Finding Error-Handling Bugs in Systems Code Using Static Analysis*

Cindy Rubio-González Ben Liblit

Computer Sciences Department, University of Wisconsin–Madison
1210 W Dayton St., Madison, Wisconsin, United States of America
{crubio, liblit}@cs.wisc.edu

ABSTRACT

Run-time errors are unavoidable whenever software interacts with the physical world. Unchecked errors are especially pernicious in operating system file management code. Transient or permanent hardware failures are inevitable, and error-management bugs at the file system layer can cause silent, unrecoverable data corruption. Furthermore, even when developers have the best of intentions, inaccurate documentation can mislead programmers and cause software to fail in unexpected ways.

We use static program analysis to understand and make error handling in large systems more reliable. We apply our analyses to numerous Linux file systems and drivers, finding hundreds of confirmed error-handling bugs that could lead to serious problems such as system crashes, silent data loss and corruption.

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

Keywords

Static program analysis, weighted pushdown systems, interprocedural dataflow analysis

1. INTRODUCTION

Incorrect error handling is a longstanding problem in many application domains. Ideally, some action should be taken whenever an error occurs (e.g., notification, attempted recovery, etc.), however that is often overlooked. Error handling accounts for a significant portion of the code in large software systems. Nevertheless error handling is not the primary concern to be implemented [1]. Error-handling code is in general the least understood, documented and tested part of a system, and as a consequence, the buggiest [5]. It is difficult to write correct error handling. Exceptional conditions must be considered during all phases of software development [18], introducing interprocedural control flow that can be difficult to reason about [3, 19, 22]. As a result, error-handling code is usually scattered across different functions and files, making software more complex and less reliable.

Modern programming languages such as Java, C++ and C# provide exception-handling mechanisms. On the other hand, C does not have explicit exception-handling support, thus programmers have to emulate exceptions in a variety of ways. The return-code idiom is among the most popular idioms used in large C programs, including operating systems. Errors are represented as simple integer codes, where each integer value represents a different kind of error. These error codes are propagated through conventional mechanisms such as variable assignments and function return values. Despite having exception-handling support, many C++ applications also adopt the return-code idiom. Unfortunately, the use of such idioms is significantly error-prone and effort-demanding.

Our goal is to use static program analysis to understand and make error handling in large systems more reliable. This includes 1) finding how error codes propagate through large and complex systems; 2) using error-propagation information to identify different kinds of error-handling bugs, many of which could lead to serious problems such as system crashes, silent data loss and corruption; 3) finding whether error-reporting program documentation accurately reflects real system behavior; and 4) extracting high-level error-handling specifications that describe how the system should recover from errors (in progress). We have applied our analyses to numerous real-world, widely-used Linux file systems such as ext3 and ReiserFS, written in C, and we have found hundreds of error-handling bugs. The trustworthiness of this kind of applications in handling errors is an upper bound on the trustworthiness of all storage-dependent user applications. We are currently extending our framework to analyze applications written in other languages.

2. ERROR-HANDLING BUGS

We focus on finding three particular kinds of error-handling bugs. The following subsections briefly describe them and present some real-world examples.

2.1 Dropped Unhandled Errors

Error-management bugs at the file system layer can cause silent, unrecoverable data corruption. We find error-code instances that vanish before proper handling is performed in

*Supported in part by AFOSR grant FA9550-07-1-0210; DoE contract DE-SC0002153; LLNL contract B580360; NSF grants CCF-0621487, CCF-0701957, and CCF-0953478; and a generous gift from the Mozilla Corporation. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the sponsoring institutions.

```
1  int txCommit(...) {
2    ...
3    if (isReadOnly(...)) {
4      rc = −EROFS;
5      ...
6      goto TheEnd;
7    } ...
8
9    if (rc = diWrite(...)) // may return EIO
10     txAbort(...);
11
12 TheEnd: return rc;
13 }
14
15 int diFree(...) {
16   ...
17   rc = txCommit(...); // rc may contain error
18   ...
19   return 0; //rc out of scope
20 }
```

Figure 1: Example of an out-of-scope error in IBM JFS

Linux file systems [25]. We identify three general cases in which unhandled errors are commonly lost: the variable holding the unhandled 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.

