Improving the Quality of Error-Handling Code in Systems Software using Function-Local Information

Suman Saha

THÈSE DE DOCTORAT DE
l'UNIVERSITÉ PIERRE ET MARIE CURIE

Spécialité

Informatique

École doctorale Informatique, Télécommunications et Électronique (Paris)

Suman SAHA

Pour obtenir le grade de
DOCTEUR de l'UNIVERSITÉ PIERRE ET MARIE CURIE

Sujet de la thèse :

Improving the Quality of Error-Handling Code in Systems Software using Function-Local Information

soutenue le 25 mars 2013

M. Gilles MULLER	Directeur de thèse
Mme. Julia LAWALL	Directeur de thèse
Mme. Sandrine BLAZY	Rapporteur
M. Laurent REVÉILLÈRE	Rapporteur
M. Olaf SPINCZYK	Examinateur
M. Yannis SMARAGDAKIS	Examinateur
M. Fabrice KORDON	Examinateur

Acknowledgements

First and foremost, I would like to express my gratitude to my both advisors Gilles Muller and Julia Lawall, for their essential guidances and insight throughout my graduate career. Their supports and encouragements throughout the thesis have been very helpful, and without our many fruitful discussions the result would surely not have been as good.

I would also like to thank my thesis jury. Both of the thesis reporters, Mme. Sandrine Blazy and M. Laurent Revéillère have spent their valuable time to improve the thesis quality. I am really thankful to them.

My sincere and warm thanks go to all my colleagues at Regal group in LIP6 Lab for their inspirations and helpful comments during my thesis. I especially thank to Gaël Thomas, Jean-Pierre Lozi, Brice Berna and Lokesh Gidra for devoting some of their precious time to help me at the some points of my work.

Special thanks to my friend, Mahfuza Farooque for encouraging and supporting me at every stage during my studies.

Needless to say, my family deserve a great deal of credit for my development. I thank my parents and younger brother for all their love and supports.

Finally, I would like to thank God for giving me ability to do this work.

Abstract

Adequate error-handling code is essential to the reliability of any systems software. On an error, such code is responsible for releasing acquired resources to restore the system to a viable state. Omitting such operations leads not only to memory leaks, but also to system crashes and deadlocks.

The C language does not provide any abstractions for exception handling or other forms of error handling, leaving programmers to devise their own conventions for detecting and handling errors. The Linux coding style guidelines suggest placing error handling code at the end of each function, where it can be reached by `goto`s whenever an error is detected. This coding style has the advantage of putting all of the error-handling code in one place, which eases understanding and maintenance, and reduces code duplication. Nevertheless, this coding style is not always applied. In the first part of the thesis, we propose an automatic program transformation that transforms error-handling code into this style. We have implemented this algorithm as a tool and have applied this tool to five directories (`drivers`, `fs`, `net`, `arch`, and `sound`) in Linux 3.6 kernel source code as well as to five widely used open-source systems software projects: PostgreSQL, Apache, Wine, Python, and PHP. This tool successfully converts 22% of the conditionals containing state-restoring error-handling code that have the scope to merge code into one, from the basic strategy to the `goto`-based strategy.

Even when error handling code is structured according to the Linux coding style guidelines, the management of the releasing of allocated resources remains a continual problem in ensuring the robustness of systems software. Finding such faults is very challenging due to the difficulty of systematically reproducing system errors and the diversity of system resources and their associated resource release operations. To address these issues, over 10 years of research has focused on macroscopic approaches that globally scan a code base for common resource-release operations. Such approaches are notorious for their high rates of false positives, while at the same time, in practice, they leave many faults undetected.

In the second part of the thesis, we propose a novel microscopic approach to finding resource-release faults in systems software, taking into account such software's diversity of resource types and resource-release operations. Rather than generalizing from the results of a complete scan of the source code, our approach achieves precision and scalability by focusing on the error-handling code of each function. Using a tool, Hector, that we have developed based on this approach, we have found 485 faults in 19 different C systems software projects, including Linux, Python, and Apache, with a false positive rate of 23%, well below the 30% that has been reported to be acceptable to developers. Some of these faults are exploitable by an unprivileged malicious user, making it possible to crash the entire system.

Contents

Chapter 1

Introduction

Contents

Any computing system may encounter errors, such as inappropriate requests from supported applications, or unexpected behavior from malfunctioning or misconfigured hardware. If the system's software, such as its operating system, programming-language runtime, or web server, does not recover from these errors correctly, they may lead to more serious failures such as a crash or a vulnerability to an attack by a malicious user. Therefore, correct error recovery is essential when a system supports long-running or critical services. Indeed, the ability to recover from errors has long been viewed as a cornerstone of system reliability [55], and much of systems code is concerned with error detection and handling. For example, 48% of Linux 2.6.34 driver code is found in functions that handle at least one error.[1]

Systems code is written in C, which unlike more modern programming languages such as Java, does not provide any specific abstractions for resource management or error-handling code. Error handling code is responsible for detecting the failure of an operation, releasing allocated resources to restore the system to a consistent state, and returning an appropriate error indicator to the calling function. Any operation that may fail must thus be followed by a conditional statement that checks for an error value and performs the appropriate operations.

Figure 5.3 shows a typical example of error handling code. At the beginning of the code excerpt, there are three conditional statements on lines 5, 9 and 14 that check different conditions. In each case, if an error is detected, the conditional branch first calls `unlock_kernel` and then returns an error indicator. The error-handling operations free data structures of various complexity, and omitting any of this code when constructing any new error-handling code that becomes needed as the function evolves will lead to memory leaks.

[1] Linux 2.6.34 was released in 2010. We focus on a version from a few years ago to prevent our contributions to the Linux kernel from the early stages of our development of Hector from interfering with our results.

```
1  {
2   ...
3   lock_kernel();
4   ...
5   if (!autofs_oz_mode(sbi)) {
6     unlock_kernel();
7     return -EACCES;
8   }
9   if (autofs_hash_lookup(dh, &dentry->d_name)) {
10    unlock_kernel();
11    return -EEXIST;
12  }
13  n = find_first_zero_bit(sbi->symlink_bitmap,AUTOFS_MAX_SYMLINKS);
14  if (n >= AUTOFS_MAX_SYMLINKS) {
15    unlock_kernel();
16    return -ENOSPC;
17  }
18  ...
19  d_instantiate(dentry, inode);
20  unlock_kernel();
21  return 0;
22 }
```

Figure 1.1: Error handling code (Linux-2.6.34/fs/autofs/root.c)

1.1 Refactoring Programming Code

A typical strategy for implementing error handling code in systems software is as shown in Figure 5.3. The strategy is to follow each operation that may encounter an error by a conditional that checks for an error result and, if one is found, performs the appropriate cleanup operations before returning from the function. We refer to this strategy as the *basic strategy*. The basic strategy, however, is error-prone, as it is easy to overlook some cleanup operations that are required, and to forget to update some existing error handling code when the function is extended with new operations that need to be undone in an error case. Furthermore, there may be substantial code duplication, as the same error handling code may be needed at many places within a function definition.

To illustrate the these issues, consider again Figure 5.3. All three error-handling conditional call the same function, `unlock_kernel`. If the protocol for using this function changes, then the code has to be adjusted in each case. Moreover, neglecting to call `unlock_kernel` is some error-handling code will potentially lead to a deadlock if the corresponding error occurs. If other resource allocations are added to the function and the error-handling code is not updated properly, there will be other kinds of resource leaks.

One style of programming that can somewhat alleviate these difficulties is to move the state-restoring operations from the individual error handling conditionals to a single labelled sequence of state-restoring operations at the end of the function. We refer to this style of programming as the *`goto`-based strategy*. In the `goto`-based strategy, each error-handling conditional only performs the operations that are specific to the identified error condition, such as printing a log message or recording an error indicator in a local variable. It then performs a `goto` that jumps to the correct position within a sequence of stat-restoring operations at the end of the function This approach localizes all of the state-restoring operations into one easily identifiable place. If the function definition is extended in a way that there are new possible error conditions, the associated error handling code only needs to jump to the right place within this sequence. If new state-changing operations are added within the function definition, the corresponding state-restoring operations only need to be added at one place

within this sequence. And finally, the duplication of code is mostly limited to the introduction of the `goto`, regardless of the complexity of the error handling process.

We consider how this code would be written if it used the `goto`-based strategy. In Figure 5.3, we observe that all of the error-handling code calls the same function, `unlock_kernel`. We also observe that there is another call of function, `unlock_kernel` on line 20 at the end of the function. Thus, the programmer can use this call instead of writing the same call in each block of error-handling code, by adding a label just before line 20 and a jump from each block of error-handling code to the label. This transformation, however, is not sufficient to obtain a correct implementation. One of the difficulties of converting the code in Figure 5.3 to use the `goto`-based strategy is that all three blocks of error-handling code return different error indicators. Therefore, it is also necessary to assign all error indicators to a single variable. Then, it becomes possible to merge all the blocks of error-handling code into one.

