Expect the Unexpected: Error Code Mismatches Between Documentation and the Real World *

Cindy Rubio-González
University of Wisconsin–Madison
crubio@cs.wisc.edu

Ben Liblit
University of Wisconsin–Madison
liblit@cs.wisc.edu

Abstract

Inaccurate documentation can mislead programmers and cause software to fail in unexpected ways. We examine mismatches between documented and actual error codes returned by 42 Linux file-related system calls. We use static program analysis to identify the error codes returned by system calls across 52 file systems, including widely-used implementations such as CIFS, ext3, IBM JFS, ReiserFS and XFS. We describe analysis optimizations that dramatically reduce run-time and memory consumption. Comparing analysis results with Linux manual pages reveals over 1,700 undocumented error-code instances affecting all file systems and system calls examined.

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.2.7 [*Software Engineering*]: Distribution, Maintenance, and Enhancement—documentation; D.4.3 [*Operating Systems*]: File Systems Management

General Terms Algorithms, Languages, Reliability, Verification

Keywords static program analysis, interprocedural dataflow analysis, weighted pushdown systems, manual pages

1. Introduction

User applications rely on systems software to run as specified. File systems play a particularly important role within systems software by storing and organizing user's data. When run-time errors do occur, user applications must be notified and respond. Thus, user applications must be aware of possible problems and be prepared to deal with them. Documentation is the programmer's main resource to learn about potential run-time errors that might arise from invoking a given system call. Unfortunately, writing and maintaining accurate code documentation is difficult. This is particularly true for large code bases such as the Linux kernel.

This paper considers whether the manual pages that document Linux kernel system calls match the real source code's behavior. We are particularly interested in Linux file-system implementations because of their importance and the large number of implementations available, which might make the task of maintaining documentation even harder. Our task is to examine the Linux source code to find the sets of error codes that system calls return and compare these against the information given in the Linux manual pages to find errors that are returned to user applications but not documented.

We generalize this problem as finding the set of error codes that each function returns, and then focus on the subset of system call functions. This requires tracing error codes through kernel code. It has been shown that error codes flow through long call chains, which has suggested the need for interprocedural static analyses. Rubio-González et al. [15] describe a static analysis that finds error propagation bugs by identifying the set of error codes that each variable may contain at each program point. Given such an analysis, we can reduce the documentation-checking problem to finding the sets of possible error codes at function exit points.

The contributions of this paper are: 1) adapting an existing error-propagation analysis to find the list of error codes that each function may return, 2) retrieving this information for system calls and comparing the list of returning error codes against the Linux manual pages, 3) describing optimizations to allow analysis of larger and more complex file systems, 4) providing real-world results for 52 Linux file-system implementations that reveal over 1,700 undocumented error-code instances returned by file-related system calls.

The paper is organized as follows: Section 2 discusses some related work. Section 3 gives a high-level description of an error-propagation analysis and how this analysis has been extended to manipulate positive error codes. Section 4 shows how to use the error-propagation framework to find the set of error codes returned by each function in a program. We discus some analysis optimizations in Section 5. We present experimental results on 52 Linux file systems in Section 6. Section 7 concludes.

2. Related Work

Studies show that programmers value accurate documentation, but neither trust nor maintain the documentation they have [9, 19]. For example, Sacramento et al. [16] found that 90% of relevant exceptions thrown by .NET assemblies (C# libraries) are undocumented. Misleading documentation can lead to coding errors [20] or even legal liability [5]. Our work bridges the gap between code and documentation, automatically identifying mismatches so that disagreements between the two may be peaceably resolved. In the spirit of Xie and Engler [23], even if we do not know which is right and

* Supported in part by AFOSR grant FA9550-07-1-0210; DoE contract DE-SC0002153; LLNL contract B580360; and NSF grants CCF-0621487, CCF-0701957, and CCF-0953478. 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.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

PASTE'10, June 5–6, 2010, Toronto, Ontario, Canada.
Copyright © 2010 ACM 978-1-4503-0082-7/10/06. . . $10.00

which is wrong, the mere presence of inconsistencies indicates that something is amiss.

Venolia [21] uses custom regular expressions to find references to software artifacts in free-form text. The referenced artifacts are extracted from compiler abstract syntax trees. Tan et al. [20] use natural-language processing to identify usage rules in source comments, then check these against actual code behavior using backtracking path exploration. Our documentation-analysis task is much easier, and can be solved using a Venolia-style purpose-built pattern-matcher. Our analysis of the corresponding source code, however, poses a greater challenge.

Prior work has measured documentation completeness, quantity, density, readability, reusability, standards adherence, and internal consistency [4, 11, 14, 17, 18]. Berglund and Priestley [1] call for automatic verification of documentation, but consider only XML validation, spell checking, and the like. None of this assesses whether the documentation's claims are actually true. For truly free-form text, nothing more may be possible. However, for some highly-structured documents, we can go beyond structural validation to content validation: affirming that the documentation is not merely well-formed, but actually truthful with respect to the code it describes.

While our work focuses on finding mismatches between code and pre-existing documentation, Buse and Weimer [3] automatically generate documentation describing the circumstances under which Java code throws exceptions. If applied to kernel code, this could help us not just list undocumented error codes, but also describe the conditions under which they arise.

3. Error Propagation Analysis

Linux (like many other operating systems) is written in C, a language that offers no exception-handling mechanism 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. For example, `EIO` (I/O error) is defined as 5. Linux uses 34 basic error codes. Error codes are negated by convention (except for the XFS Linux file system, which uses positive error codes). We say that an error is *unchecked* if no action (e.g., error notification, attempted recovery, etc.) has taken place due to its occurrence.

Rubio-González et al. [15] proposed an interprocedural, flow- and context-sensitive static program analysis that determines, at each program point, the set of unchecked error codes each variable may contain. We present a high-level description of this framework in Section 3.1. We describe in Section 3.2 an extension to this framework that allows to track positive error codes.