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

2.2 Defective Error/Pointer Interactions

Error codes are often temporarily or permanently encoded into pointer values as they propagate. Linux introduces two functions to convert (cast) error codes from integers to pointers and vice versa: ERR_PTR and PTR_ERR. The Boolean function IS_ERR is used to determine whether a pointer variable contains an error code. Error-valued pointers are not valid memory addresses, and therefore require special care by programmers. Misuse of pointer variables that store error codes can lead to serious problems such as system crashes, data corruption, unexpected results, etc. We find three classes of bugs relating error valued pointers [24]: (1) bad pointer dereferences, (2) bad pointer arithmetic, and (3) bad pointer overwrites.

Figure 2 shows an example of a bad dereference. Function fill_super in the Coda file system calls function cnode_make on line 5, which may return the integer error code ENOMEM while storing the same error code in the pointer variable root. The error is logged on line 8. If root is not NULL (line 14), then function iput in the Virtual File System (VFS) is invoked with variable root as parameter. This function dereferences the potential error-valued pointer parameter inode on line 21.

```
1  static int fill_super(...) {
2    int err;
3    inode *root = ...;
4    ...
5    err = cnode_make(&root,...); // err and root may get error
6
7    if ( err || !root ) {
8      printk("... error %d\n", err);
9      goto fail;
10   }
11   ...
12   fail:
13   ...
14   if (root) // root may contain an error
15     iput(root);
16   ...
17 }
18
19 void iput(inode *inode) {
20   if (inode) {
21     BUG_ON(inode−>i_state == ...); // bad pointer deref
22     ...
23   }
24 }
```

Figure 2: Example of a bad pointer dereference. The Coda file system propagates an error-valued pointer which is dereferenced by the VFS (function iput).

2.3 Error Code Mismatches Between Code and Documentation

Inaccurate documentation can mislead programmers and cause software to fail in unexpected ways. We consider 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 [23].

3. APPROACH

Finding error-handling bugs requires understanding how error codes propagate through systems code. We have designed and implemented an error propagation analysis (see Section 3.1) that finds the set of values that variables may contain at different program points. Given this information, the error-handling bugs described in Section 2 are detected using a second pass over the code (see Section 3.2).

3.1 Error Propagation Analysis

The main component of our framework is an interprocedural, flow- and context-sensitive static analysis that tracks errors as they propagate. The analysis resembles an over-approximating counterpart to a typical (under-approximating) copy constant propagation analysis, but with certain additional specializations for Linux code (which plan to extend for user applications). For

example, we recognize high-level error-handling patterns found in Linux, which allow us to distinguish between handled and unhandled error codes. 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.

We formulate and solve the error propagation problem using weighted pushdown systems (WPDS). A WPDS is a useful dataflow engine for problems that can be encoded with suitable weight domains, computing the meet-over-all-paths solution. A WPDS consists of three main components: a pushdown system (PDS), a bounded idempotent semiring, and a mapping from PDS rules to associated weights. The PDS models the control flow of the program. The bounded idempotent semiring defines, for example, what to do when the program control flow merges (combine operator). We define each element of the bounded idempotent semiring and transfer functions for each construct in the program [23–25].

The state of the program at each program point is represented by a mapping from variables to sets containing variables, error values (one of 34 predefined integer values), OK (used to represent all non-error constants) and/or *uninitialized* (used to represent uninitialized values). These mappings are the weights. Transfer functions define the new state of the program as a function of the old state. This is interpreted as giving the possible values of each variable following execution of a given program statement in terms of the other values of constants and variables before that statement.

After solving the WPDS, we can determine, at each program point, the set of error codes each variable might contain. Given this information, we detect a variety of error-handling related bugs using a second pass over the code.

3.2 Finding Error-Handling Bugs

We perform a poststar query [21] on the WPDS, with the beginning of the program as the starting configuration. For kernel analysis, we synthesize a main function that nondeterministically calls all exported entry points of the file system under analysis. The result is a weighted automaton. We apply the *path_summary* algorithm of Lal et al. [16] to read out weights from this automaton. We retrieve the weight representing execution from the beginning of the program to any particular point of interest. Program points of interest are determined by the kind of error-handling bug we are looking for.