The improved version of the example in Figure 5.3 is shown in Figure 5.4. This version declares a variable `ret` on line 3 at the beginning of the function. Then, it uses this variable to store the error indicators on lines 7, 11 and 16, in each block of error-handling code. The example also uses this variable `ret` to store the function return value `0`, in the non error case, on line 20. A new label `out` is added on line 22 and `goto`s are added within the blocks of error-handling code to jump to the new label. Finally, the the variable `ret` is returned on line 24.

```
1  {
2   ...
3   int ret;
4   lock_kernel();
5   ...
6   if (!autofs_oz_mode(sbi)) {
7     ret = -EACCES;
8     goto out;
9   }
10  if (autofs_hash_lookup(dh, &dentry->d_name)) {
11    ret = -EEXIST;
12    goto out;
13  }
14  n = find_first_zero_bit(sbi->symlink_bitmap,AUTOFS_MAX_SYMLINKS);
15  if (n >= AUTOFS_MAX_SYMLINKS) {
16    ret = -ENOSPC;
17    goto out;
18  }
19  ...
20  ret = 0;
21  d_instantiate(dentry, inode);
22  out:
23    unlock_kernel();
24    return ret;
25  }
```

Figure 1.2: Improved version of Example 5.3

Currently, many functions in systems software use the `goto`-based strategy. This strategy is also recommended by the Linux kernel documentation.[2] Nevertheless, a large number of functions still use the basic strategy, and a number of bugs have been found in such code. For example, in the bug-fixing patches applied to Linux 2.6.20 after its release, we have found that in around a third (12/32) of those whose only effect is to add a call to `kfree` or to an unlocking function such as `spin_unlock`, the

[2]Linux-2.6.34/Documentation/CodingStyle, Chapter 7.

bug is in code that uses the basic strategy. Most other 32 bugs were not in error-handling code. We found similar results (6/20) in the set of patches contributing to Linux 2.6.34.[3] Such bugs can persist undetected for a long time, when the handled error only rarely occurs.

To improve the structure of error handling code in system software, our first contribution is an algorithm to transform error handling code implemented according to the basic strategy so that it follows the `goto`-based strategy. This algorithm merges the state-restoring code found in each conditional into a sequence of state-restoring operations at the end of the function. We have implemented this algorithm as a tool and have applied this tool to five directories (`drivers`, `fs`, `net`, `arch`, and `sound`) in Linux 3.6 kernel source code as well as to five widely used open-source systems software projects: PostgreSQL, Apache, Wine, Python, and PHP. This tool successfully converts 22% of the conditionals containing state-restoring error-handling code from the basic strategy to the `goto`-based strategy.

1.2 Improving the Quality of Error-Handling Code

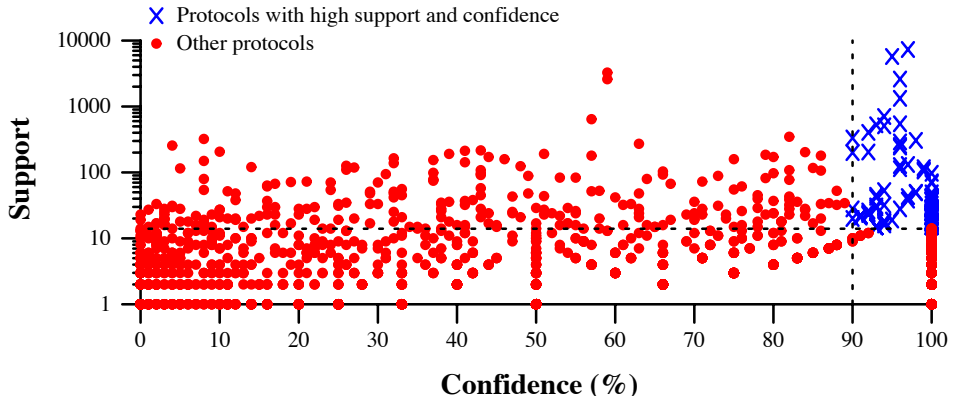
Even when error handling code is structured according to the Linux coding style guidelines, the management of the releasing of allocated resources remains a continual problem in ensuring the robustness of systems software [67].

A critical part of recovering from an error is to release any resources that the error has made incoherent or unnecessary. Omitting a needed resource release can lead to crashes, deadlocks, and resource leaks. Resource-release omission faults are a particular instance of the general problem of checking that API usage protocols are respected, that has received substantial attention [19, 43, 67, 83]. A challenge, however, is to identify the resource-release operations that are required. Indeed, systems code manipulates many different types of resources, each associated with their own dedicated operations, making it difficult for any given developer to be familiar with all of them. Furthermore, the protocol for releasing a given type of resource can vary from one subsystem to another, and can even vary within a single function, depending on the resource's state.

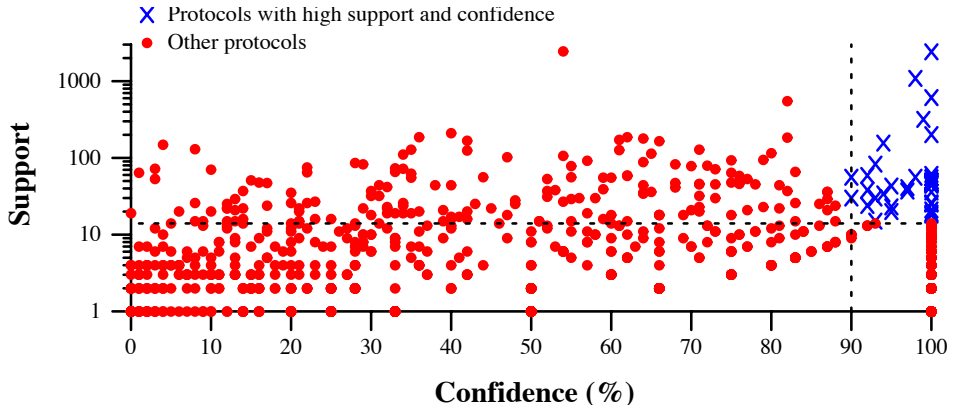
In the context of the general problem of checking API usage, a number of works have proposed to complement fault-finding tools with a preliminary phase of *specification mining* to find sets of operations that should occur together in the code [8, 27, 31, 43, 46, 49, 62, 82, 84, 90]. These approaches follow a *macroscopic* strategy, identifying common sets of operations by a global scan of the entire code base or a sufficiently large execution history. In practice, however, such global scans result in many false positives [44], which in turn lead to many false positives among the found faults. To reduce the rate of false positives, specification-mining approaches typically limit the reported results to the most frequently occurring operations. The resulting specifications, however, are insufficient to find resource-release omission faults involving rarely used functions, which are typical of systems code.

Specification mining approaches detect sets or sequences of functions that are commonly used together and that are expected to represent the required *protocol* for carrying out a particular task. Such approaches typically suffer from a high rate of false positives [44], and thus use some form of pruning and ranking to make the most likely specifications the most apparent to the user. Common metrics include *support* and *confidence*, or variants thereof [49, 62, 82, 84, 90], such as the z-ranking used by Engler *et al.* [27]. Support is the number of times the protocol is followed across the code

[3] Linux 2.6.20 patches obtained from git://git.kernel.org/pub/scm/linux/kernel/git/stable/linux-2.6.20.y.git using the command "git log -p v2.6.20..". Linux 2.6.34 patches obtained from git://git.kernel.org/pub/scm/linux/kernel/git/next/linux-next.git using the command "git log -p v2.6.33..v2.6.34".



(a) Linux



(b) Other Software (Wine, PostgreSQL, Appache, Python, and PHP)

Figure 1.3: Support and confidence of the identified protocols

base, while confidence is the percentage of occurrences of a portion of the protocol that satisfy the complete protocol. The specification-mining tool PR-Miner, for example, which has been applied to Linux code [46], has been evaluated with thresholds causing it to prune fault reports where the associated protocol does not have support of at least 15 and confidence of at least 90%.

Using the heuristics that we will present in Chapter 4 for identifying related resource acquisition and release functions, we identify 2747 potential protocols in Linux, and 1051 in the other considered software (Wine, PostgreSQL, Appache, Python, and PHP). Figure. 5.5 shows the support and confidence of each, as determined by an intraprocedural analysis. Each dot or $\times$ in this figure represents one or more protocols with the same support and confidence values. For Linux, only 3% of the protocols have both support of 15 or more and confidence of 90% or more. 88% have support below 15 and 58% have confidence below 90%. For the other software, only 3% of the protocols have both support of 15 or more and confidence of 90% or more. 81% have support below 15 and 68% have confidence below 90%. The distributions are thus quite similar at both the kernel and user level. Faults in the usage of almost all of these protocols would be overlooked in a specification mining approach using these thresholds. Lowering the thresholds could significantly increase the number of false positives. There is thus a need for a fault-detection approach that can find faults in the usage of protocols having lower support and confidence.

