FINDING ERROR-PROPAGATION BUGS IN LARGE SOFTWARE SYSTEMS USING STATIC ANALYSIS

by

Cindy Rubio González

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN–MADISON

2012

Date of final oral examination: 08/20/2012

The dissertation is approved by the following members of the Final Oral Committee:

Benjamin R. Liblit, Associate Professor, Computer Sciences

Remzi H. Arpaci-Dusseau, Professor, Electrical and Computer Engineering

Susan B. Horwitz, Professor, Computer Sciences

Shan Lu, Assistant Professor, Computer Sciences

Thomas W. Reps, Professor, Computer Sciences

To Mom, Dad, and Sister.

Acknowledgments

This dissertation would have not been completed without the continued encouragement and support of my family, friends, mentors, fellow students, professors, and many others who made it possible for me to be here today.

I dedicate this dissertation to my parents, Victor Rubio and Minerva Gonzalez, and my sister Minerva. Without their support throughout all these years far away from home, I would have not been able to complete this degree. I still remember the long hours my dad spent helping me translate graduate school information when my dream was to pursue a graduate degree in the U.S., and yet I did not speak a word of English. Also, it was my dad who encouraged me to major in Computer Engineering in the first place. Thanks to my mom and my sister for their love and patience.

I would like to express my most sincere gratitude to my advisor Prof. Ben Liblit for his guidance and unconditional support during my Ph.D. studies. He had faith in me since the first time we met, and gave me the opportunity to join his research team the summer before my first semester at UW–Madison. I was not even an official student yet and he was already taking the time to discuss and explore new research ideas with me. I was fortunate to continue being part of his team for the following six years, for which I feel extremely lucky. Ben has taught me how to conduct research while giving me the freedom to make my own decisions, which has played a pivotal role in my development as a researcher.

I would like to thank the members of my thesis committee, professors Remzi Arpaci-Dusseau, Shan Lu, Susan Horwitz, and Thomas Reps for their insightful comments and numerous sugges-

tions to improve this dissertation.

I am immensely thankful to professors Remzi Arpaci-Dusseau, Andrea Arpaci-Dusseau, Thomas Reps, and their students. My dissertation work was originated and inspired from a collaboration with Remzi, Andrea, and their student Haryadi Gunawi. I would also like to thank Tom and his students Nick Kidd and Akash Lal for letting us use their WALi library. Even after graduation, Nick and Akash took the time to continue answering our questions regarding the tool. I would also like to thank Prof. Reps for his advice throughout these years, and for encouraging me to continue pursuing my passion for music by doing a Ph.D. minor in Piano Performance. I would also like to thank professors Mark Hill and Guri Sohi for their advice.

Thanks to Andreas Gal and David Herman from Mozilla Research for their financial support, and their interest in having our tool analyze their code base. Thank you to all Linux and Mozilla developers who took the time to inspect our bug reports and gave us invaluable feedback.

During my studies, I interned at Microsoft Research twice. This was a great experience, and I would like to thank my mentor Patrice Godefroid for the invaluable opportunity, and for still answering my questions and providing support years after my internships.

Thanks to professors John Tang Boyland, Adam Brooks Webber, and Ethan Munson for introducing me to the area of Programming Languages and for all the advice and support provided during my first years of graduate school at UW–Milwaukee.

I would like to thank current and former students in my group: Piramanayagan Arumuga Nainar, Tristan Ravitch, Mark Chapman, Peter Ohmann, Anne Mulhern, Steve Jackson, and Suhail Shergill. In particular, I would like to thank Arumuga and Tristan for reviewing all my papers and sitting through each of my practice talks throughout my entire Ph.D.

Special thanks to Aditya Thakur for the insightful discussions, comments, and suggestions on my papers and presentations. I would also like to thank Arkaprava Basu, Asim Kadav, Bill Harris, Daniel Luchaup, Drew Davidson, Evan Driscoll, Junghee Lim, Kevin Roundy, Lorenzo De Carli, Matt Elder, Prathmesh Prabhu, Rich Joiner, Spyros Blanas, Tycho Andersen, Tushar Sharma, and Venkatesh Srinivasan for attending my talks and providing invaluable feedback.

Thanks to Cathrin Weiss for collaborating with us and for the fun discussions.

I would like to thank Aditya Thakur, Piramanayagan Arumuga Nainar, Michael Bebenita, Theo Benson, Shravan Rayanchu, Akanksha Baid, Yasuko Eckert, Somayeh Sardashti, Hamid Reza Ghasemi, Giridhar Ravipati, Neelam Goyal, Nitin Agrawal, Oriol Vinyals, Swaminathan Sundararaman, Vivek Shrivastava, and Massiel Cisneros for always being there for me.

I am fortunate to be part of a strong network called *Latinas in Computing*. I am particularly grateful to Gilda Garreton, Patty Lopez, Dilma Da Silva, Cecilia Aragon, Gaby Aguilera, Claris Castillo, Raquel Romano, and Rosa Enciso for sharing their experiences with me and offering their advice and help whenever needed.

I would like to thank Hugo and Emy Lopez for opening their house doors, and letting me stay when I was new to this country and did not know anyone. Without their support, I would have not stayed in Wisconsin.

Thanks to Angela Thorp and Cathy Richard for the support throughout all these years in the department.

Last but not least, I would like to thank the rest of my family, in particular, my grandparents and my brother-in-law.

Contents

Contents v

List of Tables ix

List of Figures xii

Abstract xiv

1 Introduction 1

1.1 Why Error Handling? . . . 2

1.2 Why Systems Software? . . . 3

1.3 Linux Error Management . . . 4

1.3.1 Integer Error Codes . . . 4

1.3.2 Consequences of Not Handling Errors . . . 6

1.3.3 Handled vs. Unhandled Errors . . . 6

1.4 Error-Propagation Bugs . . . 7

1.5 Contributions . . . 8

1.6 Dissertation Structure . . . 9

2 Error-Propagation Analysis 11

2.1 Weighted Pushdown Systems . . . 12

2.2 Creating the Weighted Pushdown System . . . 14

2.2.1 Pushdown System 14
2.2.2 Bounded Idempotent Semiring 14
2.2.3 Transfer Functions 17
2.3 Additional Configurable Options 23
2.3.1 Copy Mode vs. Transfer Mode 23
2.3.2 Negative vs. Positive Error Codes 24
2.3.3 Tentative vs. Non-Tentative Errors 25
2.3.4 Error-Handling Patterns 25
2.4 Solving the Dataflow Problem 26
2.5 Analysis Optimizations 26
2.5.1 Reducing Weight Size 27
2.5.2 Reducing the Number of Weights 27
2.5.3 Impact of Optimizations 28
2.6 Framework Components 30
2.6.1 Intermediate Representation 30
2.6.2 Front End 33
2.6.3 Back End 33
2.7 Summary 34

3 Dropped Errors in Linux File Systems 35
3.1 Examples of Dropped Errors 35
3.2 Finding Dropped Errors 37
3.2.1 Program Transformations 37
3.2.2 Error-Propagation Analysis 38
3.3 Describing Dropped Errors 40
3.4 Experimental Evaluation 43
3.4.1 Overwritten Errors 44
3.4.2 Out-of-Scope Errors 48

3.4.3 Unsaved Errors . . . 49
3.5 Performance . . . 50
3.6 Other File Systems . . . 51
3.7 Summary . . . 51

4 Errors Masquerading as Pointers in Linux 53
4.1 Error Transformation in the Linux Kernel . . . 54
4.2 Error-Valued Pointer Bugs . . . 55
4.2.1 Bad Pointer Dereferences . . . 55
4.2.2 Bad Pointer Arithmetic . . . 57
4.2.3 Bad Overwrites . . . 57
4.3 Error Propagation and Transformation . . . 59
4.3.1 Bounded Idempotent Semiring . . . 60
4.3.2 Transfer Functions . . . 60
4.4 Finding and Reporting Bugs . . . 66
4.5 Experimental Evaluation . . . 68
4.5.1 Bad Pointer Dereferences . . . 68
4.5.2 Bad Pointer Arithmetic . . . 73
4.5.3 Bad Overwrites . . . 75
4.5.4 False Negatives . . . 76
4.6 Performance . . . 76
4.7 Other Linux Versions and Code Bases . . . 77
4.8 Summary . . . 78

5 Error-Code Mismatches Between Code and Documentation 79
5.1 Finding Error Return Values . . . 80
5.2 Linux Manual Pages . . . 81
5.3 Experimental Evaluation . . . 81

5.3.1 Undocumented Error Codes 84
5.3.2 Performance 89
5.4 Summary 89

6 Error-Propagation Bugs in User Applications 90
6.1 Case Study: Mozilla Firefox 90
6.1.1 True Bugs 92
6.1.2 Harmless Dropped Errors 95
6.1.3 False Positives 99
6.1.4 Performance 100
6.2 Case Study: SQLite 104
6.2.1 Results 104
6.2.2 Performance 105
6.3 Summary 107

7 Related Work 108
7.1 Error Propagation and Dropped Errors 108
7.2 Errors Masquerading as Pointer Values 110
7.3 Undocumented Error Codes 111

8 Conclusions and Future Directions 113

References 117

List of Tables

2.1 Encoding of control flow as PDS rules 14

2.2 Transfer functions for assignments 18

2.3 Effectiveness of optimizations. KLOC gives the size of each file system in thousands of lines of code, including 59 KLOC of shared VFS code. 29

3.1 Average lengths of full paths and path slices 40

3.2 Summary results for the six case studies. Bug reports are broken down into overwritten, out-of-scope and unsaved. Each category is further divided into true bugs (TB) and false positives (FP). The first column under FPs corresponds to "removable" FPs (FPs that can be removed if our tool recognizes unsafe patterns). The second column corresponds to "unavoidable" FPs (FPs that cannot be automatically removed because significant human intervention is required). The last column (T) gives the total number of bug reports per bug category. Results for unsaved errors were produced in copy mode. 45

3.3 Analysis performance. KLOC gives the size of each file system in thousands of lines of code, including 60 KLOC of shared VFS code. We provide running times for extracting the WPDS textual representation of the program, solving the poststar query, and finding bugs. 50

4.1 Transfer functions for assignments in copy mode 61

4.2 Error-valued pointer dereferences. File systems, modules, and drivers producing no diagnostic reports are omitted. 69
4.3 Bad pointer arithmetic 73
4.4 Analysis performance for a subset of file systems and drivers. Sizes include 133 KLOC of shared VFS and mm code. 77

5.1 Number of file systems per system call returning undocumented errors. a:E2BIG, b:EACCES, c:EAGAIN, d:EBADF, e:EBUSY, f:EEBADHDR, g:EFAULT, h:EBIG, i:EINTR, j:EINVAL, k:EIO, l:EISDIR, m:EMFILE, n:ENLINK, o:ENFILE, p:ENODEV, q:ENOENT, r:ENOMEM, s:ENOSPC, t:ENOTDIR, u:ENXIO, v:EPERM, w:ERANGE, x:EROFS, y:ESPIPE, z:ESRC, aa:ETXTBSY, ab:EXDEV. 82
5.2 Distribution of bug reports 85
5.3 File systems with the most undocumented error codes 85
5.4 Undocumented error codes most commonly returned 86
5.5 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). 87
5.6 Analysis performance for a subset of file systems. KLOC gives the size of each file system in thousands of lines of code, including 59 KLOC of shared VFS code. . . . 88

6.1 Inspected dropped errors in Mozilla Firefox. Results are shown per component, and divided into true bugs, harmless dropped errors (H1: dropped in the process of shutting down, H2: dropped in the process of releasing resources, H3: documented by developer to be ignored, and H4: logged), and false positives (FP1: double error code, FP2: met precondition, and FP3: imprecision in our tool). 93
6.2 Analysis performance for Firefox 104

6.3 Dropped errors in SQLite (preliminary results). The reports are divided into true bugs, harmless dropped errors (H1: dropped in the process of shutting down, H2: dropped in the process of releasing resources, H3: documented by developer to be ignored, and H4: logged), and false positives (FP1: double error code, FP2: met precondition, FP3: infeasible paths, FP4: error masking, and FP5: error hierarchy). 106

6.4 Analysis performance for SQLite 107

List of Figures

1.1 Definition of basic error codes in the Linux kernel 5
1.2 Typical error-checking code example 6
1.3 High-Level Framework Components 10

2.1 An example of a weight 15
2.2 An example of applying the combine operator 16
2.3 An example of applying the extend operator 16
2.4 Removing irrelevant variables a, b and c from the weights 28
2.5 Collapsing rules 29
2.6 Sample program whose intermediate representation is shown in Figure 2.7 30
2.7 The intermediate representation for the program shown in Figure 2.6 31

3.1 Three common scenarios in which unhandled errors are lost 36
3.2 Unsaved ⇒ Out of scope ⇒ Overwrite 38
3.3 Example code fragment and corresponding diagnostic output 41
3.4 Some recurring safe patterns recognized by the analysis 46
3.5 Some recurring unsafe patterns 47

4.1 Examples of error transformation in ReiserFS 55
4.2 Example of a bad pointer dereference. The Coda file system propagates an error-valued pointer which is dereferenced by the VFS (function iput). 56

4.3 Bad pointer arithmetic found in the mm 58
4.4 Two examples of safe error-overwrite patterns 59
4.5 Example making parameter and return value passing explicit. Highlighted assignments emulate transfer functions. 64
4.6 Example of diagnostic output 67
4.7 Example of a bad pointer dereference due to a missing error check in the HFS+ file system 69
4.8 Example of an insufficient error check in the ReiserFS file system (function r_stop) leading to a bad pointer dereference in the VFS (function deactivate_super) 71
4.9 Example of a false positive found in the VFS 72
4.10 Double error code in the ext3 file system, leading to 12 overwrite false positives . . 74

5.1 Example code fragment and corresponding reports 83

6.1 Subset of macros defining errors in Firefox 91
6.2 Two examples of macros that use log errors 91
6.3 An example of a potential security bug in Firefox due to a dropped error 95
6.4 An example of a dropped error in Firefox 96
6.5 An example in which developers document that errors can be dropped 97
6.6 Example of a dropped error when shutting down 97
6.7 An example of an error dropped during the release of resources 98
6.8 An example in which emitting an error warning is sufficient 99
6.9 An example of a false positive due to a variant of the double-error-code pattern . . . 101
6.10 A second example of a false positive due to the double-error-code pattern 102
6.11 An example of a false positive due to met preconditions 103
6.12 Basic error codes used in SQLite 105

Abstract

Incorrect error handling is a longstanding problem in many large software systems. Despite accounting for a significant portion of the code, error handling is one of the least understood, documented, and tested parts of a system. Ideally, some action should be taken when a run-time error occurs (e.g., error notification, attempted recovery, etc.). Incorrect error handling in system software is especially dangerous, as it can lead to serious problems such as system crashes, silent data loss, and corruption. Most system software today is written in C, which does not provide support for exception handling. Consequently the return-code idiom is commonly used in large C programs, including operating systems: run-time errors are represented as integer codes, and these error codes propagate through the program using conventional mechanisms such as variable assignments and function return values.

In this dissertation, I present my work on developing and applying static program analyses to find error-propagation bugs in system software that uses the return-code idiom. I give an overview of an interprocedural context- and flow-sensitive analysis that tracks the propagation of errors. This analysis is formalized using weighted pushdown systems. I describe how this analysis is used to find a variety of error-propagation bugs, such as dropped errors, misused error-valued pointers, and error-code mismatches between source code and error-reporting program documentation. I present results for numerous real-world, widely-used Linux file systems such as ext3 and ReiserFS, and Linux device drivers, where we have found hundreds of confirmed error-propagation bugs. Additionally, I show that the error-propagation bugs described in this dissertation also occur in widely-used applications such as the Mozilla Firefox web browser, which is written in C++.

Chapter 1

Introduction

Incorrect error handling is an important source of critical software bugs. Ideally, some action should be taken when a run-time error occurs (e.g., error notification, attempted recovery, etc.), but that is often overlooked. Despite accounting for a significant portion of the code in large software systems, error-handling code is in general the least understood, documented and tested part of a system. Exceptional conditions must be considered during all phases of development. As a result, error-handling code is scattered across different functions and files, making software more complex. Implementing correct error handling is particularly important for system software, since user applications rely on them.

C is still the preferred language for system programming. C does not have explicit exception-handling support. Consequently the return-code idiom is commonly used in large C programs, including operating systems. Run-time errors are represented as simple integer codes, where each integer value represents a different kind of error. These error codes propagate 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, this idiom is error-prone and effort-demanding. In this dissertation, we apply static program analysis to understand how error codes propagate through software that uses the return-code idiom, with a particular emphasis on system software.

The main component of our framework is an interprocedural, flow- and context-sensitive static analysis that tracks error codes as they propagate. We formulate and solve the error-propagation problem using weighted pushdown systems (WPDS). A WPDS is a dataflow engine for problems that can be encoded with suitable weight domains, computing the meet-over-all-paths solution. Solving the WPDS reveals the set of error codes each variable might contain at each program point. This information is used to find a variety of error-propagation bugs.

1.1 Why Error Handling?

Error handling accounts for a significant portion of the code in software systems. The simplest exception handling strategy represents up to 11% of a system [20]. Weimer and Necula [70] show how error-handling code represents between 1% and 5% in a suite of open-source Java programs ranging in size from 4,000 to 1,600,000 lines of code, however between 3% and 46% of the program text is transitively reachable from error-handling code. Cristian [13] reveals that more than 66% of software represents error handling. These numbers suggest that error handling is an important part of software systems. Unfortunately, exception handling is not a priority when developing software systems [7]. Error-handling code is in general the least understood, documented and tested part of a system [13].

It is difficult to write correct error-handling code. Exceptional conditions must be considered during all phases of software development [45], introducing interprocedural control flow that can be difficult to reason about [9, 48, 54]. As a result, error-handling code is usually scattered across different functions and files and tangled with the main system's functionality [2, 3, 45]. It is not surprising that error handling is error-prone and makes software more complex and less reliable. Error-handling code is in fact the buggiest part of a system [13]. Furthermore, many system failures and vulnerabilities are due to buggy error-handling code [1, 16, 66, 70], which is hard to test [7, 63] because it is difficult to generate tests that invoke error-handling mechanisms.

Poor support for error handling is reported as one of the major obstacles for large-scale and mission-critical systems [9]. Modern programming languages such as Java, C++ and C# provide

exception-handling mechanisms. Unfortunately, there is a lack of guidance in the literature on how to use exception handling effectively [21]. On the other hand, C does not have explicit exception-handling support, thus programmers have to emulate exceptions in a variety of ways [36]. The return-code idiom is among the most popular idioms used in large C programs, including operating systems. The use of idioms is significantly error-prone and effort-demanding. The development of robust software applications is a challenging task because programs must detect and recover from a variety of faults. Error handling is the key component of any reliable software system, thus it is not optional but necessary [7].

1.2 Why Systems Software?

Buggy error handling is a longstanding problem in many application domains, but is especially troubling when it affects systems software, in particular operating-system file-management code. File systems occupy a delicate middle layer in operating systems. They sit above generic block storage drivers, such as those that implement SCSI, IDE, or software RAID; or above network drivers in the case of network file systems. These lower layers ultimately interact with the physical world, and may produce both transient and persistent errors. Error-propagation bugs at the file-system layer can cause silent data corruption from which recovery is difficult or impossible. 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. The trustworthiness of the file system in handling errors is an upper bound on the trustworthiness of all storage-dependent user applications.

Error handling in file-system code cannot simply be fixed and forgotten. File-system implementations abound, with more constantly appearing. Linux alone includes dozens of different file systems. There is no reason to believe that file system designers are running out of ideas or that the technological changes that motivate new file system development are slowing down. Given the destructive potential of buggy file systems, it is not only critical to fix error-propagation bugs, but also to create tools that automate the process of finding them.

1.3 Linux Error Management

The majority of this dissertation focuses on Linux file systems, although we also find error-propagation bugs in other code bases (see Chapter 6). Our approach combines generic program analysis techniques with specializations for Linux coding idioms. Other operating systems share the same general style, although some details may differ. This section describes error management in Linux.

1.3.1 Integer Error Codes

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

Unfortunately, Linux (like many operating systems) is written in C, which offers no exception-handling mechanisms by which an error code could be raised or thrown. Errors must propagate through conventional mechanisms such as variable assignments and function return values. Most Linux run-time errors are represented as simple integer codes. Each integer value represents a different kind of error. Macros give these mnemonic names: EIO is defined as 5, ENOMEM is 12, and so on. Linux uses 34 basic named error macros, defined as the constants 1 through 34. Figure 1.1 shows their definitions.

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

```
#ifndef _ASM_GENERIC_ERRNO_BASE_H
#define _ASM_GENERIC_ERRNO_BASE_H

#define EPERM    1      /* Operation not permitted */
#define ENOENT   2      /* No such file or directory */
#define ESRCH    3      /* No such process */
#define EINTR    4      /* Interrupted system call */
#define EIO      5      /* I/O error */
#define ENXIO    6      /* No such device or address */
#define E2BIG    7      /* Argument list too long */
#define ENOEXEC  8      /* Exec format error */
#define EBADF    9      /* Bad file number */
#define ECHILD   10     /* No child processes */
#define EAGAIN   11     /* Try again */
#define ENOMEM   12     /* Out of memory */
#define EACCES   13     /* Permission denied */
#define EFAULT   14     /* Bad address */
#define ENOTBLK  15     /* Block device required */
#define EBUSY    16     /* Device or resource busy */
#define EEXIST   17     /* File exists */
#define EXDEV    18     /* Cross-device link */
#define ENODEV   19     /* No such device */
#define ENOTDIR  20     /* Not a directory */
#define EISDIR   21     /* Is a directory */
#define EINVAL   22     /* Invalid argument */
#define ENFILE   23     /* File table overflow */
#define EMFILE   24     /* Too many open files */
#define ENOTTY   25     /* Not a typewriter */
#define ETXTBSY  26     /* Text file busy */
#define EFBIG    27     /* File too large */
#define ENOSPC   28     /* No space left on device */
#define ESPIPE   29     /* Illegal seek */
#define EROFS    30     /* Read-only file system */
#define EMLINK   31     /* Too many links */
#define EPIPE    32     /* Broken pipe */
#define EDOM     33     /* Math argument out of domain of func */
#define ERANGE   34     /* Math result not representable */

#endif
```

Figure 1.1: Definition of basic error codes in the Linux kernel

```
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 unhandled error at this point
```

Figure 1.2: Typical error-checking code example

system offers little help determining which variables actually carry errors.