3.1 Analysis Framework

Assignments propagate unchecked errors forward from one variable to another. Propagation ends when an error is overwritten, dropped, or checked by error-handling code. In general we wish to identify the set of unchecked errors that each program variable may contain at each program point. Thus, this problem resembles an over-approximating analogue of copy constant propagation [22].

We codify the analysis as a path problem over weighted pushdown systems (WPDSs) [13]. A WPDS is a useful dataflow engine for problems that can be encoded with suitable weight domains, computing the meet-over-all-paths solution. It consists of three main components: (1) a pushdown system, (2) a bounded idempotent semiring, and (3) a mapping from pushdown system rules to associated weights. The pushdown system is used to model the control flow of the program. The bounded idempotent semiring is a 5-tuple as defined in Reps et al. [13]. Finally, we define transfer functions for each construct in the program. Transfer functions define the new state of the program as a function of the old state.

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

3.1.1 Pushdown System

We model the control flow of the program with a pushdown system using the approach of Lal et al. [8]. Let P contain a single state $\{p\}$. Γ corresponds to program statements, and Δ corresponds to edges of the interprocedural control flow graph (CFG). Table 1 shows the PDS rule for each type of CFG edge.

3.1.2 Bounded Idempotent Semiring

We classify integer constants into *error constants* and *non-error constants*. Define $\mathscr{E}$ 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 $\mathscr{C}$ be the set of all constants and $\mathscr{V}$ be the set of all program variables.

Let $\mathscr{S} = (D, \oplus, \otimes, \bar{0}, \bar{1})$ be a bounded idempotent semiring. Figure 1 shows the definition for each semiring element. 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. Figure 1(b) defines the combine operator. 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. Figure 1(e) defines the extend operator, which is also applied component-wise. The extend operator is essentially composition generalized to the power set of variables and constants rather than just single variables. The neutral weight $\bar{1}$, given in Figure 1(f), maps each variable to the set containing itself, which is a power-set generalization of the identity function. Figure 1(c) gives the annihilator weight $\bar{0}$, which maps each variable to the empty set.

3.1.3 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. We describe 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.)

Consider an assignment $t = s$ where $t, s \in \mathscr{V}$ are distinct and s might contain an unchecked error code. The assignment $t = s$ leaves an unchecked error in t but removes it from s, effectively transferring ownership of unchecked error values across assignments. The following paragraphs discuss the transfer functions for assignments without function calls on the right side (see Table 2). Rubio-González et al. [15] detail transfer functions for other language constructs, including variants that treat errors as having been copied instead of transferred.

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 vari-

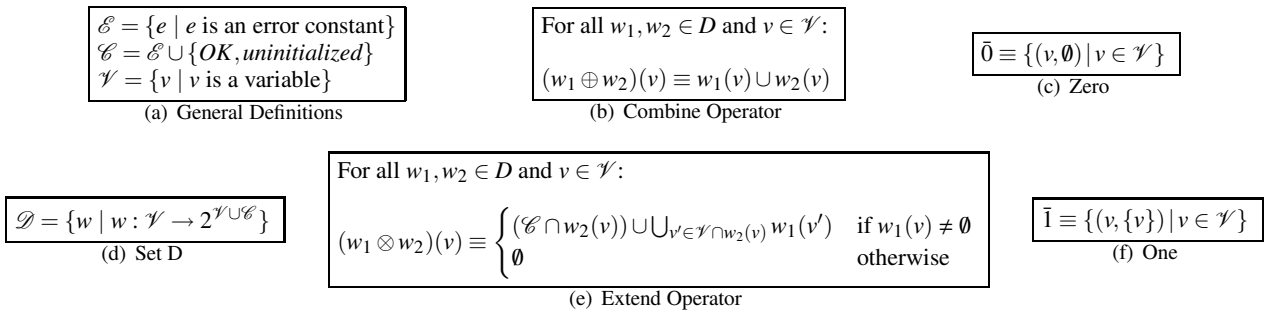

Figure 1. Definitions for weighted pushdown system representing error propagation analysis

Program Statement	Where	Transfer Function
$v = e$	$e \in \mathscr{V} \cup \mathscr{C}$	$Ident[v \mapsto \{e\}][s \mapsto \{OK\}$ for all $s \in \{e\} \cap \mathscr{V} - \{v\}]$
$v = e_1\ op\ e_2$	$e_1, e_2 \in \mathscr{V} \cup \mathscr{C}$ and op is a binary arithmetic or bitwise operator	$Ident[u \mapsto \{OK\}$ for all $u \in \{v, e_1, e_2\} \cap \mathscr{V}]$
	$e_1, e_2 \in \mathscr{V} \cup \mathscr{C}$ and op is a relational operator	$Ident[v \mapsto \{OK\}]$
$v = op\ e$	$e \in \mathscr{V} \cup \mathscr{C}$ and op is a binary arithmetic or bitwise operator	$Ident[u \mapsto \{OK\}$ for all $u \in \{v, e\} \cap \mathscr{V}]$
	$e \in \mathscr{V} \cup \mathscr{C}$ and op is the logical negation or an indirection operator	$Ident[v \mapsto \{OK\}]$

Table 2. Transfer functions for assignments

able to the set containing itself. Note that this is identical to $\bar{1}$. The transfer function for this simple assignment is $Ident[v \mapsto \{e\}][s \mapsto \{OK\}$ for $s \in \{e\} \cap \mathscr{V} - \{v\}]$. In other words, after the assignment, (1) v must now have any unchecked error code previously in e; (2) e, having relinquished responsibility, is OK or a constant; and (3) all other variables retain whatever values they had before the assignment. Special care is taken in the case that v and e are identical, in which case this transfer function reduces to identity.

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.). 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 e_1 and e_2 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 $u \in \{v, e_1, e_2\} \cap \mathscr{V}]$.

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 ($v = op\ e$) are similar: arithmetic and bitwise operators map both v and e (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 (&, $\star$), the transfer function maps v to $\{OK\}$ but leaves e unchanged.