In this thesis, we propose an alternative approach that targets specifically the properties of error-

handling code (EHC) in C systems software. We observe that *when one block of error-handling code needs a given resource release operation, nearby error-handling code typically needs the same operation.* Based on this observation, we propose a *microscopic* resource-release omission fault finding algorithm, based on a mostly intraprocedural, flow and path-sensitive analysis, that targets and exploits the properties of error-handling code. Our algorithm is resistant to false positives in the set of resource acquisition and release operations, resulting in a low rate of false positives in the fault reports, and is highly scalable. It finds resource-release omission faults *irrespective* of the number of times the associated acquisition and release operations are used together across the code base, and is independent of the strategy for identifying them. It focuses on whether a resource release is needed, based on information found in the same function, and is not led astray by information derived from other parts of the system. As a proof of concept, we provide an implementation, Hector,[4] that uses heuristics and mostly intraprocedural analysis to identify resource-related operations. Hector *does not require any fixed or user-provided list of resource-release operations* and *does not depend on the most frequent results obtained by a global scan*, but still achieves a low rate of false positives.

The main contributions of our work are:

- We highlight the fact that resource-release omission faults in error-handling code are an important problem, that may lead to crashes, resource unavailability, and memory exhaustion. Much error-handling code is rarely executed, making faults hard to find by testing.
- We show that existing tools for finding faults in systems code are unlikely to find many of these faults due to these tools' reliance on the frequency of function uses to reduce the number of false positives.
- We propose a resource-release omission fault detecting algorithm based on the observation that patterns of code found within a single function can provide insight into the requirements on the rest of the code within the same function. The applicability of the approach is illustrated by the fact that in the considered systems software, up to 43% of the code is in functions that contain multiple blocks of error-handling code.
- Using Hector, we find 485 resource-release omission faults in 19 systems software, with a false positive rate of only 23%.
- Among 485 faults, 371 resource-release omission faults in the widely used systems software Linux, PHP, Python, Apache, Wine, and PostgreSQL. 52% of the found 371 faults involve pairs of resource acquisitions and releases that are used together in the code fewer than 15 times, making the associated faults unlikely to be detected by previous specification-mining based approaches. We have submitted patches based on many of our results to the developers of the concerned software, and these patches have been accepted or are awaiting evaluation.
- We find that 257 of the 285 faults found in Linux cause memory leaks, while 9 can lead to deadlocks.
- Many of the faults in Linux we find are in initialization code, affecting *e.g.*, the installation of a hotpluggable device or the mounting of a file system. Others are found in more frequently executed functions, such as IOCTL functions or read/write functions. The faults detected by Hector in error-handling code can in practice cause system crashes and device unavailability,

[4]The first three letters of "Hector" are a permutation of "EHC."

when the system has an inconsistent view of the state of a resource. They can also lead to deadlocks and to memory leaks, which can cause the system to completely run out of memory, if the faulty code can be iterated. These conditions can make the system vulnerable to malicious attacks.

1.3 Outline of the thesis

The overall goal of this dissertation is to improve error handling in systems code. We have divided the work into two parts. The first part focuses on improving the structure of error handling code with the goal of helping to reduce the number of faults that may occur in error handling code in future. The second part focuses on finding existing faults in the error handling code of systems software.

This manuscript is organized as follows:

- **Chapter 2: Background.** This chapter is composed of several sections. First, it briefly describes different types of bugs and our targeted bugs. Second, it describes several terminologies that are used in our approaches. Third, it briefly describes of the systems software that we use in evaluating our proposed tools, and then assesses the current status of the error handling code in the considered software in the fourth section. Fifth section describes the state of the art and compares the state of the art with our work. The final section also describes my contributions to a paper on surveying the Faults in Linux that was published in [ASPLOS11].
- **Chapter 3: Improving Structure of Error Handling Code.** This chapter first presents an algorithm to improving the structure of error handling code in system software and then describes the implementation of that algorithm. *This work was published in [LCTES11].*
- **Chapter 4: Finding Faults in Error-Handling Code.** This chapter presents an algorithm to find resource-release omission faults in error-handling code and the implementation of this algorithm in the tool Hector. *A preliminary version of this work was published in [PLOS11, Operating System Review (OSR'11)]. This work was published in [DSN13].*
- **Chapter 5: Conclusion and Future Work.** To conclude the thesis, we provide an overview of the lessons learned from our work. We also provide some directions for future work in this area.

International conferences

[ASPLOS11] Nicolas Palix, Gaël Thomas, Suman Saha, Christophe Calvès, Julia Lawall and Gilles Muller. Faults in Linux: ten years later. In *16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'11).*, CA, USA, March 2011.

[LCTES11] Suman Saha, Julia Lawall, and Gilles Muller. An Approach to Improving the Structure if Error-Handling Code in the Linux Kernel. In *the ACM SIGPLAN/SIGBED Conference on Language, Compilers, Tools and Theory for Embedded Systems (LCTES'11).*, Chicago, USA, April 2011.

[DSN13] Suman Saha, Jean-Pierre Lozi, Gaël Thomas, Julia Lawall, and Gilles Muller. Hector: Detecting Resource-Release Omission Faults in Error-Handling Code for Systems Software. In *the 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), Budapest, June 2013*.

International workshop/Poster

[EuroSys12] Suman Saha, Julia Lawall, and Gilles Muller. Elicitor: Usage-Frequency Independent Detection of Resource-Release Omission Faults. In *the 7th EuroSys, poster session (EuroSys'12)*, Bern, Switzerland, April 2012.

[PLOS11] Suman Saha, Julia Lawall, and Gilles Muller. Finding Resource-Release Omission Faults in Linux. In *the 6th Workshop on Programming Languages and Operating Systems (PLOS'11)*, Portugal, October 2011. Also appeared in *OSR*, 2011.

Chapter 2

Background

Contents

The goal of this chapter is to give some background related to the thesis including the state of the art. First, we describe the types of bugs that occur in systems software and the kind of bugs that we target. We have used different programming analysis terminologies in our approaches. We briefly describe those terminologies. Third, we present a brief description of the systems software that we use in evaluating our tools, and then assess the amount of error handling code in the considered software, over a number of versions. Finally, we present the state of the art and compare the state of the art with our work.

2.1 Bugs in Systems Software

Bugs are a major cause of system failures. Indeed, more than 40% of system failures have been found to be caused by software bugs [52], making detecting bugs one of the most important research areas. *Lu et al.* [50] classify software bugs into three categories, based on the different challenges the bugs expose to detection tools:

- *Memory-related bugs*, caused by improper handling of memory objects. Malicious attackers often exploit this kind of bugs to launch security attacks. There are several types of memory-related bugs, such as
  - *Buffer overflows*, that crosses the buffer's boundary and overwrites adjacent memory,
  - *Stack smashings*, that overwrites the function return address,
  - *Memory leaks*, occur when memory is allocated but can never be freed
  - *Uninitialized reads*, occur when a memory is read before initialization and
  - *Double frees*, occur when there is an attempt to free the same memory location twice.
- *Concurrency bugs*, caused by ill-synchronized operations from multiple threads. Concurrency bugs can be further divided into following categories:
  - *Data-races*, occur when two different threads in a given program can simultaneously access a shared variable, with at least one of the accesses being a write operation,
  - *Atomicity violations*, occur when programmer fail to enclose memory accesses that should be performed atomically inside the same critical region and
  - *Deadlocks*, occur when two or more processes each wait for the others to finish, and thus none of them ever does.
- *Semantic bugs*, that are inconsistent with the original design and the programmers' intention.

Semantic bugs are particularly difficult to detect because they violate the *program semantics*, rather than the *language semantics*. Thus, in order to detect this kind of bug, a tool needs to understand the program's behavior. A general-purpose bug detecting tool is thus unlikely to be able to find many semantic bugs in a system. One common type of semantic bug is a *resource-release omission* bug, i.e., a bug that is related to resources that are allocated but not released before returning from a function. These semantic bugs are often found in error-handling code. We have targeted this type of bugs in this thesis.

2.2 Terminology

This section provides overview of some programming analysis terminology that is used in presenting work to improve the error-handling code in systems software.

Data-flow analysis Data-flow analysis is a technique to compute values at program points in a control flow graph and gather all program analysis information from paths in the graph. Data-flow analysis can be *forward* or *backward*. Forward analysis propagates information from the beginning of a given path towards the end of the path, while backward data-flow analysis is opposite, starting at end of a path and propagating information backwards towards the starting point of the path.

Intraprocedural and Interprocedural analysis Intraprocedural data-flow analysis computes some information based on the code of a single procedure. Interprocedural data-flow analysis uses calling relationships among the procedures to pass information from one procedure to another, taking into account the whole control flow graph of a program. Interprocedural analysis gives more precise compute values than intraprocedural analysis. However, Interprocedural analysis is complex in a large system.

