Improving Static Analysis for File Systems

Cindy Rubio González
CS 736 Project, University of Wisconsin–Madison
Preliminary results of this work were submitted for publication on November 15th, 2007. Coauthors of this work include: Haryadi S. Gunawi, Ben Liblit, Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau.

Abstract

Unchecked errors are problematic in general, but 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 data corruption from which recovery is difficult or impossible. 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 errors. Analysis of numerous Linux file system implementations uncovers numerous error propagation bugs. Our flow- and context-sensitive approach produces more precise results than related techniques while providing the programmer with 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, copy constant propagation, weighted pushdown systems, binary decision diagrams

1. Introduction

Run-time errors are unavoidable whenever software interacts with the physical world. Incorrect handling of errors is a longstanding problem in many application domains, but is especially troubling when it affects the file-management code of operating systems. 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. (Even the most skilled programmer cannot prevent low-level errors from occurring when hot coffee is spilled onto a laptop, or when an adventurous gopher chews through a fiber-optic cable.)

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, file systems are not so exotic that this problem can simply be fixed and forgotten. Because file systems play such a critical supporting role, there is strong motivation to specialize them for different purposes or with different assumed priorities. Thus, implementations proliferate. Well-known Unix file systems include ext2 [6], ext3 [36] (commonly the default Linux file system), ReiserFS [27], IBM's JFS [2], SGI's XFS [35], Sun's ZFS [33], Apple's HFS [1], AFS [16], NFS [30], and others, with more constantly appearing. Approaches vary in terms of performance; scalability; reliability; consistency models; and special features such as reconfigurability, networked operation, or search functionality. Linux alone includes dozens of different file system implementations. There is no reason to believe that file system designers are running out of ideas or that the technological changes which often 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 approaches are unlikely to bear fruit. Therefore, we propose a static program 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 [38], 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,

PLDI 2008 June 7–13, Tucson, Arizona

for example, include detailed witness traces that illustrate the error-fumbling steps a file system could take.

The remainder of this paper is organized as follows. Section 2 provides additional background on error management in the Linux kernel; file systems in this kernel serve as the experimental focus of our research. Section 3 presents the error propagation analysis in detail, including the encoding of this dataflow problem as a weighted pushdown system. In Section 4, we show how to extract specific error reports and detailed diagnostic information from the raw analysis results. Section 5 discusses our experimental results on 48 Linux file systems. We consider related work in Section 7, and Section 8 concludes.

2. Error Management in the Linux Kernel

In this paper we focus on file system implementations found in the Linux 2.6.15.4 kernel. Our approach uses a mixture of generic program analysis techniques and specializations for the idiomatic style of code used in Linux. Other operating systems share the same general style, although some details may differ.

2.1 Integer Error Codes

Different kinds of failure occur, and each requires different fault management. For example, in the case of an input/output error, the `EIO` error code could be generated, and a routine that receives the error code might abort a failed transaction, schedule it for later retry, release allocated buffers to prevent memory leaks, and so on. In the case of a memory shortage, the `ENOMEM` error code is raised, signaling that the system must release some memory in order to continue the operation. In the case of a full disk quota, `ENOSPC` is propagated across many file system routines to prevent new allocations.

