight three interesting cases (and bugs) that JUXTA found: `rename()`, `write_begin/end()`, and `fsync()`.

2.1 Bugs in `inode.rename()`

One might think that `rename()` is a simple system call that changes only the name of a file to another, but it has very subtle, complicated semantics. Let us consider a simple example that renames a file, "`old_dir/a`" to another, "`new_dir/b`":

```
rename("old_dir/a", "new_dir/b");
```

Upon successful completion, "`old_dir/a`" is renamed "`new_dir/b`" as expected, but what about the timestamps of involved directories and files? To precisely implement `rename()`, developers should specify the semantics of 12 different pieces of mutated state: three timestamps, `ctime` for status change, `mtime` for modification, and `atime` for access timestamps, all of which for each of four inodes, `old_dir`, `new_dir`, `a`, and `b`. In fact, POSIX partially defines its semantics: updating `ctime` and `mtime` of two directories, `old_dir` and `new_dir`:

> *"Upon successful completion, rename() shall mark for update the last data modification and last file status change timestamps of the parent directory of each file."* [55]

In the UNIX philosophy, this specification makes sense since `rename()` never changes the timestamps of both files, `a`, and `b`. But in practice, it causes serious problems, as developers believe that the status of both files (`ctime`) is changed after `rename()`.

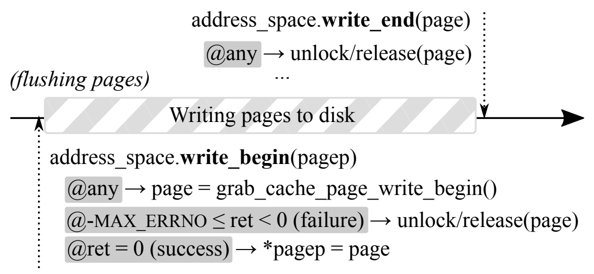
For example, a popular archiving utility, `tar`, used to have a critical problem when performing an incremental backup (`-listed-incremental`). After a user renames a file `a` to `b`, `tar` assumes file `b` is already backed up and file `a` is deleted, as the `ctime` of `b` is not updated after the rename. Then, upon extraction, `tar` deletes file `a`, which it thinks was deleted, and never restores `b` as it was skipped, thereby losing the original file that the user wanted to back up [1].

However, it is hard to say this is `tar`'s fault because the majority of file systems update the `ctime` of new and old

POSIX	Linux 4.0-rc2	Belief *	VFS *	AFFS	BFS	Coda	FAT	HFS	HFSplus	HPFS	JFFF2	RAMFS	UDF	XFS
Defined	`old_dir->i_ctime`	✓	✓	✓	✓	✓	✓	✓	✓	-	✓	✓	✓	✓
	`old_dir->i_mtime`	✓	✓	✓	✓	✓	✓	✓	✓	-	✓	✓	✓	✓
	`new_dir->i_ctime`	✓	✓	✓	✓	✓	✓	✓	✓	-	✓	✓	-	✓
	`new_dir->i_mtime`	✓	✓	✓	✓	✓	✓	✓	✓	-	✓	✓	-	✓
	`new_dir->i_atime`	-	-	-	-	-	✓	-	-	-	-	-	-	-
Undefined	`new_inode->i_ctime`	✓	-	✓	-	-	✓	-	-	-	-	-	✓	-
	`old_inode->i_ctime`	✓	✓	-	-	-	-	-	-	-	-	✓	✓	✓

Belief * supported by ext2, ext3, ext4, btrfs, F2FS, NILFS2, ReiserFS, CIFS, EXOFS, JFS, UBIFS, UFS, OCFS, GFS2, and MINIX

Table 1: Summary of `rename()` semantics of updating timestamp upon successful completion. A majority of Linux file systems update the status change timestamp (e.g., `i_ctime`/`i_mtime`) of source and destination files (e.g., `old/new_inode`) and update both status change and modification timestamps of both directories that hold source and destination files (e.g., `old/new_dir`). Note that the access timestamp of the new directory should not be updated; only FAT updates the access timestamp, implying that this is likely semantically incorrect. By observing such deviant behaviors of existing file system implementations, JUXTA can faithfully pinpoint potential semantic bugs without any concrete, precise models constructed by programmers. The most obvious deviant ones are HPFS and UDF, as the former one does not update any of the times (neither `i_ctime` nor `i_mtime`) for `old_inode` and `new_inode`, whereas the latter one only updates the `old_inode` timestamps. JUXTA's side-effect checker found six such bugs. In addition, after sending patches for the bugs, the corresponding developers fixed eighteen more similar bugs in the file system [36, 46].



address_space.**write_end**(page)
@any → unlock/release(page)
...
(flushing pages)
Writing pages to disk
address_space.**write_begin**(pagep)
@any → page = grab_cache_page_write_begin()
@-MAX_ERRNO ≤ ret < 0 (failure) → unlock/release(page)
@ret = 0 (success) → *pagep = page

Figure 1: Simplified semantics of address space operations that we extracted from 12 file system implementations in Linux. Each file system implements such operations to manipulate mappings of a file to the page cache in Linux file systems: for example, `write_begin()` to prepare page writes and `write_end()` to complete the page write operation. Grayed boxes indicate derived return conditions and their right sides represent common operations executed by all file systems under such return conditions.

files after `rename()`, though POSIX leaves this behavior undefined. In fact, assuming that `rename()` updates the `ctime` of both files is more reasonable for developers, as traditionally, `rename()` has updated `ctime` when implemented with `link()` and `unlink()` [1, 55]. Not only does Solaris still modify the `ctime` after renaming files for compatibility of its backup system (`ufsdump` and `ufsrestore`), but also modern and mainstream file systems (e.g., ext4 and btrfs) in Linux update the `ctime` after `rename()` (see Table 1).

In this paper, we propose an automatic way to extract high-level semantics and assess their correctness from multiple implementations without any specification. For example, JUXTA can summarize the semantics of Linux file systems by comparing over 50 different file system implementations in stock Linux, as shown in the summary of `rename()` semantics (and buggy behaviors) in Table 1.

2.2 Bugs in `address_space.write_begin/end()`

Linux file systems support memory-mapped files that map a file to a process' memory space. To manipulate these mappings, each file system implements a set of operations, called `address_space_operation`. As these operations are tightly integrated into the VFS, each file system should obey certain rules, ranging from as simple as returning zero upon success to as complex as requiring specific locks with specific orders and releasing pages upon error. In fact, such rules consist of 290 lines of document just for address space operations [31].

The key idea of this paper is to extract such semantics from existing file systems without any domain-specific model or specification. In this case study, we use two functions of `address_space_operation` (out of 19 functions) to explain our extracted semantics in detail: `write_begin()` to prepare page writes and `write_end()` to complete the operation. Figure 1 depicts the high-level semantics of `write_begin()` and `write_end()` that 12 file systems in Linux obey. For example, upon successful completion of `write_begin()`, they allocate a page cache, update the page pointer (`pagep`), and return zero (indicating the operation's success). Upon failure, they always unlock and release the page cache and return an error code (a range of error values). However, in `write_end()`, the allocated page cache should be unlocked and released in all possible code paths because Linux VFS starts the write operation only after `write_begin()` successfully locks the page cache.

JUXTA can derive such high-level semantics that are often not clearly specified (see §2.3) or are only implicitly followed by developers (by mimicking a mainstream file system like ext4). More importantly, JUXTA found two previously unknown violations in `write_end()` of affs and one in `write_begin()` of Ceph (all patched already) by comparing 12 different file systems that implement the address space operation.

2.3 Bugs in `man` Pages, Documents, and Comments

After comparing `fsync()` implementations of various file systems, JUXTA found that a few file systems (ext3, ext4, and OCFS2) return `-EROFS` to indicate that the file system is mounted as read-only, while the rest of the file systems never report such an error.[1] More surprisingly, the POSIX standard for `fsync()` [54] does not define `-EROFS` error, but the Linux manual does [56].

On further analysis, it turns out that this is a very subtle case that only some file system developers are aware of [58, 59], although the VFS documentation [31] and POSIX specification [54] never explicitly state this error. This corner case can happen when the status of the file system changes from read-write to read-only (e.g., remounting upon file system error). Since a file's inode flag (`i_flags`) only inherits the mount flag of its file system at `open()`, all files in use (their `i_flags`) will not reflect the current state of their file system after the status change. In other words, to avoid a disastrous situation in which a file system overwrites its disk blocks after being remounted as read-only, all `fsync()` invocations should independently check the status of the file system to see if it is read-only (`i_sb->s_flags & MS_RDONLY`) instead of trusting the inode flag.

```c
// @v4.0-rc2/fs/ubifs/file.c:1321
int ubifs_fsync(...) {
  ...
  if (sbi->ro_mount)
    // For some really strange reasons VFS does not filter out
    // 'fsync()' for R/O mounted file-systems as per 2.6.39.
    return 0;
}
```

Based on analysis using JUXTA, we found that all file systems except ext3, ext4, OCFS2, UBIFS, and F2FS never consider such errors (even the ones relying on `generic_file_fsync()`—32 files systems in total). We believe that such violations should be considered as potential bugs.

3. Overview

JUXTA's workflow is a sequence of stages (Figure 2). For each file system in Linux, JUXTA's source code merge stage first combines the entire file system module as a single large file (e.g., rescheduling symbols to avoid conflicts) for precise inter-procedural analysis in its symbolic execution engine (§4.1). Second, JUXTA symbolically enumerates all possible C-level code paths (unlike instruction-based symbolic execution engines), which generates path conditions and side-effects as symbolic expressions (§4.2). JUXTA then preprocesses the generated path conditions (§4.3, §4.4) and creates a database for all checkers (applications) to build upon (§5). The database contains rich symbolic path conditions that can be easily and quickly accessed by various other checkers for their needs.

Semantic bugs. The term *semantic bug* is used ambiguously to describe a type of bug that requires domain-specific knowledge to understand (e.g., execution context in kernel), whereas shallow bugs, like memory errors (e.g., buffer overflow), can be understood in relatively narrower contexts. One might think that memory corruption bugs are more critical from the systems security perspective, as they are often exploitable by attackers. However, semantic bugs are also serious (e.g., resulting in file system corruption and complete loss of data [30, 42]), as well as security-critical (e.g., missing an ACL check in a file system results in privilege escalation without any sophisticated exploits or attacks).