Reaching definition Reaching definitions is a data-flow analysis which statically determines that definitions may reach a given point in the code. A reaching definition of a variable `x` is `d` at a given program point `p` iff the path between program point at definition, `d` of variable `x` and the program point `p` does not have any intervening assignment that can change the value of `x`. Therefore, the variable is known to have the same value at both points. Reaching definition analysis is a forward data-flow analysis.

Live variable analysis A variable is considered as live if it holds a value that may be required for subsequent operation. Live variable analysis (or simply liveness analysis) is a backward data-flow analysis. The variable `x` is live at the point `p`, if the value of `x` at `p` could be used along some path in the control flow graph, starting at `p`; otherwise, `x` is dead at `p`.

Flow-sensitivity and Flow-insensitivity A flow-sensitive analysis takes into account the control-flow graph of a procedure in order to compute values at each point of a procedure. Such an analysis considers the order of statements in a procedure and gives precise values for each statement. In contrast, a flow-insensitive analysis determines a single value for the whole procedure. For example, a flow-insensitive pointer alias analysis may determine that "variables `x` and `y` may refer to the same location", while a flow-sensitive analysis may determine "after statement `s1`, variables `x` and `y` may refer to the same location".

Path-sensitivity and Path-insensitivity A path-sensitive analysis considers the values of condition expressions when analyzing conditional statements and loops to compute precise values at each path of the control-flow graph. For instance, if a branch contains a condition `x>0`, then on the fall-through path, the analysis would assume that `x<=0` and on the target of the branch it would assume that `x>0` indeed holds.

Constant Propagation Constant propagation analysis is to substituting the values of variables at each point of a procedure. Reaching definition analysis result is used to implement constant propagation. If a variable's all reaching definitions are the same assignment which assigns a same constant to the variable, then the variable has a constant value and can be replaced with the constant.

Alias analysis Alias analysis is an analysis that determines whether a storage location can be accessed in more that one way. Two pointers are said to be aliased if they point to the same location.

2.3 Considered Software

In order to evaluate our approach we consider a variety of open source software projects. Specifically, we consider three kinds of software, to enable answering of the following three questions:

- Do the approaches scale to very large systems?
- Are the approaches effective on different type of systems, such as programming languages, operating systems and web servers?
- Are the approaches generic enough to be applicable to any software written in the C language?

To answer the first question we choose Linux kernel, one of the largest free and open source software projects. Specifically, we focus on the Linux directories are listed in Table 2.1 (1 to 4) which have been found to be the most fault prone [27, 67]. To answer the second question we then choose several widely used applications, Table 2.1 (number 5 to 10), to evaluate how our approaches work on different type of applications. Finally, to answer the third question, we select 13 software projects from the Debian Linux distribution, focusing on large software projects written in C, Table 2.1 (number 11 to 23).

Our second contribution only detects faults when a function contains multiple blocks of error-handling code, at least one of which contains a resource-release operation. We refer to such functions as analyzable. The projects selected from Debian all contain at least 1500 C functions, of which at least 10% are analyzable. We give a brief description of each systems software and summarize of all of them in Table 2.1.

Table 2.1: Considered software

	Project	(Lines of code)	Version	Description
1	Linux `drivers`	(4.6MLoC)	2.6.34	Linux device drivers
2	Linux `sound`	(0.4MLoC)	2.6.34	Linux sound drivers
3	Linux `net`	(0.4MLoC)	2.6.34	Linux networking
4	Linux `fs`	(0.7MLoC)	2.6.34	Linux file systems
5	Wine	(2.1MLoC)	1.5.0	Windows emulator
6	PostgreSQL	(0.6MLoC)	9.1.3	Database
7	Apache httpd	(0.1MLoC)	2.4.1	HTTP server
8	Python	(0.4MLoC)	2.7.3	Python runtime
9	Python	(0.3MLoC)	3.2.3	Python runtime
10	PHP	(0.6MLoC)	5.4.0	PHP runtime
11	Samba4	(496KLoC)	4.0.0	implements the SMB protocol
12	ALSA-driver	(474KLoC)	1.0.23	Sound drivers
13	wise	(276KLoC)	2.4.1	installation packages for Windows
14	libvirt	(224KLoC)	0.9.4	Toolkit for interacting with Linux virtualization
15	GlusterFS	(193KLoC)	3.2.3	NAS file system
16	QuteCom	(188KLoC)	2.2.1	Softphone
17	wpa_supplicant	(160KLoC)	0.7.3	IEEE 802.11i authentification
18	RedHat	(151KLoC)	3.0.12	High availability clustering
19	hostapd	(134KLoC)	0.7.3	IEEE 802.11i authentification
20	LibEtPan	(94KLoC)	1.0	A mail library
21	FreeRADIUS	(82KLoC)	2.1.10	A RADIUS server
22	ALSA-lib	(76KLoC)	1.0.24.1	ALSA user-level sound library
23	IPsec	(62KLoC)	0.8.0	Internet protocol security library

Both considered versions of Python are in current use.

Linux The Linux kernel represents one of the most prominent examples of free and open-source software collaboration. Linux is widely used in numerous domains, from small devices to large systems and is becoming dominant on server platforms.

Wine Wine allows applications that are designed for Microsoft Windows to run on Unix operating systems. A survey in 2007 shows that 31.5% Linux desktop users use Wine to run Windows applications [1].

PostgreSQL PostgreSQL is an object-relational database management system. It is available in many well-known platforms including Linux, Microsoft Windows and Max OS X. Hundreds of companies have built products, solutions, web sites and tools using PostgreSQL, which is one of the world's most advanced open source database systems.

Apache httpd The Apache HTTP Server is open source web server. It is the most popular HTTP server today, used by 54.98% of all active websites [2]. Apache httpd is available for a wide variety of operating systems, including Linux, Microsoft Windows and Max OS X.

Python Python is an interpreted high-level programming language. Many large organizations like Google, Yahoo and NASA use this language.

PHP PHP is a general-purpose scripting language that is well suited for server-side web development. It is the most-used open source software in enterprises [4]. Many well-known web applications such as Drupal, Joomla, MediaWiki, Facebook and Wordpress use PHP.

Samba Samba is a re-implementation of the SMB/CIFS networking protocol. It allows file sharing between computers running on different platforms, like Windows and Unix. It is used in most Unix and Unix-like systems, such as Linux, Solaris, AIX and BSD variants.

ALSA-drivers Advanced Linux Sound Architecture (ALSA) drivers is a collection of Linux kernel modules implementing sound drivers. It was designed to use some features that are not supported by the Open Sound System [6].

Wise Wise one of the most widely used installation packages for Windows [6].

libvirt Libvirt is daemon and management tool for managing platform virtualization. It supports many virtualization technologies, including Linux KVM, Xen, and VMware ESX. Libvirt is widely used to implement the Hypervisor in a cloud-based infrastructure.

GlusterFS GlusterFS is a NAS (Network-attached storage) file system. GlusterFS is available in a variety of applications including for cloud computing, streaming media services, and content delivery networks.

QuteCom QuteCom is a open source softphone. It can be used cross-platform (Windows, Linux, Mac OS X) and integrates voice and video calls and instant messaging.

wpa_supplicant wpa_supplicant is a free software implementation of an IEEE 802.11i supplicant for Linux, FreeBSD, NetBSD and Microsoft Windows, with support for WPA and WPA2. It is suitable for both desktop/laptop computers and embedded systems.

RedHat Cluster RedHat cluster is high availability cluster that ensures service availability by monitoring other nodes of the cluster. Red Hat cluster also supports load balancing.

hostapd Hostapd is a user space daemon for wireless access point and authentication servers. There are three implementations: Jouni Malinen's hostapd, OpenBSD's hostapd and Devicescape's hostapd [6]. We use Jouni Malinen's hostapd in this thesis.

LibEtPan LibEtPab is a mail library. It provides portable and efficient framework for different kinds of mail access [6].

FreeRADIUS FreeRADIUS is the most widely deployed RADIUS server in the world. FreeRADIUS supports all the common authentication protocols [6].

ALSA-lib ALSA-lib is used by applications to access the ALSA sound interface.

IPsec Internet Protocol Security (IPsec) is a protocol suite. It has been used for securing Internet Protocol communications. It mainly implemented in the kernel.

2.4 Error handling in Systems Software

To better understand the current state of error handling code in systems software code we have analyzed the source code of the Linux kernel, Python and Wine, over a number of previous versions. Figure 2.1 shows that overall, the number of functions with error handling code is increasing in all of these software projects. Figure 2.2 shows number of functions with error-handling code per line of source code in the systems software.