Unfortunately, Linux (like many operating systems) is written in C, a language 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, and 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. This encourages overloading the return values of function calls. 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. A careful caller must check for negative return values and propagate or handle errors that arise. It is important to remember that error codes are merely integers given special meaning through coding conventions. Any `int` 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. Error propagation chains can be long, crossing many functions, modules, and software layers. If buggy code breaks this chain, higher layers receive incorrect reports regarding the outcome 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.

```
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

2.3 Distinguishing Checked from Unchecked Errors

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 which 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 which can be extended easily as necessary.

Figure 1 shows a typical fragment of Linux kernel code. Many error-handling routines eventually call `printk`, an error-logging function. Furthermore, the error code being handled is often passed as an argument in the `printk` call. Because calling `printk` is an explicit action taken by a programmer, it is reasonable to assume that the programmer is aware of the error and is handling it appropriately. Thus, while `status` may have contained an unchecked error code before the call to `printk` on line 3, we can safely assume that from the `printk` call forward, the error has been checked and is being handled. If `status` contained an unchecked error on line 2, then it contains a checked error on line 4.

Because error codes are passed as negative integers (such as `-EIO` for -5), sign-checking such as that on line 2 is common. If the condition is false, then `status` must be non-negative and therefore cannot contain an error code on line 6. When paths merge at line 8, `status` may contain a checked error or no error, but it 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 programmer guidance or 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 that marks them as taken care of. Requiring explicit programmer action to mark errors as checked would improve diagnosis by avoiding the silent propagation failures that presently occur.

3. The Error Propagation Analysis

The first task is to determine, at each program point, the set of unchecked error codes each variable might contain. Given this in-

formation, the bugs described in Section 2 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 and related constructs propagate unchecked errors forward from one location (variable) to another. Propagation ends when a variable goes out of scope or is correctly checked by error-handling code.

This problem is reminiscent of copy constant propagation [38]. However, copy constant propagation identifies the *one* constant value that a variable *must* contain (if any), whereas we identify the *set* of error code constants that a variable *may* contain. Copy constant propagation drives replacement of variables with constants in optimizing compilers, and therefore must err on the side of under-approximation. We use error propagation analysis to drive bug reporting, and therefore we prefer over-approximate so that no possible bug is overlooked.

3.1 Weighted Pushdown Systems

We use weighted pushdown systems [28] to formulate and solve the error propagation dataflow problem. A weighted pushdown system is a pushdown system that has a weight associated with each rule. These weights can be thought of as transfer functions that describe the effect of each statement on the state of the program. Such weights must be elements of a set that satisfies the bounded idempotent semiring properties. We now formally define weighted pushdown systems and related terms; Section 3.2 shows how weighted pushdown systems can be applied to solve the error propagation dataflow problem.

Definition 1. *A* **pushdown system** *is a triple $\mathcal{P} = (P, \Gamma, \Delta)$ where P and Γ are finite sets called the* **control locations** *and the* **stack alphabet***, respectively. A* **configuration** *of $\mathcal{P}$ is a pair $\langle p, w\rangle$, where $p \in P$ and $w \in \Gamma^*$. Δ contains a finite number of rules of the form $\langle p, \gamma\rangle \hookrightarrow \langle p', w\rangle$, where $p, p' \in P$, $\gamma \in \Gamma$, and $w \in \Gamma^*$, which define a transition relation $\Rightarrow$ between configurations of $\mathcal{P}$ 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. [22] and Reps et al. [28], 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, $\oplus$, $\otimes$, 0, 1), where D is a set, 0 and 1 are elements of D, and $\oplus$ (the combine operator) and $\otimes$ (the extend operator) are binary operators on D such that*

1. *$(D, \oplus)$ is a commutative monoid with 0 as its neutral element, and where $\oplus$ is idempotent (i.e., for all $a \in D$, $a \oplus a = a$).*
2. *$(D, \otimes)$ is a monoid with the neutral element 1.*
3. *$\otimes$ distributes over $\oplus$, i.e., for all $a, b, c \in D$ we have $a \otimes (b \oplus c) = (a \otimes b) \oplus (a \otimes c)$ and $(a \oplus b) \otimes c = (a \otimes c) \oplus (b \otimes c)$.*
4. *0 is an annihilator with respect to $\otimes$, i.e., for all $a \in D$, $a \otimes 0 = 0 = 0 \otimes a$.*
5. *In the partial order $\sqsubseteq$ defined by: $\forall a, b \in D$, $a \sqsubseteq b$ iff $a \oplus b = a$, there are no infinite descending chains.*

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 compute the weight of a set of paths that merge.

Property 1 in the above definition ensures that the order in which we merge paths is not relevant. Property 2 requires the existence of a weight that has no effect on the state of the program, i.e., an identity function. Property 3 allows exploring program paths in a more efficient manner. Property 5 is required in order to guarantee termination.

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

Let $\sigma = [r_1, \ldots, r_k]$ be a sequence of rules (a path in the CFG) from Δ^*. A value can be associated with σ by using function f. This value is defined as $v(\sigma) = f(r_1) \otimes \cdots \otimes f(r_k)$. In addition, for any configurations c and c' of $\mathcal{P}$, $path(c, c')$ denotes the set of all rule sequences $[r_1, \ldots, r_k]$, i.e., the set of all paths, that transform c into c'.

Definition 4. *Let $\mathcal{W} = (\mathcal{P}, \mathcal{S}, f)$ be a weighted pushdown system, where $\mathcal{P} = (P, \Gamma, \Delta)$, and let $C \subseteq P \times \Gamma^*$ be a regular set of configurations. The* **generalized pushdown successor problem** *is to find for each $c \in P \times \Gamma^*$:*

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

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

The meet over all paths value is the best possible solution to a static dataflow problem. Thus, a weighted pushdown system is a useful dataflow engine for problems that can be encoded with suitable weight domains. It has been shown how dataflow problems such as copy constant propagation can be encoded as weight domains [28]. In Section 3.2 we show how the error propagation problem can also be encoded as a weight domain. Our approach also makes extensive use of witness sets to provide programmers with detailed diagnostic traces for each potential bug; details of this process appear in Section 4.

In order to handle local variables properly, we use the extension to weighted pushdown systems proposed by Lal et al. [22]. This extension requires the definition of a merge function. A merge function 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 that 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

As discussed in Section 3.1, the weighted pushdown system consists of three main components: a pushdown system, a bounded idempotent semiring, and a mapping from pushdown system rules to associated weights. We now define these components for a weighted pushdown system that encodes the error propagation dataflow problem.

3.2.1 Pushdown System

We model the control flow of the program with a pushdown system following the standard approach described by Lal et al. [23]. Let P contain a single state $\{p\}$. Γ corresponds to program statement locations, and Δ corresponds to edges of the interprocedural CFG. Table 1 shows the PDS rule for each type of CFG edge.

Rule	Control flow modeled
$\langle p, u\rangle \hookrightarrow \langle p, v\rangle$	CFG edge $u \to v$, which is not a call
$\langle p, c\rangle \hookrightarrow \langle p, f_{enter} r\rangle$	CFG edge $c \to r$, which calls procedure f beginning at node f_{enter}
$\langle p, f_{exit}\rangle \hookrightarrow \langle p, \epsilon\rangle$	Return from procedure f at f_{exit}

Table 1. The encoding of control flow as PDS rules

3.2.2 Bounded Idempotent Semiring

We classify integer constants into two categories: *error* constants and *non-error* constants. Let $\mathcal{E}$ be the set of error constants, such as the values -1 through -34 in Linux. 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 $\mathcal{C} = \mathcal{E} \cup \{OK, uninitialized\}$ be the set of all constants. Finally, let $\mathcal{V}$ be the set of all program variables.