We perform a poststar query [13] on the WPDS, with the beginning of the program as the starting configuration. We read out weights from the resulting weighted automaton applying the *path_summary* algorithm of Lal et al. [7]. This algorithm allows us to retrieve the weight representing execution from the beginning of the program to any particular point of interest.

3.2 Positive Error Codes

Prior work assumed that error codes are always negated. We now support optional analysis of positive error codes as well. In particular, we define a new set of interchangeable transfer functions for conditional statements. As before, each branch of a conditional statement is associated with a transfer function, depending on the condition. We assume that conditional statements with short-circuiting conditions are rewritten as nested conditional statements with simple conditions.

Consider a conditional of the form $if(v > 0)$. The transfer function associated with the true branch for negative error codes is $Ident[v \mapsto \{OK\}]$. The true branch is never selected when v is negative, therefore v cannot contain an error code on that branch. The transfer function for the false branch is *Ident*. The false branch is selected when v is zero or negative, which does not reveal any additional information about v (it might contain an error code or not). Thus, variables should remain mapped to whatever values they had before the conditional. Note that the opposite holds for positive error codes. If error codes are positive, then the true branch of $if(v > 0)$ uses *Ident* and the false branch uses $Ident[v \mapsto \{OK\}]$.

Currently, the analysis can be applied in negative or positive mode. This lets us analyze code that uses positive error codes. An

example is the XFS Linux file system: it is one of the largest and most complex Linux file systems, and it uses positive error codes.

4. Finding Error Return Values

We wish to find the list of error codes returned by each function in the program. As mentioned earlier, we can now retrieve the weight at any program point. Thus, at each return statement r in function f, we retrieve the associated weight w. Let $\mathscr{E}$ be the set of all error constants and $\mathscr{R} \subseteq \mathscr{C}$ be the set of possible constant values returned by function f (if any). Then $\mathscr{R} \cap \mathscr{E}$ represents the set of error codes that may be returned when f returns at exit point r. We generate a report that includes source information, the list of returned error codes and a sample path for each of these error codes.

Sample paths describe how a particular function exit point r was reached in a way that a certain error code instance was returned. We use WPDS witness tracing information to construct these paths. A *witness set* is a set of paths that justify the weight reported for a given configuration. Rubio-González et al. [15] described the use of witness tracing information for the construction of error-propagation paths; we use witnesses here to justify each error code that a function exit point is claimed to return.

Figure 2 shows examples of return-value reports. Function `bar` returns a constant error code at line 2. Figure 2(b) shows the report produced for this return point, which consists of a single line of information, since the error was generated and returned at the same program point. On the other hand, function `foo` has two exit points (lines 9 and 10). In the first case, line 9, `foo` calls function `bar`. In the second case, `foo` returns the value contained in variable `retval`. The corresponding reports are shown in Figure 2(c). We produce three return-value reports instead of two. This is because `retval` can possibly contain two error codes (`ENOMEM` and `EPERM`), and we choose to provide a sample path for each. Of course, error propagation in real code is far more complex. Real sample paths can span thousands of lines of code, even exceeding half a million lines in one extreme case.

```
1  int bar() {
2    return -EIO;
3  }
4
5  int foo() {
6    int retval;
7    if (...) retval = -ENOMEM;
8    else if (...) retval = -EPERM;
9    else return bar();
10   return retval;
11 }
```

(a) Example code

```
ex.c:2: EIO* returned from function bar
```

(b) Report for function `bar`

```
ex.c:2: error code EIO is returned
ex.c:9: EIO* returned from function foo

ex.c:7: "retval" receives an error from ENOMEM
ex.c:10: ENOMEM* EPERM returned from function foo

ex.c:8: "retval" receives an error from EPERM
ex.c:10: ENOMEM EPERM* returned from function foo
```

(c) Reports for function `foo`

Figure 2. Example code fragment and corresponding reports

5. Analysis Optimizations

Rubio-González et al. [15] required several hours to analyze file systems approaching 100,000 lines of code, and the massive XFS file system remained completely out of reach. We have optimized the analysis core in two ways:

1. We add a preliminary flow- and context-insensitive analysis that filters out *irrelevant program variables* that cannot possibly contain any error code.
2. We compress consecutive WPDS rules that share identity weight and identical source locations.

The number of variables determines the size of the weights, and large weights can significantly degrade performance. Our return-value analysis initially considered all program variables as potential error-code holders. In reality, most program variables have nothing to do with storing error codes. Thus, we now perform a lightweight pre-analysis to find the set of program variables that can possibly contain error codes at some point during program execution, thereby keeping weights small.

This pre-analysis is flow- and context insensitive. It begins by identifying program points at which error codes are generated, i.e., those program points at which error macros are used. We identify those variables that are assigned error constants and add them to our set of *relevant variables*. A second iteration looks for variables that are assigned from relevant variables, which are also added to the relevant-variable set. We repeat this process until we reach a fixed point. Because of earlier program transformations that we do not discuss here (e.g., introducing exchange variables [15]), this approach also handles other error-flow scenarios such as function parameters and return values.

WPDS rules are used to model the control flow of the program and are associated with weights. The identity weight has no effect on the current state of the program. We also associate source information (file name and line number) to rules for use when presenting diagnostic information. Because we convert the program into three-address form, one original program statement may be split into several consecutive rules, all with identical source information. These are analyzed as distinct program points, thus increasing the number of weights to be created and calculated. We compress back consecutive rules that are associated with the identity weight and share the same source information, thereby reducing the number of weights to be calculated.

6. Experimental Evaluation

We use the CIL C front end [12] to apply preliminary source-to-source transformations on Linux kernel code, such as redefining error code macros as distinctive expressions to avoid mistaking regular constants for error codes. We also use CIL to traverse the CFG and emit a textual representation of the WPDS. Our separate analysis tool uses the WALi WPDS library [6] to perform the interprocedural dataflow analysis on this WPDS. We encode weights using *binary decision diagrams* (BDDs) [2] as implemented by the BuDDy BDD library [10].