We have selected 10 Linux versions between Linux-2.0, released in 1996 and the latest version, Linux-3.6, released in 2012. Figure 2.1(a) shows that in most directories the number of functions with error-handling code is increasing steadily, while in the `drivers` directory, the number of such functions is increasing dramatically. In Linux 2.0 there were only around 1500 such functions in the `drivers` directory, in Linux 2.6.0 there were fewer than 12 000 such functions, and in Linux 3.6 there were almost 55 000. This represents an increase of almost 8 times from Linux 2.0 to Linux 2.6.0, during which time the code size increased by only 7 times,[1] and of over 4 times from Linux 2.6.0 to Linux 3.6, during which time the code size increased by only 3 times. Furthermore, in the case of `fs`, the number of error handling functions increased by almost 5 times from Linux 2.0 to Linux 2.6.0 and the code size by 7 times, from Linux 2.6.0 to Linux 2.3.6, the number of error-handling functions grew by more than 2.5 times, while the code size grew by only 2 times. Figure 2.2(a) shows the number of error-handling code per line of code in drivers directory is increasing while that number in *fs* and *net* directories are gradually increasing. The number of error-handling code per line of code in other directories is about constant over the last few versions.

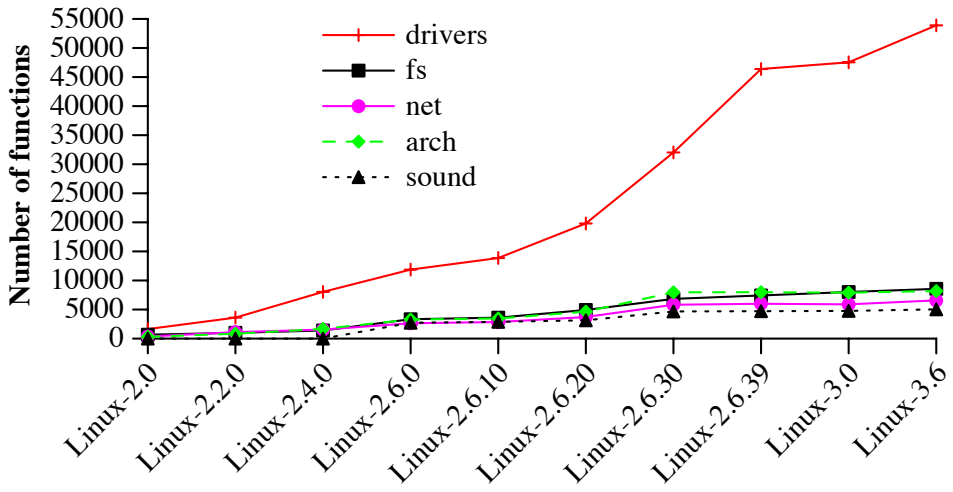
To understand the error-handling code in other systems software we have selected 10 Python versions. The two separate lines in Figure 2.1(b) represents two series of Python versions. We have selected versions between Python-0.9.1, released in 1991 and the latest version of first series, Python 2.7.3, released in 2012 and the latest version of second series, Python 3.3.0, released in 2012. Figure 2.1(b) shows that the number of functions with error handling code has grown 23 times for the first series and 12 times for the second series in Python in the last 21 years. Figure 2.2(b) shows the number of error-handling code per line of code is about constant in different versions of Python.

Figure 2.1(c) shows the number of functions with error handling code in 10 versions of Wine, between version Wine-0.9.22, released in 2006 and the latest version, Wine-1.5.14, released in 2012. The figure shows that the number of functions with error-handling code was around 7000 in Wine-0.9.22. Over the next 6 years, the number if such functions in Wine has doubled. Nevertheless, since

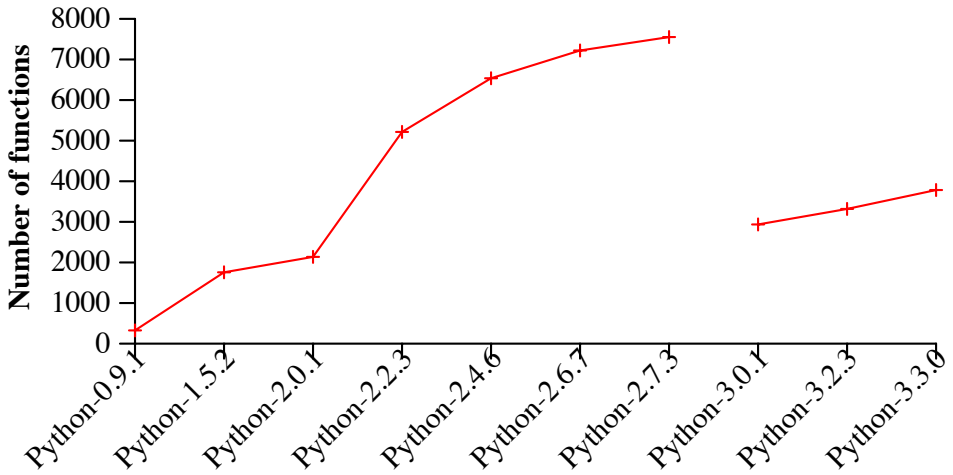
[1] The code size was measured using SLOCCount [87].

the released of Wine-1.3.28 in 2011, the increase has mostly leveled off. Figure 2.2(c) shows the number of error-handling code per line of code is about constant in different versions of Wine.

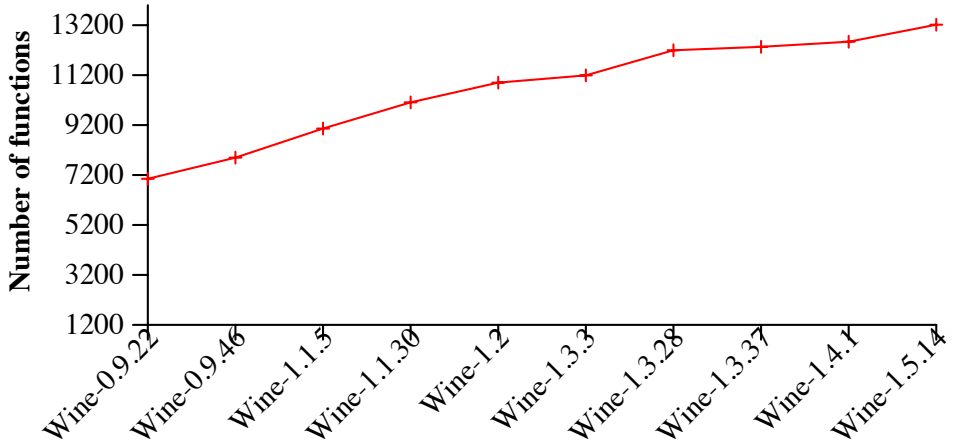
These figures suggest an overall increasing diligence in detecting and handling error conditions, which has probably been facilitated by the increasing use of defect-finding tools [67].



(a) Linux (various directories).