Let $\mathcal{S} = (D, \oplus, \otimes, 0, 1)$ be a bounded idempotent semiring as in Definition 2. Elements of D are drawn from $\mathcal{V} \to 2^{\mathcal{V} \cup \mathcal{C}}$, so each weight in D is a mapping from variables to sets containing variables, error values, OK and/or $uninitialized$. This is interpreted as giving 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 of the values it could have mapped to in either of the weights being combined. For all $w_1, w_2 \in D$ and all $v \in \mathcal{V}$:

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

The extend operator is also applied component-wise:

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

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.

The weight 1 is defined as $\{(v, \{v\}) \mid v \in \mathcal{V}\}$, which maps each variable to the set containing only itself. This weight can be seen as a power set generalization of the identity function. We define weight 0 as $\{(v, \emptyset) \mid v \in \mathcal{V}\}$, 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 $v \in \mathcal{V}$,

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

This has the effect of propagating 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 generally describe the transfer function as being associated with a specific statement. The corresponding WPDS rule weight is associated with edge from that statement to its unique successor. Conditionals have multiple outgoing edges and therefore will require multiple transfer functions.

Assignment statements. We consider two types of assignments as explained below. We leave the discussion of assignments involving function calls such as $v = f()$ for later in this section.

Simple assignments. These are assignments of the form $v = e$, where $e \in \mathcal{V} \cup \mathcal{C}$. Let $Ident$ be the function that maps each variable to the set containing itself. The transfer function for a simple assignment is then $Ident[v \mapsto \{e\}]$. In other words, v must have the value of e after this assignment, while all other variables still have whatever values they had before the assignment.

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 $e_1, e_2 \in \mathcal{V} \cup \mathcal{C}$ and op is a binary arithmetic or bitwise operator (`+`, `&`, `<<`, etc.). Define $Vars$ as the set of variables appearing on the right side of the assignment: $Vars \equiv \{e_1, e_2\} \cap \mathcal{V}$. Although error codes are represented as integers they are conceptually 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 $\mathcal{V}$ 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 $u \in Vars \cup \{v\}]$.

Consider instead an assignment of the form $v = e_1 \, op \, e_2$ where $e_1, e_2 \in \mathcal{V} \cup \mathcal{C}$ 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, we can assume that the result of the comparison, assigned to v, is a non-error. Therefore, the transfer function for this assignment is $Ident[v \mapsto OK]$.

Assignments involving unary operators ($v = op \, e_1$) are treated similarly: 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.

Conditional statements. We assume that conditional statements with compound 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 $v \in \mathcal{V}$. The transfer functions associated with the $true$ and $false$ branches are $Ident$ and $Ident[v \mapsto \{OK\}]$, respectively. 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, and thus we simply apply the identity function. On the other hand, the $false$ branch is selected when v is equal to zero. Because zero is never an error code, this tells us that v definitely does not contain an error value. In this case the $Ident[v \mapsto \{OK\}]$ transfer function reflects the new knowledge obtained about v while keeping all other variables unchanged.

Conversely, consider conditional statements of the form $if(!\, v)$, $if(v > 0)$, $if(v \geq 0)$, $if(0 < v)$, $if(0 \leq v)$, $if(v == 0)$ and $if(0 == v)$. 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 value 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 not match any of the patterns discussed above, we simply associate $Ident$ with both $true$ and $false$ branches. An example of such a pattern is $if(v_1 < v_2)$, where $v_1, v_2 \in \mathcal{V}$.

Function calls. We adopt the convention used by Reps et al. [28] 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 all function-terminating statements (e.g., `return` statements or last-block fall-through statements) have a newly-introduced per-function exit node as their unique successors. We use these dummy entry and exit nodes to manage data transfer between callers and callees, as discussed below.

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. As before, the rest of the nodes represent statements and conditions in the program.

Local variable initialization. First consider a call to a void function that takes no parameters. Let $\mathcal{L} \subseteq \mathcal{V}$ be the set of local variables and $\mathcal{G} \subseteq \mathcal{V}$ be the set of global variables in the program. Recall that transfer functions are associated with edges in the CFG. 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 $v \in \mathcal{L}]$. 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 new exchange variable is introduced for each function parameter. For example, consider a function F with formal parameters f_1 and f_2. Let $F(a_1, a_2)$ be a function call to F with actual parameters a_1 and a_2, where $a_1, a_2 \in \mathcal{V} \cup \mathcal{C}$. We introduce two global exchange variables named $F\$1$ and $F\$2$. The transfer function on the call-to-enter edge is $Ident[F\$i \mapsto \{a_i\}$ for $0 < i \leq$ number of parameters], as though each actual argument were copied into the corresponding global exchange variable. Along 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 $v \in \mathcal{L}][f_i \mapsto \{F_i\}$ for $0 < i \leq$ number of parameters], as though each formal argument were initialized with a value from the corresponding global exchange variable. Other local variables are uninitialized as before. Thus, argument passing is modeled a two-step process: first the caller copies its arguments into global exchange variables, then the callee copies from the global exchange variables into its formal parameters.

Return value passing. Lastly, suppose that function F is a non-void function. The transfer functions associated with the call-to-enter and enter-to-first-statement edges remain as given above. For the edge connecting each return node to the dummy exit node, we use the transfer function $Ident[F\$return \mapsto \{r\}]$ where $F\$return$ is a global exchange variable and $r \in \mathcal{V} \cup \mathcal{C}$ is the value being returned. The exit-to-return-site edge has the transfer function $Ident[v \mapsto \{F\$return\}]$ where $v \in \mathcal{V}$ is the variable (if any) receiving the return value in the caller. For void functions, this transfer function is just the identity function.

Other interprocedural issues. We consider functions whose implementation is not available to not have any effect on the state of the program, thus we simply apply the identity function on calls to such functions. This could easily be augmented to declare some external functions as error-code-returning via explicit programmer annotations. 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 it gives useful results in practice. We find a common pattern in the use of pointers to integer variables used to hold error values. Many functions take a pointer to the 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 indistinguishable from call-by-copy-return parameters. On the interprocedural call-to-enter edge, we copy pointed-to values from the callee into the caller, 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 either statically or via assignments of the form "`file_ops->write = ntfs_file_write`" where `ntfs_file_write` names a function, not another function pointer. In either case, 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, first 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 $Ident$.

printk. 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 variable-length parameter list function whose first parameter is always a format string. The transfer function for such a call is $Ident[v \mapsto OK$ for v in the actual `int`-typed arguments to `printk`$]$.

4. Finding and Describing Bugs

Ideally, 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. Thus, 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. We further discuss each of these scenarios below.

Overwritten. An unchecked error is overwritten when the variable that contains it is assigned another value. The new value can be anything, including another error value. If the left side of any assignment may already contain an error, then that assignment is invalid and will produce a diagnostic message from our tool. However, we explicitly allow an error value to be overwritten with itself. For example, $x =$ `EIO` is allowed if x can already have the value

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

(a) Example code