We analyze 52 file-system implementations (871 KLOC) found in the Linux 2.6.32.4 kernel. We synthesize a `main` function that nondeterministically calls all exported entry points of the file system under analysis. Our tool produces the list of basic error codes that each function may return, along with sample paths that illustrate how specific error instances reach a given function's exit points. We compare these error codes against version 2.39 of the Linux manual pages for each of 42 file-related system calls.

System Call	Error Code																												Total
	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z	aa	ab	
access	0	-	0	1	0	0	-	0	1	-	-	0	0	0	0	0	-	-	0	-	1	0	0	-	0	0	-	0	3
chdir	0	-	0	-	0	0	-	0	1	1	-	0	0	0	0	0	-	-	0	-	1	0	0	21	0	0	0	0	24
chmod	1	-	2	-	0	4	-	3	4	11	-	0	1	0	1	2	-	-	5	-	2	-	4	-	0	0	4	1	45
chown	1	-	2	-	0	4	-	3	4	11	-	0	1	0	1	2	-	-	5	-	2	-	4	-	0	0	4	1	45
chroot	0	-	0	1	0	0	-	0	1	1	-	0	0	0	0	0	-	-	0	-	1	-	0	21	0	0	0	0	25
dup	0	0	0	-	-	0	0	0	-	0	0	0	-	0	0	0	0	20	0	0	0	0	0	0	0	0	0	0	20
dup2	0	0	0	-	-	0	0	0	-	1	0	0	-	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	2
fchdir	0	-	0	-	0	0	-	0	1	0	-	0	0	0	0	0	-	-	0	-	1	0	0	21	0	0	0	0	23
fchmod	1	-	2	-	0	4	-	3	4	10	-	0	1	0	1	2	-	-	5	-	2	-	4	-	0	0	4	1	44
fchown	1	-	2	-	0	4	-	3	4	10	-	0	1	0	1	2	-	-	5	-	2	-	4	-	0	0	4	1	44
fdatasync	0	0	1	-	0	0	0	0	2	22	-	0	0	0	0	1	2	3	0	0	1	0	0	0	0	0	0	0	32
flock	0	0	21	-	0	0	0	0	-	-	21	0	0	0	0	0	21	21	0	0	0	0	0	0	0	0	0	0	84
fstat	1	-	1	-	0	1	-	0	1	2	21	0	1	0	1	2	-	-	1	-	1	0	1	1	0	0	1	1	37
fstatfs	0	-	0	-	0	0	-	0	-	0	-	0	0	0	0	21	-	-	0	-	0	0	0	0	0	0	0	0	21
fsync	0	0	1	-	0	0	0	0	2	22	-	0	0	0	0	1	2	3	0	0	1	0	0	0	0	0	0	0	32
ftruncate	1	-	21	-	0	4	-	-	-	-	-	-	1	0	1	2	-	10	5	-	2	-	4	-	0	0	-	1	52
getdents	1	1	2	-	0	1	-	0	1	-	20	0	1	0	1	2	-	12	2	-	1	1	1	1	0	0	1	1	50
ioctl	0	10	4	-	2	0	-	21	2	-	21	2	0	0	0	21	5	21	2	2	1	21	0	8	21	1	1	0	166
lchown	1	-	2	-	0	4	-	3	4	11	-	0	1	0	1	2	-	-	5	-	2	-	4	-	0	0	4	1	45
link	1	-	3	1	1	-	-	0	3	7	-	0	1	-	1	2	-	-	-	-	2	-	1	-	0	0	1	-	24
lstat	1	-	1	-	0	1	-	0	2	4	21	0	1	0	1	2	-	-	1	-	2	0	1	21	0	0	1	1	61
mkdir	1	-	3	1	1	-	-	1	3	11	21	0	1	7	1	2	-	-	-	-	2	-	2	-	0	0	2	1	60
mknod	1	-	3	1	1	-	-	1	3	-	21	0	1	1	1	2	-	-	-	-	2	-	2	-	0	0	2	1	43
mount	0	-	-	1	-	0	-	0	1	-	21	0	-	0	0	-	-	-	0	-	-	-	0	21	0	0	0	0	44
nfsservctl	0	0	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2
read	0	1	-	-	0	0	-	0	-	-	-	-	0	0	0	2	20	21	0	0	0	0	0	0	0	0	0	0	44
readlink	0	-	0	1	0	0	-	0	1	-	-	0	0	0	0	0	-	-	0	-	1	0	0	21	0	0	0	0	24
readv	0	0	0	20	0	0	0	0	0	-	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	20
rename	1	-	3	21	-	21	-	0	3	-	21	-	1	-	1	2	-	-	-	-	2	21	1	-	0	0	1	-	99
rmdir	0	-	2	1	-	1	-	0	3	-	21	21	0	0	0	1	-	-	2	-	1	-	0	-	0	0	0	0	53
select	0	0	0	-	0	0	1	0	-	-	0	0	0	0	0	0	0	-	0	0	0	0	0	0	0	0	0	0	1
stat	1	-	1	-	0	1	-	0	2	4	21	0	1	0	1	2	-	-	1	-	2	0	1	21	0	0	1	1	61
statfs	0	-	0	-	0	0	-	0	-	1	-	0	0	0	0	21	-	-	0	-	1	0	0	21	0	0	0	0	44
symlink	0	-	3	1	1	-	-	1	3	8	-	0	0	1	0	1	-	-	-	-	1	-	0	-	0	0	1	0	21
truncate	1	-	21	-	0	4	-	-	-	-	-	-	1	0	1	2	-	23	5	-	2	-	4	-	0	0	-	1	65
umount	0	-	-	1	-	0	-	0	1	-	21	0	-	0	0	-	-	-	0	-	-	-	0	21	0	0	0	0	44
unlink	1	-	2	2	21	2	-	0	3	5	-	-	1	0	1	2	-	-	3	-	2	-	1	-	0	0	2	1	49
uselib	0	-	0	0	0	0	21	0	0	21	0	0	0	0	-	0	0	21	0	0	0	0	0	0	0	0	0	0	63
ustat	0	0	0	0	0	0	-	0	1	-	2	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	5
utime	1	-	2	2	0	4	1	3	4	12	21	0	1	0	1	2	-	26	5	21	2	-	4	-	0	0	4	1	117
write	0	1	-	-	0	0	-	-	-	-	-	0	0	0	0	2	20	2	-	0	0	0	0	0	0	1	0	0	26
writev	0	0	0	20	0	0	0	0	0	-	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	20
Grand Total	17	13	105	75	27	60	24	42	65	176	274	23	17	9	17	106	70	185	52	23	43	43	43	199	21	2	38	15	1,784