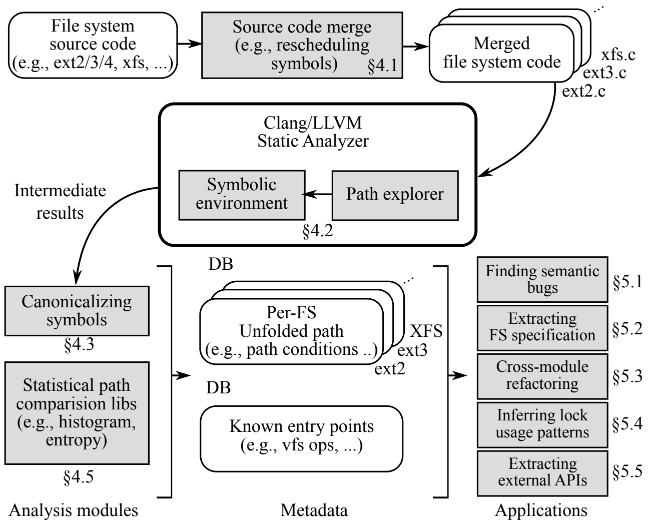

File system source code (e.g., ext2/3/4, xfs, ...)
Source code merge (e.g., rescheduling symbols) §4.1
Merged file system code
xfs.c
ext3.c
ext2.c
Clang/LLVM Static Analyzer
Symbolic environment
Path explorer
§4.2
Intermediate results
Canonicalizing symbols
§4.3
Statistical path comparision libs (e.g., histogram, entropy)
§4.5
DB
Per-FS Unfolded path (e.g., path conditions ..)
XFS
ext3
ext2
DB
Known entry points (e.g., vfs ops, ...)
Finding semantic bugs §5.1
Extracting FS specification §5.2
Cross-module refactoring §5.3
Inferring lock usage patterns §5.4
Extracting external APIs §5.5
Analysis modules
Metadata
Applications

Figure 2: Overview of JUXTA's architecture and its workflow. For each file system in Linux, the source code merge stage first combines the entire file system module into a single large file (§4.1); then JUXTA's symbolic execution engine enumerates C-level code paths with proper symbolic execution contexts (§4.2). By preprocessing the generated path conditions (§4.3, §4.4), JUXTA creates a database that all other checkers (§5) build upon.

In general, semantic bugs are difficult to detect. In our study, we measured how long each semantic bug that JUXTA detected remained undiscovered (i.e., its latent period). The result showed that their average latent period was over 6.2 years, implying that JUXTA is effective in finding long-existing semantic bugs.

To be precise, we classified semantic bugs in file systems into four categories that are similar to a recent survey on file system bugs [42]: **(S)** State, representing inconsistent state updates or checks (e.g., missing permission check); **(C)** Concurrency, indicating bugs related to execution context that might lead to deadlock or system hang (e.g., lock/unlock and GPF flag); **(M)** Memory error, representing inconsistent usage of memory-related APIs (e.g., missing a `kfree` unlike other paths); and **(E)** Error handling bugs (e.g., inconsistently handling errors). Table 5 and Table 6 incorporate these notations to describe relevant semantic bugs.

4. Design

The key insight behind our approach is that the same VFS entry functions (e.g., `inode_operations.open()`) should have the same or similar high-level semantics (e.g., handling common errors and returning standard error codes). JUXTA attempts to extract the latent VFS specification by finding the

[1] UBIFS and F2FS check the read-only status of the filesystem but return zero.

L	Type	Symbolic expression
1	FUNC	ext4_rename(old_dir, old_dentry new_dir, new_dentry, flags)
2	RETN	0
3	COND	(S#old_dir->i_sb->s_time_gran) >= (I#1000000000)
4	COND	(S#old_dir->i_sb->s_fs_info->s_mount_opt) & (C#EXT4_MOUNT_QUOTA) = 0
5	COND	(S#flags) & (C#RENAME_WHITEOUT) != 0
6	COND	(E#IS_DIRSYNC(old_dir)) = 0
7	COND	(E#IS_DIRSYNC(new_dir)) = 0
8	COND	(E#S_ISDIR(old->inode->i_mode)) = 0
9	COND	(E#ext4_add_entry(handle, new_dentry, old->inode)) = 0
10	CALL	(T#1) = ext4_add_entry(handle, new_dentry, old->inode)
11	ASSN	retval = (T#1)
12	CALL	(T#2) = ext4_current_time(old_dir)
13	ASSN	old_dir->i_mtime = (T#2)
14	ASSN	old_dir->i_ctime = (T#2)
15	CALL	ext4_dec_count(handle, old_dir)
16	CALL	ext4_inc_count(handle, new_dir)
17	CALL	(T#3) = ext4_current_time(new_dir)
⋆18	ASSN	new_dir->i_mtime = (T#3)
⋆19	ASSN	new_dir->i_ctime = (T#3)
⋆20	CALL	ext4_mark_inode_dirty(handle, new_dir)
21	CALL	ext4_mark_inode_dirty(handle, old_dir)
22	ASSN	retval = 0

Table 2: Simplified symbolic conditions and expressions of a success path (RETN=0) for ext4_rename(). In order to reach the return statement (top section), the conditions (middle section) should be satisfied under a set of side-effects (bottom section). ⋆ represents expressions related to the ext4 rename patch in Figure 3. S# means a symbolic expression, I# means an integer value, C# means a constant, and T# means a temporary variable.

commonalities of file systems and generating bug reports if it finds any deviant behavior among them. However, since file systems are large and complex software that implement different design decisions (e.g., on-disk layout), it is challenging to compare different file systems in the first place. To address these challenges, we propose two statistical path comparison methods (§4.5). In this section, we elaborate on our path-sensitive and context-aware static analysis techniques, which use symbolic execution at the compilation pipeline to compare multiple implementations at the source code level.

4.1 Merging Source Code in a File System

File systems are tightly coupled, large software. For example, ext4, XFS, and btrfs are composed of 32K, 69K, and 85K lines of code, respectively. For accurate comparison, interprocedural analysis is essential for obtaining precise symbolic execution contexts and environments. However, most publicly available static analyzers can perform interprocedural analysis of only limited scope within a code base. In particular, the Clang static analyzer, which JUXTA relies on, cannot go beyond the boundary of a single file for the inter-procedural analysis [4].

To work around this problem, we develop a *source code merge stage* that combines each file system's files into one large file, thereby enabling a simple inter-procedural analysis within a file system module. The source code merge phase first parses a build script of the file system to determine all required source files and configuration options; it then parses these source files to resolve conflicts among file-scoped symbols, such as `static` variables, functions, `struct`s, forward declarations, header files, etc. The process is largely automatic; 49 out of 54 file systems were automatically handled, and the rest required some specific rewriting (e.g., conflicted symbols defined by complex macros).

```
// [commit: 53b7e9f6] @fs/ext4/namei.c
int ext4_rename(struct inode *old_dir, ...
                struct inode *new_dir, ...) {
   ...
+  new_dir->i_ctime = new_dir->i_mtime \
+      = ext4_current_time(new_dir);
+  ext4_mark_inode_dirty(handle, new_dir);
   ...
}
```

Figure 3: A patch for a semantic bug in ext3/4: update `mtime` and `ctime` of a directory that a file is renamed to (see §2.1 and Table 2).

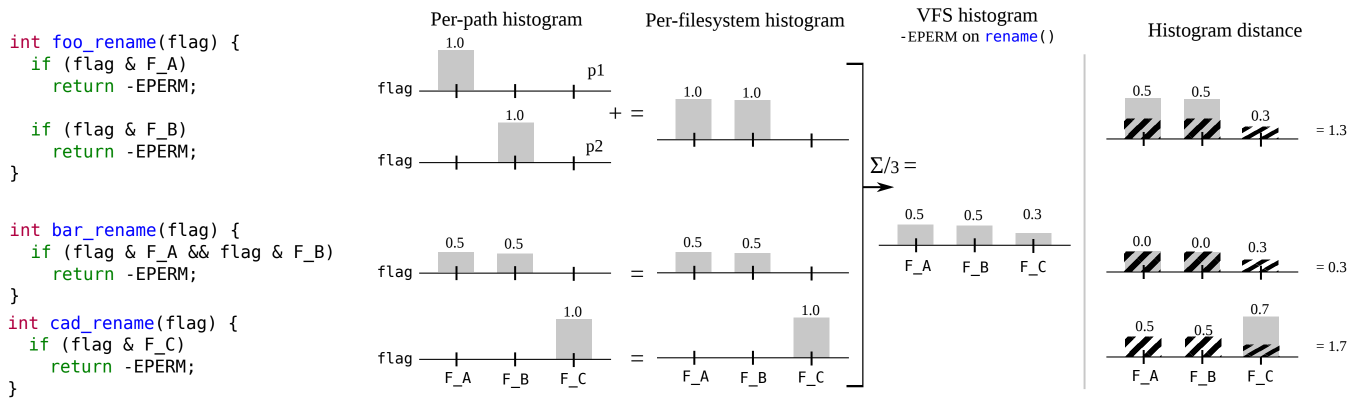
4.2 Exploring Paths to the End of a Function

To compare multiple file systems, JUXTA collects execution information for each function. It constructs a control-flow graph (CFG) for a function and *symbolically* explores a CFG from the entry to the end (typically, `return` statement). To gather execution information across functions, JUXTA inlines callee functions and creates a single CFG. To prevent the path explosion problem, we set the maximum inlined basic blocks and functions to 50 and 32, respectively, thereby limiting the size of intermediate results to less than half a terabyte. While exploring a CFG, JUXTA performs range analysis by leveraging branch conditions to narrow the possible integer ranges of variables. In JUXTA, a single execution path is represented as a five-tuple: **(1)** function name (`FUNC`), **(2)** return value (or an integer range) (`RETN`), **(3)** path conditions (integer ranges of variables) (`COND`), **(4)** updated variables (`ASSN`), and **(5)** callee functions with arguments (`CALL`). Table 2 shows a simplified (e.g., no type information) example of symbolic condition extraction for a successful path (`RETN=0`) for `rename()` in ext4.