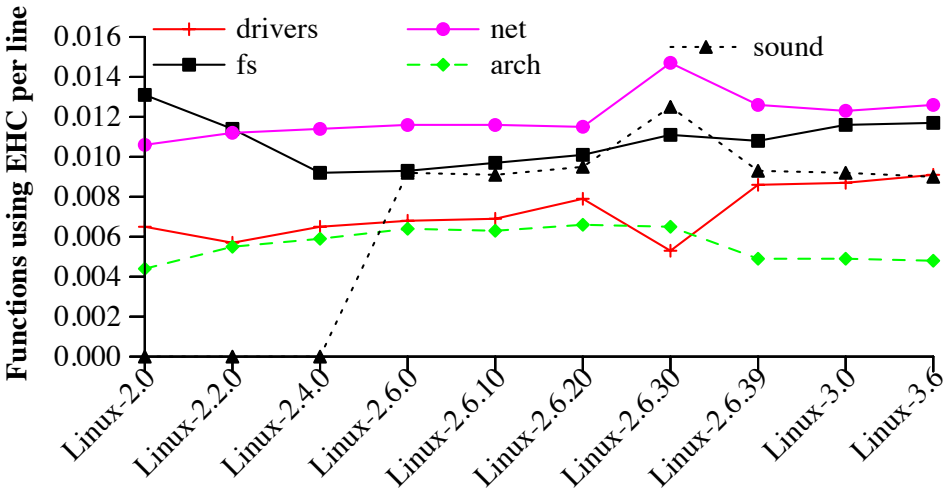


(b) Python

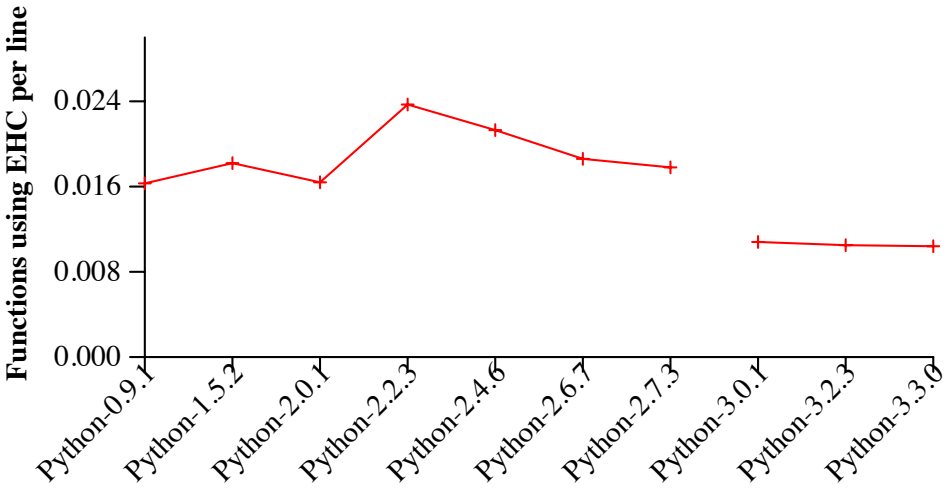


(c) Wine

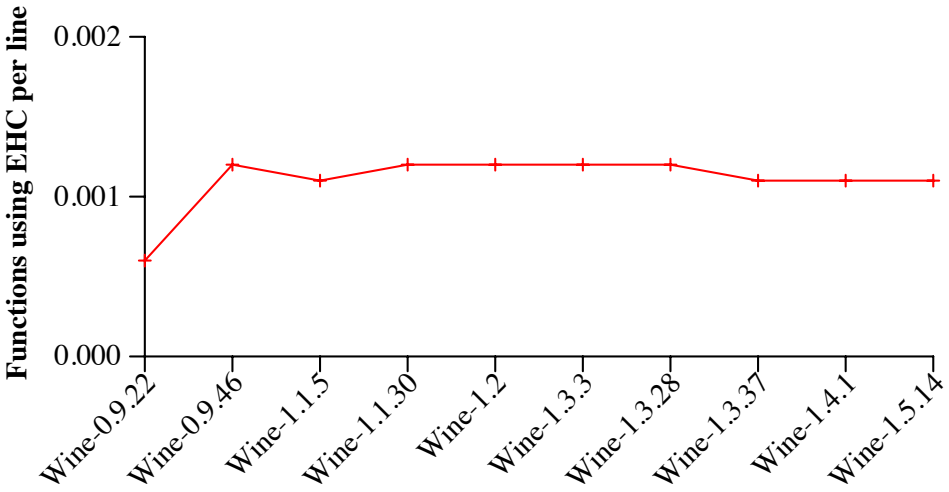
Figure 2.1: Number of functions with error handling code in different versions of systems software.



(a) Linux (various directories).



(b) Python



(c) Wine

Figure 2.2: Number of functions using error handling code (EHC) per line in different versions of systems software.

2.5 State of the Art

The goal of our work is to improve error handling code in systems software by improving the structure of error-handling code and by finding faults in error-handling code. This section briefly describes some existing work related to refactoring C code or to finding faults in systems code.

2.5.1 Refactoring C Code

Our first contribution, the transformation algorithm proposed in Chapter 3, can be considered to be a form of refactoring, since it changes the structure, but not the semantics of the code [30]. Fowler studied how refactoring can make object-oriented code simpler and easier to maintain. The refactoring process may involve moving objects from one class to another, reducing or merging code within a method, pushing some code up or down in an inheritance hierarchy, etc. Few tools, however, support refactoring of C code. Eclipse provides the CDT development environment for C and C++ code, but the support for refactoring seems to be incomplete [16].

Aspect-oriented programming is a paradigm for writing programs in a modularized way [81]. This programming paradigm breaks down the program logic into distinct *aspects* that are used to enhance the behavior of certain kinds of operations of a base program. Aspects enable the modularization of concerns such as transaction management that cut across multiple types and objects. Filho *et al.* present a technique to transform the exception handling code of a Java program into an aspect [29], providing modularity and reuse. Lippert *et al.* also present an approach to transform code related to exception handling in Java into an aspect to reduce the amount of code related to exception handling [47]. Their studies show that the use of aspect-oriented programming provides better support for incremental development, increases the reusability of the components and better support for different configurations of exceptional behaviors of the systems. Their approach can drastically reduce the code related to exception handling, by a factor of 4. Mortensen and Ghosh apply aspects to convert code that uses return codes, as done in Linux, to use C++ exception handling abstractions, to ensure that all exceptions are handled [57]. Bruntink performs a similar study on C code, using hypothetical `try` and `catch` constructs [13]. C does not provide any such abstractions.

McCloskey and Brewer propose the tool Asfact for refactoring C code [54]. As part of this work, they observe that the use of C preprocessor code (CPP) in C programs makes the code difficult to analyze and is often error-prone. The misuse of CPP may furthermore introduce bugs in a program. They have studied a number of bugs related to CPP, including (1) *Unparenthesized body*, which may trigger precedence errors that could cause the code generated by the macro to be interpreted improperly, (2) *Unparenthesized formal*, which likewise occurs when formal arguments are not surrounded by parentheses, (3) *Multiple formal uses*, which can duplicate computation and (4) *Dangling semicolon*. CPP code cannot be parsed with a standard parser. Therefore, to replace the CPP macro language, they define a new syntactic macro language that addresses the most important deficiencies of preprocessors and that eliminates many of the errors that the use of the macro language may introduce. They then provide an approach that automatically translates CPP macros, include directives, and conditional directives into semantically equivalent declarations in the proposed language. Their approach thus completely eliminates the C preprocessor from the refactoring process. It permits complex transformations to be applied directly to source code, without an initial preprocessing step. They have evaluated their approach on open-source software packages such as OpenSSH and Linux. They have also applied Asfact to the program gzip to evaluate the refactoring process. Gzip contains several

known overflows involving the `strcpy` function. Asfact found the buffer overflow vulnerabilities in Gzip and transformed all such occurrences of `strcpy` to use of `strlcpy`, a safer version.

Our approach to refactoring error-handling code, presented in Chapter 3, does not provide any alternative approach to writing error-handling code nor does it propose new programming language features. Our work addresses the issues of readability and uniformity, which the above studies have identified as critical. The proposed improvement to exception handling stays within the constructs available in C. It uses the widely recommended *goto-based* strategy to improve the structure of error-handling code.

There is some contention in the developer community about the benefit of using `gotos`. The Motor Industry Software Reliability Association (MISRA) is an organization that produces guidelines for the software developed for electronic components used in the Automotive Industry. They have published guidelines for the C programming language [56]. The use of the *goto-based* strategy to write error handling code is not allowed in MISRA C. Indeed, the use of `gotos` is considered to indicate badly constructed and incomprehensible logic, making testing difficult. Despite this, the use of *goto* has several benefits, as we describe in Chapter 3. Moreover, the Linux coding style guidelines suggest placing error handling code at the end of each function, where it can be reached by `gotos` whenever an error is detected.

2.5.2 Finding Faults in Source Code

In this section, we describe several approaches to identify faults in systems source code. First, we describe some static analysis based works that identify specifications in a large system and then use those specifications to find bugs in the systems source code. Second, we describe few dynamic analysis based approaches that find faults at the runtime of the program. Finally, we briefly describe the works that inject faults either into the systems source code or into the systems binary code to expose the faults in the systems source code.

2.5.2.1 Static Analysis

Static analysis is an analysis that is performed at compile time. The advantage of static analysis is that it analyzes all possible execution paths and variable values.

Numerous approaches have been proposed to detect the omission of certain operations in systems code. One well known technique is to use some form of data mining to extract implicit programming rules from the software source code and then to use static analysis to detect faults based on those programing rules. Engler et al.. [27] and Li et al. [46] both propose variations of this approach. Ramanathan et al. [69] integrate mining within a path-sensitive dataflow framework to define potential preconditions of a procedure. Le Goues and Weimer [44] integrate extra information about nonfunctional code characteristics, such as churn and author expertise. We present a few of the variations of this approach that have been proposed [27, 46, 51, 62, 69, 44, 88].

Engler et al. [27] use static analysis to automatically extract programming rules from source code without prior knowledge of the system. They infer program rules using hypotheses such as *a dereference of a pointer, p, implies a belief that p is non-null or a call to "unlock(l)" implies that l was locked.* They then use a statistical analysis to rank each error by its confidence from most to least likely. Essentially, the highest ranked errors will be those detected by rules with the most examples and fewest counter-examples. They consider, for example, that if a particular programming pattern is observed

in 999 out of 1000 cases, then it is a probably a valid rule, while if the pattern happens only once, it is probably a coincidence. They write six checker templates according to the extracted programing rules. They then use those templates to find contradictions of the inferred rules in the source code. Any contradiction implies the existence of a possible error in the code. Ranking calculated in terms of support and confidence is used to highlight the most probable rules. The support of a rule is the number of times the protocol is followed across the code base, while confidence is the percentage of occurrences of a portion of the protocol that satisfy the complete protocol. The approach can also use "must beliefs" derived from the user's knowledge of the semantics of the code, rather than statistics. They find hundreds of bugs in real systems such as Linux and OpenBSD. Moreover, they have used one of the checkers on Linux and OpenBSD to find security errors and found 35 security holes. Such security holes can give a malicious party control of the system.