```
example.c:11: "status" receives an error from "TENTATIVE_EIO"
example.c:12: "status" may have an unchecked error
example.c:14: "status" may have an unchecked error
example.c:4: "status" may have an unchecked error
example.c:5: "status" may have an unchecked error
example.c:14: "status" may have an unchecked error
example.c:15: "result" receives an error from "status"
example.c:17: "result" may have an unchecked error
example.c:4: "result" may have an unchecked error
example.c:5: "result" may have an unchecked error
example.c:17: "result" may have an unchecked error
example.c:18: overwriting potential unchecked error in "result"
```

(b) Complete diagnostic path trace

```
example.c:11: "status" receives an error from "TENTATIVE_EIO"
example.c:15: "result" receives an error from "status"
example.c:18: overwriting potential unchecked error in "result"
```

(c) Diagnostic path slice

Figure 2. Example code fragment and corresponding diagnostic output

EIO or OK, but is flag as invalid if x can already have any other error value. We open this loophole because we find that this is a commonly-occurring pattern universally judged to be acceptable by our operating-systems domain experts.

Out of scope. Another common scenario in which unchecked errors are lost is when variables go out of scope. In order to detect this problem, we insert assignment statements at the end of each function that 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.

Unsaved return values. 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.

4.1 Querying the Weighted Pushdown System

We perform a poststar query [28] on the weighted pushdown system, with the beginning of the program as the starting configuration. A weighted automaton is obtained as the result. In order to read out weights from this automaton, we apply the *path_summary* algorithm of Lal et al. [21]. 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 are then able to retrieve the weight for any specific point in the program, which is the weight from the beginning of the program to that particular point.

We retrieve the associated weight w for each assignment p. It is important to note that w does not include the effect of the assignment found at p itself. Our goal is to find whether the assignment at p may overwrite an error value. Let $S, T \subseteq \mathcal{C}$ respectively be the sets of possible constant values held by the source and target of the assignment, as revealed by w. Then:

1. If $T \cap \mathcal{E} = \emptyset$, then the assignment cannot overwrite any error code and need not be examined further.
2. If $T \cap \mathcal{E} = S = \{e\}$ for some single error code e, then the assignment can only overwrite an error code with the same code. As noted above, we explicitly allow this.
3. Otherwise, it is possible that this assignment will overwrite an unchecked error code with a different code. Such an assignment is incorrect, and will be presented to the programmer along with suitable diagnostic information.

We report at most one overwritten error for each assignment; we do not report every error value that might be overwritten. This can sometimes be a source of imprecision. For example, the instance chosen to be reported may actually be a false positive, fooling the programmer into believing that no real problem exists. However, a different error value potentially overwritten by the same assignment may be a true bug. On the other hand, reporting all possibly-overwritten error values might overwhelm the programmer with seemingly-redundant output.

4.2 Witnesses, Paths, and Slices

Weighted pushdown systems 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 can 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 keep track of 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 point at which the error was originated. 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. $t = x$ 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 an error value from x."
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 the error value e."
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 may contain an error value."

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 static 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.

Figure 2(a) shows an example code fragment that contains some error propagation bugs. Figure 2(b) shows a complete diagnostic path trace for one bug. Observe that this trace begins in function `load` but traverses into the `nextId` twice (lines 4 and 5) while traveling from the error code generation point (line 11) to the overwriting assignment (line 18). Figure 2(c) shows the diagnostic path slice which includes only those lines directly relevant to the error. Here we see just three events of interest: the generation of an error code in `status` on line 11, the transfer of that error code from `status` to `result` on line 15, and the overwriting assignment to `result` on line 18.

5. Experimental Evaluation

Our implementation uses the CIL C front end [26] to apply preliminary source-to-source transformations on Linux kernel code, then traverse the CFG and emit a textual representation of the weighted pushdown system. We use the WALi WPDS library [20] to perform the interprocedural dataflow analysis on this weighted pushdown system. Within the WALi-based analysis code, we encode weights (transfer functions) using *binary decision diagrams* (BDDs) [4] as implemented by the BuDDy BDD library [24]. BDDs have been used before to encode weight domains [31]. The BDD representation allows us to perform key semiring operations, such as extend and combine, in a highly efficient manner..

We analyze 48 Linux file system implementations. We omit the SGI-donated XFS file system from our study, as it uses error codes quite unlike those found elsewhere in Linux. We also exclude the hppfs, devfs, debugfs, hotfs and openpromfs file systems since they do not use the 34 basic error codes that we analyze.

Table 2 summarizes our findings. We find possible error propagation bugs in 37 out of 48 file systems, or 77% of the file systems analyzed. Among the possible bugs, out-of-scope errors dominate.

FS	Ov	Sc	Un	Total	KLOC
ext3	16	44	47	107	12
ReiserFS	8	43	28	79	24
IBM JFS	1	23	45	69	17
CIFS	18	37	12	67	21
ext2	13	15	11	39	6
SMB	13	14	9	36	6
Apple HFS+	1	17	18	36	7
JFFS v2	4	23	7	34	11
Apple HFS	1	22	11	34	5
NCP	9	19	3	31	5
Boot FS	0	18	10	28	1
Plan 9	3	17	7	27	4
procfs sup	2	19	1	22	6
AFS	2	11	6	19	7
UDF	5	7	4	16	9
FAT	3	11	2	16	4
Coda	3	12	1	16	3
ADFS	5	7	3	15	2
NFS Client	2	6	6	14	18
ISO	1	11	0	12	3
FUSE	0	10	1	11	3
Automounter	3	8	0	11	2
sysfs sup	0	10	0	10	2
Automounter4	3	7	0	10	2
Amiga FFS	0	8	1	9	3
JFFS	0	4	4	8	5
UFS	3	4	0	7	5
OS/2 HPFS	0	6	0	6	6
MSDOS	2	2	0	4	1
System V	0	1	2	3	3
Minix	0	1	2	3	4
VFAT	0	2	0	2	1
romfs sup	1	1	0	2	1
NFS Lockd	0	1	1	2	4
HugeTLB	0	2	0	2	1
EFS	0	1	0	1	1
BeOS	0	1	0	1	3
Relayfs	0	0	0	0	1
ramfs sup	0	0	0	0	1
QNX 4	0	0	0	0	2
Partitions	0	0	0	0	4
NTFS	0	0	0	0	18
NFS Server	0	0	0	0	14
NFS ACL Prot.	0	0	0	0	20
Free VxFS	0	0	0	0	2
exportfs sup.	0	0	0	0	1
devpts	0	0	0	0	1
Compr. ROM	0	0	0	0	1
Total	122	445	242	809	283

Table 2. Number of bugs found, categorized into overwritten (Ov), out of scope (Sc), and unsaved return values (Un). KLOC gives the size of each file system implementation in thousands of lines of code. File systems are sorted by total number of bugs found.

5.1 Bugs found

Here we give a few examples of true bugs found by our tool.

Example 1: Figure 3 shows a fragment of ext3 file system code in which a bug is found. An instance of an unsaved error can be found on line 3. Callers to `ext3_free_branches` have no way of knowing whether the operation succeeded or failed. We typically find that these callers cannot do anything in the event of