JUXTA's symbolic expression is rich, as it was performed at the `C` level rather than on low-level instructions (e.g., LLVM-IR). For example, it understands macros that a preprocessor (`cpp`) uses, fields of a `struct`, and function pointers. This design decision not only increases the accuracy of the analysis, but also makes the generated report more human-readable, which is critical to identifying false positives. As JUXTA explores two possible paths at each branch condition, the generated path conditions are flow-sensitive and path-sensitive. For 54 Linux file systems, JUXTA extracts 8 million execution paths and 260 million conditions that aggregate to 300 GB.

4.3 Canonicalizing Symbols

Extracted path information at the `C` level is not a form well-suited for comparison because each file system may use different symbol names for the same data (e.g., arguments). For example, ext4 uses `old_dir` and GFS2 uses `odir` as the first argument of `inode_operations.rename()`. In our *symbol canonicalization pass*, JUXTA transforms extracted path



```
int foo_rename(flag) {
  if (flag & F_A)
    return -EPERM;

  if (flag & F_B)
    return -EPERM;
}

int bar_rename(flag) {
  if (flag & F_A && flag & F_B)
    return -EPERM;
}
int cad_rename(flag) {
  if (flag & F_C)
    return -EPERM;
}
```

Figure 4: Histogram representation of `rename()` in three contrived file systems (`foo`, `bar` and `cad`) on the `-EPERM` path. Histogram-based comparison represents a universal variable (`flag`) as one dimension (x-axis). To describe `rename()`, it computes the average of all file system histograms, and deducts the average from per-file system histograms for comparison. For example, `foo` is sensitive ($+0.5$) and `cad` is insensitive (-0.5) on the `F_A` flag. Globally, `cad` behaves the most differently (deviant) from `foo` and `bar` (1.7) in terms of `-EPERM` path.

information into a directly comparable form across file systems. The key idea is to represent symbolic expressions by using universally comparable symbols such as function arguments, constants, function returns, global variables, and (some) local variables. Symbol names in inlined functions are renamed to those of the VFS entry function. For example, `old_dir` of ext4 and `odir` of GFS2 become canonicalized to the common symbol, `$A0`, indicating the first argument of a function. Please note that Table 2 is not canonicalized for clarity of exposition. After symbol canonicalization, semantically identical symbols (e.g., the same arguments of the identical VFS interface) have a matching string representation, which facilitates comparison over different file systems.

4.4 Creating the Path Database

To avoid performing expensive symbolic execution over and over again, we create a per-file system path database for applications such as checkers and specification extractors to easily use the extracted path information. The path database is hierarchically organized with function name, return value (or range), and path information (path conditions, side-effects, and callee functions). Applications can query our path database using a function name or a return value as keys. In addition, we created a VFS entry database for applications to easily iterate over the same VFS entry functions (e.g., `ext4_rename()`, `btrfs_rename()`) of the matching VFS interface function (e.g., `inode_operations.rename()`). Most file systems have multiple entry functions for several VFS operations. For example, in the case of `xattr` operations, ext4 has different entry functions for each namespace (e.g., `ext4_xattr_trusted_list()` for the trusted namespace and `ext4_xattr_user_list()` for the user namespace). In such cases, we create multiple sets of VFS entry functions so that JUXTA applications can compare functions with the same semantics. The 54 file systems in Linux kernel 4.0-rc2 have 2,424 VFS entry functions total. To handle the massive volume of the path database, JUXTA loads and iterates over the path database in parallel.

4.5 Statistical Path Comparison

Even with the canonicalized path database, it is not meaningful to compare multiple different file systems. Because all file systems employ different design decisions (e.g., disk layout) and features (e.g., snapshot), naively representing path information in a logical form (e.g., weakest precondition [10, 23]) or relying on constraint solvers [11, 22] will lead one to conclude that all file systems are completely different from each other. In this section, we describe two statistical methods to compute the deviation and commonality of different file systems from noisy path information.

Histogram-based comparison. The integer ranges, such as return values and path conditions, constitute features in model generation and comparison. To efficiently compare these multidimensional integer ranges (i.e., multiple features), we propose a *histogram-based comparison* method, which is popular in image processing [26]. The advantage of our histogram-based representation is that we can reuse well-established standard histogram operations [26]. JUXTA first encodes integer ranges into multidimensional histograms for comparison and uses distances among histograms to find deviant behaviors in different file systems. A standard (and intuitive) way to measure distance between two histograms is to measure the area size of non-overlapping regions.

Overall, JUXTA's histogram-based comparison is composed of four steps (Figure 4 illustrates an example on `rename()`): (1) it transforms each element of return values and path conditions into a single histogram (per-path histogram); (2) if the comparison involves multiple execution paths (e.g., paths that return the same value), JUXTA aggregates multiple histograms from the previous step (per-file system histogram); (3) JUXTA builds the *stereotypical path information* of a VFS interface by calculating the average histograms of multiple file systems (VFS histogram); (4) JUXTA finally compares each file system's histogram (i.e., per-file system histogram) with the averaged histogram (i.e., VFS histogram) by computing a distance between them.

In particular, in the first step, one integer range is represented as a start value, an end value, and height. Naturally, its start and end values are set to symbolic execution results in our path database (e.g., constant values determining a path condition). Then, a height value is normalized so that the area size of a histogram is always 1. Since each histogram has the same area size, the distance measure (i.e., computing non-overlapping area size between histograms) that we compute in the last step can fairly compare histograms under the constraint that each execution path exhibits the same level of importance.

In the second step, to present common features in each file system, we combine multiple per-path histograms into a per-file system histogram using a union operation on histograms. The union of two histograms is obtained by superimposing the two histograms and taking the maximum height for their overlapping regions. It is used to combine multiple instances of path information of the same function returning the same value in a file system.

Next, in the third step, we compute an average of multiple per-file system histograms to extract the generic model on each VFS interface. As with the union operation, the average of `N` histograms is obtained by stacking `N` histograms and dividing each one's height by `N`. We consider the average of multiple file systems' histograms as the stereotypical path information of a VFS interface. One nice property of the average operation is that ranges of commonly used variables retain their great magnitudes and ranges of rarely used variables (mostly, file system-specific regions) fall in magnitude.

Lastly, we measure the differences between each file system (i.e., per-file system histogram) and its stereotype (i.e., the VFS histogram). Although there are various ways to compute distance between two histograms [39, 48, 53], we used the simple yet computationally efficient method, histogram intersection distance [53]. The distance between two histograms is defined as the size of their non-overlapping regions. Also, the distance in multidimensional histogram space is defined as the Euclidean distance in each dimension. In our histogram-based framework, finding a bug is finding file system path information that is far from the stereotype (i.e., the average or VFS histogram). Figure 4 shows the histogram representation of `rename()` for three file systems on the `-EPERM` path. We explain how our checkers use this in the next section.

Entropy-based comparison. Histogram-based comparison is for the comparison of multidimensional range data. To find deviation in an event, we use information-theoretic entropy [52]. In JUXTA, such events can occur in one of two cases: how a flag is used for a function (e.g., `kmalloc(*,GFP_KERNEL)` or `kmalloc(*,GFP_NOFS)`); or how a return value of a function is handled (e.g., `ret != 0` vs `IS_ERR_OR_NULL(ret)`).

Return value	VFS interface				
	`listxattr`	`mknod`	`remount`	`rename`	`statfs`
`-EDQUOT`	JFS	-	OCFS2	-	OCFS2
`-EIO` †	JFS	-	-	ext3/JFS	-
`-EPERM`	F2FS	-	-	-	-
`-EOVERFLOW`	-	btrfs	-	-	-
`-EROFS`	-	-	ext4	-	OCFS2

Table 3: The return codes of some file systems that are not specified in the `man` page for the VFS interfaces. JUXTA marked them as deviant behaviors. An interesting point is that the POSIX manual defines `-EIO` whereas it is not mentioned in the Linux programmer's manual (marked †) for `rename()`.

JUXTA calculates the entropy of each VFS interface for an event. By definition, this entropy can be either maximum, when events completely occur in a random fashion, or zero, when only one kind occurs. So, a VFS interface whose corresponding entropy is small (except for zero) can be considered as buggy. Among the file systems that implement the VFS interface with small entropy, the file system with the least frequent event can be considered buggy. We explain how our checkers and specification extractor use this method in the next section.

Bug report ranking. JUXTA ranks all of the generated bug reports by using the following quantitative metrics. For histogram-based checkers, the occurrence of a bug is more likely for a greater distance value, whereas for entropy-based checkers, a smaller (non-zero) entropy value indicates greater heuristic confidence that a bug has been found. Programmers can leverage these generated reports by prioritizing the highest-ranked bugs first, as many of JUXTA's high-ranked reports are for true-positive bugs, as we show in §7.3.

5. Applications

In this section, we introduce eight checkers that are built on top of the database for symbolic path conditions and statistical schemes to compare different code bases.

5.1 Cross-checking File System Semantics

All file systems are different in design, features, and goals, but at the same time they share important commonalities: their external behaviors (e.g., POSIX compliance) and (for all Linux file systems) their compliance with rules defined in the VFS layer. The VFS layer defines interface functions that concrete file systems should implement, and it also defines VFS objects, such as `superblock`, `inode`, and `dentry`. Interestingly, many file systems have surprising similarities to ext2/3/4 as we can see from their file structure, function decomposition, and naming convention. As a result, a bug seen in a file system likely exists in other file systems; for example, the `rename()` bug in §2.1 was first fixed in ext3, and subsequently in ext4 [2]. Based on this intuition, we developed four checkers that cross-check file system semantics to find bugs.

Return code checker. Our first checker cross-checks the return codes of file systems for the same VFS interface, and reports whether there are deviant error codes in file systems.[2] It creates a per-file system return histogram by aggregating all return values in all paths and computes the average histogram of all file systems (i.e., VFS histogram). Distances between the VFS and per-file system histograms are measured, and VFS interfaces of file systems with large distance values are reported as bugs. Our bug reports include deviant return values by analyzing non-overlapping regions. Our checker found deviant return codes in some file systems that are not specified in the `man` page (Table 3).