1.3.2 Consequences of Not Handling Errors

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

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

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

1.3.3 Handled vs. Unhandled Errors

Figure 1.2 shows a typical fragment of Linux kernel code. Many error-handling routines call printk, an error-logging function, with the error code being handled passed as an argument. Because this is an explicit action, it is reasonable to assume that the programmer is aware of the error and is handling it appropriately. Thus, if status contained an unhandled error on line 2, we

can assume that it contains a handled error after line 3. We consider such an action sufficient to determine that the error is being handled. We do not examine the error-handling code itself to make a judgement about its effectiveness.

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 on line 8, status cannot possibly contain an unhandled error.

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-handling code may require annotation. For example, we might require that programmers assign a special EHANDLED value to variables with handled errors, or pass such variables as arguments to a special handled function to mark them as handled. Requiring explicit programmer action to mark errors as handled would improve diagnosis by avoiding the silent propagation failures that presently occur.

1.4 Error-Propagation Bugs

Our goal is to use static program analysis to find how error codes propagate through large software systems and identify a variety of error-propagation bugs:

Dropped Errors. We identify error-code instances that vanish before proper handling is performed. We learn that unhandled errors are commonly lost when the variable holding the unhandled error value (a) is overwritten with a new value, (b) goes out of scope, or (c) is returned by a function but not saved by the caller. We find dropped errors in Linux file systems, the Mozilla Firefox web browser, and the database management system SQLite.

Errors Masquerading as Pointer Values. Linux error codes are often temporarily or permanently encoded into pointer values. Error-valued pointers are not valid memory addresses, and therefore require special care. Misuse of pointer variables that store error codes can lead

to system crashes, data corruption, or unexpected results. We use static program analysis to find three classes of error-valued pointer bugs in Linux file systems and drivers: (a) bad pointer dereferences, (b) bad pointer arithmetic, and (c) bad pointer overwrites.

Error-Code Mismatches Between Code and Documentation. Inaccurate documentation can mislead programmers and cause unexpected failures. We consider whether the manual pages that document Linux kernel system calls match the real code's error-reporting behavior. We use static program analysis to find the sets of error codes that file-related system calls return and compare these to Linux manual pages to find errors that are returned to user applications but not documented.

1.5 Contributions

The overall contribution of this dissertation is the design, development and application of static program analyses to make error handling in large systems more reliable by finding error-propagation bugs. Our analyses help developers understand how error codes propagate through software and find numerous error-propagation bugs that could lead to serious problems such as silent data corruption or data loss, from which recovery is difficult or even impossible. Specifically, the contributions of this dissertation are summarized as follows:

- We characterize the error-propagation dataflow problem and encode it using weighted pushdown systems (Chapter 2).
- We propose analysis optimizations that make the error- propagation analysis highly scalable, allowing the analysis of large real-world C and C++ programs (Chapter 2).
- We show how to extract detailed and useful diagnostic error reports from the raw analysis results (Chapter 3).
- We identify high-level error-handling patterns in Linux file systems and drivers (Chapters 2 and 3).

- We identify common scenarios in which unhandled errors are commonly lost. We find 312 confirmed dropped errors in five widely-used Linux file system implementations (Chapter 3).
- We characterize error transformation in the Linux kernel and show how these transformations can lead to bugs due to error codes masquerading as pointer values. We extend the error-propagation analysis to properly model the effects of error transformation. We find 56 error-valued pointer bugs in 52 different Linux file system implementations and 4 device drivers (Chapter 4).
- We use the error-propagation analysis to find the set of error codes that file-related system calls return and compare these against the Linux manual pages. We find over 1,700 undocumented error-code instances across 52 different Linux file system implementations (Chapter 5).
- We present two case studies that show how error-propagation bugs are also common in user applications, one of them written in C++ (Chapter 6).

1.6 Dissertation Structure

The rest of this dissertation is organized as follows. First, we formalize the error-propagation analysis as a weighted pushdown system in Chapter 2. The next three chapters describe analyses that use or extend the error-propagation framework to find a varierity of error-handling bugs. We discuss dropped errors in Chapter 3, error-valued pointer bugs in Chapter 4, and error-code mismatches between code and documentation in Chapter 5. Figure 1.3 shows these different components. We present two case studies involving user applications in Chapter 6. Related work is discussed in Chapter 7. Finally, we conclude in Chapter 8.

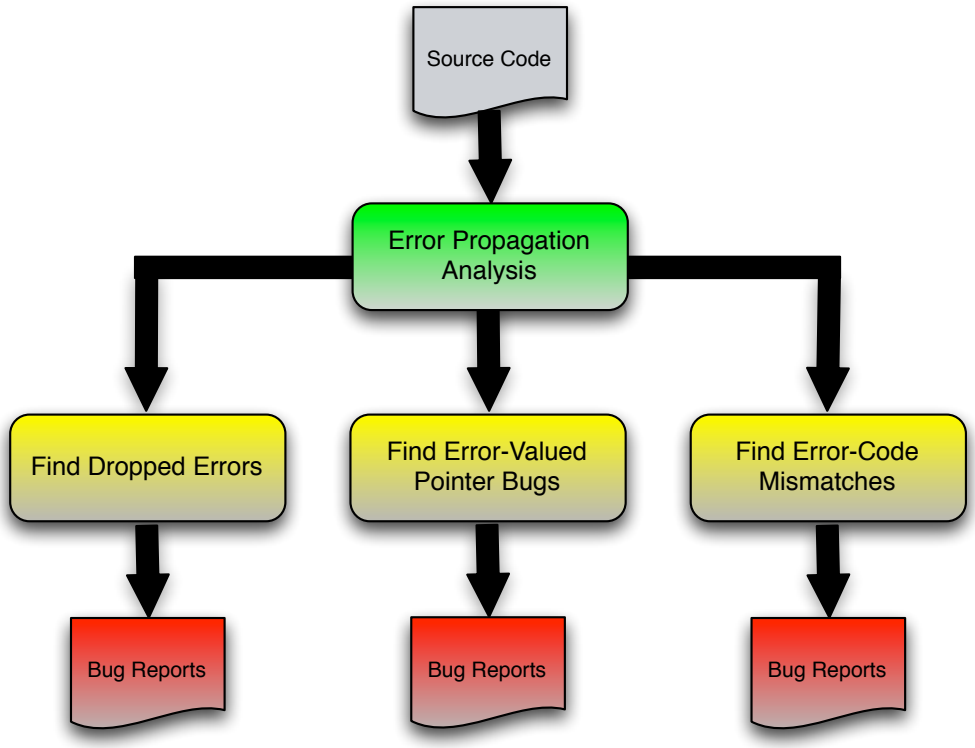

Figure 1.3: High-Level Framework Components

Chapter 2

Error-Propagation Analysis

The goal of the error-propagation analysis is to find the set of error codes that each variable may contain at each program point. The error-propagation analysis can be formulated as a forward dataflow problem. Error constants such as EIO generate unhandled error codes. Assignments propagate unhandled errors forward from one variable to another. Propagation ends when an error is overwritten, dropped, or handled by error-handling code.

This problem resembles copy constant propagation [69]. However, copy constant propagation finds *one* constant value that a variable *must* contain (if any), whereas we find the *set* of error code constants that a variable *may* contain. Copy constant propagation drives semantics-preserving optimization, and therefore under-approximates. We use the error-propagation analysis for bug reporting, and generally over-approximate so that no possible bug is overlooked.

The following sections describe weighted pushdown systems (WPDSs), and how we use WPDSs to encode the error-propagation problem. Note that the error-propagation problem can be described as a standard context-sensitive interprocedural analysis problem. We choose to cast the problem as a path problem over WPDSs because WPDSs (1) provide an algebraic formulation for handling local variables [38], and (2) support generating a witness trace as a proof of the result of solving the path problem [53]. We use witness tracing extensively to provide programmers with detailed diagnostic traces for each potential program bug (see Section 3.2).

2.1 Weighted Pushdown Systems

We use WPDSs [53] to formulate and solve the error-propagation dataflow problem. A WPDS is a pushdown system that associates a weight with each rule. Weights can serve as transfer functions that describe the effect of each statement on the program state. Such weights must be elements of a set that constitutes a bounded idempotent semiring. We now formally define WPDSs and related terms; Section 2.2 shows how WPDSs can be applied to solve the error propagation dataflow problem.

Definition 2.1. *A* **pushdown system** *is a triple* $\mathcal{P} = (P, \Gamma, \Delta)$ *where* P *and* Γ *are finite sets called the* **control locations** *and* **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* $\langle p, \gamma \rangle \hookrightarrow \langle p', w \rangle$, *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* $\langle p, \gamma w' \rangle \Rightarrow \langle p', w w' \rangle$ *for all* $w' \in \Gamma^*$.

As shown by Lal et al. [38] and Reps et al. [53], a pushdown system can be used to model the set of valid paths in an interprocedural control-flow graph (CFG).

Definition 2.2. *A* **bounded idempotent semiring** *is a quintuple* $(D, \oplus, \otimes, \bar{0}, \bar{1})$, *where* D *is a set,* $\bar{0}$ *and* $\bar{1}$ *are elements of* D, *and* $\oplus$ *(the combine operator) and* $\otimes$ *(the extend operator) are binary operators on* D *conforming to certain algebraic properties as given in Reps et al. [53].*

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

Definition 2.3. *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,* $\mathcal{S} = (D, \oplus, \otimes, \bar{0}, \bar{1})$ *is a bounded idempotent semiring, and* $f : \Delta \to D$ *is a function that assigns a value from* D *to each rule* $\mathcal{P}$.

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

Definition 2.4. *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 $c \in P \times \Gamma^*$:*

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

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

The meet over all paths value is the best possible solution to a static dataflow problem. Thus, a WPDS is a useful dataflow engine for problems that can be encoded with suitable weight domains. In Section 2.2 we show how the error propagation problem can be encoded as a weight domain.

In order to handle local variables properly, we use an extension to WPDSs proposed by Lal et al. [38]. This extension requires the definition of a *merge function*, which can be seen as a special case of the extend operator. This function is used when extending a weight w_1 at a call program point with a weight w_2 at the end of the corresponding callee. The resulting weight corresponds to the weight after the call. The difference between the merge function and a standard extend operation is that w_2 contains information about the callee's locals; this information is irrelevant to the caller. Thus, the merge function defines what information from w_2 to keep or discard before performing the extend.

Table 2.1: Encoding of control flow as PDS rules

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
$\langle p, f_{exit}\rangle \hookrightarrow \langle p, \epsilon\rangle$	Return from procedure exit f_{exit}

2.2 Creating the Weighted Pushdown System

Per definition 2.3, a WPDS consists of a pushdown system, a bounded idempotent semiring, and a mapping from pushdown system rules to associated weights. We now define these components for a WPDS that encodes the error-propagation dataflow problem.

2.2.1 Pushdown System

We model the control flow of the program with a pushdown system using the approach of Lal et al. [40]. Let P contain a single state $\{p\}$. Γ corresponds to program statements, and Δ corresponds to edges of the interprocedural CFG. Table 2.1 shows the PDS rule for each type of CFG edge.

2.2.2 Bounded Idempotent Semiring

Let $\mathcal{S} = (D, \oplus, \otimes, \bar{0}, \bar{1})$ be a bounded idempotent semiring per definition 2.2.

Set D

D is a set whose elements are drawn from $\mathcal{V} \cup \mathcal{C} \rightarrow 2^{\mathcal{V} \cup \mathcal{C}}$, where $\mathcal{V}$ is the set of program variables, and $\mathcal{C}$ is the set of constant values. Constants include error codes, the special value *OK* (used to represent all non-error values) and the special value *uninitialized* (used to represent uninitialized variables). Each element in D is called a *weight* and is a mapping from variables and constants to sets of variables and/or constants. The mapping for a variable $v \in \mathcal{V}$ gives the possible values of that variable following execution of a given program statement in terms of the values of constants and variables immediately before that statement. By design, all statement's weights always

map every constant $c \in \mathcal{C}$ to the set $\{c\}$. In other words, statements never change the values of constants. Figure 2.1 illustrates an example of a weight. Consider a program that has two variables x and y. For simplicity, we only include two error codes. OK represents all non-error values. The example shows how after the program statement $y = -EIO$, the error constant EIO flows into y, while variable x and all constants remain unchanged.

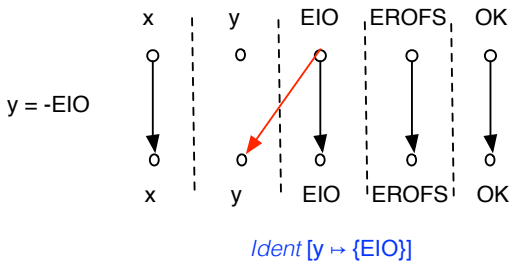

Figure 2.1: An example of a weight

Operators Combine and Extend

The combine operator ($\oplus$) is applied when conditional branches join. It summarizes the weights of a set of paths that merge. Combine is applied component-wise, where a component is a variable or constant. For all $w_1, w_2 \in D$ and $e \in \mathcal{V} \cup \mathcal{C}$:

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

In other words, combine is defined as the union of the sets of values a variable or constant is mapped to in each of the paths being merged. In the case of constants, the result is always the set containing itself. Figure 2.2 shows an example of applying the combine operator, where weights w_1 and w_2 are associated with the true and false branches of a conditional statement, respectively. The result of applying the combine operator is shown as $w_1 \oplus w_2$.

The extend operator ($\otimes$) calculates the weight of a path. It is also applied component-wise.

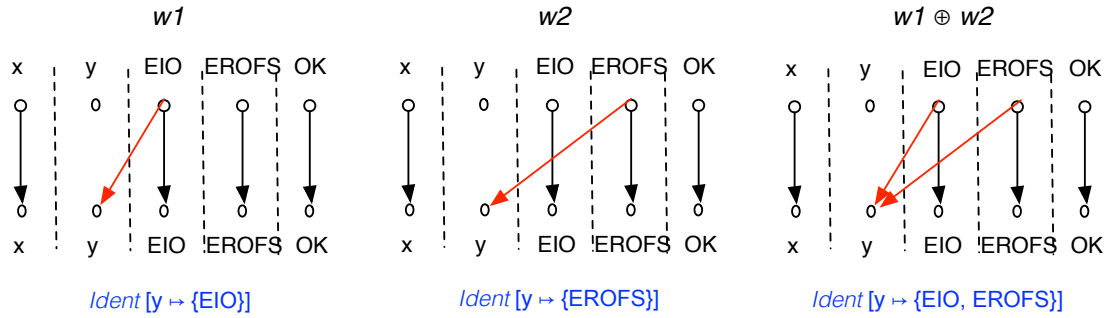

Figure 2.2: An example of applying the combine operator

For all $w_1, w_2 \in D$ and $e \in \mathcal{V} \cup \mathcal{C}$:

$$(w_1 \otimes w_2)(e) \equiv \bigcup_{e' \in w_2(e)} w_1(e')$$

The extend operator is essentially composition generalized to the power set of variables and constants rather than just single variables. Figure 2.3 shows an example of applying the extend operator.

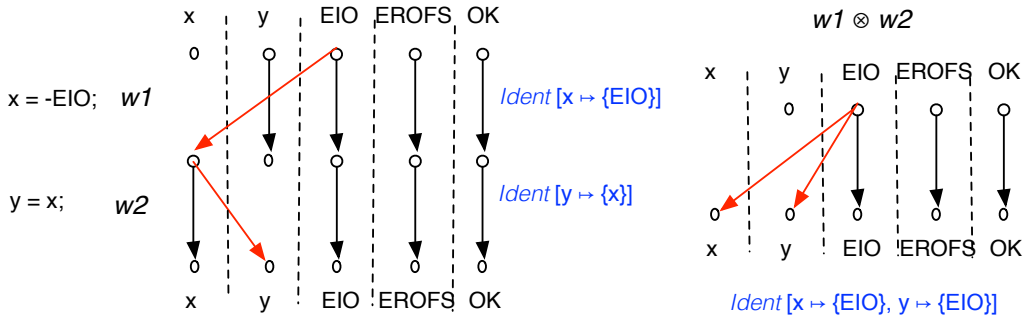

Figure 2.3: An example of applying the extend operator

Weights $\bar{0}$ and $\bar{1}$

The weights $\bar{0}$ and $\bar{1}$ are both elements of the set D. The annihilator weight $\bar{0}$ maps each variable and constant to the empty set and is the identity for the combine operator. The neutral weight

$\bar{1}$ maps each variable and constant to the set containing itself: a power-set generalization of the identity function. The weight $\bar{1}$ is the identity for the extend operator.

$$\bar{0} \equiv \{(e, \emptyset) \mid e \in \mathcal{V} \cup \mathcal{C}\} \qquad\qquad \bar{1} \equiv \{(e, \{e\}) \mid e \in \mathcal{V} \cup \mathcal{C}\}$$

Merge Function

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 $v \in \mathcal{V}$,

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

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

2.2.3 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 2.2.2. We describe transfer functions as being associated with specific program statements. The corresponding WPDS rule weight is associated with the edge from a statement to its unique successor. Conditionals have multiple outgoing edges and therefore require multiple transfer functions.

Assignments

The following paragraphs discuss the transfer functions for assignments without function calls on the right side (see Table 2.2 for a summary). We leave the discussion of assignments such as $v = f()$ for later in this section.

Table 2.2: Transfer functions for assignments

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

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. (Note that this is identical to $\bar{1}$ per Section 2.2.2.) The transfer function is $Ident[v \mapsto \{e\}]$. In other words, v must have the value of e after this assignment, while all other variables retain whatever values they had before the assignment.

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 $Donors \equiv \{e_1, e_2\} \cap \mathcal{V}$ as the set of variables on the right side of the assignment. Error codes are represented as integers but conceptually they are atomic values on which arithmetic operations are meaningless. Thus, if *op* is an arithmetic or bitwise operation, then we assume that the variables in *Donors* do not contain errors. Furthermore, the result of this operation must be a non-error value as well. Therefore, the transfer function for this assignment is $Ident[u \mapsto \{OK\}$ for all $u \in Donors \cup \{v\}]$. Note that we can easily choose to produce a warning instead.

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-error values. 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 (&, $*$), the transfer function maps v to $\{OK\}$ but leaves e unchanged.

Conditionals

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

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

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

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

For conditional statements that do not match any of the above patterns, we simply associate *Ident* with both true and false branches. An example of such a pattern is $if(v_1 < v_2)$, where $v_1, v_2 \in \mathcal{V}$.

Function calls

We adopt the convention used by Callahan [11], and later by Reps et al. [53], in which the CFG for each function has unique entry and exit nodes. The entry node is not the first statement in the

function, but rather appears just before the first statement. Likewise, we assume that function-terminating statements (e.g., **return** or last-block fall-through statements) have a synthetic per-function exit node as their unique successors. We use these dummy entry and exit nodes to manage data transfer between callers and callees.

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

Local variable initialization First consider a call to a void function that takes no parameters. Let $\mathcal{L}, \mathcal{G} \subseteq \mathcal{V}$, respectively, be the sets of all local and global variables. Recall that transfer functions are associated with CFG edges. For the edge from the callee's entry node to the first actual statement in the callee, we use the transfer function $\mathit{Ident}[v \mapsto \{\mathit{uninitialized}\}$ 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 exchange variable is introduced for each function parameter. Suppose function F has formal parameters $f_1, f_2, \ldots, f_n$. Let $F(a_1, a_2, \ldots a_n)$ be a function call to F with actual parameters $a_i \in \mathcal{V} \cup \mathcal{C}$. We introduce global exchange variables $F_1, F_2, \ldots, F_n$. The interprocedural call-to-enter edge is given the transfer function for a group of n simultaneous assignments $F_i = a_i$, exporting each actual argument into the corresponding global exchange variable. Rules for assignment transfer functions discussed earlier apply.

A similar process imports values from global exchange variables into callee formal parameters. For a callee F with formal parameters $f_1, f_2, \ldots, f_n$, the edge from the callee's entry node to the first actual statement in the callee is given the transfer function for a group of n simultaneous

assignments $f_i = F_i$, as though each formal argument were initialized with a value from the corresponding exchange variable. Other local variables are uninitialized as before.

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

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

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

Other interprocedural issues

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

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

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

Under these conditions, pointer parameters are equivalent to call-by-copy-return parameters. On the interprocedural call-to-enter edge, we copy pointed-to values from the caller to the callee, just as for simple integer parameters. On the interprocedural exit-to-return-site edge, we copy

callee values back into the caller. This extra copy-back on return is what distinguishes pointer arguments from non-pointer arguments, because it allows changes made by the callee to become visible to the caller.

Function pointers Most function pointers in Linux file systems are used in a fairly restricted manner. Global structures define handlers for generic operations (e.g., read, write, open, close), with one function pointer field per operation. Fields are populated statically or via assignments of the form "file_ops->write = ext3_file_write" where ext3_file_write identifies a function, not another function pointer. It is straightforward to identify the set of all possible implementations of a given operation. We then rewrite calls across such function pointers as **switch** statements that choose among possible implementations nondeterministically. This technique, previously employed by Gunawi et al. [26], accounts for approximately 80% of function pointer calls while avoiding the overhead and complexity of a general field-sensitive points-to analysis. The remaining 20% of calls are treated as *Ident*. Note that we analyze each file system individually; this perfectly disambiguates nearly all indirect calls in the code under study.

2.3 Additional Configurable Options

Although the overall framework described in this chapter is used throughout the dissertation, specific uses have slightly different needs. This section presents additional configurable options available depending on how the results will be used (to be discussed in subsequent chapters).

2.3.1 Copy Mode vs. Transfer Mode

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

The transfer functions described in Section 2.2.3 correspond to copy mode. As discussed earlier in this chapter, the transfer function in copy mode for a simple assignment of the form $v = e$, where $e \in \mathcal{V} \cup \mathcal{C}$, is $Ident[v \mapsto \{e\}]$. In other words, v must have the value of e after this assignment, while all other variables retain whatever values they had before the assignment. In transfer mode, let $Donor \equiv \{e\} \cap \mathcal{V} - \{v\}$ be the set containing the source variable (if any) of the assignment. Then the transfer function for a simple assignment in transfer mode is $Ident[v \mapsto \{e\}][s \mapsto \{OK\}$ for $s \in Donor]$ to transfer any errors from $Donor$ to v. In other words, after the assignment, (1) v must now have any 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.