Table 3. Number of file systems per system call, which return undocumented errors. a:`E2BIG`, b:`EACCES`, c:`EAGAIN`, d:`EBADF`, e:`EBUSY`, f:`EEXIST`, g:`EFAULT`, h:`EFBIG`, i:`EINTR`, j:`EINVAL`, k:`EIO`, l:`EISDIR`, m:`EMFILE`, n:`EMLINK`, o:`ENFILE`, p:`ENODEV`, q:`ENOENT`, r:`ENOMEM`, s:`ENOSPC`, t:`ENOTDIR`, u:`EMXIO`, v:`EPERM`, w:`ERANGE`, x:`EROFS`, y:`ESPIPE`, z:`ESRC`, aa:`ETXTBSY`, ab:`EXDEV`.

6.1 Analysis of Manual Pages

The manual pages for Linux system calls have a very consistent internal structure. We can easily identify the section listing possible errors and extract the list of error codes contained therein. This requires only basic text analysis in the style of Venolia [21] rather than sophisticated natural-language-processing algorithms as used by Tan et al. [20].

For any given system call, "`man -W 2 syscall`" prints the absolute path(s) to the raw documentation file(s) documenting *syscall*. This is typically a single GZip-compressed file named `/usr/share/man/man2/`*syscall*`.2.gz`.

The uncompressed contents of these files are human-readable text marked up using `man`-specific macros from the troff typesetting system. Section headers are annotated using "`.SH`" with the section documenting error codes always being named "`ERRORS`." Thus, "`.SH ERRORS`" marks the start of the error-documenting section of each manual page, which continues until the start of the next section, also annotated using "`.SH`."

Within the `ERRORS` section, each documented error code is named in boldface (annotated using "`.B`") followed by a brief description of the circumstances under which that error occurs. Error code names always begin with a capital letter `E` followed by one or more additional capital letters, as in `EPERM` or `ENOMEM`. These never correspond to natural English-language words within the `ERRORS` section, so it is both straightforward and highly reliable to iterate through this section and extract the names of all error codes mentioned therein.

6.2 Undocumented Error Codes

Comparing our code analysis with our documentation analysis reveals two kinds of mismatch: documented error codes not returned by any file system, and error codes returned by some file system but not mentioned in the documentation. The first case, of unused error codes, is disturbing but likely benign. We focus here on the second case, of undocumented error codes, as these can be truly disruptive.

Table 3 summarizes our results. The table shows the number of file systems that may return a given undocumented error code (columns) for each analyzed system call (rows). For example, we find that 21 file systems may return the undocumented `EIO` error (column k) for the system call `mkdir`. Note that table entries marked with a hyphen represent *documented* error codes for the respective system calls. For instance, the error code `EACCES` or permission denied (column b) is documented for the system call `chdir`. When many file systems return the same undocumented error for a given system call, this hints that the documentation may be incomplete. On the other hand, if only a few file systems return a given undocumented error code, that suggests that the documentation may be correct but the file systems are using inappropriate error codes. In either case, mismatches are signs of trouble.

The results shown in Table 3 can also be used to confirm that certain undocumented errors are indeed never returned by any file-system implementation (their count is zero). Additionally, we can retrieve the list of undocumented errors per system call.

Note that we analyze each file-system implementation separately along with the virtual file system (VFS). This leads to duplication of VFS-related reports when aggregating the results across all file systems. Unfortunately, it is not easy to determine whether a report should be attributed to the VFS. We adopt a heuristic that classifies bug reports based on the sample traces. A report is marked as file-system specific if the corresponding sample trace mentions that given file system, otherwise the report is attributed to VFS. Table 3 shows the results after removing duplicates: 1,784 undocumented error-code instances are unexpectedly returned across the 52 file systems and the VFS. [1]. Table 4 shows detailed bug-report classification results for a subset of file systems: CIFS, ext3, IBM JFS, ReiserFS and XFS. Bug reports have been sent to the corresponding developers for further inspection.

File System	FS Specific #	VFS #	Total #
CIFS	131	19	150
ext3	48	73	121
IBM JFS	44	74	118
ReiserFS	87	21	108
XFS	55	23	78

Table 4. Distribution of bug reports

File System	Total #	Unique #	Top Error	Top #
SMB	255	26	`ENODEV`	20
CIFS	131	18	`ENODEV`	26
Coda	113	20	`ENXIO`	26
ReiserFS	87	17	`EIO`	13
ext2	87	20	`EIO`	13

Table 5. File systems with the most undocumented error codes

Error Code	Total #	FS #	System Call #
`EIO`	274	21	14
`EROFS`	199	21	12
`ENOMEM`	185	26	14
`EINVAL`	176	22	21
`ENODEV`	106	21	27

Table 6. Undocumented error codes most commonly returned

If no duplicate-removal heuristic is used, 4,565 undocumented error-code instances are found. A more aggressive heuristic could mark reports as file-system specific only if the undocumented error originates in file-system code (based on sample traces). This leaves 699 instances after duplicate removal. Note that any heuristic based on sample traces will not be complete as only one sample trace is considered for each report.

It is sobering to observe that every single system call analyzed exhibits numerous mismatches; none of the 42 system calls emerges trouble-free. Likewise, not a single file system completely operates within the confines of the documented error codes for all implemented system calls. Table 5 shows the top five file systems that return the most undocumented error instances. SMB is at the top of the list with a total of 255 instances, from which we find 26 different error codes. The error code with the most instances (20) is `ENODEV` (no such device). Table 6 shows the top five undocumented error codes with the most instances across all file systems. `EIO` (I/O error) tops the list with 274 instances, accounting for 15% of all undocumented errors reported in Table 3.