Side-effect checker. To discover missing updates, our checker compares side-effects for a given VFS interface and a return value. It encodes side-effects into a histogram by mapping each canonicalized symbolic variable (e.g, the first argument of `rename()`) to a unique integer, regardless of the file systems. As two file systems update more common variables, larger overlapping regions in their histograms cause the distance between them to reduce. File system-specific variables will occur once, so their impact is naturally scaled down by averaging histograms. Thus, our checker finds deviant updates in commonly updated variables.

Function call checker. Deviant function calls can be related to either deviant behavior or a deviant condition check. Similar to the side-effect checker, our function call checker encodes function calls into histograms by mapping each function to a unique integer and finds deviant function calls by measuring the distance to the average.

Path condition checker. To discover missing condition checks, our checker encodes the path conditions of a file system into a multidimensional histogram. One unique symbolic expression is represented as one dimension of the histogram (Figure 4). For example, the path conditions of Table 2 (L3–L9) are represented as a seven-dimensional histogram. Multiple execution paths for the same return value are represented as a single histogram by aggregating each execution path. Since symbolic expressions are already canonicalized in §4.3, the same symbolic expressions are considered in the same dimension of the histogram regardless of the file system. Also, if a path condition is file system-specific, it is naturally scaled down while averaging histograms. Thus, JUXTA cross-checks common path conditions among file systems.

5.2 Extracting File System Specification

Given the enriched information from JUXTA, we extract the latent specification from the file system implementations in Linux. The extracted specification is particularly useful for novice developers who implement a file system from scratch, as it can be referred to a starting template, or even for experts who maintain the mature code base, as it gives a high-level summary of other file system implementations. Extracting

[2] In Linux kernel 4.0-rc2, 147 of 173 VFS interfaces are non-void.

```
[Specification] @inode_operations.setattr:
int setattr(struct dentry *dentry, struct iattr *attr) {
  @[CALL] (17/17) RET < 0:
          inode_change_ok(dentry->inode, attr)
  @[COND] (10/17) RET = posix_acl_chmod(...):
          attr->ia_valid & ATTR_MODE
  ...
}
```

```
// @v4.0-rc2/fs/ext3/inode.c:3241
int ext3_setattr(struct dentry *dentry, struct iattr *attr) {
  // sanity check
  error = inode_change_ok(dentry->inode, attr);
  if (error)
    return error;

  // update ACL entries.
  const unsigned int ia_valid = attr->ia_valid;
  if (ia_valid & ATTR_MODE)
    rc = posix_acl_chmod(inode, inode->i_mode);
  ...
}
```

Figure 5: JUXTA's latent specification for `setattr()`, and simplified `ext3_setattr()` code. The latent specification captures the common function call (`inode_change_ok()`) and the common flag check (`ia_valid & ATTR_MODE`) with respect to return value ranges.

latent specifications is similar to finding deviant behaviors, but its focus is more on finding common behaviors. We report side-effects, function calls, or path conditions if any one of these is commonly exhibited in most file systems.

For example, Figure 5 illustrates a latent specification of `setattr()`. According to JUXTA, `setattr()` should perform an `inode_change_ok()` check and handle its error. Although `error` is checked if it is zero, JUXTA captures that only a negative return value can be used to indicate an error. Also, 10 file systems commonly invoke `posix_acl_chmod()` if `attr` has a `ATTR_MODE` flag. One should consider following this pattern when implementing the `setattr()` interface.

5.3 Refactoring Cross-module Abstraction

The most common type of bug fixes in file systems is the maintenance patch (45%) [42]. In particular, since the VFS and file systems have been co-evolving, finding commonalities in file systems and promoting them to the VFS layer are critical to improving the maintainability of file systems. In this respect, the latent specification described in §5.2 can offer a unique refactoring opportunity because it can be used to identify redundant implementations of all VFS functions and it gives new insights beyond the cross-module boundary. More importantly, the identified code snippet can be refactored to the upper VFS layer so that each file system can benefit from it without redundantly handling the common case. For example, as shown in Figure 5, the permission checks using `inode_change_ok()` can be promoted to the VFS layer; `MS_RDONLY` (see §2.3) can also be enforced commonly at the VFS layer; `page_unlock()` and `page_cache_release()` can be uniformly handled at the VFS layer (see §2.2).

5.4 Inferring Lock Semantics

Given per-path conditions and side-effects, the lock checker emulates current locking states (e.g., which locks are held

```
// @v4.0-rc2/gfs2/glock.c:2050
int gfs2_create_debugfs_file(struct gfs2_sbd *sdp) {
    sdp->debugfs_dir = debugfs_create_dir(...);
    if (!sdp->debugfs_dir)
        return -ENOMEM;
}
```

```
// @v4.0-rc2/ubifs/debug.c:2384
int dbg_debugfs_init_fs(...) {
    dent = debugfs_create_dir(...);
    if (IS_ERR_OR_NULL(dent))
        goto out;
    ...
out:
    err = dent ? PTR_ERR(dent) : -ENODEV;
    return err;
}
```

Figure 6: GFS2 checks if the return value of `debugfs_create_dir()` is `NULL`, but UBIFS checks if its return value (pointer) is `NULL` or an error code. It turns out that `debugfs_create_dir()` can return `-ENODEV` if `DEBUG_FS` is not configured. So the caller should check for both error codes, or a system crash may result. JUXTA's error handling checker found 7 such bugs.

or released so far) and at the same time keeps track of which fields are always accessed or updated while holding a lock (e.g., `inode.i_lock` should be held when updating `inode.i_size`). Our lock checkers have two distinctive features that use path-sensitive symbolic constraints. One is a context-based promotion that promotes a function as a lock equivalent if *all* of its possible paths return while holding a lock. Our checker can help users in prioritizing the report based on context information (e.g., patterns of other VFS implementations). Based on lock orders and information from `linux/fs.h` and documentation, we could cross-validate the inferred lock semantics[3].

5.5 Inferring Semantics of External APIs

The semantics of external API invocations include two parts: the argument(s) used to invoke the API and the checks for the return value.

The incorrect use of arguments can result in serious consequences such as deadlock. For example, file systems should not use the `GFP_KERNEL` flag to allocate memory in their IO-related code because the kernel memory allocator will recursively call the file system code to write back dirty pages. To avoid such deadlock cases, `GFP_NOFS` should be used instead. In fact, this is the most common bug pattern among the concurrency bugs in file systems [42].

Handling return values is important because kernel functions have different behaviors for different return values. For example, for functions returning a pointer, some of them can return `NULL` on error, some encode an error code as the returned pointer, or in some cases, do both. If the caller misses a check, dereferencing the incorrect pointer may crash the kernel (Figure 6).

For the same VFS interface, argument usage and handling return values should be very similar because the same interface has high-level semantics. Based on this intuition, we developed two checkers that can identify such bugs without information about their complex semantics.

[3] `linux/Documentation/filesystem/locks.txt`

Component	Lines of code
Symbolic path explorer	6,180 lines of C/C++
Source code merge	1,025 lines of Python
Checkers	2,805 lines of Python
Spec. generator	628 lines of Python
JUXTA library	1,708 lines of Python
Total	12,346 lines of code

Table 4: Components of JUXTA and an estimate of their complexities in terms of lines of code.

Argument checker. The argument checker is similar to the function call checker but is specialized to understand the semantics of different parameters. Given the execution paths of the same VFS call returning a matching value, it collects invocations of external APIs and the arguments passed to the API. It then calculates entropy values based on the frequency of flags (e.g., `GFP_KERNEL` vs. `GFP_NOFS`). If the entropy value is small, this means there are only few deviations in the usage of flags and such deviations are likely to be bugs.

Error handling checker. The error handling checker is similar to the path condition checker, but it checks all file system functions besides entry functions. To identify incorrect handling of return values, including missing checks, the checker first collects the conditions for each API along all execution paths. It then calculates an entropy value for each API based on the frequency of check conditions (e.g., `ret != 0` vs `IS_ERR_OR_NULL(ret)` in Figure 6). Like the argument checker, if the entropy value is small, then there are only a few deviations in error handling and the checker reports such deviations as bugs.

6. Implementation

JUXTA is 12K LoC in total: 6K lines of C/C++ for its symbolic path explorer and 6K lines of Python for checkers, code merge, and their libraries (Table 4). We have built our symbolic path explorer by modifying Clang 3.6, a VFS entry database for Linux kernel 4.0-rc2, and eight different checkers by leveraging JUXTA's libraries.

7. Evaluation

Our evaluation answers the following questions:

- How effective is JUXTA's approach in discovering new semantic bugs in file systems in Linux? (§7.1)
- What kinds of semantic bugs can JUXTA detect in the file systems (known semantic bugs)? (§7.2)
- What are the plausible reasons for JUXTA to generate false error reports (false positives)? (§7.3)

Experimental setup. We applied JUXTA to Linux kernel 4.0-rc2, the latest version at the time of this writing, and ran all checkers for 54 file systems in stock Linux. All experiments