```
1  static void ext3_free_branches(...) {
2    ...
3    if (is_handle_aborted(handle))
4      return;
5    ...
6  }
```

Figure 3. An example of a bug found by the tool

a failure, and therefore they assume that `ext3_free_branches` always succeeds. This silent failure is clearly a problem, as callers may continue with other operations as though nothing bad had happened.

Example 2: Our analysis uncovers an interesting and potentially harmful bug in the ISO file system. If there is a read failure during a directory entry lookup, then the error is ignored, leading to a lookup failure. In this case, the error is saved but later overwritten without being checked. This situation could be problematic in the presence of a transient I/O failure as follows: if subdirectory `/tmp/dir` already exists, but `ls /tmp/dir` fails due to a transient read failure, then `mkdir /tmp/dir` will create a second directory with the same name as the one that already exists. The end result is a corrupted file system with two `/tmp/dir` subdirectories.

Example 3: Our third example also affects the ext3 file system. Function `void ext3_xattr_cache_insert` includes comments claiming that it "`Returns 0, or a negative number on failure.`" However, this is a void function. Our tool identifies a local variable within this function which may contain an unchecked error, but which goes out of scope instead of being propagated to callers. Neighboring functions have similar comments and they do return `int` error values. Clearly this function should return an `int` error value as well. Of course, this change requires corresponding changes in callers to save and either handle or propagate the returned error. Our tool can support this redesign by checking whether callers have been updated correctly.

5.2 False Positives

We have manually inspected the 122 overwritten errors reported as well as 100 of the out-of-scope errors found. We examined the path or slice provided for each of these reported errors. The inspection of overwritten errors took place in the presence of a kernel expert (and coauthor) who judged each reported issue as a true bug or a false positive.

Figure 4 shows a graphical representation of the total number of overwritten errors reported for 24 different file system implementations (implementations for which no overwritten bugs were reported are ommitted). These same numbers can also be found in Table 2, under the `Ov` column. The total number of bugs is further divided into true bugs and false positives. As an example, consider the file system implementation CIFS. A total of 18 overwritten bugs were found in this file system, however only 8 of them are true bugs while the remaining 10 are false positives. A grand total of 180 overwritten errors are found accross all the analyzed file systems. We find that 11% of these bugs are true bugs (20) whereas the remaining 89% (160) are false positives. Only a subset of the out-of-scope errors were manually inspected. We find that 43% of these bugs were false positives falling into a common pattern.

In the case of overwritten errors, false positives arise due to either idiomatic programming styles found in the kernel code or the existence of contexts in which overwriting an unchecked error becomes legal. As we discuss below, these scenarios represent a challenge to any static analysis technique. On the other hand, we find that the most of the out-of-scope false positives found are due

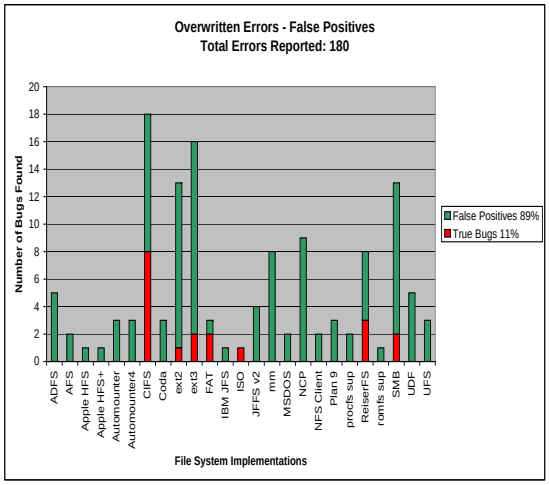

Figure 4. A graphical representation of the number of overwritten errors reported for several file system implementations. The total number of bugs founds is further broken into true bugs and false positives.

