Error Propagation Analysis for File Systems *

Cindy Rubio-González Haryadi S. Gunawi Ben Liblit
Remzi H. Arpaci-Dusseau Andrea C. Arpaci-Dusseau

Computer Sciences Department, University of Wisconsin–Madison
{crubio, haryadi, liblit, remzi, dusseau}@cs.wisc.edu

Abstract

Unchecked errors are especially pernicious in operating system file management code. Transient or permanent hardware failures are inevitable, and error-management bugs at the file system layer can cause silent, unrecoverable data corruption. We propose an interprocedural static analysis that tracks errors as they propagate through file system code. Our implementation detects overwritten, out-of-scope, and unsaved unchecked errors. Analysis of four widely-used Linux file system implementations (CIFS, ext3, IBM JFS and ReiserFS), a relatively new file system implementation (ext4), and shared virtual file system (VFS) code uncovers 312 error propagation bugs. Our flow- and context-sensitive approach produces more precise results than related techniques while providing better diagnostic information, including possible execution paths that demonstrate each bug found.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification—formal methods, reliability, validation; D.2.5 [*Software Engineering*]: Testing and Debugging—error handling and recovery; D.4.3 [*Operating Systems*]: File Systems Management

General Terms Algorithms, Languages, Reliability, Verification

Keywords static program analysis, interprocedural dataflow analysis, weighted pushdown systems, copy constant propagation, binary decision diagrams

1. Introduction

Run-time errors are unavoidable whenever software interacts with the physical world. Incorrect error handling is a longstanding problem in many application domains, but is especially troubling when it affects operating systems' file management code. File systems occupy a delicate middle layer in operating systems. They sit above generic block storage drivers, such as those that implement SCSI, IDE, or software RAID; or above network drivers in the case of network file systems. These lower layers ultimately interact with the physical world, and as such, may produce both transient and persistent errors. At the same time, implementations of specific file systems sit below generic file management layers of the operating system, which in turn relay information through system calls into user applications. No application can possibly manage errors that the file system fails to report. The trustworthiness of the file system in handling or propagating errors is an upper bound on the trustworthiness of all storage-dependent user applications.

Furthermore, this problem cannot simply be fixed and forgotten. Implementations abound, with more constantly appearing. Linux alone includes dozens of different file systems. There is no reason to believe that file system designers are running out of ideas or that the technological changes that motivate new file system development are slowing down.

Given the destructive potential of buggy file systems, it is critical that error propagation patterns be carefully vetted. However, failures in the physical layer, while inevitable, are rare enough in daily use that traditional testing is unlikely to bear fruit. Therefore, we propose a static analysis to identify certain common classes of error mismanagement. Our approach is a flow- and context-sensitive, interprocedural, forward dataflow analysis. The analysis resembles an over-approximating counterpart to a typical (under-approximating) copy constant propagation analysis [31], but with certain additional specializations for our specific problem domain. Our analysis is unsound in the presence of pointers, but has been designed for a balance of precision and accuracy that is useful to kernel developers in practice. Diagnostic reports include detailed witness traces that illustrate the error-fumbling missteps a file system could take.

The key contributions of this paper are as follows:

- We characterize the error propagation dataflow problem in its various guises (Section 2) and encode these using weighted pushdown systems (WPDSs) (Section 3).
- We show how to extract detailed diagnostic error reports from the raw analysis results (Section 4).
- We describe a high-performance implementation capable of analyzing real-world file systems (Section 5).
- We identify recurring safe and unsafe patterns and offer experimental results for several Linux file systems (Section 5).

We consider related work in Section 6, and Section 7 concludes.

2. Linux Error Management

This paper focuses on file systems in the Linux 2.6.27 kernel. Our approach combines generic program analysis techniques with specializations for Linux coding idioms. Other operating systems share the same general style, although some details may differ.

* Supported in part by AFOSR grant FA9550-07-1-0210; LLNL contract B580360; NSF grants CCF-0621487, CCF-0701957, and CNS-0720565; and an AAUW International Doctoral Fellowship. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF or other institutions.

PLDI'09, June 15–20, 2009, Dublin, Ireland.

2.1 Integer Error Codes

Different kinds of failure require different responses. For example, an input/output (I/O) error produces an `EIO` error code, which might be handled by aborting a failed transaction, scheduling it for later retry, releasing allocated buffers to prevent memory leaks, and so on. Memory shortages yield the `ENOMEM` error code, signaling that the system must release some memory in order to continue. Disk quota exhaustion propagates `ENOSPC` across many file system routines to prevent new allocations.

Unfortunately, Linux (like many operating systems) is written in C, which offers no exception handling mechanisms by which an error code could be raised or thrown. Errors must propagate through conventional mechanisms such as variable assignments and function return values. Most Linux run-time errors are represented as simple integer codes. Each integer value represents a different kind of error. Macros give these mnemonic names: `EIO` is defined as 5, `ENOMEM` is 12, and so on. Linux uses 34 basic named error macros, defined as the constants 1 through 34.

Error codes are negated by convention, so `-EIO` may be assigned to a variable or returned from a function to signal an I/O error. Return-value overloading is common. An `int`-returning function might return the positive count of bytes written to disk if a write succeeds, or a negative error code if the write fails. Callers must check for negative return values and propagate or handle errors that arise. Remember that error codes are merely integers given special meaning by coding conventions. Any `int` variable could potentially hold an error code, and the C type system offers little help determining which variables actually carry errors.

2.2 Consequences of Not Handling Errors

Ideally, an error code arises in lower layers (such as block device drivers) and propagates upward through the file system, passing from variable to variable and from callee to caller, until it is properly handled or escapes into user space as an error result from a system call. Propagation chains can be long, crossing many functions, modules, and software layers. If buggy code breaks this chain, higher layers receive incorrect information about the outcomes of file operations.

For example, if there is an I/O error deep down in the `sync()` path, but the `EIO` error code is lost in the middle, then the application will believe its attempt to synchronize with the storage system has succeeded, when in fact it failed. Any recovery routine implemented in upper layers will not be executed. “Silent” errors such as this are difficult to debug, and by the time they become visible, data may already be irreparably corrupted or destroyed.

In this paper, we are interested in how file systems propagate and handle those error codes passed up from device drivers.

2.3 Checked Versus Unchecked Errors

Some action should be taken whenever an error occurs. Actions range from simple notification to attempted recovery. We say that an error has been *checked* if such an action has taken place. There is no requirement to clear or reset an error-carrying variable after that error has been checked and handled. Once recovery code has dealt with the problem, a variable that contained `-EIO` to report an I/O error can now be seen as merely containing the integer value -5. Overwriting such a variable before it was checked is a bug, but overwriting it after it has been checked is fine. For this reason, it is useful to distinguish *unchecked error codes* from other values that might either be already-checked errors or ordinary (non-error-bearing) integers. This in turn requires recognizing correct error handling when it does occur. Recognizing error-handling code is nontrivial, given the complexity and variety of error recovery policies in modern file systems. For purposes of this analysis, we adopt a simple definition of “correct handling” that works well in many cases, and that can be extended easily as necessary.

```
1 int status = write(...);
2 if (status < 0) {
3   printk("write failed: %d\n", status);
4   // perform recovery procedures
5 } else {
6   // write succeeded
7 }
8 // no unchecked error at this point
```

Figure 1. Typical error-checking code example

Figure 1 shows a typical fragment of Linux kernel code. Many error-handling routines call `printk`, an error-logging function, with the error code being handled passed as an argument. Because this is an explicit action, it is reasonable to assume that the programmer is aware of the error and is handling it appropriately. Thus, if `status` contained an unchecked error in line 2, we can assume that it contains a checked error in line 3.

Because error codes are passed as negative integers (such as `-EIO` for -5), sign-checking such as that in line 2 is common. If the condition is false, then `status` must be non-negative and therefore cannot contain an error code in line 6. When paths merge in line 8, `status` cannot possibly contain an unchecked error. Therefore, there is no error propagation bug in this code.

Passing error codes to `printk` is common, but not universal. Code may check for and handle errors silently, or may use `printk` to warn about a problem that has been detected but not yet remedied. More accurate recognition of error-checking code may require annotation. For example, we might require that programmers assign a special `ECHECKED` value to variables with checked errors, or pass such variables as arguments to a special `checked` function to mark them as handled. Requiring explicit programmer action to mark errors as checked would improve diagnosis by avoiding the silent propagation failures that presently occur.