FS	Module	Operation	Error	Impact	#bugs	Y	S
9P	vfs_file.c	file and directory `fsync()`	`[S]` missing `MS_RDONLY` check	consistency	2	4y	P
ADFS	dir.c	file and directory `fsync()`	`[S]` missing `MS_RDONLY` check	consistency	2	>10y	P
	dir_fplus.c	data read	`[E]` incorrect return value	application	1	>10y	✓
	super.c	super operation	`[E]` incorrect return value	application	5	>10y	✓
AFS	file.c	file and directory `fsync()`	`[S]` missing `MS_RDONLY` check	consistency	1	8y	P
	super.c	mount option parsing	`[E]` missing `kstrdup()` return check	system crash	1	8y	P
AFFS	file.c	page I/O	`[C]` missing `unlock()/page_cache_release()`	deadlock	2	>10y	✓
	file.c	file and directory `fsync()`	`[S]` missing `MS_RDONLY` check	consistency	1	6y	P
	super.c	mount option parsing	`[E]` missing `kstrdup()` return check	system crash	1	10y	✓
BFS	dir.c	data read	`[E]` incorrect return value	application	2	>10y	✓
	dir.c	file and directory `fsync()`	`[S]` missing `MS_RDONLY` check	consistency	1	7y	P
blockdev	block_dev.c	file and directory `fsync()`	`[S]` missing `MS_RDONLY` check	consistency	1	6y	P
btrfs	extent_io.c	`fiemap_next_extent()`	`[E]` incorrect error handling	application	1	4y	✓
	file.c	file and directory `fsync()`	`[S]` missing `MS_RDONLY` check	consistency	1	8y	P
Ceph	caps.c	file and directory `fsync()`	`[S]` missing `MS_RDONLY` check	consistency	2	6y	P
	addr.c	page I/O	`[S]` missing `page_cache_release()`	DoS	1	5y	✓
	dir.c	`readdir()`, `symlink()`	`[E]` missing `kstrdup()` return check	system crash	2	6y	✓
	super.c	mount option parsing	`[E]` missing `kstrdup()` return check	system crash	2	6y	✓
	xattr.c	`set_xattr()`, `remove_xattr()`	`[E]` missing `kstrdup()` return check	system crash	2	6y	✓
CIFS	connect.c	mount option parsing	`[M]` missing `kfree()`	DoS	3	6y	✓
	file.c	waiting for posix lock file	`[E]` missing check	consistency	2	3y	✓
	file.c	file and directory `fsync()`	`[S]` missing `MS_RDONLY` check	consistency	2	4y	P
Coda	file.c	file and directory `fsync()`	`[S]` missing `MS_RDONLY` check	consistency	1	>10y	P
EcryptFS	file.c	file and directory `fsync()`	`[S]` missing `MS_RDONLY` check	consistency	1	4y	P
EXOFS	file.c	file and directory `fsync()`	`[S]` missing `MS_RDONLY` check	consistency	1	4y	P
ext2	file.c	file and directory `fsync()`	`[S]` missing `MS_RDONLY` check	consistency	1	6y	P
ext4	super.c	mount option parsing	`[E]` missing `kstrdup()` return check	system crash	2	5y	✓
FUSE	file.c	file and directory `fsync()`	`[S]` missing `MS_RDONLY` check	consistency	1	10y	P
GFS2	file.c	file and directory `fsync()`	`[S]` missing `MS_RDONLY` check	consistency	1	4y	P
	glock.c	debugfs file and dir creation	`[E]` incorrect error handling	system crash	5	8y	✓
HFS	dir.c	file / dir creation	`[E]` incorrect return value	application	2	>10y	✓
	inode.c	file and directory `fsync()`	`[S]` missing `MS_RDONLY` check	consistency	1	5y	P
HFSplus	dir.c	symlink and mknod creation	`[E]` incorrect return value	application	2	5y	✓
	inode.c	metadata inode sync	`[E]` missing error check	system crash	2	>10y	P
	inode.c	file and directory `fsync()`	`[S]` missing `MS_RDONLY` check	consistency	1	5y	P
HPFS	file.c	file and directory `fsync()`	`[S]` missing `MS_RDONLY` check	consistency	1	4y	P
	namei.c	rename	`[S]` missing update of `ctime` and `mtime`	application	4	>10y	✓
	super.c	mount option parsing	`[E]` missing `kstrdup()` return check	system crash	1	7y	✓
JBD2 †	transaction.c	journal transaction	`[C]` try to unlock an unheld spinlock	deadlock, consistency	2	9y	✓
JFFS2	file.c	file and directory `fsync()`	`[S]` missing `MS_RDONLY` check	consistency	1	>10y	P
JFS	file.c	file and directory `fsync()`	`[S]` missing `MS_RDONLY` check	consistency	1	>10y	P
LogFS	segment.c	`read_page_cache()`	`[E]` incorrect error handling	system crash	2	5y	P
	super.c	`read_page_cache()`	`[E]` incorrect error handling	system crash	2	6y	P
MINIX	dir.c	file and directory `fsync()`	`[S]` missing `MS_RDONLY` check	consistency	2	5y	P
NCP	file.c	file and directory `fsync()`	`[S]` missing `MS_RDONLY` check	consistency	1	>10y	P
NFS	dir.c / file.c	file and directory `fsync()`	`[S]` missing `MS_RDONLY` check	consistency	2	5y	P
	nfs4client.c	update NFS server	`[E]` missing `kstrdup()` return check	system crash	1	2y	✓
	nfs4proc.c	client ID hanlding	`[E]` missing `kstrdup()` return check	system crash	5	1y	✓
NFSD	fault_inject.c	debugfs file and dir creation	`[E]` incorrect error handling	system crash	2	4y	✓
NILFS2	file.c	file and directory `fsync()`	`[S]` missing `MS_RDONLY` check	consistency	1	6y	P
NTFS	dir.c / file.c	file and directory `fsync()`	`[S]` missing `MS_RDONLY` check	consistency	2	4y	P
OCFS2	xattr.c	get xattr list in trusted domain	`[S]` missing `CAP_SYS_ADMIN` check	security	1	6y	✓
OMFS	file.c	file and directory `fsync()`	`[S]` missing `MS_RDONLY` check	consistency	1	5y	P
QNX4	dir.c	file and directory `fsync()`	`[S]` missing `MS_RDONLY` check	consistency	1	5y	P
QNX6	file.c	file and directory `fsync()`	`[S]` missing `MS_RDONLY` check	consistency	1	3y	P
ReiserFS	dir.c / file.c	file and directory `fsync()`	`[S]` missing `MS_RDONLY` check	consistency	2	5y	P
	super.c	mount option parsing	`[E]` missing `kstrdup()` return check	system crash	1	7y	P
SquashFS	symlink.c	reading symlink information	`[E]` incorrect return value	application	2	6y	P
UBIFS	dir.c	`create/mkdir/mknod/symlink()`	`[C]` incorrect `mutex_unlock()` and `i_size` update	deadlock, application	4	<1y	✓
	file.c	page I/O	`[E]` missing `kmalloc()` return check	system crash	1	7y	P
UDF	file.c	page I/O	`[S]` missing `mark_inode_dirty()`	consistency	1	1y	P
	file.c	file and directory `fsync()`	`[S]` missing `MS_RDONLY` check	consistency	2	5y	✓
	inode.c	page I/O	`[E]` incorrect return value	application	1	8y	✓
	namei.c	`symlink()` operation	`[E]` missing return value	system crash	1	8y	✓
	namei.c	rename	`[S]` missing update of `ctime` and `mtime`	application	2	>10 y	✓
UFS	file.c	file and directory `fsync()`	`[S]` missing `MS_RDONLY` check	consistency	2	5y	P
	inode.c	update inode	`[E]` incorrect return value	application	2	8y	P
XFS	xfs_acl.c	ACL handling	`[C]` incorrect `kmalloc()` flag in `I/O` context	deadlock	3	7y	✓
	xfs_file.c	file and directory `fsync()`	`[S]` missing `MS_RDONLY` check	consistency	2	4y	P
	xfs_mru_cache.c	disk block allocation	`[C]` incorrect `kmalloc()` flag in `I/O` context	deadlock	1	8y	✓

Table 5: List of new bugs discovered by JUXTA in Linux kernel 4.0-rc2. The total number of bugs reported is 118, of which 65 have been confirmed. P in the rightmost column represents submission of a patch; a ✓represents the patch has been either reviewed or applied. Out of 54 file systems, JUXTA found bugs in 39 file systems and found one bug per 5.8K LoC. The reported semantic bugs are difficult to find, as their latent period is over 6.2 years on average.

Bug type	Cause	Detected / Total
[S] State	incorrect state update	*7 / 8
	incorrect state check	†5 / 6
[C] Concurrency	miss unlock	1 / 1
	incorrect `kmalloc()` flag	1 / 1
[M] Memory	leak on exit/failure	2 / 2
[E] Error code	miss memory error	1 / 1
	incorrect error code	2 / 2

Table 6: The completeness of JUXTA's results. We synthetically introduced 21 known bugs in Linux's file system implementations from PatchDB [42] and applied JUXTA to see if it could detect the classes of bugs. In total, JUXTA successfully detected 19 bugs out of 21 and missed two cases due to its limitations (see §7.3).

were conducted on a 64-bit Ubuntu 14.04 machine with 8×10-core Xeon E7-8870 2.4GHz and 512 GB RAM.

7.1 New Bugs

Our checkers reported 2,382 bugs total for 54 file systems in Linux. Two authors, who had no prior experience with the Linux file system code, inspected bug reports from March 19 to March 24 in 2015. They reviewed the 710 top-ranked bug reports given time constraints and found 118 new bugs in 39 different file systems (see Table 7 for details). They also made and submitted bug patches during that period; a few critical ones were quickly applied in a testing branch or in the mainline Linux. JUXTA found one bug in every 5.8K LoC. The average latent period of these bugs is over 6.2 years. This shows that JUXTA is effective in discovering long-existing semantic bugs. Some of these bugs are critical and hard-to-find; for example, though four incorrect `kmalloc()` flag bugs in XFS can cause a deadlock, they have existed for 7–8 years.

Incorrect return value. Our return code checker found that implementations of `super_operations.write_inode()` in ufs and `inode_operations.create()` in BFS have bugs not returning `-EIO`. Instead, UFS and BFS return the wrong error code, `-ENOSPC` and `-EPERM`, respectively. As previous studies [32, 42, 47] show, incorrect handling of error codes is quite common in file systems (about 10% of bugs).

Missing updates. Our side-effects checker found missing updates in HPFS. In `rename()`, HPFS does not update the `mtime` and `ctime` of new and old directories. In file systems, incorrect updates may lead to data inconsistency or malfunctioning applications. Also, incorrect state updates are the most common bugs among semantic bugs (about 40%) [42], but they are the hardest ones to find using existing static checkers without deep semantic knowledge.

Missing condition check. Our function call checker and path condition checker found a missing condition check in OCFS2. Its implementation of `xattr_handler.list()` for the trusted namespace does not have a capability check. This function is used for accessing extended attributes in the trusted namespace, which are visible and accessible only to processes that have `CAP_SYS_ADMIN` (i.e., usually super user). The missing capability checks can lead to security vulnerabilities or data corruption. Every file system operation needs proper condition checks to filter out access without capability or given improper user data (e.g., path name).