Table 7 presents more detailed results for our subset of file systems, plus the shared VFS layer. We list the undocumented errors for each system call under consideration. A file system returns a given undocumented error code if the corresponding bullet is filled (●). For some system calls such as `utime`, all file systems return the same undocumented error `ENOMEM` (among others). As discussed earlier, this hints that the documentation may be incomplete as these file systems are among the most popular and widely used. On the other hand, blame is harder to assign for other system calls

[1] The VFS is treated as a separate entity after bug-report classification. Thus, the maximum possible count in each cell of Table 3 is 53.

Call	Error	File Sys c	e	j	r	x	v
access	EBADF	○	○	○	○	○	●
chdir	EROFS	●	○	○	●	●	●
chmod	EFBIG	●	○	○	●	○	○
	ERANGE	○	●	●	○	○	○
	EEXIST	○	●	●	○	○	○
	EAGAIN	●	○	○	○	○	○
	ENOSPC	○	●	●	○	○	○
	EINVAL	●	●	●	●	○	○
	ETXTBSY	●	○	○	●	○	○
	ENODEV	●	○	○	○	○	○
chown	EFBIG	●	○	○	●	○	○
	ERANGE	○	●	●	○	○	○
	EEXIST	○	●	●	○	○	○
	EAGAIN	●	○	○	○	○	○
	ENOSPC	○	●	●	○	○	○
	EINVAL	●	●	●	●	○	○
	ETXTBSY	●	○	○	●	○	○
	ENODEV	●	○	○	○	○	○
chroot	EBADF	○	○	○	○	○	●
	EROFS	●	○	○	●	●	●
dup	ENOMEM	●	○	○	●	○	●
dup2	ENOMEM	○	○	○	○	○	●
	EINVAL	○	○	○	○	○	●
fchdir	EROFS	●	○	○	●	●	●
fchmod	EFBIG	●	○	○	●	○	○
	ERANGE	○	●	●	○	○	○
	EEXIST	○	●	●	○	○	○
	EAGAIN	●	○	○	○	○	○
	ENOSPC	○	●	●	○	○	○
	EINVAL	●	●	●	●	○	○
	ETXTBSY	●	○	○	●	○	○
	ENODEV	●	○	○	○	○	○
fchown	EFBIG	●	○	○	●	○	○
	ERANGE	○	●	●	○	○	○
	EEXIST	○	●	●	○	○	○
	EAGAIN	●	○	○	○	○	○
	ENOSPC	○	●	●	○	○	○
	EINVAL	●	●	●	●	○	○
	ETXTBSY	●	○	○	●	○	○
	ENODEV	●	○	○	○	○	○
fdatasync	ENOMEM	●	○	○	○	○	○
	EAGAIN	●	○	○	○	○	○
	ENODEV	●	○	○	○	○	○
	ENOENT	●	○	○	○	○	○
	EINVAL	●	○	○	●	●	●
flock	EAGAIN	●	○	○	●	●	●
	ENOENT	●	○	○	●	●	●
	EIO	●	○	○	●	●	●
	ENOMEM	●	○	○	●	●	●

Call	Error	File Sys c	e	j	r	x	v
fstat	EAGAIN	●	○	○	○	○	○
	ENODEV	●	○	○	○	○	○
	EIO	●	○	○	●	●	●
	EINVAL	●	○	○	○	○	○
fstatfs	ENODEV	●	○	○	●	●	●
fsync	ENOMEM	●	○	○	○	○	○
	EAGAIN	●	○	○	○	○	○
	ENODEV	●	○	○	○	○	○
	ENOENT	●	○	○	○	○	○
	EINVAL	●	○	○	●	●	●
ftruncate	ERANGE	○	●	●	○	○	○
	ENOMEM	●	●	●	●	○	○
	EEXIST	○	●	●	○	○	○
	EAGAIN	●	○	○	●	●	●
	ENOSPC	○	●	●	○	○	○
	ENODEV	●	○	○	○	○	○
getdents	EAGAIN	●	●	○	○	○	○
	ENOSPC	○	●	○	○	○	○
	ENOMEM	●	●	●	●	○	○
	EIO	●	○	○	●	○	●
	ENODEV	●	○	○	○	○	○
ioctl	EISDIR	○	○	○	○	●	○
	EFBIG	●	○	○	●	●	●
	EPERM	●	○	○	●	●	●
	ENOTDIR	○	○	○	○	●	○
	ESRC	○	○	○	○	●	○
	ENOMEM	●	○	○	●	●	●
	EACCES	●	●	●	○	●	○
	EBUSY	○	●	○	○	○	○
	ENOENT	●	○	○	○	●	○
	EAGAIN	●	○	○	○	●	○
	ENOSPC	○	●	○	○	●	○
	EROFS	○	●	●	●	●	○
	ESPIPE	●	○	○	●	●	●
	EIO	●	○	○	●	●	●
	ENODEV	●	○	○	●	●	●
lchown	EFBIG	●	○	○	●	○	○
	ERANGE	○	●	●	○	○	○
	EEXIST	○	●	●	○	○	○
	EAGAIN	●	○	○	○	○	○
	ENOSPC	○	●	●	○	○	○
	EINVAL	●	●	●	●	○	○
	ETXTBSY	●	○	○	●	○	○
	ENODEV	●	○	○	○	○	○
link	ENODEV	●	○	○	○	○	○
	EAGAIN	●	●	○	○	○	○
	EBUSY	○	○	○	●	○	○
	EINVAL	●	●	○	●	○	●
	EBADF	○	○	○	○	○	●