2.4 Error Propagation Bugs

Our goal is to find those error instances that vanish before proper checking is performed. We find three general cases in which unchecked errors are commonly lost. The variable holding the unchecked error value (1) is overwritten with a new value, (2) goes out of scope, or (3) is returned by a function but not saved by the caller. Real-world code examples for each of these follow.

Figure 2(a) illustrates an overwritten error in ext2. Function `ext2_sync_inode`, called in line 3, can return one of several errors including `ENOSPC`. The code inside the if statement in line 4 handles all errors but `ENOSPC`. Thus, if `ENOSPC` is returned then it is overwritten in line 8. This may lead to silent data loss.

Figure 2(b) depicts an out-of-scope error found in IBM JFS. `txCommit`, starting in line 1, commits any changes that its caller has made. This function returns `EROFS` if the file system is read-only. `txCommit` also may propagate `EIO` from calling `diWrite` in line 9. `diFree` calls `txCommit` in line 17, saving the return value in variable `rc`. Unfortunately, `diFree` does not check `rc` when the function exits. In fact, `diFree` always returns 0 in line 19, thereby claiming that the commit operation always succeeds. Interestingly, all other callers of `txCommit` save and propagate the return value correctly. This strongly suggests that `rc` should be returned, and that the code as it stands is incorrect.

Figure 2(c) shows an unsaved error found in ext3. Function `log_wait_commit` returns `EIO` if a transaction commit has failed (lines 5–7). In a synchronous operation, this `EIO` error code is correctly propagated to the user application. In addition to synchronous foreground I/O operations, there are also background I/O

```
int ext2_xattr_set2(...) {
  ...
  error = ext2_sync_inode(...);
  if (error && error != -ENOSPC) {
    ...
    goto cleanup;
  }
  error = 0; //overwriting error
}
```

(a) An overwritten error in ext2

```
int txCommit(...) {
  ...
  if (isReadOnly(...)) {
    rc = -EROFS;
    ...
    goto TheEnd;
  } ...

  if (rc = diWrite(...))
    txAbort(...);

 TheEnd: return rc;
}

int diFree(...) {
  ...
  rc = txCommit(...);
  ...
  return 0; //rc out of scope
}
```

(b) An out-of-scope error in IBM JFS

```
1  int log_wait_commit(...) {
2    ...
3    wake_up();
4    ...
5    if (is_journal_aborted(journal))) {
6      err = -EIO;
7      return err;
8    }
9  }
10
11 int __process_buffer(...) {
12   ...
13   log_start_commit(journal, tid);
14   log_wait_commit(journal, tid);
15   ...
16 }
```

(c) An unsaved error code in ext3

Figure 2. Three common scenarios in which unchecked errors are lost

operations that are flushed periodically to the disk. Since there is no way to communicate any related errors of background I/O operations to user applications, these errors are often dropped. One example is when a periodic timer launches a background checkpoint operation that will wrap all dirty buffers to a transaction, commit the transaction to the journal, and wait for it to finish. As shown in line 14, the I/O failure propagated by the `log_wait_commit` function is neglected by the `__process_buffer` function, which itself is called during the background checkpoint. Hence, if there is a failure, data is silently lost.

3. Analysis Formalization

The first task is to determine, at each program point, the set of unchecked error codes each variable may contain. Given this information, the bugs described in Section 2.4 can be detected using a second pass over the code. For example, error overwriting occurs when the left side of an assignment already contains an unchecked error, while error dropping occurs when a variable containing an unchecked error goes out of scope. Error propagation can be formulated as a forward dataflow problem. Error constants such as `EIO` generate unchecked error codes. Assignments propagate unchecked errors forward from one variable to another. Propagation ends when an error is overwritten, dropped, or checked by error-handling code.

This problem resembles copy constant propagation [31]. However, copy constant propagation finds *one* constant value that a variable *must* contain (if any), whereas we find the *set* of error code constants that a variable *may* contain. Copy constant propagation drives semantics-preserving optimization, and therefore under-approximates. We use error propagation analysis for bug reporting, and over-approximate so that no possible bug is overlooked.

The following subsections describe WPDSs (our formalism of choice) and how we use WPDSs to encode the error propagation problem. Note that the error propagation problem can be described as a standard context-sensitive interprocedural analysis problem. We choose to cast the problem as a path problem over WPDSs because WPDSs (1) provide an algebraic formulation for handling local variables [19], and (2) support generating a witness trace as a proof of the result of solving the path problem [25]. We use witness tracing extensively to provide programmers with detailed diagnostic traces for each potential program bug (see Section 4).

3.1 Weighted Pushdown Systems

We use WPDSs [25] to formulate and solve the error propagation dataflow problem. A WPDS is a pushdown system that associates a weight with each rule. Weights can serve as transfer functions that describe the effect of each statement on the program state. Such weights must be elements of a set that constitutes a bounded idempotent semiring. We now formally define WPDSs and related terms; Section 3.2 shows how WPDSs can be applied to solve the error propagation dataflow problem.

DEFINITION 1. *A* **pushdown system** *is a triple* $\mathscr{P} = (P, \Gamma, \Delta)$ *where* P *and* Γ *are finite sets called the* **control locations** *and* **stack alphabet**, *respectively. A* **configuration** *of* $\mathscr{P}$ *is a pair* $\langle p, w\rangle$, *where* $p \in P$ *and* $w \in \Gamma^*$. Δ *contains a finite number of rules* $\langle p, \gamma\rangle \hookrightarrow \langle p', w\rangle$, *where* $p, p' \in P$, $\gamma \in \Gamma$, *and* $w \in \Gamma^*$, *which define a transition relation* $\Rightarrow$ *between configurations of* $\mathscr{P}$ *as follows:*
If $r = \langle p, \gamma\rangle \hookrightarrow \langle p', w\rangle$, *then* $\langle p, \gamma w'\rangle \Rightarrow \langle p', ww'\rangle$ *for all* $w' \in \Gamma^*$.

As shown by Lal et al. [19] and Reps et al. [25], a pushdown system can be used to model the set of valid paths in an interprocedural control-flow graph (CFG).

DEFINITION 2. *A* **bounded idempotent semiring** *is a quintuple* $(D, \oplus, \otimes, \bar{0}, \bar{1})$, *where* D *is a set,* $\bar{0}$ *and* $\bar{1}$ *are elements of* D, *and* $\oplus$ *(the combine operator) and* $\otimes$ *(the extend operator) are binary operators on* D *conforming to certain algebraic properties as denoted in Reps et al. [25].*

Each element of D is called a *weight*. The extend operator ($\otimes$) is used to calculate the weight of a path. The combine operator ($\oplus$) is used to summarize the weights of a set of paths that merge.

DEFINITION 3. *A* **weighted pushdown system** *is a triple* $\mathscr{W} = (\mathscr{P}, \mathscr{S}, f)$ *such that* $\mathscr{P} = (P, \Gamma, \Delta)$ *is a pushdown system,* $\mathscr{S} = (D, \oplus, \otimes, \bar{0}, \bar{1})$ *is a bounded idempotent semiring, and* $f : \Delta \to D$ *is a function that assigns a value from* D *to each rule* $\mathscr{P}$.

Let $\sigma = [r_1, \ldots, r_k]$ be a sequence of rules (a path in the CFG) from Δ^*. We associate a value with σ by using function f. This value is defined as $val(\sigma) = f(r_1) \otimes \cdots \otimes f(r_k)$. For any configurations c and c' of $\mathscr{P}$, $path(c, c')$ denotes the set of all rule sequences $[r_1, \ldots, r_k]$, i.e., the set of all paths transforming c into c'.