2.3.2 Negative vs. Positive Error Codes

There are systems that define positive error codes instead of negative errors. An example is the XFS Linux file system: one of the largest and most complex Linux file systems. We 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 $Ident[v \mapsto \{OK\}]$.

2.3.3 Tentative vs. Non-Tentative Errors

A popular coding practice in the Linux kernel consists of storing potential error codes in variables before failure conditions actually hold. Error codes are used to initialize variables. These errors will be returned if a failure is eventually discovered. This phenomenon is usually contained within the function that generates the error code: error codes returned to callers generally represent real run-time errors. For certain kinds of error-propagation bugs (see Chapter 3), it is necessary to consider this practice in order to avoid a large number of false positives.

As discussed in Section 2.2.2, we classify integer constants into error constants and non-error constants. Among the error constants we further distinguish between *tentative* and *non-tentative* errors. Let *TentativeErrors* be the set of tentative error constants: integer values used to represent error codes. For each tentative error constant we define a corresponding non-tentative error constant. Let *NonTentativeErrors* be the set of all non-tentative error constants. We define $\mathcal{E} = TentativeErrors \cup NonTentativeErrors$ as the set of all error constants.

Let $E \equiv w(F_{ret}) \cap TentativeErrors$, where F_{ret} is the return exchange variable for function F and w is the weight at the end of function F. E represents the set of tentative error codes returned by function F. For all $e \in E$, we replace e with e', the corresponding non-tentative error. This transformation of returned errors from tentative to non-tentative models the coding practice described above. The analysis provides the option to make this transformation, if required.

2.3.4 Error-Handling Patterns

Error-handling patterns can be optionally provided to the analysis. If so, the analysis can distinguish between handled and unhandled errors. This distinction is crucial in certain cases. For example, dropping handled errors is harmless while dropping unhandled errors is not (see Chapter 3). On the other hand, there are other scenarios in which error handling is irrelevant (see Chapter 4 for some examples). In such cases, error-handling patterns are not required by

the analysis.

Currently, error-handling patterns are provided by developers, or found through manual inspection of the code under analysis. An example of an error-handling pattern found in the Linux kernel is described in Section 1.3.3. We consider any error values in a variable to have been handled when the variable is passed as an argument to the function printk. printk is a variadic function that logs errors. While logging alone does not correct any problems, it clearly expresses programmer awareness that a problem has occurred; presumably it is being handled as well. In general, after errors are handled, they no longer need to be tracked. The transfer function for such a call is $Ident[v \mapsto \{OK\}$ for v in the actual **int**-typed arguments to printk]. Error-handling patterns are particular to the application under analysis, however patterns across different applications often share common characteristics.

2.4 Solving the Dataflow Problem

We perform a poststar query [53] 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. [39] to read out weights from this automaton. This algorithm calculates, for each state in the automaton, the combine of all paths in the automaton from that state to the accepting state, if any. We can then retrieve the weight representing execution from the beginning of the program to any program point. We use this algorithm to find the set of values that each variable may contain at each program point.

2.5 Analysis Optimizations

Preliminary results showed that it took several hours to analyze real-world systems software components approaching 100,000 lines of code, while larger pieces of code remained completely out of reach. We optimize the analysis core in two ways. First, we reduce weight size by performing

a preliminary flow- and context-insensitive analysis. The analysis filters out *irrelevant program variables* that cannot possibly contain any error codes. Second, we reduce the number of weights by collapsing consecutive WPDS rules that share identity weight and identical source locations.

2.5.1 Reducing Weight Size

The number of variables determines the size of the weights, and large weights can significantly degrade performance. The error-propagation 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 smaller.

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 on the right-hand side of an assignment. 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 discussed in Section 2.2.3 (e.g., introducing exchange variables), this approach also handles other error-flow scenarios such as function parameters and return values.

For example, consider a program with five program variables a, b, c, x and y. If the pre-analysis finds that variables a, b and c cannot possibly contain any error codes during the execution of the program, then we can safely remove them from the weights. This is illustrated in Figure 2.4.

2.5.2 Reducing the Number of Weights

WPDS rules are used to model the control flow of the program and are associated with weights. The *Ident* 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.

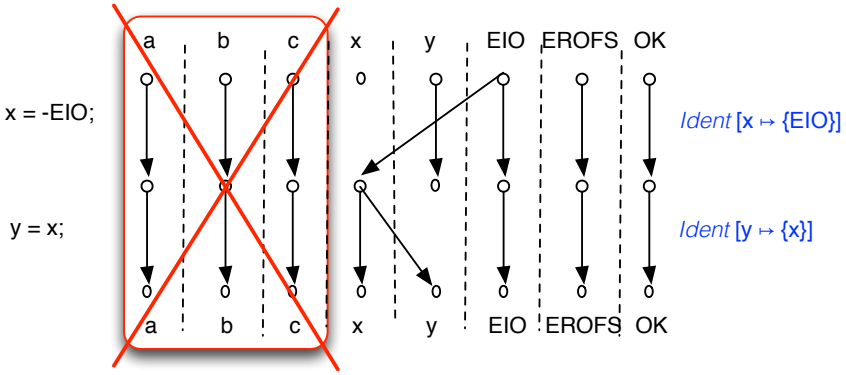

Figure 2.4: Removing irrelevant variables a, b and c from the weights

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 (same file name and line number). These are analyzed as distinct program points, thus increasing the number of weights to be created and calculated. We collapse consecutive rules that are associated with the *Ident* weight and share the same source information, thereby reducing the number of weights to be calculated.

Consider the example shown in Figure 2.5. Assume the three consecutive rules associated with the *Ident* weight share the same source information. We then collapse these into a single rule associated with the *Ident* weight without losing any information. This reduces the number of weights considerably.

2.5.3 Impact of Optimizations

Table 2.3 shows information related to the optimizations performed. For a sample of five widely-used Linux file systems, we find that about 96% of the program variables 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 collapsing 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

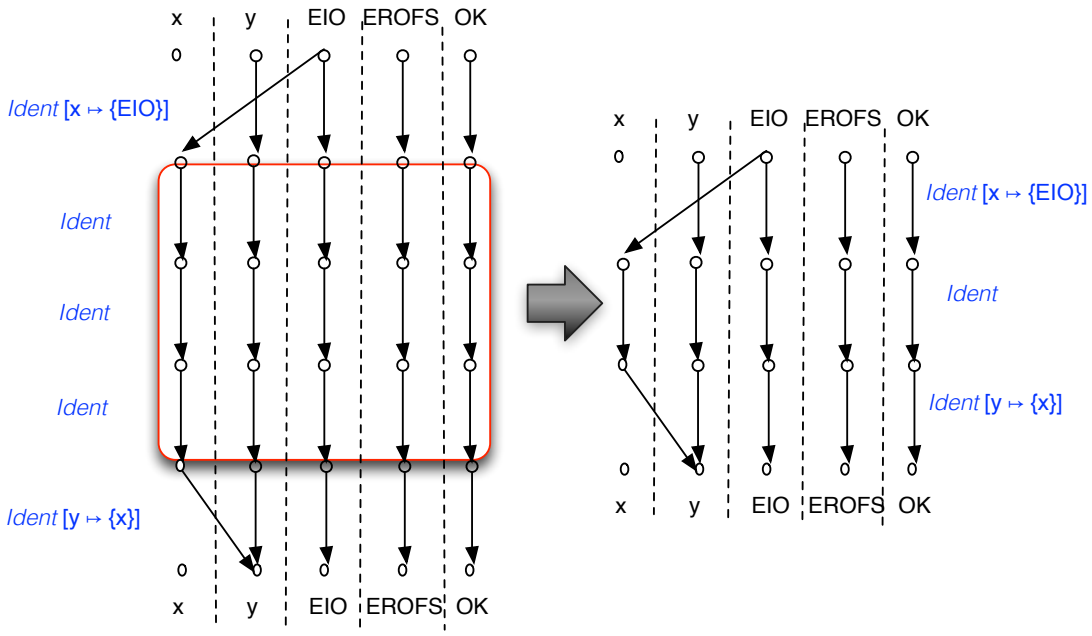

Figure 2.5: Collapsing rules

Table 2.3: Effectiveness of optimizations. KLOC gives the size of each file system in thousands of lines of code, including 59 KLOC of shared VFS code.

		Unoptimized → Optimized	
File System	KLOC	Number of Variables	Number of Rules
CIFS	90	37,504 → 1,972	117,300 → 89,131
ext3	82	38,094 → 2,119	131,274 → 91,418
IBM JFS	91	36,531 → 1,922	129,999 → 91,025
ReiserFS	86	42,249 → 1,892	143,827 → 101,958
XFS	159	55,430 → 1,076	175,683 → 137,074

139,616 to 102,121, which translates into fewer weights to calculate and consequently into a faster analysis. By performing these optimizations, the analysis runs 24 times faster on average (from hours to minutes) and requires 75% less memory with respect to the unoptimized version.

```
1  int getError() {
2    return −EIO;
3  }
4
5  int main() {
6    int status, result;
7
8    status = getError();
9    result = status;
10
11   return 0;
12 }
```

Figure 2.6: Sample program whose intermediate representation is shown in Figure 2.7

2.6 Framework Components

The error-propagation analysis requires the user to provide the source code to analyze, and the definition of error codes used by the application. Our front end produces an intermediate representation of the program, which describes the program control flow and encodes transfer functions (as defined in Section 2.2.3). The back end takes as input the intermediate representation, solves the dataflow problem, and presents the results.

The following section describes our intermediate representation. The next two sections give implementation details of our front and back ends.

2.6.1 Intermediate Representation

We extract a textual representation of the WPDS. This intermediate representation describes the program control flow and encodes the transfer functions of the different constructs used in the program under analysis.

The intermediate representation consists of a Prologue section and a sequence of Rules. The Prologue section includes the list of global and local variables in the program. Local variable names are prefixed with the function's name in which they are defined. Global variables include the exchange variables we introduce. For example, lines 2 to 12 in Figure 2.7 show the Prologue for the program from Figure 2.6. Local variables in main are prefixed with *main#*. Function

```
<WPDS>
        <Prologue>
                <Variables>
                        <Globals>
                                <var id='getError$return'/>
                        </Globals>
                        <Locals>
                                <var id='main#result'/>
                                <var id='main#status'/>
                        </Locals>
                </Variables>
        </Prologue>
        <Rule from='p' fromStack='getError.0' to='p' toStack1='getError.1'>
                <Weight basis='identityGlobals'>
                </Weight>
        </Rule>
        <Rule from='p' fromStack='getError.1' to='p'>
                <Weight basis='identity'>
                        <set to='getError$return' from='EIO' trusted='true'/>
                </Weight>
                <source line='2' file='example.c'/>
        </Rule>
        <Rule from='p' fromStack='main.0' to='p' toStack1='main.1'>
                <Weight basis='uninitialized'>
                </Weight>
        </Rule>
        <Rule from='p' fromStack='main.1' to='p' toStack1='getError.0' toStack2='main.2'>
                <Weight basis='identity'>
                </Weight>
                <source line='8' file='example.c'/>
        </Rule>
        <Rule from='p' fromStack='main.2' to='p' toStack1='main.3'>
                <Weight basis='identity'>
                        <set to='main#status' from='getError$return' trusted='false'/>
                </Weight>
                <source line='8' file='example.c'/>
        </Rule>
        <Rule from='p' fromStack='main.3' to='p' toStack1='main.4'>
                <Weight basis='identity'>
                        <set to='main#result' from='main#status' trusted='false'/>
                </Weight>
                <source line='9' file='example.c'/>
        </Rule>
        <Rule from='p' fromStack='main.4' to='p'>
                <Weight basis='identity'>
                </Weight>
                <source line='11' file='example.c'/>
        </Rule>
</WPDS>
```

Figure 2.7: The intermediate representation for the program shown in Figure 2.6

getError does not have any local variables. There are no global variables defined in this program, but there is an exchange global variable getError$return (we do not define a return exchange variable for main).

There are three types of rules: intraprocedural rules, push rules and pop rules. Intraprocedural rules model intraprocedural control flow. The attributes fromStack and toStack1 denote the from/to control locations, respectively. An example of an intraprocedural rule can be found on line 32. Push rules are used to model function calls. Push rules have an additional control location (attribute toStack2), which specifies the control location to return to after the call. An example of a push rule can be found on line 27. Pop rules model function return and only contain one attribute: fromStack. Examples of pop rules can be found in lines 17 and 44.

If source information is available, a Rule contains a Source tag with attributes line and file. Each Rule also has a Weight tag that describes the transfer function for the corresponding program statement. Weights have one of three values for the attribute basis: uninitialized, identityGlobals, or identity. uninitialized maps each global and local variable to the *uninitialized* value. This occurs only at the beginning of main (see line 24). identityGlobals maps each global variable to itself (no change) and each local variable to the *uninitialized* value (see line 14), which occurs at the beginning of each function except for main. Finally, identity maps all variables to themselves, and it is the basis value for the rest of the rules (e.g., line 39).

If required, a Weight has one or more set tags to describe transfer functions. For example, the set on line 40 describes the transfer function for the assignment on line 9 in Figure 2.6. These tags have the attribute trusted. Error overwrites (see Chapter 3) at assignments with the trusted attribute are not reported by the tool. Assignments from the original program are never trusted. On the other hand, assignments introduced by our analysis are usually trusted. For example, the assignment to the exchange variable on line 19 is trusted, while the assignment on line 9 is not.

2.6.2 Front End

The goal of the front end is to parse the source code and emit our intermediate representation of the program. In the process, the front end also applies source-to-source transformations on the source code under analysis. This includes redefining error-code macros as distinctive expressions to avoid mistaking regular constants for error codes. If a main function is not available, the front end finds the set of function entry points and creates a main function. We currently have two front ends:

1. CIL Front End. This implementation uses the CIL C front end [51]. CIL (C Intermediate Language) is a front end for the C programming Language. We use CIL version 1.3.6, and our implementation consists of 6,728 lines of OCaml code.

2. LLVM Front End. We use the LLVM Compiler Infraestructure [41]. The implementation consists of 16 LLVM passes written in 2,617 lines of C++ code. We compile the code under analysis to LLVM bitcode using Clang, which can produce LLVM bitcode for programs written in C and C++. We use LLVM and Clang version 3.0.

There is no difference between the intermediate representation produced by either front end. However, the LLVM front end (most recently implemented) allows us to obtain an intermediate representation for C++ programs.

2.6.3 Back End

The goal of the back end is to perform the dataflow analysis. We use the WALi WPDS library [37] revision 2822 to perform the interprocedural dataflow analysis on the WPDS produced by the front end. Within our WALi-based analysis code (4,744 lines of C++ code), we encode weights using *binary decision diagrams* (BDDs) [8] as implemented by the BuDDy BDD library [44] version 2.4. BDDs have been used before to encode weight domains [60]. The BDD representation allows highly-efficient implementation of key semiring operations, such as extend and combine.

We use Xerces version 3.1.1 to parse the XML intermediate representation. We write C++ code to query the WPDS and construct bug reports.

2.7 Summary

We have designed and implemented an interprocedural, flow- and context-sensitive static analysis for tracking the propagation of errors using WPDSs. The analysis finds the set of error codes that variables may contain at each program point. Our approach is based on a novel over-approximating counterpart to copy constant propagation analysis, with additional specializations for our specific problem domain. The 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 perform optimizations that allow the analysis of large real-world applications. The rest of this dissertation describes how we use the error-propagation analysis to find error-propagation bugs in widely-used software, including numerous Linux file systems and drivers.

Chapter 3

Dropped Errors in Linux File Systems

We refer to a dropped error as an error instance that vanishes before proper handling is performed. 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.

In this chapter, we give real-world examples of dropped errors (Section 3.1), show how we use the error-propagation analysis from Chapter 2 to find these kinds of bugs (Section 3.2), and present results for five widely-used Linux file systems (Section 3.4) along with performance numbers (Section 3.5).

3.1 Examples of Dropped Errors

This section presents real-world examples of dropped errors. Figure 3.1a illustrates an overwritten error in ext2. Function ext2_sync_inode, called on line 3, can return one of several errors including ENOSPC. The code inside the if statement on line 4 handles all errors but ENOSPC. Thus, if ENOSPC is returned then it is overwritten on line 8. This may lead to silent data loss.

Figure 3.1b depicts an out-of-scope error found in IBM JFS. txCommit, starting on line 1,

```
1 int ext2_xattr_set2(...) {
2   ...
3   error = ext2_sync_inode(...);
4   if (error && error != −ENOSPC) {
5     ...
6     goto cleanup;
7   }
8   error = 0; //overwriting error
9 }
```

(a) An overwritten error in ext2

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

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

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

(c) An unsaved error code in ext3

Figure 3.1: Three common scenarios in which unhandled errors are lost

commits any changes that its caller has made. This function returns EROFS if the file system is read-only. 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.

Figure 3.1c shows an unsaved error found in ext3. Function log_wait_commit returns EIO if a transaction commit has failed (lines 5–7). In a synchronous operation, this EIO error code is correctly propagated to the user application. In addition to synchronous foreground I/O operations, there are also background I/O operations that are flushed periodically to the disk. Since there is no way to communicate any related errors of background I/O operations to user applications, these errors are often dropped. One example is when a periodic timer launches a background checkpoint operation that will wrap all dirty buffers to a transaction, commit the transaction to the journal, and wait for it to finish. As shown on line 14, the I/O failure propagated by the log_wait_commit function is neglected by the ___process_buffer function, which itself is called during the background checkpoint. Hence, if there is a failure, data is silently lost.

3.2 Finding Dropped Errors

This section describes program transformations performed to simplify the problem of finding dropped errors. We present error-handling patterns we find in Linux file systems, and describe how we use the results obtained from applying the error-propagation analysis (see Chapter 2) to find dropped errors.

3.2.1 Program Transformations

We turn the three cases in which unhandled errors are commonly lost into a single case: overwritten errors. First, we turn unsaved errors into out-of-scope errors. For each function whose result

```
1 int bar(...) {
2   return −EIO;
3 }
4
5 int foo(...) {
6   ...
7   bar();
8   ...
9
10  return ...;
11 }
```

(a) Unsaved

```
1 int bar(...) {
2   return −EIO;
3 }
4
5 int foo(...) {
6   ...
7   int temp = bar();
8   ...
9
10  return ...;
11 }
```

(b) Unsaved ⇒ Out of scope

```
1 int bar(...) {
2   return −EIO;
3 }
4
5 int foo(...) {
6   ...
7   int temp = bar();
8   ...
9   temp = OK;
10  return ...;
11 }
```

(c) Out of scope ⇒ Overwritten

Figure 3.2: Unsaved ⇒ Out of scope ⇒ Overwrite

is not already being saved by the caller, we introduce a temporary local variable to hold that result. If any of these temporary variables contains unhandled errors, these are transformed into out-of-scope errors. For example, Figure 3.2a illustrates how an error is not saved on line 7. A temporary variable temp is used on line 7 (Figure 3.2b) to save the error, however variable temp goes out of scope on line 10. Thus, the unsaved error becomes an out-of-scope error.

Second, we turn out-of-scope errors into overwritten 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 unhandled error when the function ends, then the error is overwritten by the inserted assignment. A systematic naming convention for the newly-added temporary variables permits us to distinguish between unsaved and out-of-scope errors. This allows us to properly describe each bug.

The transformations discussed in this section reduce the problem of finding dropped errors to determining whether each assignment in the program may overwrite an unhandled error.

3.2.2 Error-Propagation Analysis

We run the error-propagation analysis from Chapter 2 to find the set of values each variable may contain at each program point. We use two analysis configurations. The first configuration operates in copy mode, while the second operates in transfer mode. In both, we enable error-

handling recognition, and distinguish between tentative and non-tentative errors. Section 3.4 describes how each configuration is more effective for certain kinds of bugs.

We enable error-handling recognition because we are particularly interested in finding the set of unhandled errors each variable may contain. In order to distinguish between handled and unhandled errors, we provide high-level error-handling patterns found in Linux file systems. For example, similar to our treatment of calls to function printk (discussed in Section 2.3.4), we also consider handled any errors passed as arguments to functions cERROR, cFYI, ea_bdebug, ext3_error, ext3_warning, ext4_warning, jfs_error, and reiserfs_warning. We provide additional patterns that describe scenarios in which errors are acknowledged. We refer to these as *safe patterns* since errors are likely to be handled, thus they can be safely dropped. Examples of these patterns can be found in Section 3.4.

We find that distinguishing between tentative and non-tentative errors (see Section 2.3.3) is necessary to avoid a significant number of false positives when finding dropped errors.

After running the analysis, at each assignment p we retrieve the associated weight w (representing execution from the beginning of the program to right before the assignment). 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. Note that w does not include the effect of assignment p itself. Rather, it reflects the state just before p. Then:

1. If $T \cap \mathit{NonTentativeErrors} = \emptyset$, then the assignment cannot overwrite any non-tentative error code and is not examined.

2. If $T \cap \mathit{NonTentativeErrors} = S = \{e\}$ for some single error code e, then the assignment can only overwrite an error code with the same error code and is not examined.[1]

3. Otherwise, it is possible that this assignment will overwrite an unchecked error code with a different code. Such an assignment is incorrect, and is presented to the programmer along with suitable diagnostic information.

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

Table 3.1: Average lengths of full paths and path slices

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

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

3.3 Describing Dropped Errors

WPDSs support witness tracing. As mentioned in definition 2.4, a witness set is a set of paths that justify the weight reported for a given configuration. This information lets us report not just the location of a bad assignment, but also detailed information about how that program point was reached in a way that exhibits the bug.

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

Let t be the currently-tracked target location. Each statement along the backward traversal

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

(a) Example code with overwritten error on line 19

```
Error codes: *EIO
```

(b) List of overwritten/dropped errors

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

(c) Complete diagnostic path trace

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

(d) Diagnostic path slice

Figure 3.3: Example code fragment and corresponding diagnostic output

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 unhandled error from x." If x is a return exchange variable, then we print an alternate message reflecting the fact that t receives an error code from a function call (e.g., see the message for line 14 in Figure 3.3a).

2. $t = e$ for some error constant $e \in \mathcal{E}$. We have reached the point of origin of the overwritten error. Our diagnostic trace is now complete for the bad assignment at p. We produce a final diagnostic message showing the source file name, line number, and the message "t receives error value e." If t is a return exchange variable, then we print an alternate message reflecting the fact that an error code is being returned from a function (e.g., see the message for line 7 in Figure 3.3a).