Li et al. [46] propose a method called PR-Miner that uses a data mining technique called frequent itemset mining to efficiently extract implicit programming rules from the source code of large software. Frequent itemset mining is a technique to find frequently occurring subsets of the set of items in a large database. To use frequent itemset mining on code, PR-Miner hashes each program element found in a function definition into number, and then writes the set of these numbers as a row into the itemset database. Benefiting from frequent itemset mining, PR-Miner can extract programming rules in general forms without being constrained by any fixed rule templates and thus it can find rules that can contain program elements of various types, such as functions, variables and data types. Like the work of Engler et al. [27], the hypothesis behind PR-Miner is that the programming rules usually hold for most cases and violations happen only occasionally. Based on the found programming rules, PR-Miner can find bugs by detecting violations to these rules. PR-Miner then prunes the false violations using inter-procedural analysis. PR-Miner may result in false positives if the elements in a programming rule span across multiple functions. After PR-Miner detects rule violations and prunes false positives, it ranks all remaining violations based on the support and confidence of the rules and reports them to the user. They have evaluated their approach by applying it to large systems software such as Linux, PostgreSQL and Apache, and have extracted thousands of rules from the source code of those systems. T For example, the rules may involve correlations between variables or correlations between variables and functions. They have used the extracted programming rules to find many violations in the evaluated software. PR-Miner ranked the violations according to their confidence. The authors have manually inspected only the top 60 violations for each application. They found 16 bugs in Linux, 6 bugs in PostgreSQL and 1 bug in Apache.

To reduce the rate of false positives, Le Goues and Weimer [44] consider extra information about nonfunctional code characteristics such as churn and author expertise during specification. They rely on a number of hypotheses, including (1) frequently modified code is less likely to adhere to specifications, (2) duplicated code does not represent an independent correctness argument on the part of the developer; if printf follows iter in 10 duplicated code fragments, it is not 10 times as likely that (*iter*,*printf*) is a real specification, (3) more readable code is also more likely to adhere to specifications, and (4) infeasible paths suggest pairs that are not specifications; if in some code, the programmer has made it impossible for b to follow a along a path, (a , b) is unlikely to be required. Using these hypotheses, they present an approach that automatically infers partial correctness specifications by giving less priority to duplicate code, infrequently-tested code, and code that exhibits high turnover in the version control system. Based on these criteria, they automatically identify the code that is likely to associated to program specifications in code base. They have evaluated their approach by applying it to over 800,000 lines of code, and explicitly compare it to two previous approaches. They evaluated two versions of their approach: normal miner and precise miner. They give more relaxed

threshold values for mining specifications in normal miner while they put more precise threshold values in precise miner. Their normal miner finds specifications as well as improves on the false positive rates of previous approaches by 20% for specification mining. Their precise miner finds fewer valid specifications, but it gives only a 5% false positive rate for specification mining. They were able to extract more specification than previous approaches, reducing the number of false positives by a factor of 5. This approach, however, also reduced the number of found faults. Furthermore, statistics are still used, so rarely used resource-release functions may be overlooked.

Ramanathan *et al.* integrate mining within a path-sensitive dataflow framework to identify potential preconditions for invocation of a function [69]. Their work also reduces false positives, by using path-sensitivity, which takes into account both control flow and data flow analysis. The path-sensitivity of the analysis considers the values of condition expressions when analyzing conditional statements and loops to compute precise values at each path of the control-flow graph. The path-sensitivity ensures the precedence relations among procedure calls. A precedence relation is a binary relation between procedures *a* and *b*, which specifies that a call to *b* is always preceded by a call to *a* that means all paths leading to *b* have a call to *a*. Their inter-procedural path-sensitive static analysis performs several steps to extract potential precedence relations among function calls. They first generate the control-flow graph for each function in the program and then simplify the graph by removing the nodes that do not correspond to function calls. The simplified graphs are given to a relation builder that builds relations among the function calls. They apply a simple graph walking strategy to build the relations between function calls on the simplified control-flow graph. The results of the relation builder are given to a sequence miner to mine the common sequences. The output of the sequence miner and the associated violations are ranked by a set of metrics. They have extracted programing rules and identified associated violations from five applications. They found 16 violations in Apache, 133 in Gimp, 105 in Linux, 31 in Openssh and 277 in PostgreSQL.

CP-Miner have also used another variation of mining strategies, called frequent subsequence mining, to extract copy-pasted code and identify bugs in such code in large software, including operating systems [45]. Copy-pasted code leads to bugs when programmers forget to modify terms (variables, functions, type, etc.) consistently throughout the pasted code. Moreover, copy-pasting a segment of code that contains bugs can introduce more bugs in the source code of the system. CP-Miner parses the given source code and stores sequences of code in a database. CP-Miner then uses a frequent sequence mining algorithm called CloSpan [?] to find frequent sequences in the sequence database. CP-Miner considers a sequence as a frequent when it appear in at least a specified number of times. CP-Miner uses several pruning techniques to prune false positives in extracted copy-pasted code such as (1) pruning unmappable segments, (2) pruning tiny segments, (3) pruning overlapped segments and (4) pruning segments with large gaps. CP-Miner then uses a bug detection method to find bugs in the copy-pasted code. The hypothesis behind the bug detection method is that if a programmer changes an identifier in most places of pasted code but forgets to change it in a few places, the unchanged identifier is likely to be a bug. CP-Miner was evaluated on Linux, FreeBSD, Apache and PostgreSQL. It has detected 28 copy-paste related bugs in the Linux, 23 in FreeBSD, 5 in Apache and 2 in PostgreSQL.

A program may have many inherently correlated variables, that must be accessed and updated together to avoid inconsistency in the program. Inconsistency in the values of the correlated variables can lead to a crash or other program misbehavior. Although the consistency of correlated variables is very important, little attention has been paid to this issue in previous work. Moreover, many correlated variables are just semantically correlated and do not necessarily have data dependencies, implying that traditional compilers cannot identify the relationship between them. To address these issues, MUVI detects semantic bugs and concurrency bugs by identifying multi-variable inconsistent updates, in

which only one variable is updated and other correlated variables remain unchanged [51]. MUVI automatically infers common multi-variable access correlations through static program analysis and data mining techniques. Based on the inferred multi-variable correlations, MUVI applies code analysis to detect semantic bugs, in which correlated variables are not updated in a consistent way. Then, in order to detect multi-variable related data races, MUVI extends the two classic race detection methods lock-set and happens-before [25, 60, 72]. The traditional race detectors are all designed to detect single variable races. MUVI's multi-variable extensions to data race detection methods allow it to correctly identify the root causes of previously known bugs as well as to detect new bugs. MUVI automatically identified 6449 multi-variable access correlations with an accuracy of around 83% in four real-world applications: Linux, Mozilla, MySQL, and PostgreSQL. Furthermore, MUVI found 39 new multi-variable inconsistent update bugs in these applications. Almost all of the detected bugs are semantic bugs and cannot be detected by existing memory bug detection tools. MUVI was shown to enable previous data-race detectors to identify correct root causes of four previously known bugs and detected four new multi-variable concurrency bugs in Mozilla. None of these eight bugs can be correctly identified by the original race detectors without MUVI's multi-variable extensions.

GrouMiner [62] presents new approach for mining the usage patterns of objects and classes in a code base. The approach first constructs a directed acyclic graph (DAG) describing the uses of objects and classes. In a graph, nodes represent objects' constructor calls, method calls, field access, and the branching points of control structures, and edges represent temporal usage orders and data dependencies among them. A usage pattern is considered to be a subgraph that frequently appears in the object usage graphs. GrouMiner uses a graph-based algorithm for mining the frequently appearing subgraphs in a graph dataset. In order to provide developers with coding examples, the extracted subgraphs are translated into user-friendly code skeletons. The extracted patterns can also be used to detect usage anomalies. A portion of code is considered to violate a pattern P if the corresponding object usage graph contains only an instance of strict sub-pattern of P, i.e, not all properties of P are satisfied. There are thus two main differences between GrouMiner and previous mining approaches. First, the mined patterns and code skeletons provide more information to assist developers by making apparent the usage flows among objects, including control structures (*e.g.,* conditions, loops, etc). Second, GrouMiner updates the graph datasets of detected patterns and then can find the anomalies in the new version of the system.