DEFINITION 4. *Let* $\mathscr{W} = (\mathscr{P}, \mathscr{S}, f)$ *be a weighted pushdown system, where* $\mathscr{P} = (P, \Gamma, \Delta)$, *and let* $C \subseteq P \times \Gamma^*$ *be a regular set of*

Rule	Control flow modeled
$\langle p,a\rangle \hookrightarrow \langle p,b\rangle$	Intraprocedural flow from a to b
$\langle p,c\rangle \hookrightarrow \langle p,f_{enter}r\rangle$	Call from c to procedure entry f_{enter}, eventually returning to r
$\langle p,f_{exit}\rangle \hookrightarrow \langle p,\varepsilon\rangle$	Return from procedure exit f_{exit}

Table 1. Encoding of control flow as PDS rules

configurations. The **generalized pushdown successor problem** *is to find for each* $c \in P \times \Gamma^*$*:*

- $\delta(c) \equiv \oplus\{val(\sigma) \mid \sigma \in path(c',c), c' \in C\}$
- *a* **witness set** *of paths* $w(c) \subseteq \cup_{c' \in C}\, path(c',c)$ *such that* $\oplus_{\sigma \in w(c)} val(\sigma) = \delta(c)$.

The *generalized pushdown successor problem* is a forward reachability problem. It finds $\delta(c)$, the combine of values of all paths between configuration pairs, i.e., the meet over all paths value for each configuration pair. A corresponding witness set $w(c)$ is a subset of inspected paths such that their combine is $\delta(c)$. This set can be used to justify the resulting $\delta(c)$.

The meet over all paths value is the best possible solution to a static dataflow problem. Thus, a WPDS is a useful dataflow engine for problems that can be encoded with suitable weight domains. In Section 3.2 we show how the error propagation problem can be encoded as a weight domain.

In order to handle local variables properly, we use an extension to WPDSs proposed by Lal et al. [19]. This extension requires the definition of a *merge function*, which can be seen as a special case of the extend operator. This function is used when extending a weight w_1 at a call program point with a weight w_2 at the end of the corresponding callee. The resulting weight corresponds to the weight after the call. The difference between the merge function and a standard extend operation is that w_2 contains information about the callee's locals; this information is irrelevant to the caller. Thus, the merge function defines what information from w_2 to keep or discard before performing the extend.

3.2 Creating the Weighted Pushdown System

Per Definition 3, a WPDS consists of a pushdown system, a bounded idempotent semiring, and a mapping from pushdown system rules to associated weights. We now define these components for a WPDS that encodes the error propagation dataflow problem.

3.2.1 Pushdown System

We model the control flow of the program with a pushdown system using the approach of Lal et al. [21]. Let P contain a single state $\{p\}$. Γ corresponds to program statements, and Δ corresponds to edges of the interprocedural CFG. Table 1 shows the PDS rule for each type of CFG edge.

3.2.2 Bounded Idempotent Semiring

We classify integer constants into *error constants* and *non-error constants*. Among the error constants we further distinguish tentative from non-tentative errors for reasons we discuss further in Section 4.1. Let *TentativeErrors* be the set of tentative error constants: integer values used to represent error codes. For each tentative error constant define a corresponding *non-tentative* error constant. Let *NonTentativeErrors* be the set of all non-tentative error constants. Define $\mathscr{E} = \mathit{TentativeErrors} \cup \mathit{NonTentativeErrors}$ as the set of all error constants. For purposes of this analysis, all non-error constants can be treated as a single value, which we represent as *OK*. We also introduce *uninitialized* to represent uninitialized values. Let $\mathscr{C} = \mathscr{E} \cup \{OK, \mathit{uninitialized}\}$ be the set of all constants. Finally, let $\mathscr{V}$ be the set of all program variables.

Let $\mathscr{S} = (D, \oplus, \otimes, \bar{0}, \bar{1})$ be a bounded idempotent semiring per Definition 2. Elements of D are drawn from $\mathscr{V} \to 2^{\mathscr{V} \cup \mathscr{C}}$, so each weight in D is a mapping from variables to sets containing variables, error values, *OK* and/or *uninitialized*. This gives the possible values of v following execution of a given program statement in terms of the values of constants and variables before that statement.

The combine operator is applied component-wise, with each variable v mapping to any value it could have mapped to in either of the weights being combined. For all $w_1, w_2 \in D$ and $v \in \mathscr{V}$:

$$(w_1 \oplus w_2)(v) \equiv w_1(v) \cup w_2(v)$$

The extend operator is also applied component-wise:

$$(w_1 \otimes w_2)(v) \equiv (\mathscr{C} \cap w_2(v)) \cup \bigcup_{v' \in \mathscr{V} \cap w_2(v)} w_1(v')$$

where $w_1(v) \neq \emptyset$, otherwise $(w_1 \otimes w_2)(v) \equiv \emptyset$. This definition is essentially function composition generalized to the power set of variables and constants rather than just single variables.

Define the neutral weight $\bar{1}$ as $\{(v,\{v\}) \mid v \in \mathscr{V}\}$, which maps each variable to the set containing itself. This is a power-set generalization of the identity function. Define the annihilator weight $\bar{0}$ as $\{(v,\emptyset) \mid v \in \mathscr{V}\}$, mapping each variable to the empty set.

Finally, the merge function is defined as follows. Let w_1 be the weight of the caller just before the call, and let w_2 be the weight at the very end of the callee. Then for any variable $v \in \mathscr{V}$,

$$merge(w_1(v), w_2(v)) \equiv \begin{cases} w_1(v) & \text{if } v \text{ is a local variable} \\ w_2(v) & \text{if } v \text{ is a global variable} \end{cases}$$

This propagates any changes that the callee made to globals while discarding any changes that the callee made to locals.

3.2.3 Transfer Functions

Each control-flow edge in the source program corresponds to a WPDS rule and therefore needs an associated weight drawn from the set of transfer functions D defined in Section 3.2.2. In the following discussion of specific source constructs, we describe most transfer functions as being associated with specific statements. The corresponding WPDS rule weight is associated with the edge from a statement to its unique successor. Conditionals have multiple outgoing edges and therefore require multiple transfer functions.

Assignments Here we consider only assignments without function calls on the right side. We leave the discussion of assignments such as $v = f()$ for later in this section.

Copy mode versus transfer mode Our analysis has two chief modes of operation: *copy mode* and *transfer mode*. Consider an assignment $t = s$ where $t, s \in \mathscr{V}$ are distinct and s might contain an unchecked error code. In copy mode, assignment copies errors: after the assignment $t = s$, both t and s contain unchecked errors. In transfer mode, the assignment $t = s$ leaves an unchecked error in t but removes it from s, effectively transferring ownership of unchecked error values across assignments. Discussion that follows refers to both copy and transfer modes unless otherwise stated.

Simple assignments These are assignments of the form $v = e$, where $e \in \mathscr{V} \cup \mathscr{C}$. Let *Ident* be the function that maps each variable to the set containing itself. (Note that this is identical to $\bar{1}$ per Section 3.2.2.) The transfer function for a simple assignment in copy mode is then $\mathit{Ident}[v \mapsto \{e\}]$. In other words, v must have the value of e after this assignment, while all other variables retain whatever values they had before the assignment. In transfer mode, let $\mathit{Donor} \equiv \{e\} \cap \mathscr{V} - \{v\}$ be the set containing the source variable (if any) of the assignment. Then the transfer function for a simple assignment in transfer mode is $Ident[v \mapsto \{e\}][s \mapsto \{OK\}$ for $s \in Donor]$ to transfer any unchecked errors from *Donor* to v.

Complex assignments These are assignments in which the assigned expression e is not a simple variable or constant. We assume that the program has been converted into three-address form, with no more than one operator on the right side of each assignment.

Consider an assignment of the form $v = e_1\, op\, e_2$ where $e_1, e_2 \in \mathscr{V} \cup \mathscr{C}$ and *op* is a binary arithmetic or bitwise operator (`+`, `&`, `<<`, etc.). Define $Donors \equiv \{e_1, e_2\} \cap \mathscr{V}$ as the set of variables on the right side of the assignment. Error codes are represented as integers but conceptually they are atomic values on which arithmetic operations are meaningless. Thus, if *op* is an arithmetic or bitwise operation, then we can safely assume that the variables in *Donors* do not contain errors. Furthermore, the result of this operation must be a non-error as well. Therefore, the transfer function for this assignment is $Ident[u \mapsto \{OK\}$ for all $u \in Donors \cup \{v\}]$.

Consider instead an assignment of the form $v = e_1\, op\, e_2$ where $e_1, e_2 \in \mathscr{V} \cup \mathscr{C}$ and *op* is a binary relational operator (`>`, `==`, etc.). Relational comparisons are meaningful for error codes, so we cannot assume that e_1 and e_2 are non-errors. However, the Boolean result of the comparison cannot be an error. Therefore, the transfer function for this assignment is $Ident[v \mapsto \{OK\}]$.