3. Anything else. We continue the backward path traversal, retaining t as the tracked target location. Additionally, we produce diagnostic output showing the source file name, line number, and the message "t has unhandled error."

If all diagnostic output mentioned above is presented to the programmer, then the result is a step-by-step trace of every program statement from the origin of an error value to its overwriting at p. If diagnostic output is omitted for case 3, then the trace shows only key events of interest, where the error value was passed from one variable to another. We term this a *path slice*, as it is analogous to a program slice that retains only the statements relevant to a particular operation. In practice, we find that the concise path slice provides a useful overview while the complete witness path trace helps to fill in details where gaps between relevant statements are large enough to make intervening control flow non-obvious. Table 3.1 shows that slicing significantly reduces path lengths. Across all five file systems and the shared virtual file system, slicing shrinks paths by an average ratio of 5.7 to 1.

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

Figure 3.3a shows an example code fragment that has an error-propagation bug in transfer mode. Figure 3.3b lists the error codes that may be overwritten/dropped at line 19. The error code to which the rest of the diagnostic information corresponds is marked with an asterisk. EIO is the only error code that may be overwritten in this example. Figure 3.3c shows the complete diagnostic path trace. Observe that this trace begins in function getError, which is called from load on line 14. Execution eventually traverses into nextId (line 3) while traveling from the error-code-generation point (line 7) to the overwriting assignment (line 19). Figure 3.3d shows the diagnostic path slice that includes only those lines directly relevant to the error. Here we see just four events of interest: the generation of an error code, which is returned by function getError on line 7; the transfer of that error to status on line 14; the transfer of that error code from status to result on line 16; and the assignment to result on line 19.

3.4 Experimental Evaluation

We present the results of our analysis on four local file systems (ReiserFS, IBM JFS, ext3 and ext4), one network file system (CIFS), and the virtual file system (VFS) in the Linux 2.6.27 kernel.

Our analysis reports 501 bugs in total, of which 312 are judged true bugs following manual inspection of the reports. IBM JFS and ReiserFS reports were inspected by the file systems' respective developers. CIFS and ext4 developers inspected a subset of their corresponding reports. A local domain expert assessed the rest, including the reports for ext3.

Developer response has been positive:

> I think this is an excellent way of detecting bugs that happen rarely enough that there are no good reproduction cases, but likely hit users on occasion and are otherwise impossible to diagnose. [14]

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

In the discussion that follows, we present results for each bug category. All results reported are for transfer mode unless explicitly stated otherwise. Table 3.2 summarizes our findings. We identify and describe safe patterns that we use to refine our tool. We also describe false positives in detail. Note that these are only "false" positives in that developers and our local expert judge that errors are safely overwritten, out of scope or unsaved. The fact that errors are overwritten, out of scope or unsaved is real, and in this sense the analysis is providing correct, precise information for the questions it was designed to answer.

3.4.1 Overwritten Errors

Developers and our local expert identify 25 overwritten true bugs out of 69 reports. We find that EIO and ENOMEM are the most commonly overwritten error codes. EIO signals I/O errors, including write failures that may lead to data loss. ENOMEM is used when there is insufficient memory. Figure 3.1a shows an overwritten error found in ext2.

Our tool recognizes four recurring patterns that represent safe overwrites. Figure 3.4a shows the most common recurring pattern found across all five file systems. Here, line 1 compares err with a specific error code. If they match, then line 3 clears err or assigns it a different error code. Overwriting one error code with another does not always represent a bug. For example, an error code generated in one layer of the operating system may need to be translated into a different code when passed to another layer. This clearly depends on the context and the error codes involved. In this case, we can see that the programmer acknowledges that err contains a

Table 3.2: Summary results for the six case studies. Bug reports are broken down into overwritten, out-of-scope and unsaved. Each category is further divided into true bugs (TB) and false positives (FP). The first column under FPs corresponds to "removable" FPs (FPs that can be removed if our tool recognizes unsafe patterns). The second column corresponds to "unavoidable" FPs (FPs that cannot be automatically removed because significant human intervention is required). The last column (T) gives the total number of bug reports per bug category. Results for unsaved errors were produced in copy mode.

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

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

(a) Specific error code

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

(b) Special function

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

(c) Replacement

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

12    goto retry;
```

(d) Retries

Figure 3.4: Some recurring safe patterns recognized by the analysis

specific error code before performing the assignment. We choose to trust the programmer in this particular scenario, thus we assume that overwriting the error code contained in err is safe.

Figure 3.4b shows the second common pattern, found in both ReiserFS and ext3. In this case the programmer acknowledges that something might be wrong by calling a function such as reiserfs_warning in the case of ReiserFS. The call is usually followed by an assignment that may overwrite an error code. We choose to allow overwrites that occur immediately after such calls.

The third pattern, shown in Figure 3.4d, appears in both ext3 and ext4. Here variable err may receive an error code from a function call (the function could return different error codes) on line 9. Our tool initially reported an overwrite at that line in the case of a retry. We observe that the **goto** statement on line 12 is always located inside an **if** statement. In addition, the variable being overwritten always appears in the condition (line 11), making it possible to identify the variable that needs to be cleared before retrying.

The last pattern is shown in Figure 3.4c. Both variables retval and err might contain error codes at line 1. Thus a potential overwrite would be reported on line 2 when the error stored in err replaces that in retval. In this case, we can see that the programmer acknowledges that those variables might contain error codes before performing the assignment: the assignment occurs

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

(a) Replacement

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

(b) Precedence/scope

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

(c) Precedence/overwrite

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

(d) Redundancy

Figure 3.5: Some recurring unsafe patterns

only if both variables are nonzero. We trust the programmer in this particular scenario and assume that overwriting the error code contained in retval is safe.

Most false positives arise from overwriting one error code with another error code without clear knowledge that an error may be overwritten. Unfortunately, there is no formal error hierarchy, which prevents us from automatically differentiating between correct and incorrect overwrites. We identify two unsafe patterns in which error codes are commonly overwritten. We find that 27 out of 44 false positives obey the pattern shown in Figure 3.5a. In this case, only the variable err is acknowledged to be nonzero on line 1. We do not consider the overwrite on line 2 to be safe because it is not clear that the developer is aware of the overwrite. Our tool reports the potential bug and developers must determine its validity. The second unsafe pattern is shown in Figure 3.5c. If both ret and ret2 contain error codes at line 4, then ret2 is overwritten

in line 7. In this case, ret has precedence over ret2. We find that 9 out of the remaining 17 false positives fall into this category. We can recognize these patterns and mark these reports in the future. This would allow developers to prioritize the reports or skip certain categories altogether if considered safe. We call these false positives "removable." The 8 remaining false positives required significant human intervention to determine their safeness; we call these "unavoidable."

3.4.2 Out-of-Scope Errors

Out-of-scope errors are the least common. A total of 66 bug reports concern out-of-scope errors. Among these, 18 true bugs are identified. Figure 3.1b shows an out-of-scope error found in IBM JFS. Most of these bugs relate to ignoring I/O errors. We identify four recurring safe patterns for out-of-scope errors, of which three are variants of those discussed in Section 3.4.1.

The first pattern appears in CIFS, ext4 and IBM JFS. This pattern is similar to that shown in Figure 3.4a, however if the variable holds a specific error code, then zero or a different error code is returned on line 3, i.e., there is a return statement instead of an assignment. We also trust the programmer in this case and err is not reported to go out of scope at this line.

The second pattern, shown in Figure 3.4d, appears in ext3 and ext4. Without recognizing this pattern, our tool would report that err is out of scope on line 6. This is not the case when err is cleared before retrying.

The third pattern has already been shown in Figure 3.4b, however there is a subtle difference. In this case, reiserfs_warning takes the variable that is about to go out of scope as a parameter. As a general approach for this pattern, we clear any variable that is passed as a parameter to reiserfs_warning and similar functions.

The fourth pattern concerns *error transformation*: changes in how errors are represented as they cross software layers. Integer error codes may pass through structure fields, be cast into other types, be transformed into null pointers, and so on. Our analysis does not track errors across all of these representations. As a result, error codes are not propagated when transformed, yielding out-of-scope false positives. We also find that transformation from integers to pointers

predominates [2]. This transformation uses the ERR_PTR macro, which takes the error to be transformed as parameter. As in the case of functions such as reiserfs_warning, we clear any variable that is passed as a parameter to ERR_PTR.

Ad hoc error precedence is the main source of false positives. Figure 3.5b presents one example. Both ret and err may be assigned error codes on lines 6 and 7, respectively. Variable ret is propagated regardless the contents of err, unless it does not contain an error code, i.e., ret has precedence over err. Our tool produces an out-of-scope report for err on line 12. This could be a bug or not depending on the context. We find that 41 out of 48 false positives exhibit this pattern. We can recognize this pattern to provide more information in the future. As for overwrites, the "false" positives here are not indications of analysis imprecision, but rather are based on a human expert's judgment that some errors can fall out of scope safely.

3.4.3 Unsaved Errors

Unsaved errors predominate in all five file systems. Developers and our local expert identify 269 true bugs among 366 unsaved bug reports in copy mode. Transfer mode produces 48% fewer false positives but misses 33% of the true bugs found in copy mode. The most common unsaved error code is EIO, followed by ENOSPC and ENOMEM. Figure 3.1c shows an unsaved error found in ext3.

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

The main source of false positives concerns *error paths*: paths along which an error is already being returned, so other errors may be safely ignored. The second most common source of false positives is due to the fact that there is another way to detect the problem, which we term

[2] We address this kind of error transformation in Chapter 4.

Table 3.3: Analysis performance. KLOC gives the size of each file system in thousands of lines of code, including 60 KLOC of shared VFS code. We provide running times for extracting the WPDS textual representation of the program, solving the poststar query, and finding bugs.

		Analysis Time (min:sec)				
File System	KLOC	WPDS	Poststar Query	Finding Bugs	Total	Memory (MB)
CIFS	91	1:09	0:24	0:04	1:31	271
ext3	83	1:15	0:24	0:08	1:47	273
ext4	97	1:33	0:31	0:12	2:16	358
IBM JFS	93	1:14	0:24	0:05	1:43	304
ReiserFS	88	1:15	0:28	0:06	1:49	320

redundant error reporting. Figure 3.5d shows an example from ReiserFS. The sync_dirty_buffer call on line 4 may return an error code, but checking its parameter tbh on line 7 is sufficient in this case. However, it is still possible for a more specific error to be dropped leading to loss of information about what exactly went wrong. A few false positives arise when callers know that the callee returns an error code only if certain preconditions are not met. Callers that have already established those preconditions know that success is assured and therefore ignore the return value.

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

3.5 Performance

Our experiments use a dual 3.2 GHz Intel processor workstation with 3 GB RAM. Table 3.3 shows code sizes, the time required to analyze each file system, and memory usage. We divide the analysis into three phases: (1) extracting a textual WPDS representation of the kernel code, (2) solving the poststar query, and (3) finding bugs and traversing the witness information to produce diagnostic output.

The analysis is quite cheap. Extracting the textual WPDS representation is the most expensive

phase, nevertheless the overall analysis running time ranges from only 1 minute 31 seconds for CIFS to 2 minutes 16 seconds for ext4, while using 305 MB of memory on average. The analysis is clearly suitable for regular use by kernel developers using widely-available hardware.

3.6 Other File Systems

We have performed the analysis on 43 other Linux file systems. Together, these account for 250 thousand additional lines of kernel code (KLOC). However we have not manually inspected the results. We have also analyzed different Linux versions, and find that file-system code evolves significantly in each release. This demonstrates that fixing this class of bugs is not a one-time operation. Rather, kernel developers need robust tools to ensure that existing error-propagation bugs are fixed, and also that new bugs are not introduced as implementations change over time.

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

> We've found one legitimate problem. ... We call a non-void function (that can return some critical error codes) and don't assign the return value, dropping some nice things such as EASSERT, EABOUND, and EEBADHDR on the ground. We would have expected the compiler or [another code-checking tool] to catch that, actually... We're going to rerun on a big update to the code, soon. [25]

3.7 Summary

In this chapter, we characterize and present real-world examples of dropped errors in Linux file systems. We show how we use the error-propagation analysis from Chapter 2 to find unsaved, out-of-scope and overwritten unhandled errors. We describe how to construct useful and detailed diagnostic information using WPDS witness information. Finally, we present results for five

widely-used Linux file systems: CIFS, ext3, ext4, IBM JFS, and ReiserFS, where we find 312 nontrivial bugs. False positives arise, but many of these can be ascribed to a small number of recurring unsafe patterns that should also be amenable to automated analysis; we identify several such patterns in our detailed case studies. We also find that the same patterns repeat among different file system implementations.

Chapter 4

Errors Masquerading as Pointers in Linux

Error codes are often temporarily or permanently encoded into pointer values as they propagate. Error-valued pointers are not valid memory addresses, and therefore require special care by programmers. Improper use of pointer values in systems code can have serious consequences such as system crashes, data corruption, and unexpected results. We identify three classes of bugs relating to error-valued pointers: (1) bad pointer dereferences, (2) bad pointer arithmetic, and (3) bad pointer overwrites.

In this chapter, we further describe the transformation from integer error codes to error-valued pointers (Section 4.1), characterize the three kinds of error-valued pointer bugs (Section 4.2), describe how to extend the error-propagation analysis from Chapter 2 to track errors in pointer variables (Section 4.3), and how to use the results to find error-valued pointer bugs (Section 4.4). Finally, we present experimental results for 52 different Linux file systems, and 4 Linux drivers (Section 4.5).

4.1 Error Transformation in the Linux Kernel

Error transformation refers to changes in error representation as errors propagate across software layers. Integer error codes may be cast into other types. In particular, integer error codes are often cast to pointer values. To be clear, these are not pointers that refer to the locations of error codes. Rather, the numeric value of the pointer itself is actually a small integer error code rather than a proper memory address. As offensive as this may seem from a type-system perspective, it is nevertheless a well-accepted practice found throughout the Linux kernel. 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.

Figure 4.1a shows an example of integer-to-pointer error transformation. Function open_xa_dir returns a pointer value. Variable xaroot may receive an error-valued pointer from a function call on line 4. Function IS_ERR on line 6 tests the return value. If it is an error, the error-valued pointer is returned. Additionally, function ERR_PTR is called on lines 16 and 22 to transform integer error codes into pointers.

Figure 4.1b illustrates the opposite transformation, from pointer to integer. Function reiserfs_listxattr returns an integer value. An error constant is returned on line 6. Also, variable dir may receive an error-valued pointer from a call to function open_xa_dir (shown in Figure 4.1a). If it is an error, then function PTR_ERR transforms the error from a pointer to an integer on line 13.

The preceding examples, though simplified, already illustrate how tricky it can be to follow error flows manually. Errors propagate through long call chains, transforming several times before being handled. This makes error tracking quite challenging in large systems such as the Linux kernel. Thus, supporting error transformation is crucial to building a more complete understanding of error propagation and how the system recovers from run-time errors.

```
1  struct dentry *open_xa_dir(...) {
2   struct dentry *xaroot;
3   ...
4   xaroot = ...;
5
6   if (IS_ERR(xaroot))
7    return xaroot;
8   ...
9   int err;
10
11  if (...) {
12   err = ...;
13
14   if (err) {
15    ...
16    return ERR_PTR(err);
17   }
18  }
19
20  if (...) {
21   ...
22   return ERR_PTR(-ENODATA);
23  }
24
25  ...
26  return ...;
27 }
```

(a) **int**-to-pointer

```
1  int reiserfs_listxattr(...) {
2   struct dentry *dir;
3   int err = 0;
4
5   if (...)
6    return -EINVAL;
7
8   ...
9   dir = open_xa_dir(...);
10  ...
11
12  if (IS_ERR(dir)) {
13   err = PTR_ERR(dir);
14
15   if (err == -ENODATA) {
16    ...
17    err = ...;
18   }
19
20   goto out;
21  }
22  ...
23
24  out:
25  ...
26  return err;
27 }
```

(b) pointer-to-**int**

Figure 4.1: Examples of error transformation in ReiserFS

4.2 Error-Valued Pointer Bugs

We concentrate on finding bugs due to the improper use of error-holding pointers. The following subsections present three kinds of pointer-related bugs: bad pointer dereferences, bad pointer arithmetic, and bad overwrites.

4.2.1 Bad Pointer Dereferences

A bad pointer dereference occurs when a possibly–error-valued pointer is dereferenced, since an error value is not a valid memory address. Figure 4.2 shows an example. Function fill_super in

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

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 VFS is invoked with variable root as parameter. This function dereferences the potential error-valued pointer parameter inode on line 21.

Our goal is to find the program locations at which these bad pointer dereferences may occur. We identify the program points at which pointer variables are dereferenced, i.e., program points where the indirection ($*$) or arrow ($->$) operators are applied. Let us assume for now that we are able to retrieve the set of values each pointer variable may contain at any location l in the program. Thus, at each dereference of variable v, we retrieve the associated set of values $\mathcal{N}_l$, which corresponds to the set of values v may contain right before the dereference at l. Let $\mathcal{E}$

be the finite set of all error constants. Let OK be a single value not in $\mathcal{E}$ that represents all non-error values. Let $\mathcal{C} = OK \cup \mathcal{E}$ be the set of all values. Then $\mathcal{N}_l \subseteq \mathcal{C}$, and the set of error codes that variable v contains before the dereference is given by $\mathcal{N}_l \cap \mathcal{E}$. If $\mathcal{N}_l \cap \mathcal{E} \neq \emptyset$, then we report the bad pointer dereference.

4.2.2 Bad Pointer Arithmetic

Although error codes are stored in integer and pointer variables, these codes are conceptually atomic symbols, not numbers. Error-valued pointers should never be used to perform pointer arithmetic. For example, incrementing or decrementing a pointer variable that holds an error code will not result in a valid memory address. Similarly, subtracting two pointer variables that may contain error values will not yield the number of elements between both pointers as it would with valid addresses. Figure 4.3 shows an example of bad pointer arithmetic found in the mm. Callers of function kfree (line 3) may pass in a pointer variable that contains the error code ENOMEM, now in variable x. The variable is further passed to function virt_to_head_page when it is invoked on line 6. Finally, this function uses x to perform some pointer arithmetic on line 11, without first checking for any errors.

We aim to identify the program points at which such bad pointer arithmetic occurs. We find the program locations at which pointer arithmetic operators addition ($+$), subtraction ($-$), increment ($++$), or decrement ($--$) are used. For each variable operand v in a given pointer arithmetic operation at program location l, we retrieve the set of values $\mathcal{N}_l$ that v may contain right before the operation. We report a problem if $\mathcal{N}_l \cap \mathcal{E} \neq \emptyset$ for any operand v.

4.2.3 Bad Overwrites

Bad overwrites occur when error values are overwritten before they have been properly acknowledged by recovery/reporting code. Our goal is to find bad overwrites of error-valued pointers or error values stored in pointed-to variables. The latter can occur either when the variable is assigned through a pointer dereference or when the pointer variable is assigned a different value,

```
1  #define virt_to_page(addr) (mem_map + (((unsigned long)(addr)−PAGE_OFFSET) >> ...))
2
3  void kfree(const void *x) {  // may be passed an error
4    struct page *page;
5    ...
6    page = virt_to_head_page(x);  // passing error
7    ... // use page
8  }
9
10 struct page *virt_to_head_page(const void *x) {
11   struct page *page = virt_to_page(x);  // macro from line 1
12   return compound_head(page);
13 }
```

Figure 4.3: Bad pointer arithmetic found in the mm

which may or may not be a valid address value.

In general, bad overwrites are more challenging to identify than bad pointer dereferences and bad pointer arithmetic. Most error-valued overwrites are safe or harmless, whereas (for example) error-valued pointer dereferences always represent a serious problem. Also, the consequences of a bad overwrite may not be noticed immediately: the system may appear to continue running normally.

We do not attempt to identify or validate recovery code. Rather, we simply look for indications that the programmer is at least checking for the possibility of an error. If the check is clearly present, then presumably error handling or recovery follows. As mentioned earlier, an error code may be safely overwritten after the error code has been handled or checked. Figure 4.4 shows examples in which it is safe to overwrite error codes that have been checked. In Figure 4.4a, err may receive one of several error codes on line 4. If this variable contains an error code on line 6, then we continue to the next iteration of the loop, where the error code is overwritten the next time line 4 is run. Overwriting an error code with the exact same error code is considered to be harmless, but the problem here is that different error codes might be returned by successive calls to function get_error. A similar pattern is illustrated in Figure 4.4b.

In order to find bad overwrites, we identify the program points at which assignments are made to potentially–error-carrying storage locations. At each such assignment to pointer variable

```
1  int *err;
2  ...
3  while(...) {
4    err = get_error();
5
6    if (IS_ERR(err)) {
7      continue;
8    }
9  ...
10 }
```

(a) Loop