```
1  err = -EIO; // represents a null buffer head
2  if (!bh)
3    goto cleanup;
```

Figure 5. Typical misplacement of assignments

to error copying. Finally, we also find that a mix of false positives arise due to our incomplete definition of error-handling functions.

5.2.1 Problematic Coding Styles

We begin by discussing programming styles found to confuse our tool. As mentioned above, these coding styles are particularly problematic when looking for overwritten errors. We further divide these into three categories: initializing with an error code, misplacing assignments, and overwriting with a different error code.

Variable initialization using error codes. Programmers occasionally initialize a local variable with an error code before any error has actually occurred. The error code is kept in the variable if a problem eventually occurs; otherwise it is overwritten. This appears to have been done as a means of refactoring. For example, a common scenario is found when several branches return the same default error. Instead of assigning the default error in each of those branches, the error code is assigned at the beginning of the function and overwritten if none of the failing branches is taken, i.e., the operation was successful.

Misplaced assignments. This is the source of most false positives. This situation is similar to the initialization problem in that an error code is assigned in advance. The error is assigned before it is known whether an error condition is actually met. Figure 5 shows a typical example. The assignment on line 1 should be postponed until just before line 3, at which point we are certain that the buffer head is null are about to jump to cleanup code.

```
1  if (rc == -EAGAIN)
2    rc = -EHOSTDOWN;
```

Figure 6. Overwriting an error code with a different error code

```
1  int err;
2  for (...) {
3    bh = getfrag(..., &err);
4    if (!bh)
5      ...
6    if (bh && !buffer_update())
7      ...
8  }
```

Figure 7. Redundant error reporting