Assignments with unary operators ($v = op\, e_1$) are similar: arithmetic and bitwise operators map both v and e_1 (if a variable) to $\{OK\}$. However, C programmers often use logical negation to test for equality to 0. So when *op* is logical negation (`!`) or an indirection operator (`&`, `*`), the transfer function maps v to $\{OK\}$ but leaves e_1 unchanged.

Conditionals Assume that conditional statements with short-circuiting conditions are rewritten as nested conditional statements with simple conditions. A transfer function is then associated with each branch of a conditional statement. The transfer function to be applied on each branch depends upon the condition.

Consider a conditional statement of the form $if(v)$, where $v \in \mathscr{V}$. The true branch is selected when v is not equal to zero, which does not reveal any additional information about v: it may or may not contain an error value. In this case, variables should remain mapped to whatever values they had before. On the other hand, the false branch is selected when v is equal to zero. Because zero is never an error code, v definitely does not contain an error. Thus the transfer functions associated with the true and false branches are *Ident* and $Ident[v \mapsto \{OK\}]$, respectively.

Conversely, consider conditionals of the forms $if(v > 0)$, $if(v \geq 0)$, $if(0 < v)$, $if(0 \leq v)$, $if(0 == v)$, $if(v == 0)$, and $if(!v)$. In all of these cases, the transfer function associated with the true branch is $Ident[v \mapsto \{OK\}]$. The true branch is never selected when v is negative, so v cannot contain an error on that branch. The transfer function for the false branch is the identity function *Ident*.

Lastly, consider conditional statements such as $if(v < 0)$, $if(v \leq 0)$, $if(0 > v)$ and $if(0 \geq v)$. We associate the transfer function *Ident* with the true branch and $Ident[v \mapsto \{OK\}]$ with the false branch. In each of these cases, the false branch is only selected when v is non-negative, which means that v cannot contain an error code.

For conditional statements that do match none of the above patterns, we simply associate *Ident* with both true and false branches. An example of such a pattern is $if(v_1 < v_2)$, where $v_1, v_2 \in \mathscr{V}$.

Function calls We adopt the convention used by Callahan [3], and later by Reps et al. [25], in which the CFG for each function has unique entry and exit nodes. The entry node is not the first statement in the function, but rather appears just before the first statement. Likewise, we assume that function-terminating statements (e.g., `return` or last-block fall-through statements) have a synthetic per-function exit node as their unique successors. We use these dummy entry and exit nodes to manage data transfer between callers and callees.

CFGs for individual functions are combined together to form an interprocedural CFG. Furthermore, each CFG node n that contains a function call is split into two nodes: a *call* node n_1 and a *return-site* node n_2. There is an interprocedural *call-to-enter edge* from n_1 to the callee's entry node. Similarly, there is an interprocedural *exit-to-return-site edge* from the callee's exit node to n_2.

Local variable initialization First consider a call to a void function that takes no parameters. Let $\mathscr{L}, \mathscr{G} \subseteq \mathscr{V}$ respectively be the sets of all local and global variables. Recall that transfer functions are associated with CFG edges. For the edge from the callee's entry node to the first actual statement in the callee, we use the transfer function $Ident[v \mapsto \{uninitialized\}$ for $v \in \mathscr{L}]$. When a function begins executing, local variables are uninitialized while global variables retain their old values.

Parameter passing Now consider a call to a void function that takes one or more parameters. We introduce new global variables, called *exchange variables*, to convey actual arguments from the caller into the formal parameters of the callee. One exchange variable is introduced for each function parameter. Suppose function F has formal parameters $f_1, f_2, \ldots, f_n$. Let $F(a_1, a_2, \ldots a_n)$ be a function call to F with actual parameters $a_i \in \mathscr{V} \cup \mathscr{C}$. We introduce global exchange variables $F_1, F_2, \ldots, F_n$. The interprocedural call-to-enter edge is given the transfer function for a group of n simultaneous assignments $F_i = a_i$, exporting each actual argument into the corresponding global exchange variable. Rules for assignment transfer functions discussed earlier apply, including the transfer-mode variant which maps each a_i to $\{OK\}$ after the assignments.

A similar process imports values from global exchange variables into callee formal parameters. For a callee F with formal parameters $f_1, f_2, \ldots, f_n$, the edge from the callee's entry node to the first actual statement in the callee is given the transfer function for a group of n simultaneous assignments $f_i = F_i$, as though each formal argument were initialized with a value from the corresponding exchange variable. Other local variables are uninitialized as before.

Thus, argument passing is modeled a two-step process: first the caller exports its arguments into global exchange variables, then the callee imports these exchange variables into its formal parameters.

Return value passing Lastly, suppose that function F returns non-void. Let $r \in \mathscr{V} \cup \mathscr{C}$ be the value being returned by some `return` r statement, and let F_{ret} be a per-function global exchange variable. Then the edge connecting this return statement node to the dummy exit node is given the transfer function for an assignment $F_{ret} = r$.

As defined in Section 3.2.2, there are two kinds of error constants: tentative and non-tentative. Error codes are initially tentative and become non-tentative as soon as returned from a function. Let $E \equiv w(F_{ret}) \cap TentativeErrors$, where w is the weight at the end of function F. For all $e \in E$, we replace e with e', the corresponding non-tentative error. We discuss the need for two kinds of error constants in Section 4.1.

Let $v \in \mathscr{V}$ be the variable receiving the return value in the caller. Then the interprocedural exit-to-return-site edge from F's exit node is given the transfer function for an assignment $v = F_{ret}$.

Other interprocedural issues We consider functions whose implementation is not available to not have any effect on the state of the program. Thus the weight across any such call is simply *Ident*.

For functions with variable-length parameter lists, we apply the above transfer functions but we only consider the formal parameters explicitly declared.

Pointers Our treatment of pointers is both unsound and incomplete, but is designed for simplicity and to give useful results in practice. We find that many functions take a pointer to a callee-local variable where an error code, if any, should be written. Thus we only consider pointer parameters and ignore other pointer operations. We assume that inside a function, pointer variables have no aliases and are never changed to point to some other variable.

Under these conditions, pointer parameters are equivalent to call-by-copy-return parameters. On the interprocedural call-to-enter edge, we copy pointed-to values from the caller to the callee, just as for simple integer parameters. On the interprocedural exit-to-return-site edge, we copy callee values back into the caller. This extra copy-back on return is what distinguishes pointer arguments from non-pointer arguments, because it allows changes made by the callee to become visible to the caller.

Function pointers Most function pointers in Linux file systems are used in a fairly restricted manner. Global structures define handlers for generic operations (e.g., read, write, open, close), with one function pointer field per operation. Fields are populated statically or via assignments of the form "`file_ops->write = ext3_file_write`" where `ext3_file_write` identifies a function, not another function pointer. It is straightforward to identify the set of all possible implementations of a given operation. We then rewrite calls across such function pointers as `switch` statements that choose among possible implementations nondeterministically. This technique, previously employed by Gunawi et al. [11], accounts for approximately 80% of function pointer calls while avoiding the overhead and complexity of a general field-sensitive points-to analysis. The remaining 20% of calls are treated as *Ident*. Note that we analyze each file system individually; this perfectly disambiguates nearly all indirect calls in the code under study.

Error-handling functions As suggested in Section 2.3, we consider any error values in a variable to have been checked when the variable is passed as an argument to `printk`. `printk` is a variadic function whose first parameter is always a format string. The transfer function for such a call is $\mathit{Ident}[v \mapsto \{\mathit{OK}\}$ for v in the actual `int`-typed arguments to `printk`$]$. This also applies to error-handling functions specific to each file system under study: `ext3_error`, `jfs_error`, etc.

4. Finding and Describing Bugs

Here we describe how we query the WPDS to find error propagation bugs. We also describe how witness information is used to construct paths and slices that better describe the bugs found.

4.1 Querying the Weighted Pushdown System

We perform a poststar query [25] on the WPDS, with the beginning of the program as the starting configuration. For kernel analysis, we synthesize a `main` function that nondeterministically calls all exported entry points of the file system under analysis. The result is a weighted automaton. We apply the *path_summary* algorithm of Lal et al. [20] to read out weights from this automaton. This algorithm calculates, for each state in the automaton, the combine of all paths in the automaton from that state to the accepting state, if any. We can then retrieve the weight representing execution from the beginning of the program to any particular point of interest.