```
1  int *err;
2  ...
3
4  retry:
5    ...
6    err = get_error();
7
8    if (err == ERR_PTR(-EIO))
9      goto retry;
10 ...
```

(b) Goto

Figure 4.4: Two examples of safe error-overwrite patterns

v at location l, we retrieve the set of values $\mathcal{N}_l$ that variable v may contain. If $\mathcal{N}_l \cap \mathcal{E} \neq \emptyset$, then we report the bad overwrite. A generalization of this strategy also allows us to check indirect assignments across pointers, as in "$*v = \ldots$"; we give further details on this extension in Section 4.3.1.

4.3 Error Propagation and Transformation

We require error-propagation information to find the bugs described in Section 4.2. The error-propagation analysis described in Chapter 2 tracks how integer error codes propagate. However, the analysis does not support error transformation, which is necessary to find bad pointer dereferences, bad pointer arithmetic, and bad assignments to pointer variables. For example, it assumes that error propagation ends if the error is transformed into a pointer. In Figure 4.1a, even though an error may be assigned on line 4, the analysis does not actually track error flow into variable xaroot because it is a pointer variable. Similarly, no pointer error value is recognized as being returned at lines 16 and 22 because the analysis always clears the actual argument to any calls to function IS_ERR. Thus, no pointer error value is identified as returned by function open_xa_dir on line 9 in Figure 4.1b.

We extend the error-propagation framework to support error transformation. The following subsections describe new definitions for some WPDS components. In particular, we modify one of

the elements of the bounded idempotent semiring, and replace the original transfer functions with a new suite of functions that take into consideration pointer variables and error transformation.

4.3.1 Bounded Idempotent Semiring

Our definition of the bounded idempotent semiring $\mathcal{S} = (D, \oplus, \otimes, \bar{0}, \bar{1})$ in Section 2.2.2 remains unchanged, except for the set D. In Section 2.2.2 we defined D as a set whose elements are drawn from $\mathcal{V} \cup \mathcal{C} \rightarrow 2^{\mathcal{V} \cup \mathcal{C}}$, where $\mathcal{V}$ is the set of all program variables, and $\mathcal{C}$ the set of constants (error codes, *OK* and the *uninitialized* value). The error-propagation analysis described in Chapter 2 does not track errors stored in pointer variables. Even in the special case of integer pointer parameters, error codes are stored in the integer variable pointed to, not in the pointer variable itself. Now we need to track errors stored in pointer variables. This uncovers a new requirement: distinguishing between an error code stored in a pointer variable v and an error stored in $*v$. We introduce a *dereference variable* $*v$ for each pointer variable v. This allows us to distinguish and track error codes stored in either "level."

We replace dereference expressions with the corresponding dereference variables before performing the error-propagation analysis. Thus, the set $\mathcal{V}$ is now redefined as the set of all program variables and dereference variables. Even though the number of variables can increase considerably due to dereference variables, this does not represent a problem in practice since we only keep those variables that are truly relevant to our analysis (see analysis optimizations in Section 2.5).

4.3.2 Transfer Functions

Transfer functions define the new state of the program as a function of the old state. As discussed in Section 2.2.1, PDS rules correspond to edges in the interprocedural CFG. Each PDS rule is associated with a *weight* or *transfer function*. Although here we describe weights as being associated with specific program statements, they are in fact associated with the edges from a statement to its successors.

Table 4.1: Transfer functions for assignments in copy mode

Pattern	Where	Transfer Function
$v = e$	$e \in \mathcal{V} \cup \mathcal{C}$ and v is of type int	$Ident[v \mapsto \{e\}]$
$v = e$	$e \in \mathcal{V} \cup \mathcal{C}$ and v is of pointer type but e is not	$Ident[v \mapsto \{e\}][*v \mapsto \{OK\}]$
$*v = e$	$*v \in \mathcal{V}$ and $e \in \mathcal{V} \cup \mathcal{C}$	$Ident[v \mapsto \{OK\}][*v \mapsto \{e\}]$
$v_1 = v_2$	$v_1, v_2 \in \mathcal{V}$ and v_1 and v_2 are of pointer type	$Ident[v_1 \mapsto \{v_2\}][*v_1 \mapsto \{*v_2\}]$
$v_1 = \&v_2$	$v_1, v_2 \in \mathcal{V}$ and v_1 is of pointer type	$Ident[v_1 \mapsto \{OK\}][*v_1 \mapsto \{v_2\}]$
$v = e_1 \ \ op \ \ e_2$	$e_1, e_2 \in \mathcal{V} \cup \mathcal{C}$ and op is a binary arithmetic, bitwise or logical operator	$Ident[v \mapsto \{OK\}]$
$v = op \ \ e$	$e \in \mathcal{V} \cup \mathcal{C}$ and op is a unary arithmetic, bitwise, or logical operator	$Ident[v \mapsto \{OK\}]$

The transfer functions discussed in this section correspond to copy mode (see Section 2.3.1). All transfer functions share one key assumption: that pointer variables have no aliases inside a function. This makes our approach to pointers both unsound and incomplete, however it is simple and gives good results in practice.

Assignments

Table 4.1 shows the transfer functions for assignments. For the purpose of this discussion, we classify these into three groups. First consider assignments of the form $v = e$, where $e \in \mathcal{V} \cup \mathcal{C}$ and v is of type **int**. Let *Ident* be the function that maps each variable and constant to the set containing itself, which is identical to $\bar{1}$. The transfer function for such an assignment is $Ident[v \mapsto \{e\}]$. In other words, v must have the value of e after this assignment, while all other variables retain whatever values they contained before the assignment, including e.

Next consider assignments that involve pointer or dereference variables. In either case, we need to update mappings at two levels. For example, for assignments of the form $*v = e$, where $*v$ is the dereference variable corresponding to pointer variable v and $e \in \mathcal{V} \cup \mathcal{C}$, the transfer function is $Ident[v \mapsto \{OK\}][*v \mapsto \{e\}]$. We map the dereference variable to any values e may contain. At the same time, we assume that the corresponding pointer variable contains a valid address, i.e., v is mapped to the OK value. The opposite occurs with assignments of the form $v = e$, where v is of some pointer type and $e \in \mathcal{V} \cup \mathcal{C}$ and not a pointer variable. In this case, variable v is mapped to whatever values e may contain, which must be non-address values. We assume that the corresponding dereference variable $*v$ does not contain an error since v does not hold a valid address. Transfer functions for pointer-related assignments of the form $v_1 = v_2$ and $v_1 = \&v_2$ can also be found in Table 4.1.

Lastly, consider assignments 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, bitwise, or logical operator. The program is converted into three-address form, with no more than one operator on the right side of each assignment. As noted earlier, error codes should be treated as atomic symbols, not numbers. Thus, we assume that the result of

those operations is a non-error value. The transfer function is $Ident[v \mapsto \{OK\}]$, which maps the receiver variable v to the OK non-error value. The same transfer function applies for assignments of the form $v = op\ \ e$, where op is a unary arithmetic, bitwise, or logical operator.

Function Calls

We primarily focus on parameter passing and value return for the case of non-void functions. Note that we transform the interprocedural CFG so that each function has a dummy entry node just before the first statement. We refer to the edge from the function call to this entry node as the *call-to-enter edge*. Each function also has a unique exit node. The edge from this node back to the call site is referred to as the *exit-to-return edge*.

Parameter Passing This is modeled as a two-step process: first the caller exports its arguments into global exchange variables, then the callee imports these exchange variables into its formal parameters. Exchange variables are global variables introduced for the sole purpose of value passing between callers and callees. There is one exchange variable for each function formal parameter.

Suppose function F has formal parameters $f_1, f_2, \ldots, f_n$, where some formal parameters may be of pointer type. Let $F(a_1, a_2, \ldots a_n)$ be a function call to F with actual parameters $a_i \in \mathcal{V} \cup \mathcal{C}$. We introduce a global exchange variable $F\$i$ for each formal parameter. We also introduce a global dereference exchange variable $F\${*}i$ for each formal parameter of pointer type. The interprocedural call-to-enter edge is given the transfer function for a group of n simultaneous assignments $F\$i = a_i$, exporting each actual argument into the corresponding global exchange variable. Rules for assignment transfer functions apply. This means that, in the case of pointer arguments, we pass in the values of dereference variables when applicable.

The edge from the callee's entry node to the first actual statement in the callee is given the transfer function for a group of n simultaneous assignments $f_i = F\$i$. Note that since the transfer functions for assignments are applied, this group additionally includes an assignment of the form $*f_i = F\${*}i$ for each parameter of pointer type, where $*f_i$ is a dereference local variable

```
1
2
3
4  void foo(int *a) {
5
6
7    *a = -5;
8
9    return;
10 }
11
12 int main() {
13   int x = 0;
14
15
16   foo(&x);
17
18
19   x = 6;
20   return 0;
21 }
```

(a) Original

```
1  int* foo$1;
2  int foo$*1;
3
4  void foo(int* a) {
5    int *a;
6    a = foo$1; *a = foo$*1;
7    *a = -5;
8    foo$1 = a; foo$*1 = *a;
9    return;
10 }
11
12 int main() {
13   int x = 0;
14
15   foo$1 = OK; foo$*1 = x;
16   foo(&x);
17   x = foo$*1;
18
19   x = 6;
20   return 0;
21 }
```

(b) Transformed

Figure 4.5: Example making parameter and return value passing explicit. Highlighted assignments emulate transfer functions.

corresponding to pointer formal parameter f_i. This step initializes each formal argument with a value from the corresponding exchange variable. For pointer variables, both the pointer and the corresponding dereference variable are initialized.

Figure 4.5 shows an example illustrating the idea behind pointer parameter passing. Consider the code fragment in Figure 4.5a as though it is transformed into the code fragment in Figure 4.5b. The goal is to make parameter passing explicit. Function foo has one pointer parameter. We declare the corresponding pointer exchange and dereference exchange variables on lines 1 and 2, respectively. A dereference variable corresponding to the original pointer parameter is also declared on line 5. Exchange-variable assignments on lines 6 and 15 emulate the effects of the corresponding parameter-passing transfer functions.

Return Value Passing We also introduce a global return exchange variable $F\$\text{ret}$ for any non-void function F. This variable is used to pass the function result value from the callee to the caller. Thus, for non-void functions, the edges from the callee's last statements to the exit node are given the transfer function $\mathit{Ident}[F\$\text{ret} \mapsto \{e\}]$, where e is the return expression. The interprocedural exit-to-return edge is given the transfer function $\mathit{Ident}[r \mapsto \{F\$\text{ret}\}]$, where $r \in \mathcal{V}$ is the variable in which the caller stores the result of the call, if any.

In addition, we copy back certain other values upon function return. Many functions take a pointer to a caller-local variable where (at any of the two levels) an error code, if any, should be written. In particular, formal dereference variables are copied back into their corresponding dereference exchange variables. The edges from the callee's last statements to the exit node are additionally given the transfer function for a group of at most n simultaneous assignments $F\${*}i = {*}f_i$. Finally, dereference exchange variable values are copied back to any actual variables at the caller's side. The interprocedural exit-to-return edge is given the transfer function for a group of at most n simultaneous assignments ${*}a_i = F\${*}i$, where a_i is a pointer variable or $a_i = F\${*}i$, where a_i is an *address-of* expression. The idea is illustrated on lines 8 and 17 in Figure 4.5b.

Error-Transformation Functions

We attribute a special meaning to calls to the function IS_ERR. As mentioned earlier, this Boolean function is used to test whether a variable contains a pointer error value. Typically, such calls are part of a conditional expression. Depending on the branch taken, we can deduce what the outcome is. If the true branch is selected, then we know that the pointer definitely contained an error value. Conversely, when the false branch is chosen, the pointer cannot possibly contain an error. Therefore, we map this pointer to *OK* in the false branch.

Since our analysis supports error-valued pointers, calls to error-transformation functions ERR_PTR and PTR_ERR are treated as regular function calls, i.e., we apply the transfer functions for parameter passing and value return as discussed in Section 4.3.2.

4.4 Finding and Reporting Bugs

We run the error-propagation and transformation analysis in two different configurations depending on the bugs to be found. The first configuration operates in copy mode with error-handling pattern recognition disabled; this finds bad pointer dereferences and bad pointer arithmetic. We use copy mode because dereferencing (or performing pointer arithmetic using) any copy of a pointer error value is equally bad. Thus, all copies of an error must be considered. Likewise, we disable error-handling pattern recognition because even after handling, an error code remains an invalid address which must not be dereferenced or used in pointer arithmetic.

The second configuration uses transfer mode with error-handling pattern recognition enabled (we use the error-handling patterns described in Sections 3.2.2 and 3.4). We use this configuration when finding bad overwrites. It is common for an error instance to be copied into several variables while only one copy is propagated and the rest can be safely overwritten. In Section 3.4.1 we found that transfer mode leads to significantly fewer false positives when finding overwritten integer error codes. We find that this also holds for pointer error values. We enable error-handling pattern recognition because we are only interested in finding overwrites of unhandled error codes, thus handled errors must be identified.

We identify program locations and variables of interest as explained in Section 4.2 and use the analysis results to determine which of those represent error-valued pointer bugs. Each bug report consists of a sample trace that illustrates how a given error reaches a particular program location l at which the error is dereferenced, used in pointer arithmetic, or overwritten. We use WPDS witness sets to construct these sample paths.

Figure 4.6 shows a more detailed version of the VFS bad pointer dereference from Figure 4.2. The error ENOMEM is first returned by function iget in Figure 4.6a and propagated through three other functions (cnode_make, fill_super and iput, in that order) across two other files (shown in Figure 4.6b and Figure 4.6c). The bad dereference occurs on line 1325 of file fs/inode.c in Figure 4.6c. The sample path produced by our tool is shown in Section 4.4. This path is automatically filtered to show only program points directly relevant to the propagation of the

```
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) Diagnostic output

Figure 4.6: Example of diagnostic output

error. We also provide an unfiltered sample path, not shown 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.

4.5 Experimental Evaluation

We analyzed 52 file systems (including widely-used implementations such as ext3 and ReiserFS), the VFS, the mm, and 4 heavily-used device drivers (SCSI, PCI, IDE, ATA) found in the Linux 2.6.35.4 kernel. We analyze each file system and driver separately along with both the VFS and mm. We have reported all bugs to Linux kernel developers.

4.5.1 Bad Pointer Dereferences

Our tool produces 41 error-valued pointer dereference reports, of which 36 are true bugs. We report only the first of multiple dereferences of each pointer variable within a function. In other words, as soon as a variable is dereferenced in a function, any subsequent dereferences made in this function or its callees are not reported by the tool. Similarly, we do not report duplicate bugs resulting from analyzing shared code (VFS and mm) multiple times.

Table 4.2 shows the number of error-valued pointer dereferences found per file system, module, and driver. Note that the location of a bad dereference sometimes differs from the location where a missing error-check ought to be added. For example, the mm contains a dereference that is only reported when analyzing the Coda, NTFS, and ReiserFS file systems. We count this as a single bad dereference located in the mm. So far, Coda developers have confirmed that this potential error-valued dereference is due to a missing error check in a Coda function. This is likely to be the case for the other two file systems. On the other hand, most of the other dereferences found in shared code are reported when analyzing any file-system implementation. This suggests that the error checks might be needed within the shared code itself.

We classify true dereference bugs into four categories depending on their source:

Table 4.2: Error-valued pointer dereferences. File systems, modules, and drivers producing no diagnostic reports are omitted.

	Number of Diagnostic Reports		
Location	True Bugs	False Positives	Total
AFFS	4	0	4
Coda	0	1	1
devpts	1	0	1
FAT	0	1	1
HFS+	1	0	1
mm	15	0	15
NTFS	3	0	3
PCI	1	0	1
ReiserFS	3	0	3
SCSI	1	0	1
VFS	7	3	10
Total	36	5	41

```
1  struct bnode *bnode_split(...) {
2   struct bnode *node = ...;
3
4   if (IS_ERR(node))
5     return node;
6   ...
7   if (node->next) {
8     struct bnode *next = bnode_find(..., node->next);
9     next->prev = node->this; // bad dereference
10    ...
11  }
12 }
```

Figure 4.7: Example of a bad pointer dereference due to a missing error check in the HFS+ file system

Missing Check

We refer to a missing error check when there is no check at all before dereferencing a potential error-valued pointer. 17 out of 36 (47%) true dereference bugs are due to a missing check. Figure 4.7 shows an example found in the HFS+ file system. Function and variable names have been shortened for simplicity. Function bnode_split calls function bnode_find on line 8, which is expected to return the *next* node. However, function bnode_find may also return one of two

error codes: EIO or ENOMEM. Because of this, callers of function bnode_find must check the pointer result value for errors before any dereferences. Instead, function bnode_split dereferences the result value immediately on line 9, without checking for any errors.

Insufficient Check

We define an insufficient check as any check that does not include a call to function IS_ERR involving the variable being dereferenced. This is the second-most-common scenario leading to error-valued pointer dereferences, accounting for 11 out of 36 true bugs (31%). We identify two variants of insufficient checks. In the first case, the pointer dereference is preceded by a check for NULL but not for an error code (6 bugs). In the second case, there is an error check, but it involves an unrelated pointer variable (5 bugs).

Figure 4.8 shows an example of the first variant. The pointer variable p may receive the error code ENOMEM on line 4. If so, the **while** loop on line 5 is entered, then exits on line 8 since the condition on line 7 is true. Pointer p is passed as parameter to function r_stop on line 11, which checks it for NULL before calling function deactivate_super with variable v as a parameter. Since v contains an error code, the function deactivate_super is indeed called, which then dereferences the error-valued pointer on line 21.

Double Error Code

First identified by Gunawi et al. [26], a double-error-code report refers to a situation in which there are two ways to report an error: by storing an error in a pointer parameter or passing it through the function return value. Action is often taken upon the function return value, which may or may not be checked for errors. At the same time, a copy of the error is left in the pointer argument and dereferenced later. Sometimes this pointer is checked, but only for the NULL value. We find 5 (14%) true error-valued dereferences due to double error codes. An example of a double error code can be found in Figure 4.2 (simplified version) or Figure 4.6 (extended version including diagnostics).

```
1  static int traverse(...) {
2    void *p;
3    ...
4    p = m->op->start(...); // may receive error
5    while (p) {
6      ...
7      if (IS_ERR(p))
8        break;
9      ...
10   }
11   m->op->stop(..., p); // passing error
12   ...
13 }
14
15 static void r_stop(..., void *v) {
16   if (v)
17     deactivate_super(v); // passing error
18 }
19
20 void deactivate_super(struct super_block *s) {
21   if (!atomic_add_unless(&s->s_active, ...)) { // bad deref
22     ...
23   }
24 }
```

Figure 4.8: Example of an insufficient error check in the ReiserFS file system (function r_stop) leading to a bad pointer dereference in the VFS (function deactivate_super)

Global Variable

This category refers to the case in which an error code is stored in a global pointer variable. Only 3 error-valued dereferences fall into this group. In the first situation, the global pointer variable devpts_mnt (declared in the devpts file system) may be assigned one of two error codes: ENOMEM or ENODEV. This variable is dereferenced in a function eventually called from function devpts_kill_index, which is an entry-point function to our analysis, i.e., no function within the analyzed code invokes it. The second and third cases are similar and refer to the VFS global pointer variable pipe_mnt. This variable may be assigned one of six error codes, including ENOMEM and EIO. Variable pipe_mnt is dereferenced in a function eventually called from the system call pipe and also from entry-point function exit_pipe_fs.

```
1  int ___break_lease(...) {
2    struct file_lock *new_fl;
3    int error = 0;
4    ...
5    new_fl = lease_alloc(...); // may receive error
6    ...
7
8    if (IS_ERR(new_fl) && !i_have_this_lease
9      && ((mode & O_NONBLOCK) == 0)) {
10     error = PTR_ERR(new_fl);
11     goto out;
12   }
13   ...
14
15   if (i_have_this_lease || (mode & O_NONBLOCK)) {
16     error = -EWOULDBLOCK;
17     goto out;
18   }
19   error = wait_event_interrupt(new_fl->fl_wait, ...);
20   ...
21 out:
22   ...
23   return error;
24 }
```

Figure 4.9: Example of a false positive found in the VFS

False Positives

Finally, we identified 5 out of 41 reports (12%) to be false positives. Figure 4.9 illustrates an example. Pointer variable new_fl may receive an error code on line 5. There are two conditionals on lines 8 and 9 and on line 15. Variable new_fl is checked for errors in the first conditional, but the call to function IS_ERR is part of a compound conditional statement. Our tool correctly recognizes that even though there is an error, the whole expression may not evaluate to true. Nonetheless, the two conditionals are complementary: the conditional statement on line 15 evaluates to true if that on lines 8 and 9 was false, thereby covering all possibilities. The analysis does not detect this, so the dereference on line 19 is reported. This scenario is found twice.

Other false positives arise when (1) the error check is not exhaustive, but the missing error codes cannot possibly reach that program point; (2) there is a double error code and one is checked before dereferencing the other; and (3) a copy of the error is made and checked before

Table 4.3: Bad pointer arithmetic

	Number of Diagnostic Reports		
Location	True Bugs	False Positives	Total
Coda	0	1	1
mm	15	0	15
ReiserFS	1	0	1
Total	16	1	17

dereferencing the original variable. We can easily remove (1) since we have information regarding what error codes reach a given program point. Similarly, we can remove (3) by running the analysis in transfer mode. On the other hand, the false positives resulting from (2) and the example described in Figure 4.9 would require more effort to remove.

4.5.2 Bad Pointer Arithmetic

Table 4.3 shows the results of our analysis of pointer arithmetic applied to pointers whose values are actually error codes, not addresses. Our tool reports 17 instances of bad pointer arithmetic. We identify 16 true bugs: 15 from the mm and 1 from the ReiserFS file system. Note that we only report the first instance in which an error-valued pointer is used to perform pointer arithmetic. Subsequent bad uses, including bad dereferences, are not reported. Similarly, if the error-valued pointer is first dereferenced, subsequent uses in pointer arithmetic are not reported.

As with bad pointer dereferences in Section 4.5.1, most of the bad pointer-arithmetic instances are due to missing checks (75% or 12 out of 16 reports). The remaining bad pointer operations are surrounded by conditionals, but none of them include checks for errors in the operands. The majority of the reports involve pointer additions (69% or 11 out of 16 reports), while the rest involve subtraction. We find no bad increments or decrements.

In all cases but one, the error-valued pointer is assumed to contain a valid address that is used to calculate another address. The one exception is a calculation involving an error-valued pointer that determines the function return value. In all situations, the error-valued pointer may contain the error ENOMEM. There are two cases in which the pointer may additionally contain

```
1  struct buffer_head *ext3_getblk(..., int *errp) {
2    int err;
3    ...
4    err = ext3_get_blocks_handle(...); // may receive error
5    ...
6    *errp = err; // copy error
7
8    if (!err && ...) {
9      ...
10   }
11   return NULL;
12 }
13
14 struct buffer_head *ext3_bread(..., int *err) {
15   struct buffer_head * bh;
16   bh = ext3_getblk(..., err); // err has an error
17
18   if (!bh)
19     return bh;
20   ... // code leads to overwrites
21 }
```

Figure 4.10: Double error code in the ext3 file system, leading to 12 overwrite false positives

the EFAULT error code, which (ironically) denotes a bad address.

Most cases, including all those in the mm, are solely triggered by the SCSI driver. An example is shown in Figure 4.3. Callers of function kfree (line 3) may pass in a pointer variable that contains the error code ENOMEM, now in variable x. The variable is further passed to function virt_to_head_page when it is invoked on line 6. Finally, this function uses x to perform some pointer arithmetic on line 11, without first checking for any errors.

A false positive is found in the Coda file system. Function cnode_make calls a function that may return an error code and also stores it in a pointer parameter (double error code). If the return value is any error code but ENOENT, then cnode_make further propagates the error to its callers. Otherwise, the function proceeds to call a function that uses the pointer parameter to perform pointer arithmetic. This would lead to bad pointer arithmetic if the pointer parameter could contain ENOENT; however, we find that this is not the case.