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

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

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

Different program points and rules are used to find bad error/-pointer interactions [24] and error-code mismatches between documentation and real code [23].

3.3 Sample Bug Report

In addition to the exact program location at which error-handling bugs from Section 2 are found, our tool also produces a sample path for each bug describing how a given error code could have reached a particular program point.

Figure 3 shows a more detailed version of the VFS bad pointer dereference from Figure 2. The error ENOMEM is first returned by function iget in Figure 3a and propagated through three other functions (cnode_make, fill_super and iput, in that order) across two other files (shown in Figure 3b and Figure 3c). The bad dereference occurs on line 1325 of file fs/inode.c in Figure 3c. The sample path produced by our tool is shown in Figure 3d. This path is automatically filtered to show only program points directly relevant to the propagation of the error. We also provide an unfiltered sample path, not included here, showing every single step from the program point at which the error is generated (i.e., the error macro is used) to the program point at which the problem occurs. We list all other error codes, if any, that may also reach there.

4. EXPERIMENTAL EVALUATION

Our implementation uses the CIL C front end [20] to apply preliminary source-to-source transformations on Linux kernel code. This includes redefining error code macros as distinctive expressions to avoid mistaking regular constants for error codes. We then traverse the CFG and emit a textual representation of the WPDS. Our separate analysis tool uses the WALi WPDS library [15] to perform the interprocedural dataflow analysis on this WPDS. Within our WALi-based analysis code, we encode weights using *binary decision diagrams* (BDDs) [2] as implemented by the BuDDy BDD library [17].

We analyze 52 different Linux file systems and 4 drivers separately along with the VFS and the Memory Management module (mm). The following sections present the results for each kind of error-handling bug.

4.1 Dropped Unhandled Errors

Developers and a local expert manually inspected each bug report produced for five widely-used file systems (CIFS, ext3,

```
58 inode * iget(...) {
   ...
67   if (!inode)
68     return ERR_PTR(-ENOMEM);
   ...
81 }
   ...
89 int cnode_make(inode **inode, ...) {
   ...
101   *inode = iget(sb, fid, &attr);}
102   if ( IS_ERR(*inode) ) {
103     printk("...");
104     return PTR_ERR(*inode);
105   }
```

(a) File fs/coda/cnode.c

```
143 static int fill_super(...) {
   ...
194   error = cnode_make(&root, ...);
195   if (error || !root) {
196     printk("... error %d\n", error);
197     goto error;
198   }
   ...
207  error:
208   bdi_destroy(&vc->bdi);
209  bdi_err:
210   if (root)
211     iput(root);
   ...
216 }
```

(b) File fs/coda/inode.c

```
1322 void iput(inode *inode) {
1323
1324   if (inode) {
1325     BUG_ON(inode->i_state == ...);
1326
1327     if (...)
1328       iput_final(inode);
1329   }
1330 }
```

(c) File fs/inode.c

```
fs/coda/cnode.c:68: an unchecked error may be returned
fs/coda/cnode.c:101:"*inode" receives an error from function "iget"
fs/coda/cnode.c:104:"*inode" may have an unchecked error
fs/coda/inode.c:194:"root" may have an unchecked error
fs/coda/inode.c:211:"root" may have an unchecked error
fs/inode.c:1325: Dereferencing variable inode, which may contain error code ENOMEM
```

(d) Sample trace

Figure 3: Example of diagnostic output

ext4, IBM JFS and ReiserFS) along with the VFS in the Linux 2.6.27 kernel, confirming 312 true bugs. Table 1 shows the results per bug category. We find that 86% of the dropped unhandled errors correspond to the category of unsaved errors. The most common unsaved error code is EIO, followed by ENOSPC and ENOMEM. Close inspection reveals serious inconsistencies in use of some functions' return values. For example, we find one function whose returned error code is unsaved at 35 call sites, but saved at 17 others. In this particular example, 9 out of the 35 bad calls are true bugs; the rest are false positives. When we alerted developers, some suggested they could use annotations to explicitly mark cases where error codes are intentionally ignored.