We turn the three cases in which error codes are commonly lost into a single case: overwritten errors. For out-of-scope errors, we insert assignment statements at the end of each function. These extra statements assign *OK* to each local variable except for the variable being returned (if any). Thus, if any local variable contains an unchecked error when the function ends, then the error is overwritten by the inserted assignment and our analysis detects the problem. In the case of unsaved errors, for each function whose result is not already being saved by the caller, we introduce a temporary local variable to hold that result. These temporaries are overwritten with *OK* at the end of the function, as described above. Thus, unsaved return values are transformed into out-of-scope bugs. A systematic naming convention for these newly-added temporary variables lets us distinguish the two cases later so that they can be described properly in diagnostic messages. Thus, both out-of-scope and unsaved errors are ultimately turned into overwritten errors.

	CIFS	ext3	ext4	IBM JFS	ReiserFS	VFS
Full path	14.7	66.6	70.4	16.7	17.9	22.6
Path slice	6.0	8.1	8.3	4.7	3.8	5.8

Table 2. Average lengths of full paths and path slices

Our goal is to find whether each assignment may overwrite an error value. At each assignment p we retrieve the associated weight w. Let $S, T \subseteq \mathscr{C}$ respectively be the sets of possible constant values held by the source and target of the assignment, as revealed by w. Note that w does not include the effect of assignment p itself. Rather, it reflects the state just before p. Then:

1. If $T \cap \mathit{NonTentativeErrors} = \emptyset$, then the assignment cannot overwrite any non-tentative error code and is not examined.
2. If $T \cap \mathit{NonTentativeErrors} = S = \{e\}$ for some single error code e, then the assignment can only overwrite an error code with the same error code and is not examined.[1]
3. Otherwise, it is possible that this assignment will overwrite an unchecked error code with a different code. Such an assignment is incorrect, and is presented to the programmer along with suitable diagnostic information.

Observe that we only report overwrites of non-tentative errors. We find that overwrites of tentative errors are rarely true bugs. This is due to coding conventions such as storing potential error codes in variables before failure conditions actually hold. This phenomenon is usually contained within the function that generates the error code: error codes returned to callers generally represent real run-time errors. Our transformation of returned errors from tentative to non-tentative models this coding practice; ignoring it would have tripled our false-positive count. We list all error codes that could be possibly overwritten at each bad assignment, then select one for detailed path reporting as described next.

4.2 Witnesses, Paths, and Slices

WPDSs support witness tracing. As mentioned in Definition 4, a witness set is a set of paths that justify the weight reported for a given configuration. This information lets us report not just the location of a bad assignment, but also detailed information about how that program point was reached in a way that exhibits the bug.

For each program point p containing a bad, error-overwriting assignment, we retrieve a corresponding set of witness paths. Each witness path starts at the beginning of the program and ends at p. We select one of these paths arbitrarily and traverse it backward, starting at p and moving along reversed CFG edges toward the beginning of the program. During this backward traversal, we track a single special *target location* which is initially the variable overwritten at p. The goal is to stop when the target is directly assigned the error value under consideration, i.e., when we have found the error's point of origin. This allows us to present only a relevant suffix of the complete witness path.

Let t be the currently-tracked target location. Each statement along the backward traversal of the selected witness path has one of the following forms:

[1] We open this loophole because we find that this is a commonly-occurring pattern judged to be acceptable by file-system developers.

```
1  int nextId() {
2    static int id;
3    return ++id;
4  }
5
6  int getError() {
7    return -EIO;
8  }
9
10 int load() {
11   int status, result = 0;
12
13   if (nextId())
14     status = getError();
15
16   result = status;
17
18   if (nextId())
19     result = -EPIPE;
20
21   return result;
22 }
```

(a) Example code

```
Error codes: *EIO
```

(b) List of overwritten/dropped errors

```
example.c:7: unchecked error "EIO" is returned
example.c:14: "status" receives unchecked error from function "getError"
example.c:16: "result" receives unchecked error from "status"
example.c:18: "result" has unchecked error
example.c:3: "result" has unchecked error
example.c:18: "result" has unchecked error
example.c:19: overwriting unchecked error in "result"
```

(c) Complete diagnostic path trace

```
example.c:7: unchecked error "EIO" is returned
example.c:14: "status" receives unchecked error from function "getError"
example.c:16: "result" receives unchecked error from "status"
example.c:19: overwriting unchecked error in "result"
```

(d) Diagnostic path slice

Figure 3. Example code fragment and corresponding diagnostic output

1. $t = x$ for some other variable $x \in \mathcal{V}$. Then the overwritten error value in t must have come from x. We continue the backward path traversal, but with x as the new tracked target location instead of t. Additionally, we produce diagnostic output showing the source file name, line number, and the message "t receives unchecked error from x." If x is a return exchange variable, then we print an alternate message reflecting the fact that t receives an error code from a function call (e.g., see the message for line 14 in Figure 3(a)).
2. $t = e$ for some error constant $e \in \mathcal{E}$. We have reached the point of origin of the overwritten error. Our diagnostic trace is now complete for the bad assignment at p. We produce a final diagnostic message showing the source file name, line number, and the message "t receives error value e." If t is a return exchange variable, then we print an alternate message reflecting the fact that an error code is being returned from a function (e.g., see the message for line 7 in Figure 3(a)).
3. Anything else. We continue the backward path traversal, retaining t as the tracked target location. Additionally, we produce diagnostic output showing the source file name, line number, and the message "t has unchecked error."

If all diagnostic output mentioned above is presented to the programmer, then the result is a step-by-step trace of every program statement from the origin of an error value to its overwriting at p. If diagnostic output is omitted for case 3, then the trace shows only key events of interest, where the error value was passed from one variable to another. We term this a *path slice*, as it is analogous to a program slice that retains only the statements relevant to a particular operation. In practice, we find that the concise path slice provides a useful overview while the complete witness path trace helps to fill in details where gaps between relevant statements are large enough to make intervening control flow non-obvious. Table 2 shows that slicing significantly reduces path lengths. Across all five file systems and the shared virtual file system, slicing shrinks paths by an average ratio of 5.7 to 1.

Note that we only provide diagnostic output for one overwritten error code per bad assignment. If the bad assignment may overwrite more than one error code, then we choose one arbitrarily. The instance chosen may not be a true bug, fooling the programmer into believing that no real problem exists. A different error value potentially overwritten by the same assignment may be a true bug. However, providing diagnostic output for all error values might overwhelm the programmer with seemingly-redundant output.

Figure 3(a) shows an example code fragment that has an error propagation bug in transfer mode. Figure 3(b) lists the error codes that may be overwritten/dropped at a particular program point. The error code which the rest of the diagnostic information correspond to is marked with an asterisk. `EIO` is the only error code that may be overwritten in this example. Figure 3(c) shows the complete diagnostic path trace. Observe that this trace begins in function `getError`, which is called from `load` in line 14. Execution eventually traverses into `nextId` (line 3) while traveling from the error code generation point (line 7) to the overwriting assignment (line 19). Figure 3(d) shows the diagnostic path slice which includes only those lines directly relevant to the error. Here we see just four events of interest: the generation of an error code, which is returned by function `getError` in line 7; the transfer of that error to `status` in line 14; the transfer of that error code from `status` to `result` in line 16; and the assignment to `result` in line 19.

5. Experimental Evaluation

Our implementation uses the CIL C front end [24] to apply preliminary source-to-source transformations on Linux kernel code. This includes redefining error code macros as distinctive expressions to avoid mistaking regular constants for error codes. We then traverse the CFG and emit a textual representation of the WPDS. Our separate analysis tool uses the WALi WPDS library [18] to perform the interprocedural dataflow analysis on this WPDS. Within our WALi-based analysis code, we encode weights using *binary decision diagrams* (BDDs) [2] as implemented by the BuDDy BDD library [22]. BDDs have been used before to encode weight domains [27]. The BDD representation allows highly-efficient implementation of key semiring operations, such as extend and combine.