4.5.3 Bad Overwrites

Our tool produced 7 reports describing overwrites of error-valued pointer variables. As with other kinds of bugs, we eliminate duplicated reports that belong to shared code (VFS and mm). We identified three true bugs located in the mm. In two cases, an error is stored in a global variable, which is overwritten later without first being checked for errors. In the remaining case, the error is stored in a static local variable. Three out of the four false positives are found to be duplicates but located in file-system specific code. This is due to cloned (copied and pasted) code. We are not able to recognize this automatically, thus we count these as multiple reports. These overwrites are located in the ext2, System V, and UFS file systems and are due to complex loop conditions. The other false positive is found in the mm.

The tool reported 31 cases in which errors contained in dereference variables are overwritten, among which we only identify 1 true bug in the SCSI driver. The remaining false positives are associated with the ext3 (15 reports), UDF (12 reports), and UFS (2 reports) file systems and the SCSI (1 report) driver. There is complete overlap between reports belonging to ext3 and UDF due to cloned code. Double error codes, as discussed in Section 4.5.1, caused most false positives (87%). Figure 4.10 shows an example. An error returned on line 4 is copied to the formal parameter ∗errp on line 6. Function ext3_getblk then returns NULL. The caller ext3_bread stores the returned value in bh, which is further returned on line 19. However, because we are tracking variable err and not variable bh, the analysis chooses the path that skips the conditional of line 18 and eventually leads to 12 overwrites. The same piece of code is found in file-system–specific code for both ext3 and UDF, accounting for every false positive in the latter. Note that we find no overwrites of error-valued dereference variables due to assignments to pointer variables.

We find considerably fewer overwrites than dropped errors due to overwrites of integer error codes (Section 3.4.1). One difference between integer and pointer error values is that there is an explicit error check function for the latter (IS_ERR). The existence of such a function may influence developers into being more aware of error checking, thus contributing to fewer bugs. Another reason might be that although error-valued pointers are part of many propagation

chains, these errors may ultimately end up back in **int** variables.

4.5.4 False Negatives

We identify three possible sources of false negatives: function pointers, aliasing, and structure fields. We adopt a technique previously employed by Gunawi et al. [26], which exploits the fact that function pointers in Linux file systems are used in a fairly restricted manner, allowing us to identify the set of all possible implementations of a given file-system operation. Calls across such functions pointers are rewritten as **switch** statements that choose among possible implementations nondeterministically. This technique accounts for approximately 80% of function pointer calls. We treat the remaining calls as *Ident*. Thus, if any function that propagates errors is called through one of these unresolved function pointers, then subsequent error-valued pointer dereferences or other misuses are not detected. Similarly, we do not perform a points-to analysis. If a pointer variable p is assigned another pointer variable, which later receives an error code, the analysis cannot determine that p may also contain an error code. Finally, our analysis is not field sensitive, thus it does not currently track errors stored in structure fields.

4.6 Performance

We used a dual 3.2 GHz Intel Pentium 4 processor workstation with 3 GB RAM to run our experiments. We analyzed 1,538,082 lines of code, including white space and comments. Counting reanalysis of the VFS and mm as used by multiple file systems, we processed 8,875,522 lines of code in total. Table 4.4 shows the size (in thousands of lines of code) for those file systems and drivers in which bugs are found. The table also includes running time and memory usage for the two different analysis configurations described in Section 4.4. Overall, we find that the analysis scales and performs quite well even with the added burden of tracking pointer-typed variables and their corresponding dereference variables.

Finally, we find that an average of 42% of the variables that hold errors at some point during execution are pointer variables. This shows that error transformation is not merely an anomaly;

Table 4.4: Analysis performance for a subset of file systems and drivers. Sizes include 133 KLOC of shared VFS and mm code.

		Bad Dereferences & Arithmetic		Bad Overwrites	
File System	KLOC	Time (min:sec)	Memory (GB)	Time (min:sec)	Memory (GB)
AFFS	137	2:48	0.86	3:17	0.87
Coda	136	2:54	0.83	3:15	0.84
devpts	134	2:36	0.81	3:06	0.82
FAT	140	3:06	0.88	3:21	0.90
HFS+	143	2:54	0.86	3:31	0.87
NTFS	162	4:12	1.37	4:39	1.39
PCI	191	3:24	1.00	3:55	1.02
ReiserFS	161	4:06	1.36	4:37	1.37
SCSI	703	11:00	2.42	13:04	2.52
Avg FS	-	2:54	0.87	3:16	0.89
Avg Drivers	-	5:24	1.44	6:18	1.50

it is critical to understanding how error propagation really works.

4.7 Other Linux Versions and Code Bases

We also analyzed the Linux kernel 2.6.38.3, which was released seven months after the version discussed throughout this section. The results show that 9 bad dereferences reported in Section 4.5.1 are no longer present in the newer kernel, but 8 new bad dereferences are introduced. We find that 6 bad pointer dereferences are fixed by adding the appropriate error checks while code for the rest has simply been removed. An example of a bad pointer dereference that has been fixed is that shown in Figure 4.7. Bugs related to bad pointer arithmetic and bad pointer overwrites remain the same in both versions. This demonstrates that finding and fixing these kinds of bugs is not a one-time operation. New bugs are introduced as code evolves.

Inspection of several other code bases reveals that FreeBSD, OpenSolaris, and Xen (hypervisor and guest) also define and use functions that convert error codes between integers and pointers, including an IS_ERR function to check for errors in pointers. Our tool could be used to analyze these and other similar code bases.

4.8 Summary

In this chapter, we described three kinds of bugs arising from error codes masquerading as pointer values: bad dereferences, bad pointer arithmetic, and bad overwrites. We showed how to extend the error-propagation analysis from Chapter 2 to account for error transformation as in the Linux kernel in order to find these bugs. We applied the analysis to 52 Linux file system implementations, the VFS, the mm and 4 drivers, finding a total of 56 true bugs. Hiding error codes in pointers may seem distasteful, but it is by no means uncommon: we find that 42% of the variables that may contain error codes are pointer variables. Thus, understanding the behavior of error-valued pointers is an important component to having a more complete understanding of how errors propagate in large systems such as the Linux kernel.

Chapter 5

Error-Code Mismatches Between Code and Documentation

User applications rely on systems software to run as specified. 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. Inaccurate documentation can mislead programmers and cause software to fail in unexpected ways. Unfortunately, writing and maintaining accurate code documentation is difficult. This is particularly true for large code bases such as the Linux kernel.

In this chapter, 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 makes 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.

5.1 Finding Error Return Values

We find the set of error codes that each function returns, and then focus on the set of file-related system-call functions. We apply the error-propagation analysis from Chapter 2 to find the set of possible error codes at function exit points. We run the analysis in transfer mode, with error-handling recognition disabled. At each return statement r in function f, we retrieve the associated weight w. Let $\mathcal{E}$ be the set of all error constants and $\mathcal{R} \subseteq \mathcal{C}$ be the set of possible constant values returned by function f (if any). Then $\mathcal{R} \cap \mathcal{E}$ 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. We described the use of witness-tracing information for the construction of error-propagation paths in Section 3.3; we use witnesses here to justify each error code that a function exit point is claimed to return.

Figure 5.1 shows examples of return-value reports. Function bar returns a constant error code at line 2. Figure 5.1b 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 12 and 13). In the first case, line 12, foo calls function bar. In the second case, foo returns the value contained in variable retval. The corresponding reports are shown in Figure 5.1c. 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.

5.2 Linux 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 [68] rather than sophisticated natural-language-processing algorithms as used by Tan et al. [65].

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.

5.3 Experimental Evaluation

We analyzed 52 file-system implementations and the VFS (871 KLOC in total) found in the Linux 2.6.32.4 kernel. We ran the error-propagation analysis to find the set of 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 compared these error codes against version 2.39 of the Linux manual pages for each of 42 file-related system calls. We found 1,784 error-code mismatches

Table 5.1: Number of file systems per system call returning undocumented errors. a:E2BIG, b:EACCES, c:EAGAIN, d:EBADF, e:EBUSY, f:EEBADHDR, g:EFAULT, h:EBIG, i:EINTR, j:EINVAL, k:EIO, l:EISDIR, m:EMFILE, n:ENLINK, o:ENFILE, p:ENODEV, q:ENOENT, r:ENOMEM, s:ENOSPC, t:ENOTDIR, u:ENXIO, v:EPERM, w:ERANGE, x:EROFS, y:ESPIPE, z:ESRC, aa:ETXTBSY, ab: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	y	z	aa	ab	Total
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
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

```
1  int bar() {
2    return −EIO;
3  }
4
5  int foo() {
6    int retval;
7    if (...)
8      retval = −ENOMEM;
9    else if (...)
10     retval = −EPERM;
11   else
12     return bar();
13   return retval;
14 }
```

(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:12: EIO* returned from function foo

ex.c:8: "retval" receives an error from ENOMEM
ex.c:13: ENOMEM* EPERM returned from function foo

ex.c:10: "retval" receives an error from EPERM
ex.c:13: ENOMEM EPERM* returned from function foo
```

(c) Reports for function foo

Figure 5.1: Example code fragment and corresponding reports

among all file systems and system calls. The following sections describe these results in more detail.

5.3.1 Undocumented Error Codes

Comparing our code analysis with our documentation analysis reveals two kinds of mismatches: 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 5.1 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 5.1 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 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

Table 5.2: Distribution of bug reports

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 5.3: File systems with the most undocumented error codes

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

the report is attributed to VFS. Table 5.1 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 5.2 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.

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 5.3 shows the top five file systems that return the most undocumented error instances.

[1]The VFS is treated as a separate entity after bug-report classification. Thus, the maximum possible count in each cell of Table 5.1 is 53.

Table 5.4: Undocumented error codes most commonly returned

Error Code	Total	File Systems	System Calls
EIO	274	21	14
EROFS	199	21	12
ENOMEM	185	26	14
EINVAL	176	22	21
ENODEV	106	21	27

SMB is at the top of the list with a total of 255 instances, from which we find 26 different undocumented error codes. For SMB, the error code with the most instances (20) is ENODEV (no such device). Table 5.4 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 5.1.

Table 5.5 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 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.

Table 5.5: 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).

Call	Error	File System 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	●	○	○	●	●	●
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	●	○	○	○	●	○
ioctl	ENOSPC	○	●	○	○	●	○
	EROFS	○	●	●	●	●	○
	ESPIPE	●	○	○	●	●	●
	EIO	●	○	○	●	●	●
	ENODEV	●	○	○	●	●	●
lchown	EFBIG	●	○	○	●	○	○
	ERANGE	○	●	●	○	○	○
	EEXIST	○	●	●	○	○	○
	EAGAIN	●	○	○	○	○	○
	ENOSPC	○	●	○	○	○	○
	EINVAL	●	●	●	●	○	○
	ETXTBSY	●	○	○	●	○	○
	ENODEV	●	○	○	○	○	○
link	ENODEV	●	○	○	○	○	○
	EAGAIN	●	●	○	○	○	○
	EBUSY	○	○	○	●	○	○
	EINVAL	●	●	○	●	○	●
	EBADF	○	○	○	○	○	●
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	●	○	○	○	○	○

Table 5.5: 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). (continued)

Call	Error	File System c	e	j	r	x	v
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 System 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	●	○	○	○	○	○

Call	Error	File System c	e	j	r	x	v
unlink	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 5.6: Analysis performance for a subset of file systems. KLOC gives the size of each file system in thousands of lines of code, including 59 KLOC of shared VFS code.

File System	KLOC	Time (min:sec)	Memory (MB)
CIFS	90	3:01	246
ext3	82	2:24	267
IBM JFS	91	2:30	287
ReiserFS	86	2:36	309
XFS	159	4:05	491

5.3.2 Performance

We performed our experiments on a dual 3.2 GHz Intel processor workstation with 3 GB RAM. Table 5.6 shows the sizes of file systems (in thousands of lines of code) and the time and memory required to analyze each. We restricted our focus to the five popular file systems presented in detail in Table 5.5. We give the total running time, which includes (1) extracting a textual WPDS representation, (2) solving the poststar query, and (3) traversing witnesses to produce the sample paths. For the file systems under consideration, the total running time ranges from 2 minutes 24 seconds for ext3 to just over 4 minutes for XFS. The analysis consumes between 246 MB and 291 MB of memory for CIFS and XFS, respectively.

5.4 Summary

In this chapter, we used the error-propagation analysis from Chapter 2 to find the set of error codes returned by each function in a program. We analyzed 52 Linux file systems, including CIFS, ext3, IBM JFS, ReiserFS and XFS. After retrieving the results for 42 file-related system calls, we compared against the Linux manual pages, finding 1,784 undocumented error instances across all file systems. Sometimes undocumented errors may be attributed to particular file-system implementations (e.g., only a handful of implementations return the error code). Other times, the mismatch may be attributed to the documentation (e.g., most file systems return the error code).

Chapter 6

Error-Propagation Bugs in User Applications

The purpose of this chapter is to show that (1) error handling is also important for user applications, (2) the error-propagation bugs described in this dissertation are not exclusive to Linux file systems and drivers, and (3) error-propagation bugs can also be found in widely-used C++ applications that use the return-code idiom. We apply the error-propagation analysis to find dropped errors (Chapter 3) in two widely-used user applications: Mozilla Firefox and SQLite. The following sections describe the results.

6.1 Case Study: Mozilla Firefox

Our first case study is the Mozilla Firefox web browser. Firefox is written in C++, however it uses the return-code idiom. Figure 6.1 shows a subset of macros that define error codes used in Firefox. For example, the macro NS_ERROR_UNEXPECTED defines the error that is used when an unexpected error occurs. Error codes have type nsresult, which is a **typedef** for **unsigned long**. We consider 49 different error codes. Firefox also defines several heavily-used macros that log errors. Figure 6.2 shows two examples. NS_ENSURE_TRUE takes a parameter x and an error code ret. If x is NULL, then a warning is printed, and the error code is returned.

```
#define NS_ERROR_BASE                          ((nsresult) 0xC1F30000)

/* Returned when an instance is not initialized */
#define NS_ERROR_NOT_INITIALIZED               (NS_ERROR_BASE + 1)

/* Returned when an instance is already initialized */
#define NS_ERROR_ALREADY_INITIALIZED           (NS_ERROR_BASE + 2)

/* Returned by a not implemented function */
#define NS_ERROR_NOT_IMPLEMENTED               ((nsresult) 0x80004001L)

/* Returned when a given interface is not supported. */
#define NS_NOINTERFACE                         ((nsresult) 0x80004002L)
#define NS_ERROR_NO_INTERFACE                  NS_NOINTERFACE

#define NS_ERROR_INVALID_POINTER               ((nsresult) 0x80004003L)
#define NS_ERROR_NULL_POINTER                  NS_ERROR_INVALID_POINTER

/* Returned when a function aborts */
#define NS_ERROR_ABORT                         ((nsresult) 0x80004004L)

/* Returned when a function fails */
#define NS_ERROR_FAILURE                       ((nsresult) 0x80004005L)

/* Returned when an unexpected error occurs */
#define NS_ERROR_UNEXPECTED                    ((nsresult) 0x8000ffffL)

/* Returned when a memory allocation fails */
#define NS_ERROR_OUT_OF_MEMORY                 ((nsresult) 0x8007000eL)

/* Returned when an illegal value is passed */
#define NS_ERROR_ILLEGAL_VALUE                 ((nsresult) 0x80070057L)
#define NS_ERROR_INVALID_ARG                   NS_ERROR_ILLEGAL_VALUE

/* Returned when a class doesn't allow aggregation */
#define NS_ERROR_NO_AGGREGATION                ((nsresult) 0x80040110L)
```

Figure 6.1: Subset of macros defining errors in Firefox

```
#define NS_ENSURE_TRUE(x, ret)                                  \
   PR_BEGIN_MACRO                                               \
     if (NS_UNLIKELY(!(x))) {                                   \
       NS_WARNING("NS_ENSURE_TRUE(" #x ") failed");             \
       return ret;                                              \
     }                                                          \
   PR_END_MACRO

#define NS_ENSURE_STATE(state)                                  \
   NS_ENSURE_TRUE(state, NS_ERROR_UNEXPECTED)
```

Figure 6.2: Two examples of macros that use log errors

We ran the analysis from Chapter 3 to find dropped errors in Mozilla Firefox version mozilla-central 155f67c2c578 (the current trunk at the time of running the analysis). This version is in-between the official releases 13.0.1 and 14.0.1 (the most recent official release at the time of writing this dissertation). Our tool found a total of 1,388 dropped errors. We have manually inspected 1,029 bug reports (74%). The results are summarized in Table 6.1. We divide the reports into three groups: true bugs, harmless dropped errors, and false positives. The following sections describe each category in more detail.

6.1.1 True Bugs

We define true bugs as dropped errors that (1) are completely ignored, or (2) are logged, but error-logging is not sufficient. We classify 486 out of 1,029 inspected dropped errors as true bugs. This accounts for 47% of the total number of inspected bug reports. We identified a subset of 261 unique true-bug instances, and have reported them to Mozilla developers. The rest of the reports will be submitted as soon as we receive feedback on the initial set of bug reports.

True bugs are located in 23 out of 26 Firefox components (listed in Table 6.1). The components with the most true bugs are editor (88 reports), netwerk (67 reports), and content (59 reports). We find that the number of bugs per component is not proportional to the component's size. The most bug-dense component is embedding, while the js component is the least bug-dense.

51% of the dropped errors are not even logged. The remaining 49% are logged. Most of these errors are logged right before they are generated. For example, developers make heavy use of error-generator macros such as NS_ENSURE_TRUE, NS_ENSURE_STATE, and NS_ENSURE_ARG.

We find that 63% of the functions that ignore callees' error return values have return types nsresult or NS_IMETHODIMP (which is defined as nsresult). As mentioned earlier in this chapter, nsresult is the type of error codes. Thus, these functions could continue to propagate these ignored errors without requiring changes to any function signatures in the application. Many of these functions propagate errors in other scenarios. Others return NS_OK (an nsresult value

Table 6.1: Inspected dropped errors in Mozilla Firefox. Results are shown per component, and divided into true bugs, harmless dropped errors (H1: dropped in the process of shutting down, H2: dropped in the process of releasing resources, H3: documented by developer to be ignored, and H4: logged), and false positives (FP1: double error code, FP2: met precondition, and FP3: imprecision in our tool).

			Harmless Dropped Errors					False Positives				
Component	KLOC	True Bugs	H1	H2	H3	H4	Total	FP1	FP2	FP3	Total	Grand Total
accessible	62	1	1	0	0	40	41	1	1	0	2	44
caps	7	5	0	0	0	6	6	0	4	0	4	15
chrme	3	0	0	0	0	0	0	0	0	0	0	0
content	380	59	3	8	6	47	64	16	1	2	19	142
docshell	20	5	0	0	1	2	3	7	0	3	10	18
dom	187	88	9	0	0	23	32	37	1	2	40	160
editor	71	31	0	2	0	20	22	4	1	2	7	60
embedding	22	36	0	0	0	7	7	5	0	0	5	48
extensions	40	11	0	0	0	2	2	0	1	0	1	14
gfx	578	12	0	0	0	2	2	1	0	1	2	16
intl	49	3	0	0	0	0	0	0	0	0	0	3
js	342	2	0	0	0	14	14	2	0	1	3	19
layout	358	25	0	0	1	40	41	5	13	0	18	84
modules	71	7	0	2	1	5	8	8	0	2	10	25
netwerk	139	67	10	1	7	14	32	0	1	16	17	116
parser	59	14	0	0	1	1	2	0	0	2	2	18
profile	1	0	0	0	0	0	0	0	0	0	0	0
rdf	15	14	0	0	0	2	2	0	0	0	0	16
security	141	16	1	0	2	1	4	1	0	1	2	22
startupcache	2	1	0	0	0	1	1	0	0	0	0	2
storage	15	6	0	0	0	6	6	1	0	0	1	13
toolkit	134	58	2	2	7	19	30	1	3	3	7	95
view	4	0	0	0	0	0	0	0	0	0	0	0
widget	165	10	0	0	1	47	48	1	0	0	1	59
xpcom	126	14	3	0	5	6	14	6	4	1	11	38
xpfe	9	1	0	0	0	0	0	0	0	0	0	1
Grand Total	3000	486	29	15	32	305	381	96	30	36	162	1029

representing a non-error) no matter what the outcome is. We also find that 24% of the functions whose error return values are ignored are used in an inconsistent manner: some callers save the error while others ignore it. Engler et al. [17] have shown that inconsistencies are often bug indicators.

Preliminary feedback from developers on a small subset of bug reports suggests that dropping the error NS_ERROR_OUT_OF_MEMORY is not critical: there is really nothing left to do if the application runs out of memory. We find that 86 out of 486 true-bug reports (17%) correspond to NS_ERROR_OUT_OF_MEMORY dropped errors only. The top three dropped error codes are NS_ERROR_FAILURE, NS_ERROR_UNEXPECTED, and NS_ERROR_INVALID_POINTER.

So far, developers have identified two potential security-related bugs among our reports in the dom and xpcom components. One of the problems has been fixed. The original code is shown in Figure 6.3a. Function DashArrayToJSVal may return one of two error codes: NS_ERROR_OUT_OF_MEMORY or NS_ERROR_FAILURE. The error is dropped on line 3 while failing to store it in the parameter error (this component declares a class ErrorResult with a data member to store the error code). Ignoring the error could cause mozDash to remain uninitialized, which could lead to a potential security vulnerability. Figure 6.3b shows the code after developers fixed the bug. The error is simply stored in the parameter error. Note that, although the fix was trivial, developers actively discussed this bug for more than a week, and a first patch that proposed a different fix was rejected. Meanwhile, we were asked to keep this in strict confidence. The fix of the second security bug is in progress and has been classified as *security-moderate*.

Another example of a true bug is shown in Figure 6.4. An ignored error could cause persistent information about the cache to be lost silently. Function Close in Figure 6.4a may return one of two error codes: NS_ERROR_UNEXPECTED or NS_ERROR_NOT_INITIALIZED. FlushHeader may return either error when called on line 12. The constant NS_ERROR_UNEXPECTED can also be assigned to variable rv on line 16, which is returned on line 21. Function Shutdown_Private in Figure 6.4b ignores both errors when calling Close on line 11. Note that Shutdown_Private's