Call	Error	File Sys c	e	j	r	x	v
lstat	EAGAIN	●	○	○	○	○	○
	ENODEV	●	○	○	○	○	○
	EROFS	●	○	○	●	●	●
	EIO	●	○	○	●	●	●
	EINVAL	●	○	○	○	○	●
mkdir	EBADF	○	○	○	○	○	●
	EFBIG	○	○	○	●	○	○
	ERANGE	○	○	●	○	○	○
	EMLINK	○	●	●	○	○	○
	EBUSY	○	○	○	●	○	○
	ETXTBSY	○	○	○	●	○	○
	EAGAIN	●	●	○	○	○	○
	ENODEV	●	○	○	○	○	○
	EIO	●	○	○	●	●	●
	EINVAL	●	●	●	●	●	○
mknod	EBADF	○	○	○	○	○	●
	EFBIG	○	○	○	●	○	○
	ERANGE	○	○	●	○	○	○
	EMLINK	○	○	●	○	○	○
	EBUSY	○	○	○	●	○	○
	ETXTBSY	○	○	○	●	○	○
	EAGAIN	●	●	○	○	○	○
	EIO	●	○	○	●	●	●
	ENODEV	●	○	○	○	○	○
mount	EBADF	○	○	○	○	○	●
	EROFS	●	○	○	●	●	●
	EIO	●	○	○	●	●	●
nfsservctl	EFAULT	○	○	○	○	○	●
	EINVAL	○	○	○	○	○	●
read	ENOMEM	●	○	○	●	○	●
	ENOENT	●	○	○	●	○	●
	ENODEV	●	○	○	○	○	○
readlink	EBADF	○	○	○	○	○	●
	EROFS	●	○	○	●	●	●
readv	EBADF	●	○	○	●	○	●
rename	EBADF	●	○	○	●	●	●
	EPERM	●	○	○	●	●	●
	EEXIST	●	○	○	●	●	●
	EAGAIN	●	●	○	○	○	○
	EIO	●	○	○	●	●	●
	ENODEV	●	○	○	○	○	○
rmdir	EISDIR	●	○	○	●	●	●
	EBADF	○	○	○	○	○	●
	EEXIST	○	○	●	○	○	○
	EAGAIN	●	○	○	○	○	○
	EIO	●	○	○	●	●	●
	ENODEV	●	○	○	○	○	○
select	EFAULT	○	○	○	○	○	●

Call	Error	File Sys c	e	j	r	x	v
stat	EAGAIN	●	○	○	○	○	○
	ENODEV	●	○	○	○	○	○
	EROFS	●	○	○	●	●	●
	EIO	●	○	○	●	●	●
	EINVAL	●	○	○	○	○	●
statfs	EROFS	●	○	○	●	●	●
	ENODEV	●	○	○	●	●	●
symlink	EBADF	○	○	○	○	○	●
	EFBIG	○	○	○	●	○	○
	EMLINK	○	○	●	○	○	○
	EBUSY	○	○	○	●	○	○
	EAGAIN	●	●	○	○	○	○
	ENODEV	●	○	○	○	○	○
	ETXTBSY	○	○	○	●	○	○
	EINVAL	●	●	●	●	○	○
truncate	ERANGE	○	●	●	○	○	○
	ENOMEM	●	○	○	●	●	●
	EEXIST	○	●	●	○	○	○
	EAGAIN	●	○	○	●	●	●
	ENOSPC	○	●	●	○	○	○
	ENODEV	●	○	○	○	○	○
umount	EBADF	○	○	○	○	○	●
	EROFS	●	○	○	●	●	●
	EIO	●	○	○	●	●	●
unlink	EBADF	○	○	○	○	○	●
	ETXTBSY	●	○	○	○	○	○
	EBUSY	●	○	○	●	●	●
	EEXIST	○	○	●	○	○	○
	EAGAIN	●	○	○	○	○	○
	EINVAL	●	○	○	○	○	○
	ENODEV	●	○	○	○	○	○
uselib	ENOMEM	●	○	○	●	●	●
	EFAULT	●	○	○	●	●	●
	EINVAL	●	○	○	●	●	●
ustat	ENODEV	○	○	○	○	○	●
utime	EBADF	○	○	○	○	○	●
	EFBIG	●	○	○	●	○	○
	EFAULT	○	○	○	○	○	●
	ENOTDIR	●	○	○	●	●	●
	ENOMEM	●	●	●	●	●	●
	EEXIST	○	●	●	○	○	○
	ETXTBSY	●	○	○	●	○	○
	EAGAIN	●	○	○	○	○	○
	ENOSPC	○	●	●	○	○	○
	ENODEV	●	○	○	○	○	○
	ERANGE	○	●	●	○	○	○
	EIO	●	○	○	●	●	●
	EINVAL	●	●	●	●	○	●
write	ENOMEM	●	○	○	○	○	○
	ENOENT	●	○	○	●	○	●
	ENODEV	●	○	○	○	○	○
writev	EBADF	●	○	○	●	○	●

Table 7. Undocumented error codes returned per system call. Bullets mark undocumented error codes returned (●) or not returned (○) by CIFS (c), ext3 (e), IBM JFS (j), ReiserFS (r), XFS (x), and VFS (v).

File System	KLOC	Unoptimized → Optimized # of Variables	# of Rules	Time Used	Memory Used
CIFS	90	37,504 → 1,972	117,300 → 89,131	1:00:12 → 0:05:16	1.69 GB → 0.40 GB
ext3	82	38,094 → 2,119	131,274 → 91,418	1:32:04 → 0:05:50	1.73 GB → 0.48 GB
IBM JFS	91	36,531 → 1,922	129,999 → 91,025	1:50:30 → 0:05:49	1.54 GB → 0.52 GB
ReiserFS	86	42,249 → 1,892	143,827 → 101,958	1:36:43 → 1:49:31	1.87 GB → 0.53 GB
XFS	159	55,430 → 1,076	175,683 → 137,074	4:17:13 → 1:12:11	2.26 GB → 0.98 GB

Table 8. Analysis performance and effectiveness of optimizations. KLOC gives the size of each file system in thousands of lines of code, including 59 KLOC of shared VFS code. Time is given as h:mm:ss.

such as `mknod`. For `fdatasync`, we posit mistakes on both sides: `EINVAL` may be incorrectly omitted from the documentation, and CIFS may be returning a variety of inappropriate error codes.