We present the results of our analysis on four local file systems (ReiserFS, IBM JFS, ext3 and ext4), one network file system (CIFS), and common virtual file system (VFS) code used by all others.[2]

Our analysis reports 501 bugs in total, of which 312 are judged true bugs following manual inspection of the reports. IBM JFS and ReiserFS reports were inspected by the file systems' respective

[2] We exclude shared memory-management code (mm), as it slows the analysis significantly while containing very few error-propagation bugs.

```
1 if (err == -EIO) {
2   ...
3   err = ...; //safe
4 }
```

(a) Specific error code

```
1 reiserfs_warning(...);
2 err = -EIO; //safe
```

(b) Special function

```
1 if (retval && err)
2   retval = err; //safe
```

(c) Replacement

```
1  int err;
2  ...
3  retry:
4  ...
5  if (...)
6    return ...;
7    //err is safely out of scope
8
9    err = ...; //safe
10 ...
11 if (err == -ENOSPC && ...)
12   goto retry;
```

(d) Retries

Figure 4. Some recurring safe patterns recognized by the analysis

developers. CIFS and ext4 developers inspected a subset of their corresponding reports. A local domain expert who is also coauthor of this paper assessed the rest, including the reports for ext3.

Developer response has been positive:

> I think this is an excellent way of detecting bugs that happen rarely enough that there are no good reproduction cases, but likely hit users on occasion and are otherwise impossible to diagnose. [5]

Our local expert reports spending an average of five minutes to accept or reject each path trace. We find that unsaved error propagation bugs are the most common. In general we find that transfer mode yields better results than copy mode in the sense that it produces fewer false positives.

In the discussion that follows, we present results for each bug category. All results reported are for transfer mode unless explicitly stated otherwise. Table 3 summarizes our findings. We identify and describe safe patterns that we use to refine our tool. We also describe false positives in detail. Note that these are only "false" positives in that developers and our local expert judge that errors are safely overwritten, out of scope or unsaved. The fact that errors are overwritten, out of scope or unsaved is real, and in this sense the analysis is providing correct, precise information for the questions it was designed to answer.

5.1 Overwritten Bug Reports

Developers and our local expert identify 25 overwritten true bugs. We find that `EIO` and `ENOMEM` are the most commonly overwritten error codes. `EIO` signals I/O errors, including write failures that may lead to data loss. `ENOMEM` is used when there is insufficient memory.

Our tool recognizes four recurring patterns that represent safe overwrites. Figure 4(a) shows the most common recurring pattern found across all five file systems. Here, line 1 compares `err` with a specific error code. If they match, then line 3 clears `err` or assigns it a different error code. Overwriting one error code with another does not always represent a bug. For example, an error code generated in one layer of the operating system may need to be translated into a different code when passed to another layer. This clearly depends on the context and the error codes involved. In this case, we can see that the programmer acknowledges that `err` contains a specific error code before performing the assignment. We choose to trust the programmer in this particular scenario, thus we assume that overwriting the error code contained in `err` is safe.

Figure 4(b) shows the second common pattern, found in both ReiserFS and ext3. In this case the programmer acknowledges that something might be wrong by calling a function such as

```
1 if (err)
2   retval = err; //unsafe
```

(a) Replacement

```
1  int ret, err;
2  ret = ...;
3
4  if (ret) goto out;
5
6  ret = ...;
7  err = ...;
8
9  if (!ret && err)
10   ret = err;
11
12 out: return ret;
13 // err out of scope
```

(b) Precedence/scope

```
1 ret = ...;
2 ret2 = ...;
3
4 if (ret == 0)
5    ret = ret2;
6 ...
7 ret2 = ...; //unsafe
```

(c) Precedence/overwrite

```
1  buffer_head *tbh = NULL;
2  ...
3  if (buffer_dirty(tbh))
4    sync_dirty_buffer(tbh);
5    // unsaved error
6
7  if (!buffer_uptodate(tbh)) {
8    reiserfs_warning(...);
9    retval = -EIO;
10 }
```

(d) Redundancy

Figure 5. Some recurring unsafe patterns

`reiserfs_warning` in the case of ReiserFS. The call is usually followed by an assignment that may overwrite an error code. We choose to allow overwrites that occur immediately after such calls.

The third pattern, shown in Figure 4(d), appears in both ext3 and ext4. Here variable `err` may receive an error code from a function call (the function could return different error codes) in line 9. Our tool initially reported an overwrite at that line in the case of a retry. We observe that the `goto` statement in line 12 is always located inside an `if` statement. In addition, the variable being overwritten always appears in the condition (line 11), making it possible to identify the variable that needs to be cleared before retrying.

The last pattern is shown in Figure 4(c). Both variables `retval` and `err` might contain error codes at line 1. Thus a potential overwrite would be reported in line 2 when the error stored in `err` replaces that in `retval`. In this case, we can see that the programmer acknowledges that those variables might contain error codes before performing the assignment: the assignment occurs only if both variables are nonzero. We trust the programmer in this particular scenario and assume that overwriting the error code contained in `retval` is safe.

Most false positives arise from overwriting one error code with another error code without clear knowledge that an error may be overwritten. Unfortunately, there is no formal error hierarchy, which prevents us from automatically differentiating between correct and incorrect overwrites. We identify two unsafe patterns in which error codes are commonly overwritten. We find that 27 out of 44 false positives obey the pattern shown in Figure 5(a). In this case, only the variable `err` is acknowledged to be nonzero in line 1. We do not consider the overwrite in line 2 to be safe because it is not clear that the developer is aware of the overwrite. Our tool reports the potential bug and developers must determine its validity. The second unsafe pattern is shown in Figure 5(c). If both `ret` and `ret2` contain error codes at line 4, then `ret2` is overwritten in line 7. In this case, `ret` has precedence over `ret2`. We find that 9 out of the remaining 17 false positives fall into this category. We can recognize these patterns and mark these reports in the future. This would allow developers to prioritize the reports or skip certain categories altogether if considered safe. We call these false positives "removable." The 8 remaining false positives required significant human intervention to determine their safeness; we call these "unavoidable."

Bug category	CIFS TB	CIFS FP	CIFS T	ext3 TB	ext3 FP	ext3 T	ext4 TB	ext4 FP	ext4 T	IBM JFS TB	IBM JFS FP	IBM JFS T	ReiserFS TB	ReiserFS FP	ReiserFS T	VFS TB	VFS FP	VFS T
Overwritten	8	1+5	14	5	5+0	10	5	10+0	15	2	7+0	9	3	2+0	5	2	11+3	16
Out of scope	2	0+0	2	5	6+0	11	3	7+0	10	2	0+1	3	3	12+1	16	3	16+5	24
Unsaved	12	11+4	27	69	16+2	87	68	39+1	108	58	0+3	61	24	6+5	35	38	10+0	48
Total	22	12+9	43	79	27+2	108	76	56+1	133	62	7+4	73	30	20+6	56	43	37+8	88

Table 3. Summary results for the six case studies. Bug reports are broken down into overwritten, out-of-scope and unsaved. Each category is further divided into true bugs (TB) and false positives (FP). The first column under FPs corresponds to "removable" FPs (FPs that can be removed if our tool recognizes unsafe patterns). The second column corresponds to "unavoidable" FPs (FPs that cannot be automatically removed because significant human intervention is required). The last column (T) gives the total number of bug reports per bug category. Results for unsaved errors were produced in copy mode.

5.2 Out-of-Scope Bug Reports

Out-of-scope bugs are the least common. A total of 66 bug reports concern out-of-scope errors. Among these, 18 true bugs are identified. Figure 2(b) shows an out-of-scope bug found in IBM JFS. Most of these bugs relate to ignoring I/O errors. We identify four recurring safe patterns for out-of-scope errors, of which three are variants of those discussed in Section 5.1.

The first pattern appears in CIFS, ext4 and IBM JFS. This pattern is similar to that shown in Figure 4(a), however if the variable holds a specific error code, then zero or a different error code is returned in line 3, i.e., there is a return statement instead of an assignment. We also trust the programmer in this case and `err` is not reported to go out of scope at this line.

The second pattern, shown in Figure 4(d), appears in ext3 and ext4. Without recognizing this pattern, our tool would report that `err` is out of scope in line 6. This is not the case when `err` is cleared before retrying.