```
1  rc = ext3_mark_inode_dirty();
2  error = rc;  // the error value is transferred
3  rc = inode_setattr(); // legal overwrite
```

Figure 8. An example illustrating the copy of an error code

Overwriting an error code with another error code. Situations arise in which overwriting an error code with another error code does not represent a bug. For example, an error code generated in one layer of the operating system may need to be translated into a different error code when received by another layer. Allowing such assignments clearly depends on the context and the error codes involved. Unfortunately, there is no formal error hierarchy, which makes difficult, if not impossible, to automatically differentiate between correct and incorrect overwrites. Figure 6 shows a simple example of translating one error code into a different one.

Overwriting an error code may also be allowed when there is another way to detect the problem. We refer to this as *redundant error reporting*. In Figure 7 we can see that `err` may be overwritten inside the loop when passed as parameter to the `getfrag` function on line 3. However, it so happens that `getfrag` returns null whenever it writes an error code into its pointer argument. Thus, the `!bh` tests are sufficient and checking `err` as well would be redundant. The only potential problem is that a more specific error in `err` could be overwritten with a more general one, leading to loss of information about what exactly went wrong.

5.2.2 Making Copies of Errors

One more scenario is found to produce false positives, which are out-of-scope related. A variable containing an error code may be copied into other variables. In practice, checking at least one of these copies suffices. Our tool is more restrictive and expects each copy to be eventually checked. Figure 8 shows an example. A copy of an error value is made on line 2. The variable holding the first copy is then overwritten on line 3, which is not a bug. This example suggests that the last variable assigned the error value is the one that must be checked. Unfortunately, this is not always true, and it is difficult to determine which overwrites can be allowed on the assumption that some other copy will eventually be checked. Solving this may require incorporating the concept of ownership used in static leak detection by Heine [12]. Transferring an error value can also transfer ownership of that error. Exactly one variable has ownership on an error value at any given program point, and that is the copy that must be checked.

5.2.3 Error-handling Functions

File system implementations define certain functions that handle errors. Thus, a call to such a function usually means that the an error has been checked. Some of these functions take the variable containing the error value as parameter, in which case it is clear which error is being handled. Other error-handling functions do not take the error value as parameter. Knowing which error should be considered checked after such a call requires deeper understanding of the surrounding context. A first attempt to solve this problem is simply to identify the calls to these functions and consider their parameters as checked, as is already done for `printk`.

5.3 Performance

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.

The largest file system analyzed (in lines of code) is ReiserFS, at 24 KLOC. The total analysis running time for this file system is 6 hours, 8 minutes, 51 seconds. The time is further divided into each phase. Phase 1 (WPDS extraction) takes 8 seconds. Phase 2 (poststar solving) takes 1 hour, 20 minutes, 7 seconds. Phase 3 (bug identification and reporting) takes 4 hours, 48 minutes, 36 seconds. By contrast, one of the smallest file systems in which more bugs are found is Boot FS, at 1 KLOC. Analysis of this file system takes a total 1 hour, 7 minutes, 14 seconds. Phase 1 takes 6 seconds; Phase 2 takes 17 minutes, 28 seconds; and Phase 3 takes 49 minutes, 38 seconds.

6. Improving the Precision of the Analysis

As discussed in Section 5, our tool is able to report interesting, serious and otherwise-difficult-to-find bugs, however it also suffers from a high rate of false positives. Our focus in this section is to improve the precision of our analysis by reducing the number of false positives. In particular, our initial goal is to reduce the number of false positives for the category of overwritten errors.

The remainder of this section describes a new approach that reduces the number of overwritten false positives in about 77%. Unfortunately, this new approach introduces the risk of false negatives, although we find that their occurrence rate is somewhat low (5%). At the same time, the 95% of the originally reported true bugs are still found and reported.

6.1 New Approach

This approach focuses on eliminating two out of the three sources of false positives discussed in Section 5.2.1. We are referring to initialization using error codes and misplaced assignments. We further inspect the false positives that fall into these categories and find an interesting pattern: these problematic coding styles mostly occur within functions generating a given error code. We find that callers to these functions instead limit themselves to propagate the error code. This brings an interesting point to our attention: it might be acceptable to overwrite an error code within the function that originates it but not once the error code is returned to a caller.

This general observation leads us to revisit the design of our analysis. Rather than tracking down a single category of error codes, we now introduce two different categories: $tentative$ and $nontentative$. An error code is referred as $tentative$ whithin the function that generates it and it can be overwritten. As soon as the error code is returned from the originating function, a $tentative$ error is transformed into a $nontentative$ one and it should no longer be overwritten. Both kinds of error codes are treated differently when determining whether an unchecked error code might be overwritten or not.

The basics of our new approach remain as explained in Section 3.2. Modifications and additions are further explained in the following subsections.

6.1.1 Tentative and NonTentative Error Codes

We still consider only the 34 basic error codes used in Linux, however internally we are tracking down 68 different error codes. For each basic error code, we simply consider two versions: *tentative* and *nontentative*. For example, the error code `EIO` results in a *tentative* `EIO` and a *nontentative* `EIO`.

6.1.2 Transfer Functions

Transfer functions remain as discussed in Section 3.2.3. However, we need to change the definition of $\mathcal{E}$, which was introduced earlier in Section 3.2.2. As discussed in the previous subsection, we now have two kinds of error codes. We define $\mathcal{E}$ as the set containing the *tentative* constant error codes. The reason for this is that all instances of error codes in the source code are translated into their corresponding *tentative* codes. This means that the textual representation of the kernel code as a weighted pushdown system (which includes the transfer functions for each program statement) can only contain *tentative* (as opposed to *nontentative*) error codes. Later when operations such as extend, combine and merge are applied in order to solve the dataflow problem, an internal transformation from *tentative* to *nontentative* codes will take place if required. This is explained in the next subsection.

6.1.3 Merge Function

A *tentative* error code is transformed into a *nontentative* error only when it is returned from a function. At the transfer function level, we would be able to make this transformation only when a function specifically returns a constant error code. Unfortunately, functions not only return constants, but variables and more complex expressions. Consider the case in which a variable is returned, we would need to know the specific *tentative* error codes the variable might contain in order to transformate them into their corresponding *nontentative* codes. As it can be seen, this information is not available at this point. On the other hand, this information is available when applying the merge function, discussed in Section 3.1. Thus, we add the capability of error transformation to the merge function.

At each particular point in which the merge function is applied, we have information about the callee and the caller whose weights are about to be extended. As mentioned earlier, we introduce a global variable to hold the return value of a given function (functionName$return). Thus, we can find the values that the global variable calleeName$return may contain when returning from the callee and before we extend with the caller. For any of those values which happen to be *tentative* error codes, we simply replace them with their corresponding *nontentative* codes.

6.1.4 Finding and Describing Bugs

We still report overwrites of *tentative* errors, however we differentiate them from overwrites of *nontentative* errors, which are more likely to be true bugs. This allows us to rank the bugs reported. Thus, the programmer could start inspecting the overwrites of *nontentative* error codes and then, depending on the time available, proceed to inspect the rest of the bug reports. On the other hand, the programmer could choose to filter the bugs found and only look at those involving *nontentative* error codes.

We still produce a single path/slice to demonstrate the existence of a overwrite at a particular point in the program. Unfortunately, this is not always the best idea as several error codes might be overwritten at that particular point and we might end up choosing an error code that leads to a false positive when choosing a different error code might have demonstrated the existence of a true bug. We now give priority to *nontentative* errors. Thus, if many error codes are being overwritten at the same program point, then we prefer to report the overwrite of a *nontentative* error code over the overwrite of a *tentative* one.