It is also possible that implementation and documentation are both correct, but that our analysis claims an error code can be returned when it actually cannot. The effect of such false positives can be multiplied if a single analysis-fooling code construct is copied and pasted into many file systems. The sample paths presented for each error code may help programmers recognize if this is happening; further study of this possibility is left for future work and pending feedback from developers.

6.3 Performance

We perform our experiments on a dual 3.2 GHz Intel processor workstation with 3 GB RAM. Table 8 shows the sizes of file systems (in thousands of lines of code) and the time and memory required to analyze each. We restrict our focus to the five popular file systems presented in detail in Table 7. We present running times without and with the optimizations discussed in Section 5. We give the total running time, which includes 1) extracting the push-down system, 2) solving the poststar query, and 3) traversing witnesses to produce the sample paths. The second phase is the most expensive, consuming roughly 76% of the running time before optimizations. For the five file systems under consideration, the running time after optimizations ranges from just over five minutes (CIFS, ext3 and IBM JFS) to less than two hours (ReiserFS). IBM JFS shows the most significant difference between running times, with optimizations making the analysis nineteen times faster. Note that running times decrease in general, except for ReiserFS. We find that the running time for the second phase is indeed reduced considerably, from 1:35:28 to 0:02:25. However the usually cheap third phase becomes fairly expensive, from 0:00:19 to 1:45:39. This is because sample traces are chosen non-deterministically. In this particular case, and no other, extremely long traces happen to be chosen when running the optimized analysis, slowing down the running time significantly. Deterministically selecting short (or minimal) paths remains future work.

Table 8 shows additional information related to the optimizations performed. We find that about 96% of the variables in the file-system implementations under consideration (including the VFS) cannot possibly contain error codes. Filtering out irrelevant variables reduces their count from an average of $41,961$ to just $1,796$. As a consequence, the size of the weights is reduced considerably, boosting performance. Similarly, rule compression leads to a 27% decrease in the number of rules used to model the control flow of the file systems. The number of rules decreases from an average of $139,616$ to $102,121$, which translates into fewer weights to calculate and consequently into a faster analysis. The analysis also dramatically decreases memory usage, saving an average of 1.24 GB with respect to the unoptimized version.

7. Conclusions

We describe how to adapt an existing error-propagation analysis to find the list of error codes returned by each program function. We analyze 52 Linux file systems, including CIFS, ext3, IBM JFS, ReiserFS and XFS. After retrieving data for 42 file-related system calls, we compare against the Linux manual pages, finding 1,784 undocumented error instances across all file systems. We also find that the analysis optimizations described in this paper contribute to decreasing running time and memory consumption considerably.

8. References

[1] E. Berglund and M. Priestley. Open-source documentation: in search of user-driven, just-in-time writing. In *SIGDOC*, pages 132–141, 2001.

[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] R. P. L. Buse and W. Weimer. Automatic documentation inference for exceptions. In B. G. Ryder and A. Zeller, editors, *ISSTA*, pages 273–282. ACM, 2008.

[4] L. H. Etzkorn, W. E. H. Jr., and C. G. Davis. Automated reusability quality analysis of oo legacy software. *Information & Software Technology*, 43(5):295–308, 2001.

[5] C. Kaner. Liability for defective documentation. In S. B. Jones and D. G. Novick, editors, *SIGDOC*, pages 192–197. ACM, 2003.

[6] N. Kidd, T. Reps, and A. Lal. WALi: A C++ library for weighted pushdown systems. `http://www.cs.wisc.edu/wpis/wpds/`, 2009.

[7] 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.

[8] 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.

[9] T. Lethbridge, J. Singer, and A. Forward. How software engineers use documentation: The state of the practice. *IEEE Software*, 20(6): 35–39, 2003.

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

[11] S. N. I. Mount, R. M. Newman, R. J. Low, and A. Mycroft. Exstatic: a generic static checker applied to documentation systems. In S. R. Tilley and S. Huang, editors, *SIGDOC*, pages 52–57. ACM, 2004.

[12] 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.

[13] 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.

[14] G. Robles, J. M. González Barahona, and J. L. Prieto Martínez. Assessing and evaluating documentation in libre software projects. In T. Wasserman and M. Pal, editors, *Workshop on Evaluation Frameworks for Open Source Software (EFOSS)*, Como, Italy, June 2006. International Federation for Information Processing.

[15] C. Rubio-González, H. S. Gunawi, B. Liblit, R. H. Arpaci-Dusseau, and A. C. Arpaci-Dusseau. Error Propagation Analysis for File Systems. In *Proceedings of the ACM SIGPLAN 2009 Conference on Programming Language Design and Implementation*, Dublin, Ireland, June 15–20 2009.

[16] 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.

[17] C. Schönberg, F. Weitl, M. Jaksic, and B. Freitag. Logic-based verification of technical documentation. In U. M. Borghoff and B. Chidlovskii, editors, *ACM Symposium on Document Engineering*, pages 251–252. ACM, 2009.

[18] D. Schreck, V. Dallmeier, and T. Zimmermann. How documentation evolves over time. In M. D. Penta and M. Lanza, editors, *IWPSE*, pages 4–10. ACM, 2007.

[19] J. Singer. Practices of software maintenance. In *ICSM*, pages 139–145, 1998.

[20] L. Tan, D. Yuan, G. Krishna, and Y. Zhou. /*icomment: bugs or bad comments?*/. In T. C. Bressoud and M. F. Kaashoek, editors, *SOSP*, pages 145–158. ACM, 2007.

[21] G. Venolia. Textual allusions to artifacts in software-related repositories. In S. Diehl, H. Gall, and A. E. Hassan, editors, *MSR*, pages 151–154. ACM, 2006.

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

[23] Y. Xie and D. R. Engler. Using redundancies to find errors. *IEEE Trans. Software Eng.*, 29(10):915–928, 2003.