The third pattern has already been shown in Figure 4(b), however there is a subtle difference. In this case, `reiserfs_warning` takes the variable that is about to go out of scope as parameter. As a general approach for this pattern, we clear any variable that is passed as parameter to `reiserfs_warning` and similar functions.

The fourth pattern concerns *error transformation*: changes in how errors are represented as they cross software layers. Integer error codes may pass through structure fields, be cast into other types, be transformed into null pointers, and so on. Our analysis does not track errors across all of these representations. As a result, error codes are not propagated when transformed, yielding out-of-scope false positives. We also find that transformation from integers to pointers predominates. This transformation uses the `ERR_PTR` macro, which takes the error to be transformed as parameter. As in the case of functions such as `reiserfs_warning`, we clear any variable that is passed as parameter to `ERR_PTR`.

Ad hoc error precedence is the main source of false positives. Figure 5(b) presents one example. Both `ret` and `err` may be assigned error codes in lines 6 and 7, respectively. Variable `ret` is propagated regardless the contents of `err`, unless it does not contain an error code, i.e., `ret` has precedence over `err`. Our tool produces an out-of-scope report for `err` in line 12. This could be a bug or not depending on the context. We find that 41 out of 48 false positives exhibit this pattern. We can recognize this pattern to provide more information in the future. As for overwrites, the "false" positives here are not indications of analysis imprecision, but rather are based on a human expert's judgment that some errors can fall out of scope safely.

5.3 Unsaved Bug Reports

Unsaved bugs predominate in all five file systems. Developers and our local expert identify 269 true bugs among 366 unsaved bug reports in copy mode. Transfer mode produces 48% fewer false positives but misses 33% of the true bugs found in copy mode. The most common unsaved error code is `EIO`, followed by `ENOSPC` and `ENOMEM`. Figure 2(c) shows an unsaved bug found in ext3.

File system	KLOC	Analysis time (h:mm:ss) Phase 1	Phase 2	Phase 3	Total
CIFS	91	0:00:12	1:14:18	0:01:10	1:15:40
ext3	83	0:00:12	1:37:00	0:01:06	1:38:18
ext4	97	0:00:13	3:36:54	0:01:22	3:38:29
IBM JFS	93	0:00:12	1:16:01	0:01:00	1:17:13
ReiserFS	88	0:00:12	2:28:59	0:01:19	2:30:30

Table 4. Analysis performance. KLOC gives the size of each file system in thousands of lines of code, including 60 KLOC of shared VFS code.

Close inspection reveals serious inconsistencies in use of some functions' return values. For example, we find one function whose returned error code is unsaved at 35 call sites, but saved at 17 others. In this particular example, 9 out of the 35 bad calls are true bugs; the rest are false positives. When we alerted developers, some suggested they could use annotations to explicitly mark cases where error codes are intentionally ignored.

The main source of false positives concerns *error paths*: paths along which an error is already being returned, so other errors may be safely ignored. The second most common source of false positives is due to the fact there is another way to detect the problem, which we term *redundant error reporting*. Figure 5(d) shows an example from ReiserFS. The `sync_dirty_buffer` call in line 4 may return an error code, but checking its parameter `tbh` in line 7 is sufficient in this case. However, it is still possible for a more specific error to be dropped leading to loss of information about what exactly went wrong. A few false positives arise when callers know that the callee returns an error code only if certain preconditions are not met. Callers that have already established those preconditions know that success is assured and therefore ignore the return value.

We find that error paths and redundant error reporting describe 82 out of 97 false positives. We consider these removable since we can recognize these unsafe patterns and provide additional information for the developer to decide about their safety. The remaining 15 unavoidable false positives correspond to met preconditions.

5.4 Performance

Our experiments use a dual 3.2 GHz Intel processor workstation with 2.96 GB RAM. We divide the analysis into three phases: (1) extracting a textual weighted pushdown representation of the kernel code, (2) solving the poststar query, and (3) finding bugs and traversing the witness information to produce diagnostic output.

Table 4 shows the sizes and the time required to analyze each file system. While turnaround time is not fast enough for interactive use, the analysis is clearly suitable for regular use by kernel developers using widely-available hardware. Extracting the pushdown system (phase 1) is quite cheap, as is traversing witnesses to produce diagnostic output (phase 3). Solving the poststar query (phase 2) is the most costly operation, and is also the operation that is most generic to WPDSs. Thus, future improvements to this phase will further boost both the performance of this specific tool as well as any number of other WPDS-based analyses. Furthermore, this phase involves the application of hundreds of thousands of semiring operations. Operation performance is given by the weight domain representation. The BDD representation is known for already providing a highly efficient implementation, however BDD performance is notoriously fickle; additional tuning or alternate implementations may yield substantial improvements.

5.5 Other File Systems

We have performed the analysis on 43 other Linux file systems and verified that it runs correctly. Together, these account for 250 thousand additional lines of kernel code (KLOC). However we have not identified recurring safe patterns, and therefore we refrain from reporting detailed results. Our implementation readily accommodates pattern changes with a modicum of OCaml coding.

We have also analyzed different Linux versions, and find that file system code evolves significantly in each release. This demonstrates that fixing this class of bugs is not a one-time operation. Rather, kernel developers need robust tools to ensure that existing error propagation bugs are fixed, and also that new bugs are not introduced as implementations change over time.

The NASA/JPL Laboratory for Reliable Software is currently using our implementation to check code in the Mars Science Laboratory. JPL builds upon the VxWorks real-time operating system, not Linux, but was able to tune the tool themselves without difficulty. To date our tool has found one error-propagation bug in "flying" code (code used for space missions):

> We've found one legitimate problem. . . . We call a non-void function (that can return some critical error codes) and don't assign the return value, dropping some nice things such as `EASSERT`, `EABOUND`, and `EEBADHDR` on the ground. We would have expected the compiler or [another code-checking tool] to catch that, actually. . . We're going to rerun on a big update to the code, soon. [10]

6. Related work

The problem of unchecked function return values is longstanding, and is seen as especially endemic in C due to the wide use of return values to indicate success or failure of system calls. LCLint statically checks for function calls whose return value is immediately discarded [6], but does not trace the flow of errors over extended paths. GCC 3.4 introduced a `warn_unused_result` annotation for functions whose return values should be checked, but again enforcement is limited to the call itself: storing the result in a variable that is never subsequently used is enough to satisfy GCC. Neither LCLint nor GCC analyzes deeply enough to uncover bugs along extended propagation chains.

It is tempting to blame this problem on C, and argue for structured exception handling instead. Language designs for exception management have been under consideration for decades [8, 23]. Setting aside the impracticality of reimplementing existing operating systems in new languages, static verification of proper exception management has its own difficulties. C++ exception throwing declarations are explicitly checked at run time only, not at compile time. Java's insistence that most exceptions be either caught or explicitly declared as thrown is controversial [29, 30]. Frustrated Java programmers are known to pacify the compiler by adding blanket `catch` clauses that catch and discard all possible exceptions. C# imposes no static validation; Sacramento et al. [26] found that 90% of relevant exceptions thrown by .NET assemblies (C# libraries) are undocumented. Thus, while exceptions change the error propagation problem in interesting ways, they certainly do not solve it.

Numerous proposals detect or monitor error propagation patterns at run time, typically during controlled in-house testing with fault-injection to elicit failures [4, 7, 9, 13–17, 28]. Work by Guo et al. [12] on dynamic abstract type inference could be used to distinguish error-carrying variables from ordinary integers, but this approach also requires running on real (error-inducing) inputs. In contrast to these dynamic techniques, our approach offers the stronger assurances of static analysis, which become especially important for critical software components such as operating system kernels. Storage errors are rare enough to be difficult to test dynamically, but can be catastrophic when they do occur. This is precisely the scenario in which intensive static analysis is most suitable.

Gunawi et al. [11] highlight error code propagation bugs in file systems as a special concern. Gunawi's proposed Error Detection and Propagation (EDP) analysis is essentially a type inference over the file system's call graph, classifying functions as generators, propagators, or terminators of error codes. Our approach uses a more precise analysis framework that offers flow- and context-sensitivity. The difference is not merely theoretical: we have compared the two in detail and while Gunawi's EDP finds 97% of our true unsaved bugs, it also produces 2.75 times more false positives. Furthermore, EDP finds no overwrites and just one of our true out-of-scope bugs. EDP runs faster, producing results in a matter of seconds. However, it does not produce detailed diagnostic information; WPDS witness traces (Section 4) offer a level of diagnostic feedback not possible with EDP's whole-function-classification approach.