```
1 JS::Value nsCanvasRenderingContext2DAzure::GetMozDash(JSContext* cx, ErrorResult& error) {
2   JS::Value mozDash;
3   DashArrayToJSVal(CurrentState().dash, cx, &mozDash);
4   return mozDash;
5 }
```

(a) Dropped error introduces potential security vulnerability

```
1 JS::Value nsCanvasRenderingContext2DAzure::GetMozDash(JSContext* cx, ErrorResult& error) {
2   JS::Value mozDash;
3   error = DashArrayToJSVal(CurrentState().dash, cx, &mozDash);
4   return mozDash;
5 }
```

(b) Code after bug fix

Figure 6.3: An example of a potential security bug in Firefox due to a dropped error

return type is nsresult, thus these errors could be further propagated by the function. Instead, Shutdown_Private always returns NS_OK. The most troublesome part is that the errors are not even logged.

6.1.2 Harmless Dropped Errors

We classify dropped errors as harmless if (1) developers have documented the fact that it is OK to drop them, (2) the application is already shutting down, (3) the application is at the end of the process of releasing resources, and (4) emitting a warning is sufficient. We identify 381 harmless dropped errors, which accounts for 37% of the total number of inspected bug reports.

Figure 6.5 shows an example in which, according to developers' comments in the code, it is OK to drop errors. Function Check may return errors when called on line 13; however, the comment says that errors returned by Check are ignored in this case as it is preferable to return incomplete results rather than failing altogether. Unfortunately, only 32 out of 381 harmless dropped errors are documented. Ideally, all harmless dropped errors should be documented by developers.

Figure 6.6 shows an example of errors dropped in the process of shutting down the application. Function Shutdown may return error NS_ERROR_UNEXPECTED when called on line 11. In this

```
1  nsresult nsDiskCacheMap::Close(bool flush) {
2    nsresult rv = NS_OK;
3
4    if (mMapFD) {
5      // close block files
6      rv = CloseBlockFiles(flush);
7      if (NS_SUCCEEDED(rv) && ...) {
8        // write the map records
9        rv = FlushRecords(false);
10       if (NS_SUCCEEDED(rv)) {
11         mHeader.mIsDirty = false;
12         rv = FlushHeader();
13       }
14     }
15     if (... && (NS_SUCCEEDED(rv)))
16       rv = NS_ERROR_UNEXPECTED;
17
18     mMapFD = nsnull;
19   }
20   ...
21   return rv;
22 }
```

(a) Close may return error codes

```
1  nsresult nsDiskCacheDevice::Shutdown_Private(bool flush) {
2    CACHE_LOG_DEBUG(("CACHE: disk ... [%u]\n", flush));
3
4    if (Initialized()) {
5      // check cache limits in case we need to evict.
6      EvictDiskCacheEntries(mCacheCapacity);
7
8      (void) nsCacheService::SyncWithCacheIOThread();
9
10     // write out persistent information about the cache.
11     (void) mCacheMap.Close(flush);
12
13     mBindery.Reset();
14     mInitialized = false;
15   }
16   return NS_OK;
17 }
```

(b) Shutdown_Private ignores errors returned by Close and always returns NS_OK

Figure 6.4: An example of a dropped error in Firefox

```
1  nsresult nsUrlClassifierDBServiceWorker::DoLookup(const nsACString& spec,
2                                                    nsIUrlClassifierLookupCallback* c) {
3    ...
4    nsAutoPtr<nsTArray<nsUrlClassifierLookupResult> > results;
5    results = new nsTArray<nsUrlClassifierLookupResult>();
6    if (!results) {
7      c->LookupComplete(nsnull);
8      return NS_ERROR_OUT_OF_MEMORY;
9    }
10
11   // we ignore failures from Check because we'd rather return the
12   // results that were found than fail.
13   Check(spec, *results);
14   ...
15   return NS_OK;
16 }
```

Figure 6.5: An example in which developers document that errors can be dropped

```
1  void Navigator::Invalidate() {
2    mWindow = nsnull;
3
4    if (mPlugins) {
5      mPlugins->Invalidate();
6      mPlugins = nsnull;
7    }
8    ...
9
10   if (mPowerManager) {
11     mPowerManager->Shutdown();
12     mPowerManager = nsnull;
13   }
14
15   ...
16 }
```

Figure 6.6: Example of a dropped error when shutting down

case, the error is logged. Note that the caller has **void** return type. 29 out of 381 reports fall into this category. There are 14 errors dropped while releasing resources. An example is shown in Figure 6.7. Function Destroy may return the NS_ERROR_OUT_OF_MEMORY error on line 8 in Figure 6.7a. This error is dropped on line 4 in Figure 6.7b. Note that the error is logged, and the caller has **void** return type.

Finally, the remaining reports (305 out of 381) fall into the general category in which,

```
1 NS_IMETHODIMP nsFrameLoader::Destroy() {
2
3   // most removal done, 50 lines of code
4
5   if ((mNeedsAsyncDestroy || !doc ||
6         NS_FAILED(doc->FinalizeFrameLoader(this))) && mDocShell) {
7     nsCOMPtr<nsIRunnable> event = new nsAsyncDocShellDestroyer(mDocShell);
8     NS_ENSURE_TRUE(event, NS_ERROR_OUT_OF_MEMORY);
9     NS_DispatchToCurrentThread(event);
10
11    // Let go of our docshell now that the async destroyer holds on to the docshell
12    mDocShell = nsnull;
13  }
14
15  return NS_OK;
16 }
```

(a) Function Destroy may return an error

```
1 void nsObjectLoadingContent::RemovedFromDocument() {
2   if (mFrameLoader) {
3     // XXX This is very temporary and must go away
4     mFrameLoader->Destroy();
5     mFrameLoader = nsnull;
6
7     // Clear the current URI
8     mURI = nsnull;
9   }
10  ...
11 }
```

(b) Function RemovedFromDocument ignores an error returned by function Destroy

Figure 6.7: An example of an error dropped during the release of resources

regardless the context, an error warning is sufficient. An example is illustrated in Figure 6.8. Function TakeFocus calls function SetFocus on line 8 in Figure 6.8b. Function SetFocus uses the macro NS_ENSURE_ARG on line 5 in Figure 6.8a. This macro ensures that newFocus is not NULL. If it is NULL, then an error warning is emitted and the function returns the error NS_ERROR_INVALID_ARG. Function TakeFocus ignores this error and returns NS_OK. Note that all callers of SetFocus ignore the error.

```
1 NS_IMETHODIMP nsFocusManager::SetFocus(nsIDOMElement* aElement, ...) {
2
3   ...
4   nsCOMPtr<nsIContent> newFocus = do_QueryInterface(aElement);
5   NS_ENSURE_ARG(newFocus);
6
7   SetFocusInner(newFocus, aFlags, true, true);
8
9   return NS_OK;
10 }
```

(a) Function SetFocus may return an error

```
1 NS_IMETHODIMP Accessible::TakeFocus() {
2   ...
3   nsIContent* focusContent = mContent;
4   ...
5   nsCOMPtr<nsIDOMElement> element(do_QueryInterface(focusContent));
6   nsFocusManager* fm = nsFocusManager::GetFocusManager();
7   if (fm)
8     fm->SetFocus(element, 0);
9
10  return NS_OK;
11 }
```

(b) Function TakeFocus ignores an error returned by SetFocus

Figure 6.8: An example in which emitting an error warning is sufficient

6.1.3 False Positives

We identified 162 false positives, which account for 16% of the total number of inspected reports. We divide false positives into three categories: (1) double error code, (2) met preconditions, and (3) imprecision in our tool.

The first category consists of a variant of the double-error-code scenario discussed in Section 4.2.1. An error is dropped, however there is an alternative way for the caller to know that an error arises. A total of 96 out of 162 false positives fall into this category. An example is illustrated in Figure 6.9. The NULL pointer variable stack is passed to function CreateStack on line 9 in Figure 6.9b. Function CreateStack in Figure 6.9a may return error NS_ERROR_FAILURE on lines 3 and 7. In either case, the variable stack's contents are not modified. Thus, when the function returns, it is sufficient for the caller to check the variable stack to know whether the

stack was created successfully (see line 10 in Figure 6.9b). If not, stack must still be NULL and the current stack is not updated.

Another example is shown in Figure 6.10. Function GetRuleNodeForContent in Figure 6.10a returns the error NS_ERROR_UNEXPECTED on line 9. The caller GetCSSStyleRules in Figure 6.10b drops the error on line 8. However, the function checks variable ruleNode, which is previously passed as an argument to GetRuleNodeForContent (and initialized to nsnull). The comment inside the conditional says how this could fail and suggests to bail out by returning NS_OK. This is similar to the scenario discussed in Section 6.1.2. In this case, the comments reinforce the fact that checking variable ruleNode is sufficient to determine whether the operation has failed.

The second category consists of cases in which a precondition is met so that the callee cannot possibly return errors, and thus the caller can safely ignore the return value. 30 out of 162 reports correspond to this category. An example is shown in Figure 6.11. Function AllocateContents is called with count 0 on line 13 in Figure 6.11a. Function AllocateContents cannot possibly return an error when the count is 0 (see the conditional on line 3 in Figure 6.11b). Thus, the caller safely ignores the return value.

Finally, the third category includes reports that do not actually represent dropped errors. These are issues/patterns that need to be fixed/incorporated in the tool. There are 36 out of 162 reports in this category. Most cases are related to errors saved only when in DEBUG mode. We did not compile the code in this mode. The rest are related to the tracking of errors in temporary variables. This issue has to be further investigated and fixed.

6.1.4 Performance

Our analysis of the Mozilla Firefox code base (3 million lines of code) was run on a Core i7 3 GHz machine with 192 GB RAM. The analysis is divided into three phases: (1) extracting a textual representation of the WPDS (Section 2.6.1), (2) collapsing the WPDS rules (Section 2.5.2), and (3) solving the dataflow problem and producing diagnostic information (Sections 2.4 and 3.2).

```
1  nsresult XPCJSStack::CreateStack(JSContext* cx, nsIStackFrame** stack) {
2    if (!cx)
3      return NS_ERROR_FAILURE;
4
5    JSStackFrame *fp = NULL;
6    if (!JS_FrameIterator(cx, &fp))
7      return NS_ERROR_FAILURE;
8
9    return XPCJSStackFrame::CreateStack(cx, fp, (XPCJSStackFrame**) stack);
10 }
```

(a) Function CreateStack may return an error

```
1  NS_IMETHODIMP nsXPConnect::GetCurrentJSStack(nsIStackFrame * *aCurrentJSStack) {
2    NS_ASSERTION(aCurrentJSStack, "bad param");
3    *aCurrentJSStack = nsnull;
4
5    JSContext* cx;
6    // is there a current context available?
7    if (NS_SUCCEEDED(Peek(&cx)) && cx) {
8      nsCOMPtr<nsIStackFrame> stack;
9      XPCJSStack::CreateStack(cx, getter_AddRefs(stack));
10     if (stack) {
11       ...
12       NS_IF_ADDREF(*aCurrentJSStack = stack);
13     }
14   }
15   return NS_OK;
16 }
```

(b) Function GetCurrentJSStack checks the parameter instead of the return value

Figure 6.9: An example of a false positive due to a variant of the double-error-code pattern

Table 6.2 shows the running time and memory usage for each phase. The analysis takes a total of 5 hours to run. Extracting the WPDS alone takes 4 hours 5 minutes. We have not attempted to optimize the LLVM-based front end to make it more efficient. Profiler information reveals that half of the time is spent on printing the WPDS file. Collapsing the WPDS rules takes 39 minutes. This analysis optimization reduces the number of rules from 19,933,777 to 7,998,022. As a result, the analysis itself runs on a significantly smaller WPDS file, taking only 17 minutes. Without collapsing the rules, the analysis runs out of memory on a machine with 192 GB of memory. Solving the dataflow problem and producing diagnostic information is the most memory-intensive

```
1  nsresult inDOMUtils::GetRuleNodeForContent(nsIContent* aContent,
2                                             nsIAtom* aPseudo,
3                                             nsStyleContext** aStyleContext,
4                                             nsRuleNode** aRuleNode) {
5    *aRuleNode = nsnull;
6    *aStyleContext = nsnull;
7
8    if (!aContent->IsElement()) {
9      return NS_ERROR_UNEXPECTED;
10   }
11   ...
12   return NS_OK;
13 }
```

(a) Function GetRuleNodeForContent returns an error

```
1  NS_IMETHODIMP inDOMUtils::GetCSSStyleRules(nsIDOMElement *aElement,
2                                             const nsAString& aPseudo,
3                                             nsISupportsArray **_retval) {
4    ...
5    nsRuleNode* ruleNode = nsnull;
6    ...
7
8    GetRuleNodeForContent(..., &ruleNode);
9    if (!ruleNode) {
10     // This can fail for content nodes that are not in the document or
11     // if the document they're in doesn't have a presshell. Bail out.
12     return NS_OK;
13   }
14   ...
15   return NS_OK;
16 }
```

(b) Function GetCSSStyleRules ignores a returned error but checks ruleNode instead

Figure 6.10: A second example of a false positive due to the double-error-code pattern

```
1  const void* nsRuleNode::ComputeContentData(void* aStartStruct,
2                                              const nsRuleData* aRuleData, ...) {
3    ...
4    // content: [string, url, counter, attr, enum]+, normal, none, inherit
5    const nsCSSValue* contentValue = aRuleData->ValueForContent();
6    switch (contentValue->GetUnit()) {
7    case eCSSUnit_Null:
8      break;
9    case eCSSUnit_Normal:
10   case eCSSUnit_None:
11   case eCSSUnit_Initial:
12     // "normal", "none", and "initial" all mean no content
13     content->AllocateContents(0);
14     break;
15    ...
16   }
17   ...
18 }
```

(a) Function AllocateContents is called with a count of 0

```
1  nsresult nsStyleContent::AllocateContents(PRUint32 aCount) {
2    DELETE_ARRAY_IF(mContents);
3    if (aCount) {
4      mContents = new nsStyleContentData[aCount];
5      if (! mContents) {
6        mContentCount = 0;
7        return NS_ERROR_OUT_OF_MEMORY;
8      }
9    }
10   mContentCount = aCount;
11   return NS_OK;
12 }
```

(b) Function AllocateContents does not return errors if the count is 0

Figure 6.11: An example of a false positive due to met preconditions

Table 6.2: Analysis performance for Firefox

Task	Time (h:mm:ss)	Memory (GB)
Extracting WPDS	4:05:23	28.6
Collapsing rules	0:39:17	6.9
Solving problem	0:17:28	38.7

task (38.7 GB).

6.2 Case Study: SQLite

Our second case study is SQLite. SQLite is a relational database management system library that is extensively used in widely-deployed applications such as Mozilla Firefox, Chrome, Skype, and Dropbox. SQLite is written in C and uses the return-code idiom. Figure 6.12 shows the list of basic error codes used by SQLite. For example, SQLITE_READONLY defines the error used when there is an attempt to write a read-only database. This section presents results for the current official release 3.7.13.

6.2.1 Results

Our tool produced a total of 197 bug reports. We have manually inspected all reports and classified them into three categories: true bugs, harmless dropped errors, and false positives. Table 6.3 summarizes our findings.

We identified 49 potential true bugs. These include 44 unsaved errors, 4 overwritten errors, and 1 out-of-scope error. We found 36 harmless dropped errors. As with Firefox, we divided harmless dropped errors into four groups (see Table 6.3). Finally, 112 reports are false positives. We divided these into five groups. As with Firefox, the most common source of false positives is double error codes (FP1) with 50 out of 112 reports. The second most common source of false positives is due to infeasible paths (FP3) with 34 out of 112 reports. We found two additional kinds of false positives. The first is found when inspecting overwritten error reports. In this case, errors are overwritten while masking them (FP4). The second is found in overwritten and