Describing bugs also suffered modifications as we no longer keep track of a single error code (the one being overwritten). Error codes may transform, thus we need to make sure we do not lose track of the current error code to be tracked down.

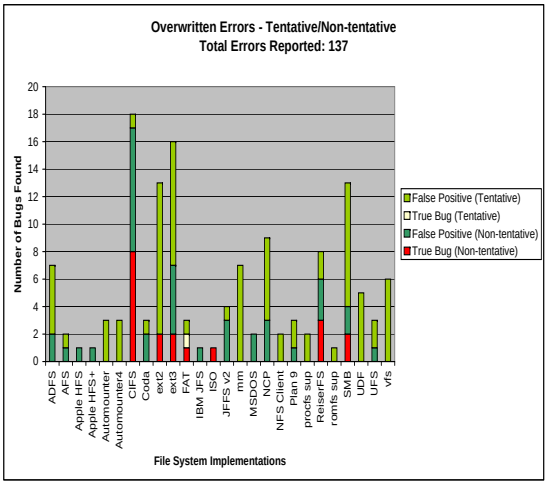

Figure 9. A graphical representation of the number of overwritten errors found by the new tool. Bugs are now divided into four categories: 1) *nontentative* true bugs, 2) *nontentative* false positives, 3) *tentative* true bugs and 4) *tentative* false positives. A total of 137 bugs are found accross all the file system implementations analyzed.

6.2 New Results

Figure 9 contain the new results obtained. The bugs found are divided into four categories: 1) *nontentative* true bugs, 2) *nontentative* false positives, 3) *tentative* true bugs and 4) *tentative* false positives. We are still able to report all the true bugs found in the previous analysis while reducing the number of bug reports from 180 to 137. We also find that 19 out of the 20 true bugs are overwrites to *nontentative* errors. Thus, experimental results show that our assumption that error codes can be overwritten within the function that generates them gives good results in practice. It is possible for the programmer to give priority to *nontentative* related bugs as they are more likely to be true bugs. The programmer still has the option to go through all bug reports. This can be beneficial since it allows the programmer to decide what and how many bug reports to examine. On the other hand, 117 out of the 137 bugs reported are still false positives, a high rate that can overwhelm the programmer.

Figure 10 presents the results obtained when only considering *nontentative* related bugs. We find that the number of false positives is reduced in 77% while finding 95% of the true bugs. By filtering the results, we fail to report one true bug. Unfortunately, less false positives usually means more false negatives. A compromise is necessary.

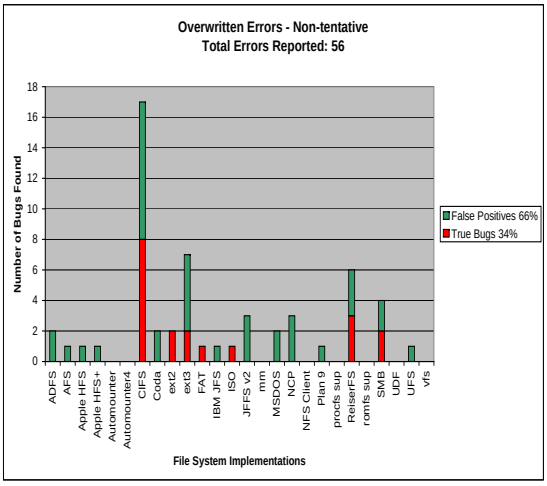

Figure 10. A graphical representation of the number of *nontentative* overwritten errors. We only consider *nontentative* related bugs, reducing the number of false positives from 160 to 37 while finding 19 out of the 20 true bugs.

6.3 Reducing Out-of-Scope False Positives

Finally, we briefly describe an approach to reduce the number of out-of-scope false positives. The idea is as follows. We earlier found that the main source of false positives for this category of errors is the copying of error codes. Our tool requires every copy to be checked, which is too restrictive. The problem is that we do not have a way to distinguish between different instances of the same error code. It is fundamental to be able to recognize those instances that are copies of a given error code.

First of all, we can introduce a natural new category of error codes: *checked* and *unchecked* errors. Only then we would be able to determine which out-of-scope bugs to report as potential bugs.

The second addition to the current design of our analysis is to make each error constant found in the source code unique. For example, each use of `EIO` in the source code would lead to a unique `EIO` error code. This would allow us to easily identify the copies of a given error code when solving the dataflow analysis problem. Checking a particular `EIO` instance would imply checking the other copies of this error code and promoting all of them to *checked*. Later in the phase of finding bugs, we would obviously not report out-of-scope variables holding *checked* error codes. This is likely to eliminate all the false positives identified so far. A further study should be conducted as out-of-scope errors have not been inspected in the presence of a kernel expert, thus we are lacking of confirmation about real bugs. At the same time, it is more obvious to spot out-of-scope false positives than false positives for overwrites.

7. 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 [7], but does not attempt to 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 which 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 adopting structured exception handling instead. Language designs for exception management have been under consideration for decades [9, 25]. 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 [34, 37]. 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. found that 90% of relevant exceptions thrown by .NET assemblies (C# libraries) are undocumented [29]. 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 [5, 8, 10, 13, 14, 15, 18, 19, 32]. 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.

Recent work by Gunawi et al. [11] highlights 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. WPDS witness traces (Section 4) offer a level of diagnostic feedback not possible with Gunawi's whole-function-classification approach.

Bigrigg and Vos [3] 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 283 thousand lines of kernel code.

The FiSC system of Yang et al. [39] 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.

8. Future Work and Conclusions

Besides continuing tuning our tool to produce better results, other future work includes support for error transformation and asynchronous paths. *Error transformation* refers to changes in how errors are represented as they propagate across software layers. Integer error codes may pass through structure fields, be cast into other types, be transformed into null pointers, and so on. An improved analysis should be able to track errors across all of these representations. Operating systems are concurrent programs; *asynchronous paths* arise from concurrently-executing code that may produce er-

rors or otherwise interact with the code under inspection. Recent work by Jhala et al. [17] may be applicable here.

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 an extended weighted pushdown system, and poststar queries on this system allow detailed diagnosis of a variety of error mismanagement bugs. We have applied our implementation to four dozen Linux file systems and found non-trivial bugs. False positives arise, but many of these arise from a small number of recurring patterns that should also be amenable to automated analysis.

We also designed and implemented a variation of the analysis in order to reduce the number of false positives for overwrites. Our approach successfully reduced the number of false positives in 77% while finding 95% of the true bugs. An important lesson learned is that even when automatization does not seem feasible, we can always study the source code and extract valuable information that can be used to tune our analysis. We also examined 100 out-of-scope bugs and identified a recurrent pattern that serves as a source of false positives. We propose a new variation of our analysis that could remove the most of them.

Each positive step toward eliminating error propagation bugs increases the trustworthiness of file systems and, in turn, of computer systems as a whole.

References

[1] Apple. Technical note TN1150. `http://developer.apple.com/technotes/tn/tn1150.html`, Mar. 2004.

[2] S. Best. JFS Overview. `http://www.ibm.com/developerworks/library/l-jfs.html`, 2000.

[3] 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.

[4] 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.

[5] 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.

[6] R. Card, T. Ts'o, and S. Tweedie. Design and Implementation of the Second Extended Filesystem. In Proceedings of the First Dutch International Symposium on Linux, 1994.

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

[8] 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.

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

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

[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] D. L. Heine. *Static memory leak detection*. PhD thesis, Stanford University, Stanford, CA, USA, 2005. Adviser-Monica S. Lam.

[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] J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West. Scale and Performance in a Distributed File System. *ACM Transactions on Computer Systems*, 6(1), Feb. 1988.

[17] R. Jhala and R. Majumdar. Interprocedural analysis of asynchronous programs. In M. Hofmann and M. Felleisen, editors, *POPL*, pages 339–350. ACM, 2007.

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

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

[20] N. Kidd, T. Reps, and A. Lal. WALi: A C++ library for weighted pushdown systems, 2007. To be released.

[21] 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.

[22] 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.

[23] 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.

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

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

[26] 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.

[27] H. Reiser. ReiserFS. www.namesys.com, 2004.

[28] 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.

[29] 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.

[30] R. Sandberg. The Design and Implementation of the Sun Network File System. In *Proceedings of the 1985 USENIX Summer Technical Conference*, pages 119–130, Berkeley, CA, June 1985.

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

[32] 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.

[33] Sun Microsystems. ZFS: The last word in file systems. `http://www.sun.com/2004-0914/feature/`, 2006.

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

[35] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, and G. Peck. Scalability in the XFS File System, Jan. 1996.

[36] S. C. Tweedie. EXT3, Journaling File System. `olstrans.sourceforge.net/release/OLS2000-ext3/OLS2000-ext3.html`, July 2000.

[37] 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.

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

[39] 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.