Bigrigg and Vos [1] describe a dataflow analysis for detecting bugs in the propagation of errors in user applications. Their approach augments traditional def-use chains with intermediate check operations: correct propagation requires a check between each definition and subsequent use. This is similar to our tracking of error values from generation to eventual handling or accidental discarding. Bigrigg and Vos apply their analysis manually, whereas we have a working implementation that is interprocedural, context-sensitive, and has been applied to 516 thousand lines of kernel code.

The FiSC system of Yang et al. [32] uses software model checking to check for a number of file-system-specific bugs. Relative to our work, FiSC employs a richer (more domain-specific) model of file system behavior, including properties of on-disk representations. However, FiSC does not check for error propagation bugs and has been applied to only three of Linux's many file systems.

7. Conclusions

We have designed and implemented an interprocedural, flow- and context-sensitive static analysis for tracking the propagation of errors through file systems. Our approach is based on a novel over-approximating counterpart to copy constant propagation analysis, with additional specializations for our unusual problem domain. The analysis is encoded as a WPDS, and poststar queries on this system allow detailed diagnosis of a variety of error mismanagement bugs. We present six case studies, including four widely-used Linux file systems and one relatively new file system. We find 312 nontrivial bugs. False positives arise, but many of these can be ascribed to a small number of recurring unsafe patterns that should also be amenable to automated analysis; we identify several such patterns in our detailed case studies. We also find that the same patterns repeat among different file system implementations.

The unstructured nature of C error reporting creates a significant analysis challenge. Programmer intent is often implicit, and our findings show that current practice (manual inspection and testing) is insufficient. For good or ill, implementing operating systems in C is also part of the status quo, and this is unlikely to change soon. Analyses such as that we describe here can go a long way toward eliminating error propagation bugs. This increases the trustworthiness of file systems and, in turn, of computer systems as a whole.

Acknowledgments

We are grateful to the anonymous reviewers for their constructive suggestions, and to Nicholas Kidd, Akash Lal, and Thomas W. Reps for their invaluable WPDS support. We thank developers Andreas Dilger (ext4), Steve French (CIFS), David Kleikamp (JFS), Jeff Mahoney (ReiserFS), and Alex D. Groce (JPL) for their prompt and insightful feedback on the reports produced by our tool. The first author also wishes to thank the National Council on Science and Technology of Mexico and the Secretariat of Public Education for their financial support.

References

[1] M. W. Bigrigg and J. J. Vos. The set-check-use methodology for detecting error propagation failures in I/O routines. In *Workshop on Dependability Benchmarking*, Washington, DC, June 2002.

[2] R. E. Bryant. Binary decision diagrams and beyond: enabling technologies for formal verification. In R. L. Rudell, editor, *ICCAD*, pages 236–243. IEEE Computer Society, 1995.

[3] D. Callahan. The program summary graph and flow-sensitive interprocedural data flow analysis. In *PLDI*, pages 47–56, 1988.

[4] G. Candea, M. Delgado, M. Chen, and A. Fox. Automatic failure-path inference: A generic introspection technique for Internet applications. In *Proceedings of the The Third IEEE Workshop on Internet Applications (WIAPP '03)*, pages 132–141, San Jose, California, June 2003. IEEE.

[5] A. Dilger. Error propagation bugs in ext4. Personal communication, Nov. 2008.

[6] D. Evans. *LCLint User's Guide*. University of Virginia, May 2000.

[7] C. A. Flanagan and M. Burrows. System and method for dynamically detecting unchecked error condition values in computer programs. United States Patent #6,378,081 B1, Apr. 2002.

[8] J. B. Goodenough. Structured exception handling. In *POPL*, pages 204–224, 1975.

[9] T. Goradia. Dynamic impact analysis: A cost-effective technique to enforce error-propagation. In *ISSTA*, pages 171–181, 1993.

[10] A. D. Groce. Problem solved. Personal communication, Jan. 2009.

[11] H. S. Gunawi, C. Rubio-González, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and B. Liblit. EIO: Error handling is occasionally correct. In *6th USENIX Conference on File and Storage Technologies (FAST '08)*, San Jose, California, Feb. 2008.

[12] P. J. Guo, J. H. Perkins, S. McCamant, and M. D. Ernst. Dynamic inference of abstract types. In L. L. Pollock and M. Pezzè, editors, *ISSTA*, pages 255–265. ACM, 2006.

[13] M. Hiller, A. Jhumka, and N. Suri. An approach for analysing the propagation of data errors in software. In *DSN*, pages 161–172. IEEE Computer Society, 2001.

[14] M. Hiller, A. Jhumka, and N. Suri. Propane: an environment for examining the propagation of errors in software. In *ISSTA*, pages 81–85, 2002.

[15] M. Hiller, A. Jhumka, and N. Suri. Epic: Profiling the propagation and effect of data errors in software. *IEEE Trans. Computers*, 53(5): 512–530, 2004.

[16] A. Jhumka, M. Hiller, and N. Suri. Assessing inter-modular error propagation in distributed software. In *SRDS*, pages 152–161. IEEE Computer Society, 2001.

[17] A. Johansson and N. Suri. Error propagation profiling of operating systems. In *DSN*, pages 86–95. IEEE Computer Society, 2005.

[18] N. Kidd, T. Reps, and A. Lal. WALi: A C++ library for weighted pushdown systems. `http://www.cs.wisc.edu/wpis/wpds/download.php`, 2008.

[19] A. Lal, T. W. Reps, and G. Balakrishnan. Extended weighted pushdown systems. In K. Etessami and S. K. Rajamani, editors, *CAV*, volume 3576 of *Lecture Notes in Computer Science*, pages 434–448. Springer, 2005.

[20] A. Lal, N. Kidd, T. W. Reps, and T. Touili. Abstract error projection. In H. R. Nielson and G. Filé, editors, *SAS*, volume 4634 of *Lecture Notes in Computer Science*, pages 200–217. Springer, 2007.

[21] A. Lal, T. Touili, N. Kidd, and T. Reps. Interprocedural analysis of concurrent programs under a context bound. Technical Report 1598, University of Wisconsin–Madison, July 2007.

[22] J. Lind-Nielsen. BuDDy - A Binary Decision Diagram Package. http://sourceforge.net/projects/buddy, 2004.

[23] B. Liskov. A history of CLU. In *HOPL Preprints*, pages 133–147, 1993.

[24] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In R. N. Horspool, editor, *CC*, volume 2304 of *Lecture Notes in Computer Science*, pages 213–228. Springer, 2002.

[25] T. W. Reps, S. Schwoon, S. Jha, and D. Melski. Weighted pushdown systems and their application to interprocedural dataflow analysis. *Sci. Comput. Program.*, 58(1-2):206–263, 2005.

[26] P. Sacramento, B. Cabral, and P. Marques. Unchecked exceptions: Can the programmer be trusted to document exceptions? In *Second International Conference on Innovative Views of .NET Technologies*, Florianópolis, Brazil, Oct. 2006. Microsoft.

[27] S. Schwoon. *Model-Checking Pushdown Systems*. PhD thesis, Technical Univ. of Munich, Munich, Germany, July 2002.

[28] K. G. Shin and T.-H. Lin. Modeling and measurement of error propagation in a multimodule computing system. *IEEE Trans. Computers*, 37(9):1053–1066, 1988.

[29] Sun Microsystems, Inc. Unchecked exceptions – the controversy. `http://java.sun.com/docs/books/tutorial/essential/exceptions/runtime.html`, Aug. 2007.

[30] M. van Dooren and E. Steegmans. Combining the robustness of checked exceptions with the flexibility of unchecked exceptions using anchored exception declarations. In R. Johnson and R. P. Gabriel, editors, *OOPSLA*, pages 455–471. ACM, 2005.

[31] M. N. Wegman and F. K. Zadeck. Constant propagation with conditional branches. In *POPL*, pages 291–299, 1985.

[32] J. Yang, P. Twohey, D. R. Engler, and M. Musuvathi. Using model checking to find serious file system errors. *ACM Trans. Comput. Syst.*, 24(4):393–423, 2006.