```
#define SQLITE_ERROR        1   /* SQL error or missing database */
#define SQLITE_INTERNAL     2   /* Internal logic error in SQLite */
#define SQLITE_PERM         3   /* Access permission denied */
#define SQLITE_ABORT        4   /* Callback routine requested an abort */
#define SQLITE_BUSY         5   /* The database file is locked */
#define SQLITE_LOCKED       6   /* A table in the database is locked */
#define SQLITE_NOMEM        7   /* A malloc() failed */
#define SQLITE_READONLY     8   /* Attempt to write a readonly database */
#define SQLITE_INTERRUPT    9   /* Operation terminated by sqlite3_interrupt()*/
#define SQLITE_IOERR       10   /* Some kind of disk I/O error occurred */
#define SQLITE_CORRUPT     11   /* The database disk image is malformed */
#define SQLITE_NOTFOUND    12   /* Unknown opcode in sqlite3_file_control() */
#define SQLITE_FULL        13   /* Insertion failed because database is full */
#define SQLITE_CANTOPEN    14   /* Unable to open the database file */
#define SQLITE_PROTOCOL    15   /* Database lock protocol error */
#define SQLITE_EMPTY       16   /* Database is empty */
#define SQLITE_SCHEMA      17   /* The database schema changed */
#define SQLITE_TOOBIG      18   /* String or BLOB exceeds size limit */
#define SQLITE_CONSTRAINT  19   /* Abort due to constraint violation */
#define SQLITE_MISMATCH    20   /* Data type mismatch */
#define SQLITE_MISUSE      21   /* Library used incorrectly */
#define SQLITE_NOLFS       22   /* Uses OS features not supported on host */
#define SQLITE_AUTH        23   /* Authorization denied */
#define SQLITE_FORMAT      24   /* Auxiliary database format error */
#define SQLITE_RANGE       25   /* 2nd parameter to sqlite3_bind out of range */
#define SQLITE_NOTADB      26   /* File opened that is not a database file */
#define SQLITE_ROW        100   /* sqlite3_step() has another row ready */
#define SQLITE_DONE       101   /* sqlite3_step() has finished executing */
```

Figure 6.12: Basic error codes used in SQLite

out-of-scope error reports (FP5). This arises when an error is overwritten with another (different) error.

6.2.2 Performance

We ran the analysis of SQLite (138,243 lines of code) on a Core i7 3 GHz machine with 192 GB RAM. Table 6.4 shows the running time and memory usage. The analysis takes a total of 3 minutes 39 seconds to run, while using 566 MB of memory. As with our first case study, we give a breakdown of running time and memory using for extracting the WPDS, collapsing rules, and solving the dataflow problem and producing diagnostic information. Again, the most expensive phase is producing the textual WPDS representation.

Table 6.3: Dropped errors in SQLite (preliminary results). The reports are divided into true bugs, harmless dropped errors (H1: dropped in the process of shutting down, H2: dropped in the process of releasing resources, H3: documented by developer to be ignored, and H4: logged), and false positives (FP1: double error code, FP2: met precondition, FP3: infeasible paths, FP4: error masking, and FP5: error hierarchy).

		Harmless Dropped Errors					False Positives						
Bug Category	True Bugs	H1	H2	H3	H4	Total	FP1	FP2	FP3	FP4	FP5	Total	Grand Total
Unsaved	44	6	4	8	7	25	36	4	7	0	0	47	188
Overwritten	4	2	0	1	3	6	3	0	26	2	11	42	52
Out of Scope	1	3	0	0	2	5	11	0	1	0	11	23	29
Grand Total	49	11	4	9	12	36	50	4	34	2	22	112	197

Table 6.4: Analysis performance for SQLite

Task	Time (m:ss)	Memory (MB)
Extracting WPDS	3:14	196
Collapsing rules	0:11	61
Solving problem	0:14	566

6.3 Summary

We applied the error-propagation analysis to find dropped errors in two widely-used user applications: Mozilla Firefox and SQLite. The results show that error handling is not only important and challenging in systems software, but also in user applications. As with systems software, error-propagation bugs are abundant; however, not all of them represent real problems for the application. Developers agree that fixing all dropped errors would have a positive impact on the overall quality of the application. Unfortunately, human resources are limited and developers prefer to focus on the "real" problems. Simply filing all bug reports is not an option, thus determining the impact of dropped errors beforehand is crucial. This is a difficult and effort-demanding task, in particular when one is not familiar with the code base under analysis.

This problem could be alleviated by providing the tool with more fine-grained error-handling specifications. The high-level error-handling specification used when analyzing systems software (error logging) no longer applied to the user applications presented in this chapter. For example, most of the errors in Firefox are logged before they start to propagate, and error logging is not always sufficient.

In both user applications, we found several cases in which program comments indicate that it is OK to drop errors in particular scenarios. It would be ideal to have developers document all similar instances. Comments might not be the best alternative to document harmless dropped errors, but it is at least a good start.

So far, the feedback from developers continues to be positive. We have received a suggestion for our tool to be used during code review, and there is interest in using the tool to analyze patches to determine whether they could introduce new dropped errors.

Chapter 7

Related Work

In this chapter, we describe other work related to the analysis of the propagation of errors, and the different kinds of error-propagation bugs discussed in this dissertation.

7.1 Error Propagation and Dropped Errors

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 [19], but does not trace the flow of errors over extended paths. GCC 3.4 introduced a warn_unused_result annotation for functions whose return values should be checked, but again enforcement is limited to the call itself: storing the result in a variable that is never subsequently used is enough to satisfy GCC. Neither LCLint nor GCC analyzes deeply enough to uncover bugs along extended propagation chains.

It is tempting to blame this problem on C, and argue for structured exception handling instead. Language designs for exception management have been under consideration for decades [23, 46]. 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 all checked exceptions be either caught or explicitly declared as thrown is controversial [64, 67]. 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. [57] found that 90% of relevant exceptions thrown by .NET assemblies (C# libraries) are undocumented. Thus, while exceptions change the error-propagation problem in interesting ways, they certainly do not solve it. Furthermore, widely-used applications written in C++ still use the return-idiom code, not exceptions.

There are numerous proposals for techniques to detect or monitor error-propagation patterns at run time, typically during controlled in-house testing with fault-injection to elicit failures [12, 22, 24, 28–30, 32, 33, 61]. Work by Guo et al. [27] 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. [26] highlight dropped errors in file systems as a special concern. Gunawi's proposed Error Detection and Propagation (EDP) analysis is essentially a type inference over the file system's call graph, classifying functions as generators, propagators, or terminators of error codes. Our approach uses a more precise analysis framework that offers flow- and context-sensitivity. The difference is not merely theoretical: we have compared the two in detail and while Gunawi's EDP finds 97% of our true unsaved errors, it also produces 2.75 times more false positives. Furthermore, EDP finds no overwrites and just one of our true out-of-scope errors. EDP runs relatively faster, producing results in a matter of seconds. However, it does not produce detailed diagnostic information; WPDS witness traces (Section 3.3) offer a level of diagnostic feedback not possible with EDP's whole-function-classification approach.

Bigrigg and Vos [6] 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 millions of lines of kernel code.

The FiSC system of Yang et al. [73] 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 dropped errors and has been applied to only three of Linux's many file systems.

7.2 Errors Masquerading as Pointer Values

Engler et al. [17] infer programmer beliefs from systems code and check for contradictions. They offer six checkers, including a NULL-consistency checker that reveals an error-valued pointer dereference. They also provide an IS_ERR-consistency checker, which reveals that NULL checks are often omitted when checking for errors. We do not infer beliefs. Instead, we track error codes to find what pointer variables may hold them and then report those that are used improperly, including but not limited to pointer dereferences.

Lawall et al. [42] use Coccinelle [52] to find bugs in Linux. Their case study identifies and classifies functions based on their known return values: a valid pointer, NULL, ERR_PTR, or both. The tool reports program points at which inappropriate or insufficient checks are detected. This can reveal some error-valued dereferences. However, dereferences made at functions that cannot be classified by the tool cannot possibly be found, and only 6% of the functions are classified as returning ERR_PTR or both ERR_PTR and NULL. Also, dereferences of error-valued pointers that are never returned by a function or further manipulated cannot be found. Our approach uses an interprocedural flow- and context-sensitive dataflow analysis that allows us to track error-pointer values regardless of their location and whether or not they are transformed.

Although identifying missing or inappropriate checks [17, 42] can lead to finding and fixing potential problems, our tool instead reports the exact program location at which problems might occur due to misuse of error-valued pointers. Our bug reports also help programmers find the program points at which error checks should be added in order to fix the problems reported. These tools aim to find a wider range of bugs; their discovery of missing or inappropriate error checks is only an example case study of a generic capability. Our tool is more specialized: it finds more specific kinds of bugs than Engler et al. [17] and Lawall et al. [42], and is more precise in finding these bugs.

Zhang et al. [74] use type inference to find violations of the principle of complete mediation, such as the requirement that Linux Security Modules authorization must occur before any controlled operation is executed. IS_ERR can be thought of as a mediating check that must appear before any potentially–error-carrying pointer is used. We believe our technique can be adapted to find other mediation violations as well. Our approach can be more precise as it is context-sensitive. Furthermore, we could provide detailed sample traces describing how such violations might occur.

Numerous efforts (e.g., [4, 10, 15, 17, 31, 35, 47, 50, 72]) have focused on finding NULL pointer dereferences using varied approaches. Our problem is a generalization of the NULL dereference problem, where instead of just one invalid pointer value, we are tracking 34 of them. However, our problem is also more complex. Error codes might transform during propagation, which does not occur with NULL pointers. In addition, while dereferencing and using NULL values in pointer arithmetic is as bad as using error values, overwriting NULL is perfectly benign. Overwriting unhandled error values, however, may have serious consequences.

7.3 Undocumented Error Codes

Studies show that programmers value accurate documentation, but neither trust nor maintain the documentation they have [43, 62]. For example, Sacramento et al. [57] found that 90% of relevant exceptions thrown by .NET assemblies (C# libraries) are undocumented. Misleading

documentation can lead to coding errors [65] or even legal liability [34]. 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 [71], even if we do not know which is right and which is wrong, the mere presence of inconsistencies indicates that something is amiss.

Venolia [68] 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. [65] 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 [18, 49, 55, 58, 59]. Berglund and Priestley [5] 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 [9] 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.

Chapter 8

Conclusions and Future Directions

In this dissertation, we applied static program analysis to understand how error codes propagate through software that uses the return-code idiom. We described the main component of our framework: an interprocedural, flow- and context-sensitive static analysis that tracks the propagation of errors, which we formulated and solved using weighted pushdown systems. We showed how we use the error-propagation analysis to find different kinds of error-propagation bugs:

Dropped Errors. We found error-code instances that vanish before proper handling is performed. We learned that unhandled errors are commonly lost when the variable holding the unhandled error value (a) is overwritten with a new value, (b) goes out of scope, or (c) is returned by a function but not saved by the caller. We found 312 confirmed dropped errors in five widely-used Linux file systems, including ext3 and ReiserFS. We also found numerous dropped errors in two user applications: the Mozilla Firefox web browser, and the database management system SQLite. Mozilla Firefox is written in C++, however it also uses the return-code idiom. We have submitted a subset of the bug reports to Firefox developers. Two security vulnerabilities due to dropped errors have been confirmed so far.

Errors Masquerading as Pointers. We found misuses of pointer variables that store error codes. We identified three classes of error-valued pointer bugs in Linux file systems and drivers:

(a) bad pointer dereferences, (b) bad pointer arithmetic, and (c) bad pointer overwrites. We found 56 true bugs among 52 different Linux file systems and 4 device drivers. We found that bad pointer dereferences are the most common error-valued pointer bugs. We ran the analysis on a newer code version, and found that a few reported bugs had been fixed. However, as the code evolves, new bugs are introduced.

Error-Code Mismatches Between Code and Documentation. We considered whether the manual pages that document Linux kernel system calls match the real code's behavior regarding returned error codes. We found the sets of error codes that Linux file-related system calls return and compared these to the Linux manual pages to find errors that are returned to user applications but not documented. We found a total of 1,784 undocumented error-code instances across 52 different Linux file systems and 42 file-related system calls.

In all of the above, bug reports included a trace that illustrates how the problem might arise. In total, our tool has analyzed over 5 million lines of code. Although this work was mainly focused on Linux, the analyses can also be applied to other programs. As an example, we presented results for two additional case studies involving user applications: Mozilla Firefox and SQLite. Additionally, the NASA/JPL Laboratory for Reliable Software has used our tool to check code in the Mars Science Laboratory, where it found a critical bug in code used for space missions. As an interesting side note, the Mars rover Curiosity landed successfully the day this chapter was written.

We identified and addressed multiple technical challenges while developing and applying these static program analyses to real-world applications. For example, performance and scalability became an issue given the size of the systems under analysis. We devised two extremely effective optimizations that allowed the analyses to run 24 times faster (under 5 minutes on average for Linux file systems and drivers), requiring 75% less memory. One of these optimizations consisted of filtering out program variables that cannot possibly contain error codes. Another challenge was to reduce the number of false positives. By manually inspecting bug reports, we found patterns

that described common sources of false positives. We reduced the number of false positives by hundreds once the tool recognized these patterns.

The feedback received from developers has been positive and encouraging:

"Thanks for your efforts!" — Jeff Mahoney (ReiserFS)

"This sounds interesting - please forward them to me." — Steve French (CIFS)

"Thank you for helping to improve JFS!" — David Kleikamp (IBM JFS)

"So that is a nice find." — Jan Harkes (Coda)

"Thank you for looking into this. It's a great idea." — Matthew Wilcox (FS)

"Ew, this [bug] is hard to figure out." — Matthew Wilcox (FS)

"I think this is an excellent way of detecting bugs that happen rarely enough that there are no good reproduction cases, but likely hit users on occasion and are otherwise impossible to diagnose." — Andreas Dilger (ext4)

The unstructured nature of C error reporting creates a significant analysis challenge. Programmer intent is often implicit, and our findings show that current practice (manual inspection and testing) is insufficient. For good or ill, implementing operating systems in C is also part of the status quo, and this is unlikely to change soon. Furthermore, our additional case studies confirmed that error handling is also important and challenging in user applications. The error-propagation bugs described in this dissertation are common and not exclusive to C programs. Furthermore, our analyses can be useful not only in finding and fixing existing problems, but also in preventing the introduction of new bugs as the code evolves.

A challenge still remains. Static program analysis tools have to prove useful in practice to be worth developers' time. Often, the precision of such tools can be improved by incorporating domain-specific knowledge. Unfortunately, finding this knowledge represents a difficult task. Error handling is often not documented, and developers are the only available source when trying to understand, for example, how the program is supposed to recover from errors in a particular scenario. This problem becomes even more challenging when analyzing large code bases: there

might be hundreds or thousands of developers spread across the world, and it is likely that no one is familiar with the entire code base.

This dissertation uncovers several potential future directions to make static program analysis more appealing in practice. We need to invest more time developing techniques that infer program domain-specific knowledge automatically. The goal is to use this knowledge to improve the precision of static analysis tools: producing not only fewer reports, but the reports that describe the most relevant bugs. For example, the majority of the dropped errors found by our tool describe real dropped errors, however developers do not find them equally critical. Ideally, we could have a tool that learns facts from the code under analysis itself, or at least accepts feedback on the most recently produced reports to decide what to focus on when re-analyzing a program, or when analyzing a different version.

Another future direction is to develop techniques to automate the process of inspecting bug reports produced by static program analysis tools. Such techniques should find similarities between bug reports and classify the results accordingly. That would reduce the time spent in inspecting bug reports significantly while allowing relevant problems to be found faster. Last but not least, effort could also be spent in proposing language extensions that provide developers with better and more effective ways to encode error handling in existing applications without the need to rewrite them entirely. At least, a mechanism should be proposed to easily document error-handling code.

In this dissertation, we described the use of static analysis to find error-propagation bugs in widely-used software, in particular system software. Our results show that static analysis is an effective way to find bugs that rarely occur (and as a consequence are difficult to reproduce), but when they do occur they can have catastrophic consequences. Analyses such as those we described here can go a long way toward improving not only system software reliability, but user applications too. Eliminating error-propagation bugs increases the trustworthiness of computer systems as a whole.

References

[1] Beware: 10 common web application security risks. Technical Report 11756, Security Advisor Portal, January 2003.

[2] Acharya, Mithun, and Tao Xie. Mining API error-handling specifications from source code. In Chechik, Marsha, and Martin Wirsing, editors, *FASE*, volume 5503 of *Lecture Notes in Computer Science*, pages 370–384. Springer, 2009. ISBN 978-3-642-00592-3.

[3] Adams, Bram, and Kris De Schutter. An aspect for idiom-based exception handling: (using local continuation join points, join point properties, annotations and type parameters). In Bergmans, Lodewijk, Johan Brichau, Erik Ernst, and Kris Gybels, editors, *SPLAT*, volume 217 of *ACM International Conference Proceeding Series*, page 1. ACM, 2007.

[4] Babic, Domagoj, and Alan J. Hu. Calysto: scalable and precise extended static checking. In Schäfer, Wilhelm, Matthew B. Dwyer, and Volker Gruhn, editors, *ICSE*, pages 211–220. ACM, 2008. ISBN 978-1-60558-079-1.

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

[6] Bigrigg, Michael W., and Jacob 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.

[7] Bruntink, Magiel, Arie van Deursen, and Tom Tourwé. Discovering faults in idiom-based exception handling. In Osterweil, Leon J., H. Dieter Rombach, and Mary Lou Soffa, editors, *ICSE*, pages 242–251. ACM, 2006. ISBN 1-59593-375-1.

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

[9] Buse, Raymond P. L., and Westley Weimer. Automatic documentation inference for exceptions. In Ryder and Zeller [56], pages 273–282. ISBN 978-1-60558-050-0.

[10] Bush, William R., Jonathan D. Pincus, and David J. Sielaff. A static analyzer for finding dynamic programming errors. In *Softw., Pract. Exper.*, 30(7):775–802, 2000.

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

[12] Candea, George, Mauricio Delgado, Michael Chen, and Armondo 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.

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

[14] Dilger, Andreas. Error propagation bugs in ext4. Personal communication, November 2008.

[15] Dillig, Isil, Thomas Dillig, and Alex Aiken. Static error detection using semantic inconsistency inference. In Ferrante, Jeanne, and Kathryn S. McKinley, editors, *PLDI*, pages 435–445. ACM, 2007. ISBN 978-1-59593-633-2.

[16] Dowson, Mark. The ariane 5 software failure. In *SIGSOFT Softw. Eng. Notes*, 22(2):84, 1997. ISSN 0163-5948. doi: http://doi.acm.org/10.1145/251880.251992.

[17] Engler, Dawson R., David Yu Chen, and Andy Chou. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *SOSP*, pages 57–72, 2001.

[18] Etzkorn, Letha H., William E. Hughes Jr., and Carl G. Davis. Automated reusability quality analysis of OO legacy software. In *Information & Software Technology*, 43(5):295–308, 2001.

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

[20] Filho, Fernando Castor, Nélio Cacho, Eduardo Figueiredo, Raquel Maranhão, Alessandro Garcia, and Cecília M. F. Rubira. Exceptions and aspects: the devil is in the details. In Young, Michal, and Premkumar T. Devanbu, editors, *SIGSOFT FSE*, pages 152–162. ACM, 2006. ISBN 1-59593-468-5.

[21] Filho, Fernando Castor, Alessandro Garcia, and Cecília M. F. Rubira. Extracting error handling to aspects: A cookbook. In *ICSM*, pages 134–143. IEEE, 2007.

[22] Flanagan, Cormac A., and Michael Burrows. System and method for dynamically detecting unchecked error condition values in computer programs. United States Patent #6,378,081 B1, April 2002.

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

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

[25] Groce, Alex D. Problem solved. Personal communication, January 2009.

[26] Gunawi, Haryadi S., Cindy Rubio-González, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Ben Liblit. EIO: Error handling is occasionally correct. In *6th USENIX Conference on File and Storage Technologies (FAST '08)*, San Jose, California, February 2008.

[27] Guo, Philip J., Jeff H. Perkins, Stephen McCamant, and Michael D. Ernst. Dynamic inference of abstract types. In Pollock, Lori L., and Mauro Pezzè, editors, *ISSTA*, pages 255–265. ACM, 2006.

[28] Hiller, Martin, Arshad Jhumka, and Neeraj Suri. An approach for analysing the propagation of data errors in software. In *DSN*, pages 161–172. IEEE Computer Society, 2001.

[29] Hiller, Martin, Arshad Jhumka, and Neeraj Suri. Propane: an environment for examining the propagation of errors in software. In *ISSTA*, pages 81–85, 2002.

[30] Hiller, Martin, Arshad Jhumka, and Neeraj Suri. Epic: Profiling the propagation and effect of data errors in software. In *IEEE Trans. Computers*, 53(5):512–530, 2004.

[31] Hovemeyer, David, and William Pugh. Finding more null pointer bugs, but not too many. In Das, Manuvir, and Dan Grossman, editors, *PASTE*, pages 9–14. ACM, 2007. ISBN 978-1-59593-595-3.

[32] Jhumka, Arshad, Martin Hiller, and Neeraj Suri. Assessing inter-modular error propagation in distributed software. In *SRDS*, pages 152–161. IEEE Computer Society, 2001.

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

[34] Kaner, Cem. Liability for defective documentation. In Jones, Susan B., and David G. Novick, editors, *SIGDOC*, pages 192–197. ACM, 2003. ISBN 1-58113-696-X.

[35] Karthik, S., and H. G. Jayakumar. Static analysis: C code error checking for reliable and secure programming. In Ardil, Cemal, editor, *IEC (Prague)*, pages 434–439. Enformatika, Çanakkale, Turkey, 2005. ISBN 975-98458-6-5.

[36] Kelley, Al, and Ira Pohl. *A book on C (4th ed.): programming in C.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1998. ISBN 0-201-18399-4.

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

[38] Lal, Akash, Thomas W. Reps, and Gogul Balakrishnan. Extended weighted pushdown systems. In Etessami, Kousha, and Sriram K. Rajamani, editors, *CAV*, volume 3576 of *Lecture Notes in Computer Science*, pages 434–448. Springer, 2005.

[39] Lal, Akash, Nicholas Kidd, Thomas W. Reps, and Tayssir Touili. Abstract error projection. In Nielson, Hanne Riis, and Gilberto Filé, editors, *SAS*, volume 4634 of *Lecture Notes in Computer Science*, pages 200–217. Springer, 2007.

[40] Lal, Akash, Tayssir Touili, Nicholas Kidd, and Thomas Reps. Interprocedural analysis of concurrent programs under a context bound. Technical Report 1598, University of Wisconsin–Madison, July 2007.

[41] Lattner, Chris, and Vikram S. Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *CGO*, pages 75–88. IEEE Computer Society, 2004. ISBN 0-7695-2102-9.

[42] Lawall, Julia L., Julien Brunel, Nicolas Palix, René Rydhof Hansen, Henrik Stuart, and Gilles Muller. WYSIWIB: A declarative approach to finding API protocols and bugs in Linux code. In *DSN*, pages 43–52. IEEE, 2009.

[43] Lethbridge, Timothy, Janice Singer, and Andrew Forward. How software engineers use documentation: The state of the practice. In *IEEE Software*, 20(6):35–39, 2003.

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

[45] Lippert, Martin, and Cristina Videira Lopes. A study on exception detection and handling using aspect-oriented programming. In *ICSE*, pages 418–427, 2000.

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

[47] Loginov, Alexey, Eran Yahav, Satish Chandra, Stephen Fink, Noam Rinetzky, and Mangala Gowri Nanda. Verifying dereference safety via expanding-scope analysis. In Ryder and Zeller [56], pages 213–224. ISBN 978-1-60558-050-0.

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

[49] Mount, S. N. I., Robert M. Newman, R. J. Low, and A. Mycroft. Exstatic: a generic static checker applied to documentation systems. In Tilley, Scott R., and Shihong Huang, editors, *SIGDOC*, pages 52–57. ACM, 2004. ISBN 1-58113-809-1.

[50] Nanda, Mangala Gowri, and Saurabh Sinha. Accurate interprocedural null-dereference analysis for Java. In *ICSE*, pages 133–143. IEEE, 2009. ISBN 978-1-4244-3452-7.

[51] Necula, George C., Scott McPeak, Shree Prakash Rahul, and Westley Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In Horspool, R. Nigel, editor, *CC*, volume 2304 of *Lecture Notes in Computer Science*, pages 213–228. Springer, 2002.

[52] Padioleau, Yoann, Julia L. Lawall, René Rydhof Hansen, and Gilles Muller. Documenting and automating collateral evolutions in Linux device drivers. In Sventek, Joseph S., and Steven Hand, editors, *EuroSys*, pages 247–260. ACM, 2008. ISBN 978-1-60558-013-5.

[53] Reps, Thomas W., Stefan Schwoon, Somesh Jha, and David Melski. Weighted pushdown systems and their application to interprocedural dataflow analysis. In *Sci. Comput. Program.*, 58(1-2):206–263, 2005.

[54] Robillard, Martin P., and Gail C. Murphy. Regaining control of exception handling. Technical report, University of British Columbia, Vancouver, BC, Canada, 1999.

[55] Robles, Gregorio, JesÃžs M. González Barahona, and Juan Luis Prieto Martínez. Assessing and evaluating documentation in libre software projects. In Wasserman, Tony, and Murugan

Pal, editors, *Workshop on Evaluation Frameworks for Open Source Software (EFOSS)*, Como, Italy, June 2006. International Federation for Information Processing.

[56] Ryder, Barbara G., and Andreas Zeller, editors. *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2008, Seattle, WA, USA, July 20-24, 2008*, 2008. ACM. ISBN 978-1-60558-050-0.

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

[58] Schönberg, Christian, Franz Weitl, Mirjana Jaksic, and Burkhard Freitag. Logic-based verification of technical documentation. In Borghoff, Uwe M., and Boris Chidlovskii, editors, *ACM Symposium on Document Engineering*, pages 251–252. ACM, 2009. ISBN 978-1-60558-575-8.

[59] Schreck, Daniel, Valentin Dallmeier, and Thomas Zimmermann. How documentation evolves over time. In Penta, Massimiliano Di, and Michele Lanza, editors, *IWPSE*, pages 4–10. ACM, 2007. ISBN 978-1-59593-722-3.

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

[61] Shin, Kang G., and Tein-Hsiang Lin. Modeling and measurement of error propagation in a multimodule computing system. In *IEEE Trans. Computers*, 37(9):1053–1066, 1988.

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

[63] Sinha, Saurabh, and Mary Jean Harrold. Analysis and testing of programs with exception-handling constructs. In *IEEE Transactions on Software Engineering*, 26:849–871, 2000.

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

[65] Tan, Lin, Ding Yuan, Gopal Krishna, and Yuanyuan Zhou. /*icomment: bugs or bad comments?*/. In Bressoud, Thomas C., and M. Frans Kaashoek, editors, *SOSP*, pages 145–158. ACM, 2007. ISBN 978-1-59593-591-5.

[66] Toy, W.N. Fault-tolerant design of local ess processors. In *Proceedings of IEEE*, pages 1126–1145. IEEE Computer Society, 1982.

[67] van Dooren, Marko, and Eric Steegmans. Combining the robustness of checked exceptions with the flexibility of unchecked exceptions using anchored exception declarations. In Johnson, Ralph, and Richard P. Gabriel, editors, *OOPSLA*, pages 455–471. ACM, 2005.

[68] Venolia, Gina. Textual allusions to artifacts in software-related repositories. In Diehl, Stephan, Harald Gall, and Ahmed E. Hassan, editors, *MSR*, pages 151–154. ACM, 2006. ISBN 1-59593-397-2.

[69] Wegman, Mark N., and F. Kenneth Zadeck. Constant propagation with conditional branches. In *POPL*, pages 291–299, 1985.

[70] Weimer, Westley, and George C. Necula. Finding and preventing run-time error handling mistakes. In Vlissides, John M., and Douglas C. Schmidt, editors, *OOPSLA*, pages 419–431. ACM, 2004. ISBN 1-58113-831-8.

[71] Xie, Yichen, and Dawson R. Engler. Using redundancies to find errors. In *IEEE Trans. Software Eng.*, 29(10):915–928, 2003.

[72] Xie, Yichen, Andy Chou, and Dawson R. Engler. Archer: using symbolic, path-sensitive analysis to detect memory access errors. In *ESEC / SIGSOFT FSE*, pages 327–336. ACM, 2003.

[73] Yang, Junfeng, Paul Twohey, Dawson R. Engler, and Madanlal Musuvathi. Using model checking to find serious file system errors. In *ACM Trans. Comput. Syst.*, 24(4):393–423, 2006.

[74] Zhang, Xiaolan, Antony Edwards, and Trent Jaeger. Using CQUAL for static analysis of authorization hook placement. In Boneh, Dan, editor, *USENIX Security Symposium*, pages 33–48. USENIX, 2002. ISBN 1-931971-00-5.