Lock bugs. JUXTA found eight lock bugs in AFFS, Ceph, ext4/JBD2, and UBIFS, all of which are fixed in the upstream Linux repository. Unlike traditional lock checkers, including the built-in Sparse, JUXTA can reason about correct lock states from other file system implementations: all paths of `write_end()` commonly unlock and release a page in most file systems, except two paths in AFFS. A similar bug was found in `write_begin()` of Ceph. Note that two unlock bugs in ext4/JBD2 were typical; two unlocks were not paired correctly due to its complex `if/else` structure. Detecting lock bugs is not only extremely difficult (requiring deep understanding of context), but the consequences of these bugs are critical: any lock bug can lead to the deadlock of the entire file system and result in loss of cached data.

Inconsistent use of external APIs. Static analyzers typically lack execution context and domain-specific knowledge unless a special model is given. After comparing usage patterns of external APIs in file system code, our checker reported a deviant usage of `kmalloc()`: regarding a `vfs_mount()` implementation, other file systems commonly use `kmalloc()` with `GFP_NOFS`, but XFS uses `kmalloc()` with `GFP_KERNEL`. Our checker, without complex semantic correctness knowledge,[4] can *statically* identify such errors.

Similarly, JUXTA found 50 misuses of external APIs (e.g., memory allocation like `kstrdup()`, incorrect error handling of the `debugfs` interface, etc.) in AFS, AFFS, Ceph, DLM, ext4, GFS2, HPFS, HFSplus, NFS, OCFS2, UBIFS, and XFS, which all result in dereferencing invalid pointers and thus potentially cause system crashes.

7.2 Completeness

To evaluate JUXTA's completeness, we collected 21 known file system semantic bugs from PatchDB [42]. We synthesized these bugs into the Linux Kernel 4.0-rc2 to see if JUXTA could identify them. Table 6 shows the result of JUXTA execution, as well as bug categorizations. JUXTA was able to identify 19 out of 21 bugs. JUXTA missed one bug due to the complex structure of a buggy function (marked $*$) that our symbolic executor failed to explore. Another bug was introduced too deeply from the entry point function, and the error condition was not visible with our statistical comparison schemes (marked †).

7.3 False Positives

As explained in §4.5, JUXTA ranks bug reports by quantitative code deviation metrics. Table 7 shows the number of

[4] The `GFP_KERNEL` flag is okay in most cases but it should not be used in the I/O related code of file systems. With the `GFP_KERNEL` flag, the VM memory allocator `kswapd` will call file system code to write dirty pages for free memory so it results in a deadlock. Our fix in XFS changes the flag to `GFP_NOFS` to prevent the recursive call of file system code.

Checker	# reports	# verified	New bugs	# rejected
Return code checker	573	150	2	0
Side-effect checker	389	150	6	0
Function call checker	521	100	5	0
Path condition checker	470	150	46	2
Argument checker	56	10	4	0
Error handling checker	242	100	47	21
Lock checker	131	50	8	1
Total	2,382	710	118	24

Table 7: Bugs reported by each JUXTA checker and bugs manually verified. For bugs with high confidence after manual verification, we provided patches to corresponding developer communities. Some are confirmed as true bugs and some are revealed as non-bugs.

generated bug reports and statistics for each checker. We carefully examined the top-ranked 710 bug reports of the total 2,382 found by JUXTA. Of these 710, we identified 142 as potential bugs and the remaining 568 as false positives. We submitted bug reports for the 142 potential bugs to the respective file systems' developers, who confirmed that 118 of these were bona fide bugs, while 24 were not—i.e., false positives. Thus, of the 710 bug reports we examined carefully, 118 were bona fide bugs and the aggregated 592 were false positives, for an overall false positive rate of 83%.[5] As noted in §4.5, because JUXTA ranks the bugs it reports, the programmer can prioritize investigating the highest-ranked bug reports, which are most likely to be true positives. Figure 7 illustrates that there are many true positives among the bug reports that JUXTA ranks highly.

To further understand what causes JUXTA to generate false positive reports, we first classified bug reports and bug statistics by each checker and then further classified the false bug reports.

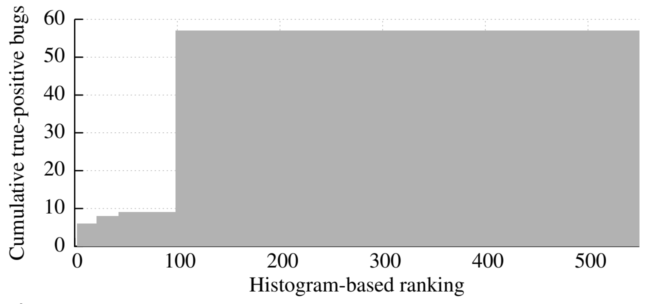
7.3.1 Classification by Checkers

Since each checker is designed to find different types of bugs, it is difficult to directly compare the effectiveness of each checker by simply comparing the number of new bugs found. In this regard, we instead focus on the negative aspects of JUXTA here, i.e., the rejected bugs. All of these rejected bugs are file-system specific so JUXTA (and even the file system maintainers) considered them as bugs.

The path condition checker found two incorrect condition checks in `xattr`-related code in F2FS. But they are actually correct since they are used only for a particular `xattr` that is only inherent to F2FS; its usage pattern was contrary to the normal convention.

The error handling checker found 21 incorrect error handling codes in OCFS2 and debugfs. They were once confirmed, but after further investigation they were reverted since it is not possible to build OCFS2 without debugfs, although most file systems check such combinations to support kernel builds without debugfs.

[5] We do not yet know the false-positive rate for the remaining lower-ranked 1,672 bugs, as we have not yet investigated the ground truth for these bugs.



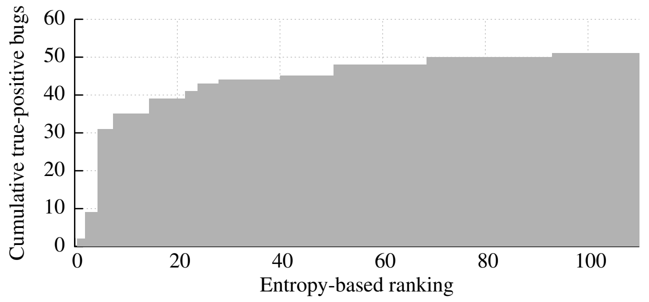


Figure 7: Cumulative true-positive bugs, sorted by ranking reported by JUXTA. For histogram-based checkers, bug reports are ranked in descending order, while for entropy-based checkers, bug reports (except for ones with zero entropy) are ranked in ascending order. JUXTA's statistical path comparison and bug report ranking can effectively save programmers effort by letting them investigate the highest-ranked reports first.

Finally, our lock checker found an incorrect implementation of a `write_end()` of `address_space` operation in UDF, which does not unlock the page cache. But it is actually correct because the code is for a special case, in which data is inlined to an `inode` structure and thus data does not have a corresponding page.

7.3.2 Classification by Types

Different file system types. Each file system supports a different set of features: SquashFS is designed as a read-only file system; ramfs uses main memory without backing storage; Ceph is designed as a distributed file system. JUXTA reports that Ceph has no capability check for file system operations. But since Ceph relies on servers for the capability check (i.e., it does not trust its clients), JUXTA's bug reports are false alarms. If JUXTA compares file systems only of the same type, our approach might incur fewer false positives.

Different implementation decision. Even in the same type of file system, different implementation decisions could cause false alarms in JUXTA. For example, btrfs is the only file system returning `-EOVERFLOW` at `mkdir()`. It occurs when there is no room in a lead node of the tree structure in btrfs. Since the POSIX specification on `mkdir()` [3] does not have the case of returning `-EOVERFLOW`, there is no matching error code for this case.

Redundant codes. For `symlink()`, JUXTA reports that F2FS does not check the length of the symbolic link, but many other file systems, including ext2 and NILFS2, have the

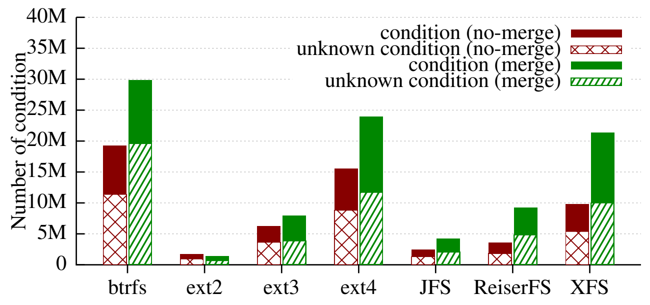


Figure 8: When JUXTA performs an inter-procedure analysis within a file system module, the symbolic path conditions contain two times more concrete expressions.

length check. However, surprisingly, since the VFS function `symlinkat()` already checks the length of a symbolic link before calling `symlink()`, the length check in ext2 and NILFS2 is redundant. Considering that VFS and concrete file systems are co-evolving, such redundant code could exist. Also, it shows that JUXTA can be extended to a refactoring suggestion tool by comparing file system behavior across layers (e.g., VFS, file system, and block layer).

Incorrect symbolic execution results. Besides the above true deviant cases, most false positives are caused by the inaccurate symbolic expressions or contexts. There are two major sources of inaccuracy: JUXTA unrolls a loop only once so analysis results related to loops could be inaccurate; JUXTA limits the depth of inlining to prevent path explosion and to terminate the analysis within a reasonable time. As in Figure 8, around 50% of path conditions are *unknown* due to the uninlined function calls, although JUXTA improves the precision of symbolic expressions with its code merge (50% more concrete expressions).

7.4 Performance

For our experiments, we use an 80-core Xeon server with 512 GB RAM. When analyzing 54 Linux file systems composed of roughly 680K lines of pure code, it takes roughly five hours: 30 minutes for merging (e.g., resolving conflicts) each file system's code into one large file for precise analysis, another 30 minutes for path exploration, which generates 300 GB of intermediate results. JUXTA requires two hours to preprocess and create the database (another 300 GB) for path conditions and side-effects that all other checkers rely on (a one-time cost). Then, all checkers take roughly two hours total to scan all possible paths in the database. Since the generated database consists of checker-neutral data, we will make it publicly available. This will allow other programmers to easily develop their own checkers. Also, this result shows that JUXTA can scale to even larger system code within a reasonable time budget.