Note that false positives correspond to cases in which the developers and local expert judge that erros are safely overwritten, out of scope or unsaved. The claim that unhandled errors are lost is true, and in this sense the analysis is providing correct, precise information for the questions it was designed to answer. Rubio-González et al. [25] describe these results in greater detail, including a breakdown per file system.

Table 1: Dropped unhandled errors for subset of file systems

Bug Category	True Bugs	False Positives	Total
Overwritten	25	44	69
Out of Scope	18	48	66
Unsaved	269	97	366
Total	312	189	501

Table 2: Defective error/pointer interactions

Bug Category	True Bugs	False Positives	Total
Bad Dereference	36	5	41
Bad Arithmetic	16	1	17
Pointer Overwrite	4	34	38
Total	56	40	96

4.2 Defective Error/Pointer Interactions

We identify 56 true bugs among 52 Linux file system implementations (including widely-used file systems such as ext3 and ReiserFS) and 4 drivers (SCSI, PCI, IDE, ATA) along with the VFS and mm in the Linux 2.6.35.4 kernel. Table 2 shows the results per bug category. We find that bad pointer dereferences are the most common (64% of the true bugs). We find that most bad pointer dereferences are due to dereferencing pointer variables without first checking them for errors. In many cases there is a check for NULL, however the error check is missing.

Most false positives (85%) correspond to overwrites and are due to cloned code. In other words, a few false-positive reports are generated multiple times when analyzing file systems separately. The reports correspond to code copied among different implementations. Because our tool does not automatically identify these "duplicated" reports, we count them multiple times, hence the high number. Rubio-González and Liblit [24] discuss the results in more detail.

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

	Error Code																							
SysCall	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	Total
chdir	0	-	0	-	0	0	0	1	1	-	0	0	0	0	0	-	-	0	1	0	21	0	0	0	24
chown	1	-	2	-	0	4	3	4	11	-	0	1	0	1	2	-	-	5	2	4	-	0	4	1	45
mkdir	1	-	3	1	1	-	1	3	11	21	0	1	7	1	2	-	-	-	2	2	-	0	2	1	60
read	0	1	-	-	0	0	0	-	-	-	-	0	0	0	2	20	21	0	0	0	0	0	0	0	44
rmdir	0	-	2	1	-	1	0	3	-	21	21	0	0	0	1	-	-	2	1	0	-	0	0	0	53
write	0	1	-	-	0	0	-	-	-	-	0	0	0	0	2	20	2	-	0	0	0	1	0	0	26

Table 4: Analysis performance for a subset of file systems and drivers when finding bad pointer dereferences and bad pointer arithmetic. Sizes include 133 KLOC of shared VFS and mm code.

File System	KLOC	Time (min:sec)	Mem (GB)
Coda	136	2:54	0.83
FAT	140	3:06	0.88
NTFS	162	4:12	1.37
PCI	191	3:24	1.00
ReiserFS	161	4:06	1.36
SCSI	703	11:00	2.42

4.3 Error Code Mismatches Between Code and Documentation

We find the set of error codes that 42 Linux file-related system calls return across 52 different file system implementations and the VFS in the Linux 2.6.32.4 kernel. We compare these error codes against version 2.39 of the Linux manual pages for each of the 42 file-related system calls. We report 1,784 undocumented error-code instances affecting all file systems and system calls examined.

Table 3 shows the results for a subset of system calls and error codes. 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 j) 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. Rubio-González and Liblit [23] discuss the results.

4.4 Performance

We use a dual 3.2 GHz Intel Pentium 4 processor workstation with 3 GB RAM to run our experiments. As mentioned earlier, we run our analyses on each file system implementation and driver separately along with shared code (VFS and mm). We apply important optimizations that allow our analyses to run in a matter of minutes [23]. Table 4 shows running time and memory usage for a subset of file systems and drivers when running the analysis that finds bad pointer dereferences and bad pointer arithmetic. The table also includes the size (in thousands of lines of code). Running times range from 2 minutes 54 seconds to 11 minutes for this subset of modules. Memory usage ranges from 0.83 to 2.42 GB.