8. Discussion

As a static analyzer, JUXTA is neither sound nor complete: it reports bugs incorrectly (§7.3) and misses some bugs (§7.2). However, the reports Juxta generates are useful, as JUXTA found one real bug per 5.8K LoC in mature Linux file systems.

JUXTA's approach in general is not well-suited for discovering bugs that are specific to a file system's design, logic or implementation. However, we found that our external API checker helps a lot in finding such bugs without file system specific knowledge (e.g., incorrectly handling the error codes of `debugfs` APIs). We believe that JUXTA's approach can also be applied in detecting implementation-specific bugs (across the entire Linux code base) and perhaps by abstracting and comparing the way each file system interacts with its neighboring or underlying layers (e.g., block devices).

JUXTA's approach can be considered a general mechanism to explore two different semantically equivalent implementations without special or domain-specific knowledge of their internal model. Therefore, JUXTA has huge potential for other application domains that have multiple implementations of common features. All modern Web browsers, for example, implement (or support) the W3C DOM/HTML [57] and ECMAScript specifications [27]. Using JUXTA's approach, their compatibility can be systematically examined and summarized in terms of standard interfaces.

We believe JUXTA is promising for code bases with multiple implementations that comply with similar specifications, such as standard POSIX libraries, TCP/IP network stacks, and UNIX utilities (busybox, coreutils, or Cygwin). JUXTA's approach is also a good fit for self-regression testing (in the spirit of Poirot [37]), in which one could treat multiple previous versions and a new version as semantically equivalent implementations.

9. Related work

Two large bodies of previous research motivate the development of JUXTA: one is bug studies in systems code, OSes [15, 17, 41, 45, 67], file systems [42, 68], and distributed systems [69]; the other is a body of work that changed our perspective on bugs—for example, viewing them as deviant behavior [24, 29] or as a different behavior among multiple implementations [6, 25]. While previous research focuses on shallow bugs [24, 29, 32, 68], relies on manual inspection [6, 25], or requires special models to find bugs [9, 64, 66, 68], JUXTA statistically compares multiple implementations having the same semantics (i.e., the same VFS interface) to find deep semantic bugs.

Broadly, previous research on validating or enforcing system rules (or software invariants) can be classified into three approaches: model checking, formal proof, and automatic testing.

Model checking. Meta Compilation [5, 28] proposed using system-specific compiler extensions to check domain-specific rules. JUXTA incorporates its workflow but adopts path-sensitive and context-aware analysis as proposed by ESP [21]. Model checking has been explored in the context of file systems by FiSC [66], by eXplode [64] with a better reduction technique [33], and by EIO [32, 47] for error propagation. SDV [7] performs model checking on device drivers and CMC [44] checks the correctness of popular protocols. Since Juxta statistically infers a latent model by comparing various implementations, it naturally incurs a higher false positive rate than typical model checks, but Juxta's approach is general (requiring no specific models), and thus has huge potential for other application domains that have multiple implementations.

Specification and formal proof. Formal methods have been applied to system software such as compilers [40], browsers [35], file systems [16], device drivers [49], operating system components such as BPF [61], and the entire operating system [38] in order to provide strong guarantees of their high-level invariants (e.g., security properties or functional correctness). The key emphasis of Juxta is that, unlike these solutions that require programmers to provide formal models or specifications, Juxta derives such models directly from existing implementations and then automatically identifies deviant behaviors from the extracted model.

Symbolic execution. One promising way to explore all possible paths of a program is to use symbolic execution [9, 12–14, 60, 62], which Juxta employs at its core. KLEE [13] is a pioneer in this space and proposes an idea of validating the compatibility of multiple software implementations such as busybox and coreutils. However, symbolic execution cannot be directly applied to compare multiple file systems, as they are all completely different from each other but behave similarly. Juxta's goal is to identify such *latent* rules that all file systems implement in their own manner. Woodpecker [20] and EXE [14, 65] used symbolic execution with given system rules to validate file system implementations, but they largely focus on finding shallow bugs (e.g., memory leaks), unlike Juxta, which focuses on semantic bugs.

Inferring models. Unlike the three aforementioned techniques that require manual models or specific rules in validating code, a few projects focus on inferring such rules, with minimum user involvement or in a fully automated way: finite state machines [19], copy-and-pasted code snippets [41], locking patterns (at runtime) [50], inconsistent code patterns [29, 63], network protocols [9], `setuid` implementations (manually) [25], and error handling [24, 32]. However, Juxta infers its model by comparing multiple existing implementations that obey implicit rules.

10. Conclusion

In this paper, we propose Juxta, a static analysis tool that extracts high-level semantics by comparing and contrasting multiple implementations of semantically equivalent software. We applied Juxta to 54 different file systems in Linux and found 118 previously unknown semantic bugs in 39 different file systems (one bug per 5.8K), of which 30 bugs have existed for more than 10 years. Not only do our empirical results look promising, but the design of Juxta is general enough to easily extend to any software that has multiple implementations.

11. Acknowledgment

We would like to thank our shepherd, Brad Karp, and the anonymous reviewers for their helpful feedback, which substantially improved the content and presentation of this paper. We also would like to acknowledge the proofreading efforts of our operations staff. This research was supported by the NSF award DGE-1500084, the ONR under grant N000141512162, the DARPA Transparent Computing program under contract No. DARPA-15-15-TC-FP-006, and the ETRI MSIP/IITP[B0101-15-0644].

References

[1] Skipped files with `-listed-incremental` after rename, 2003. http://osdir.com/ml/gnu.tar.bugs/2003-10/msg00013.html.

[2] Fix update of `mtime` and `ctime` on rename, 2008. http://linux-ext4.vger.kernel.narkive.com/Cc13bI74/patch-ext3-fix-update-of-mtime-and-ctime-on-rename.

[3] `mkdir()`, 2013. The IEEE and The Open Group, The Open Group Base Specifications Issue 7, IEEE Std 1003.1, 2013 Edition http://pubs.opengroup.org/onlinepubs/9699919799/functions/mkdir.html.

[4] Checker developer manual, 2015. http://clang-analyzer.llvm.org/checker_dev_manual.html#idea.

[5] Ashcraft, K., and Engler, D. Using programmer-written compiler extensions to catch security holes. In *Proceedings of the 23rd IEEE Symposium on Security and Privacy (Oakland)* (Oakland, CA, May 2002), pp. 143–160.

[6] Avizienis, A. The N-Version approach to fault-tolerant software. *IEEE Transactions of Software Engineering 11*, 12 (Dec. 1985), 1491–1501.

[7] Ball, T., Bounimova, E., Cook, B., Levin, V., Lichtenberg, J., McGarvey, C., Ondrusek, B., Rajamani, S. K., and Ustuner, A. Thorough static analysis of device drivers. In *Proceedings of the ACM EuroSys Conference* (Leuven, Belgium, Apr. 2006), pp. 73–85.

[8] Bessey, A., Block, K., Chelf, B., Chou, A., Fulton, B., Hallem, S., Henri-Gros, C., Kamsky, A., McPeak, S., and Engler, D. A few billion lines of code later: Using static analysis to find bugs in the real world. *Communications of the ACM 53*, 2 (Feb. 2010), 66–75.

[9] Brumley, D., Caballero, J., Liang, Z., Newsome, J., and Song, D. Towards automatic discovery of deviations in binary implementations with applications to error detection and fingerprint generation. In *Proceedings of the 16th Usenix Security Symposium (Security)* (Boston, MA, Aug. 2007), pp. 15:1–15:16.

[10] Brumley, D., Wang, H., Jha, S., and Song, D. Creating vulnerability signatures using weakest preconditions. In *Proceedings of the 20th IEEE Computer Security Foundations*

Symposium (Washington, DC, USA, 2007), CSF '07, IEEE Computer Society, pp. 311–325.

[11] Brummayer, R., and Biere, A. Boolector: An efficient SMT solver for bit-vectors and arrays. In *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2009, pp. 174–177.

[12] Bucur, S., Ureche, V., Zamfir, C., and Candea, G. Parallel symbolic execution for automated real-world software testing. In *Proceedings of the ACM EuroSys Conference* (Salzburg, Austria, Apr. 2011), pp. 183–198.

[13] Cadar, C., Dunbar, D., and Engler, D. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation (OSDI)* (San Diego, CA, Dec. 2008), pp. 209–224.

[14] Cadar, C., Ganesh, V., Pawlowski, P. M., Dill, D. L., and Engler, D. R. EXE: Automatically generating inputs of death. In *Proceedings of the 13th ACM Conference on Computer and Communications Security* (Alexandria, VA, Oct.–Nov. 2006), pp. 322–335.

[15] Chen, H., Cutler, C., Kim, T., Mao, Y., Wang, X., Zeldovich, N., and Kaashoek, M. F. Security bugs in embedded interpreters. In *Proceedings of the 4th Asia-Pacific Workshop on Systems (APSys)* (2013), pp. 17:1–17:7.

[16] Chen, H., Ziegler, D., Chlipala, A., Kaashoek, M. F., Kohler, E., and Zeldovich, N. Towards certified storage systems. In *Proceedings of the 15th Workshop on Hot Topics in Operating Systems (HotOS)* (May 2015).

[17] Chou, A., Yang, J., Chelf, B., Hallem, S., and Engler, D. An empirical study of operating systems errors. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP)* (Chateau Lake Louise, Banff, Canada, Oct. 2001), pp. 73–88.

[18] Corbet, J., Kroah-Hartman, G., and McPherson, A. Linux Kernel Development: How Fast is it Going, Who is Doing It, What Are They Doing and Who is Sponsoring the Work, 2015. http://www.linuxfoundation.org/publications/linux-foundation/who-writes-linux-2015.

[19] Corbett, J. C., Dwyer, M. B., Hatcliff, J., Laubach, S., Pasareanu, C. S., and Zheng, H. Bandera: Extracting finite-state models from Java source code. In *Proceedings of the 22nd international conference on software engineering (ICSE)* (2000), pp. 439–448.