5. RELATED WORK

Numerous proposals detect or monitor error propagation patterns at run time, typically during controlled in-house testing with fault-injection to elicit failures [4, 6, 7, 10–14, 26]. Work by Guo et al. [9] on dynamic abstract type inference could be used to distinguish error-carrying variables from ordinary integers, but this approach also requires running on real (error-inducing) inputs. In contrast to these dynamic techniques, our approach offers the stronger assurances of static analysis, which become especially important for critical software components such as operating system kernels. Storage errors are rare enough to be difficult to test dynamically, but can be catastrophic when they do occur. This is precisely the scenario in which intensive static analysis is most suitable.

Gunawi et al. [8] highlight error code propagation bugs in file systems as a special concern. Gunawi's proposed Error Detection and Propagation (EDP) analysis is essentially a type inference over the file system's call graph, which finds dropped unhandled errors. Our approach uses a more precise analysis framework that offers flow- and context-sensitivity, finding a larger number of bugs and producing more detailed diagnostic information.

The FiSC system of Yang et al. [27] 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 the kinds of bugs we find and has been applied to only three of Linux's many file systems.

6. CONCLUSIONS

We have contributed to the Linux community by reporting hundreds of error-handling bugs. Although we have focused on Linux so far, this work is not specific to Linux file systems and drivers but can also be applied to other applications that use the return-code idiom. Our tool has been used by the NASA/JPL Laboratory for Reliable Software to check code in the Mars Science Laboratory, where it has found a critical unsaved error in code used for space missions. Another example is the Mozilla code base, which is written in C++, but uses the return-code idiom extensively. We are currently developing a new LLVM-based front end that will allow us to analyze this application and many others.

Our ultimate goal is to use our analyses to automatically extract error-handling specifications that describe how a system detects and recovers from run-time errors. These specifications could be used in many settings. We are particularly interested in testing of error-handling code in large applications.

7. RESEARCH PHILOSOPHY

It is important for me to conduct challenging research that not only has an impact for me, but for many other people as well. That is what I like the most about my current research work. I like the fact that widely-used applications can be made more reliable by using our static program analyses. That is a great way to give back!

My goal is to conduct high quality research. I have always been encouraged to find my own research problems. This has been challenging but also very beneficial. I have learned a lot in the process and it has helped me to stay focused and motivated. My research experience so far has been very rewarding and I consider myself fortunate to work as a research assistant in a research-oriented institution under my advisor's supervision.

8. REFERENCES

[1] M. Bruntink, A. van Deursen, and T. Tourwé. Discovering faults in idiom-based exception handling. In L. J. Osterweil, H. D. Rombach, and M. L. Soffa, editors, *ICSE*, pages 242–251. ACM, 2006.

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

[5] F. Cristian. Exception handling. In *Dependability of Resilient Computers*, pages 68–97, 1989.

[6] 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.

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

[8] 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.

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

[10] 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.

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

[12] 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.

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

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

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

[16] 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.

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

[18] M. Lippert and C. V. Lopes. A study on exception detection and handling using aspect-oriented programming. In *ICSE*, pages 418–427, 2000.

[19] R. Miller and A. Tripathi. Issues with exception handling in object-oriented systems. In *In Object-Oriented Programming, 11th European Conference (ECOOP*, pages 85–103. Springer-Verlag, 1997.

[20] 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.

[21] 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.

[22] M. P. Robillard and G. C. Murphy. Regaining control of exception handling. Technical report, University of British Columbia, Vancouver, BC, Canada, 1999.

[23] C. Rubio-González and B. Liblit. Expect the unexpected: error code mismatches between documentation and the real world. In S. Lerner and A. Rountev, editors, *PASTE*, pages 73–80. ACM, 2010.

[24] C. Rubio-González and B. Liblit. Defective error/pointer interactions in the linux kernel. In F. Tip, editor, *International Symposium on Software Testing and Analysis*, Toronto, Ontario, Canada, July 17–21 2011. To appear.

[25] C. Rubio-González, H. S. Gunawi, B. Liblit, R. H. Arpaci-Dusseau, and A. C. Arpaci-Dusseau. Error propagation analysis for file systems. In M. Hind and A. Diwan, editors, *PLDI*, pages 270–280. ACM, 2009.

[26] 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.

[27] 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.