[20] Cui, H., Hu, G., Wu, J., and Yang, J. Verifying systems rules using rule-directed symbolic execution. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (Houston, TX, Mar. 2013), pp. 329–342.

[21] Das, M., Lerner, S., and Seigle, M. ESP: Path-sensitive program verification in polynomial time. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation* (Berlin, Germany, June 2002), pp. 57–68.

[22] De Moura, L., and Bjørner, N. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 337–340.

[23] Dijkstra, E. W. *A discipline of programming*, vol. 1. Prentice-Hall Englewood Cliffs, 1976.

[24] Dillig, I., Dillig, T., and Aiken, A. Static error detection using semantic inconsistency inference. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Diego, CA, June 2007), pp. 435–445.

[25] Dittmer, M. S., and Tripunitara, M. V. The UNIX process identity crisis: A standards-driven approach to setuid. In *Proceedings of the 21st ACM Conference on Computer and Communications Security* (Scottsdale, Arizona, Nov. 2014), pp. 1391–1402.

[26] Duda, R. O., Hart, P. E., and Stork, D. G. *Pattern classification*. John Wiley & Sons, 2012.

[27] ECMA International. ECMAScript Language Specification, June 2011. http://www.ecmascript.org/docs.php.

[28] Engler, D., Chelf, B., Chou, A., and Hallem, S. Checking system rules using system-specific, programmer-written compiler extensions. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation (OSDI)* (San Diego, CA, Oct. 2000).

[29] Engler, D., Chen, D. Y., Hallem, S., Chou, A., and Chelf, B. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP)* (Chateau Lake Louise, Banff, Canada, Oct. 2001), pp. 57–72.

[30] 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 (FAST)* (2012).

[31] Gooch, R. Overview of the linux virtual file system, 2007. https://www.kernel.org/doc/Documentation/filesystems/vfs.txt.

[32] Gunawi, H. S., Rubio-González, C., Arpaci-Dusseau, A. C., Arpaci-Dussea, R. H., and Liblit, B. EIO: Error handling is occasionally correct. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST)* (2008), pp. 14:1–14:16.

[33] Guo, H., Wu, M., Zhou, L., Hu, G., Yang, J., and Zhang, L. Practical software model checking via dynamic interface reduction. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)* (Cascais, Portugal, Oct. 2011), pp. 265–278.

[34] Hunt, G. C., and Larus, J. R. Singularity: Rethinking the Software Stack. *ACM SIGOPS Operating Systems Review 41*, 2 (April 2007), 37–49.

[35] Jang, D., Tatlock, Z., and Lerner, S. Establishing browser security guarantees through formal shim verification. In *Proceedings of the 21st Usenix Security Symposium (Security)* (Bellevue, WA, Aug. 2012), pp. 113–128.

[36] Kara, J. fs/udf/namei.c at v4.1, 2015. https://github.com/torvalds/linux/commit/3adc12e9648291149a1e3f354d0ad158fc2571e7.

[37] Kim, T., Chandra, R., and Zeldovich, N. Efficient patch-based auditing for web application vulnerabilities. In *Proceedings of the 10th Symposium on Operating Systems Design and Implementation (OSDI)* (Hollywood, CA, Oct. 2012).

[38] Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., and Winwood, S. seL4: Formal verification of an os kernel. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)* (Big Sky, MT, Oct. 2009), pp. 207–220.

[39] Kullback, S., and Leibler, R. A. On information and sufficiency. *The annals of mathematical statistics* (1951), 79–86.

[40] Leroy, X. Formal certification of a compiler back-end or: Programming a compiler with a proof assistant. In *Proceedings of the 33rd ACM Symposium on Principles of Programming Languages* (Charleston, South Carolina, Jan. 2006), pp. 42–54.

[41] Li, Z., Lu, S., Myagmar, S., and Zhou, Y. CP-Miner: A tool for finding copy-paste and related bugs in operating system code. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI)* (San Francisco, CA, Dec. 2004).

[42] Lu, L., Arpaci-Dusseau, A. C., Arpaci-Dusseau, R. H., and Lu, S. A study of Linux file system evolution. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST)* (2013), pp. 31–44.

[43] Madhavapeddy, A., Mortier, R., Rotsos, C., Scott, D., Singh, B., Gazagnaire, T., Smith, S., Hand, S., and Crowcroft, J. Unikernels: Library operating systems for the cloud. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (Houston, TX, Mar. 2013), pp. 461–472.

[44] Musuvathi, M. S., Park, D., Park, D. Y. W., Chou, A., Engler, D. R., and Dill, D. L. CMC: A pragmatic approach to model checking real code. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI)* (Boston, MA, Dec. 2002).

[45] Palix, N., Thomas, G., Saha, S., Calvès, C., Lawall, J., and Muller, G. Faults in Linux: Ten years later. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (Newport Beach, CA, Mar. 2011), pp. 305–318.

[46] Patocka, M. [Patch] hpfs: update ctime and mtime on directory modification, 2015. `https://lkml.org/lkml/2015/9/2/552`.

[47] Rubio-González, C., Gunawi, H. S., Liblit, B., Arpaci-Dusseau, R. H., and Arpaci-Dusseau, A. C. Error propagation analysis for file systems. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation* (Dublin, Ireland, June 2009), pp. 270–280.

[48] Rubner, Y., Tomasi, C., and Guibas, L. J. The earth mover's distance as a metric for image retrieval. *International journal of computer vision 40*, 2 (2000), 99–121.

[49] Ryzhyk, L., Chubb, P., Kuz, I., Le Sueur, E., and Heiser, G. Automatic device driver synthesis with Termite. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)* (Big Sky, MT, Oct. 2009), pp. 73–86.

[50] Savage, S., Burrows, M., Nelson, G., Sobalvarro, P., and Anderson, T. Eraser: A dynamic data race detector for multi-threaded programs. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles* (Saint-Malo, France, Oct. 1997), pp. 27–37.

[51] Serebryany, K., Bruening, D., Potapenko, A., and Vyukov, D. AddressSanitizer: A Fast Address Sanity Checker. In *Proceedings of the 2012 ATC Annual Technical Conference (ATC)* (Boston, MA, June 2012), pp. 309–318.

[52] Shannon, C. E. A mathematical theory of communication. *Bell system technical journal 27* (1948).

[53] Swain, M. J., and Ballard, D. H. Color indexing. *International journal of computer vision 7*, 1 (1991), 11–32.

[54] The IEEE and The Open Group. `fsync()`, 2013. The Open Group Base Specifications Issue 7, IEEE Std 1003.1, 2013 Edition, `http://pubs.opengroup.org/onlinepubs/9699919799/functions/fsync.html`.

[55] The IEEE and The Open Group. `rename()`, 2013. The Open Group Base Specifications Issue 7, IEEE Std 1003.1, 2013 Edition, `http://pubs.opengroup.org/onlinepubs/9699919799/functions/rename.html`.

[56] The Linux Programming Interface. `fync()`, 2014. Linux's Programmer's Manual, `http://man7.org/linux/man-pages/man2/fsync.2.html`.

[57] The World Wide Web Consortium (W3C). Document Object Model (DOM) Level 2 HTML Specification, Jan. 2003. `http://www.w3.org/TR/DOM-Level-2-HTML/Overview.html`.

[58] Torvalds, L. `fs/ubifs/file.c` at v4.0-rc2, 2015. `https://github.com/torvalds/linux/blob/v4.0-rc2/fs/ubifs/file.c#L1321`.

[59] Torvalds, L. `inlucde/linux/fs.h` at v4.0-rc2, 2015. `https://github.com/torvalds/linux/blob/v4.0-rc2/include/linux/fs.h#L1688`.

[60] Wang, X., Chen, H., Jia, Z., Zeldovich, N., and Kaashoek, M. F. Improving integer security for systems with KINT. In *Proceedings of the 10th Symposium on Operating Systems Design and Implementation (OSDI)* (Hollywood, CA, Oct. 2012), pp. 163–177.

[61] Wang, X., Lazar, D., Zeldovich, N., Chlipala, A., and Tatlock, Z. Jitk: A trustworthy in-kernel interpreter infrastructure. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI)* (Broomfield, Colorado, Oct. 2014), pp. 33–47.

[62] Wang, X., Zeldovich, N., Kaashoek, M. F., and Solar-Lezama, A. Towards optimization-safe systems: Analyzing the impact of undefined behavior. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)* (Farmington, PA, Nov. 2013), pp. 260–275.

[63] YAMAGUCHI, F., GOLDE, N., ARP, D., AND RIECK, K. Modeling and discovering vulnerabilities with code property graphs. In *Proceedings of the 35th IEEE Symposium on Security and Privacy (Oakland)* (San Jose, CA, May 2014), pp. 590–604.

[64] YANG, J., SAR, C., AND ENGLER, D. explode: A lightweight, general system for finding serious storage system errors. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)* (Seattle, WA, Nov. 2006), pp. 10–10.

[65] YANG, J., SAR, C., TWOHEY, P., CADAR, C., AND ENGLER, D. Automatically generating malicious disks using symbolic execution. In *Proceedings of the 27th IEEE Symposium on Security and Privacy (Oakland)* (Oakland, CA, May 2006), pp. 243–257.

[66] YANG, J., TWOHEY, P., AND DAWSON. Using model checking to find serious file system errors. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI)* (San Francisco, CA, Dec. 2004), pp. 273–288.

[67] YIN, Z., MA, X., ZHENG, J., ZHOU, Y., BAIRAVASUNDARAM, L. N., AND PASUPATHY, S. An empirical study on configuration errors in commercial and open source systems. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)* (Cascais, Portugal, Oct. 2011), pp. 159–172.

[68] YOSHIMURA, T., AND KONO, K. Who writes what checkers?—learning from bug repositories. In *Proceedings of the 10th Workshop on Hot Topics in System Dependability (HotDep)* (Broomfield, CO, Oct. 2014).

[69] YUAN, D., LUO, Y., ZHUANG, X., RODRIGUES, G. R., ZHAO, X., ZHANG, Y., JAIN, P. U., AND STUMM, M. Simple testing can prevent most critical failures: An analysis of production failures in distributed data-intensive systems. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI)* (Broomfield, Colorado, Oct. 2014), pp. 249–265.