Wu *et al.* identify resource acquisition and release operations in Java code by analysis of method definitions [88]. They combine a number of features, such as the frequency of NULL assignments within a method definition and a method's calls to known resource-release methods, to characterize probable acquisition and release functions. The analysis is interprocedural. Their approach takes into account both the source code and API documentation of the library to extract the specifications. In their work, they address three main issues while mining resource-releasing specifications from API libraries. First, relying on propagation based on method-calling relationships (starting from known basic specifications concerning low-level resources) alone is not sufficient to mine precise specifications. For example, a method definition may have several resource-releasing operations but the method might do acquisition actually. Second, information embedded in API method definitions may help to identify the resource-acquiring/releasing operation. For example, comments in API methods may contain important information, so comment analysis might be helpful to identify resource-acquiring/releasing operation. Third, resource-releasing operations exhibit some common features, such as following some naming convention. However, they have found that resource-acquiring operations are unlikely to exhibit any common features. They have applied their tool to eight open source libraries to mine specifications from them. They achieve a high degree of precision and recall, but

report only one fault based on these results.

icomment [77] automatically analyzes comments in source code to extract program rules. The extracted rules are then used to find inconsistencies between the source code and the comments, indicating either bad comments or bugs in the source code. icomment uses several techniques to perform the task, including natural language processing (NPL), machine learning, statistics analysis and program analysis. They use natural language processing to tag each word as "verb", "noun", etc. in comments, and NPL is also used to parse a comment into main clauses, sub-clauses, etc. Statistics techniques are then used to find frequently appearing correlated words in the comments. Machine learning techniques are then used to generate a model from a subset of the comments in one software codebase and then to use that model to analyze other comments in the software. The interesting part of this work is that a model trained using the comments in one software can be used for another software. Finally, program analysis is used to detect inconsistencies between code and comments, rank rules and prune false positives. They evaluated icomment on Linux, Mozilla, Wine and Apache. They extracted 1832 rules from comments with 90.8-100% accuracy and detect 60 inconsistencies between code and comment. Among the 60 inconsistencies, 33 are new bugs and 27 are bad comments. The overall false positive rate of icomment is 38.8%.

Kremenek *et al.* present Annotation Factor Grapps (AFGs), a group of probabilistic graphical models for mining resource acquisition/releasing functions in code base [42]. They gather all program analysis information and incorporate into AFG. They have evaluated their approach effectiveness on five code bases: SDL, OpenSSH, GIMP, and the OS kernels for Linux and Mac OS X (XNU). Lawall *et al.* propose a declarative approach to find API protocols and bugs in Linux [43].

Our tool, Hector, presented in Chapter 4, does not rely on a separate specification mining phase. Instead, it finds faults based on inconsistent local information, rather than a global analysis of the software. Hector can find faults in the use of protocols that occur rarely and thus are likely to be pruned or given a low rank by other approaches. Moreover, Hector does not require any known resource-release functions; this is a particular advantage for Linux, which manages a very wide range of types of resources and which does not rely on standard libraries. The analyses required are furthermore less costly, as interprocedural analysis is limited to a single file.

2.5.2.2 Dynamic Analysis

Dynamic analysis is an analysis that is performed by executing the targeted program. In order to analysis the program behavior the program need to be fed by sufficient number of test inputs. Although the dynamic analysis reveals subtle defect or vulnerabilities whose cause is too complex to be discovered by static analysis, it is often difficult to find test inputs that are sufficient to take into account all possible execution paths.

Dynamic binary analysis (DBA) tools analyze programs at runtime at the level of machine code. DBA tools are often implemented using dynamic binary instrumentation (DBI), whereby the analysis code is added to the original code of the client program at run-time. Valgrind [59] is a DBI framework that only reports on faults in code that is actually executed. Valgrind loads the client program to recompile the client's machine code. The core disassembles the code block into an intermediate representation (IR). IR then is instrumented with the analysis code and the whole code converted back into machine code by the core. The translated code is stored in a code cache to be returned as necessary. The translated code then can be executed to find faults in them.

Purify [34] is a dynamic tool that finds memory access errors at run-time. It inserts a function

call, CATCH_ME instruction, on which the programmer can set a breakpoint, into the object code of a program, before every load and store, to trap every memory access that a program makes. The function called, in conjunction with `malloc` and `free`, maintains a table that keeps the possible states for each byte in the heap and stack. Three states are possible : (1) Unallocated (unwritable and unreadable), (2) Allocated but uninitialized (writable but unreadable) and (3) Allocated and initialized (writable and readable). Purify catches array bounds violations by allocating a *red-zone* at the beginning and end of each block returned by `malloc`. Purify records the bytes in the red-zone as unallocated (unwritable and unreadable) , and thus any access of these bytes would be treated as an array bound error by Purify. In order to catch reads of uninitialized automatic variables, Purify sets the state of the stack frame to the allocated-but-uninitialized state after a function entry. Any inconsistent access of memory causes a diagnostic message to be printed and the function to be called, on which the programmer can set a breakpoint. There are two parts of a garbage collector: a garbage detector and a garbage reclaimer. Purify makes an novel change in garbage detector to achieve some of the benefits of garbage collector. Here, the garbage detector is a subroutine library that can help to identify and fix memory leaks during development. Purify calls garbage detector instead of providing an automatic garbage detector, to identify memory leaks. In order to marks all blocks that are referenced by a particular data, Purify recursively follows potential pointers from the data and stack segments into the heap. Purify then steps through the heap and reports allocated blocks that no longer seem to be referenced by the program. However, Purify and most other dynamic analysis tools incur high run-time overhead. These tools can slow down a program by up to 20 times [93]. Therefore, it is not possible to use this kind of dynamic tool during production runs.

SafeMem [68] proposes a low overhead dynamic analysis tool. It detects buffer overflows and accesses to freed memory, which lead to memory corruption. It uses existing ECC (Error-Correcting Code) protection in a novel way to detect memory corruption. ECC protection is used for error detection and correction when hardware memory errors occur. SafeMem uses padding at both ends of each buffer and ECC protection is used to guard these paddings. Any access to the padding is reported as a buffer overflow bug. SafeMem also detect memory leaks on-the-fly during production runs. In order to develop an approach to detect such kind of bugs the author studied about the life time of Memory object, the period from the allocation of a memory object to its deallocation. Their study shows that most dynamic memory objects conform to some expected lifetime. If any memory objects whose lifetime exceeds the expected maximal lifetime then SafeMem considers the situation as memory leak. Their detection process of memory leaks has three steps: (1) Dynamically analyze the memory usage behavior of the program. The step collects lifetime informations and memory usage information including the number of current live objects, the last allocation time, and the total memory space currently occupied by this memory object group. (2) Detect potential memory leaks based on observed usage characteristics. For example, SafeMem compares current time and the allocation time of a memory to check whether the memory usage is dynamically growing. (3) Use ECC protection to prune false positives. In order to detect accesses to freed memory, SafeMem also uses ECC protection to monitor all free memory buffers and its its lifetime using ECC protections. SafeMem is evaluated on seven applications that contain memory leaks and memory corruption bugs. It detected all tested bugs with low overhead (1.6% - 14.4%).

Baratloo *et al.* propose approach to detect and handle buffer overflow vulnerabilities, more specifically stack smashing attack at run-time [11]. A stack smashing attack occur when a malicious attacker is able to overwrite the return address of a function call. Their approach estimates the the maximum buffer size by realizing that such local buffers cannot extend beyond the end of the current stack frame. The measuring of the buffer maximum size is performed at run-time after the start of the function in

which the buffer is accessed. The maximum size of the buffer limits the buffer writes within the estimated buffer size. Thus, the return address of any function call on the stack can not be overwritten. Their approach also verifies the return address of a function call on the stack before use. They have evaluated their approach by applying on the Linux and found several known attacks. The performance overhead of these libraries range from negligible to 15%.

CRED (C Range Error Detector) is a dynamic buffer overflow detector [71]. CRED first checks the bounds of a pointer, if the address is out-of-bounds (a violation of the ANSI C standard) CRED creates a special OOB (out-of-bounds) object in the heap by calling special `malloc`. CRED replaces all out-of-bounds pointer value with the address of the OOB object and stores the actual pointer value and the information of referent object that the pointer is intended to reference in the OOB object. The OOB object is not recorded as a regular object in the object table, but its address is entered into the OOB hash table. When a pointer is dereferenced, CRED checks if it points to an object in the object table to an unchecked object. If neither is the case, it is an illegal reference and the error message is printed. If a pointer is used in an arithmetic and comparison operation, CRED first checks if it points to an object in the object table or to an uncheck object. If neither is the case, CRED checks the OOB hash table to determine if is an out-of-bounds value. CRED then performs the desired operation on the actual out-of-bound value retrieved from OOB. CRED was evaluated by integrating it into the Jones and Kelly checker that identify any out-of-bounds address for gcc 3.3.1 [38]. CRED was evaluated on over 20 open-source programs, comprising over 1.2 million lines of code. CRED is reported to be effected against a tested of 20 different buffer overflows attacks.

Most dynamic checkers suffer from two limitations [93]. First, they incur high run-time overhead because of their large instrumentation cost. Moreover, dynamic tools do not have accurate information where the code needs to be instrumented. Therefore they may instrument more places than necessary that leads to false positives. Second, most dynamic tools rely on compilers and pre-processing tools to insert instrumentation. Some accesses to a monitored location may be missed by the instrumentation tool due to imperfect variable disambiguation that leads to false negatives. In order to overcome these limitations, Zhou *et al.* propose a a low overhead dynamic tool, iWatcher for software debugging [93]. It provides two system calls: one to start monitoring a memory location and another one to stop monitoring. These calls can be inserted automatically by an instrumentation tool or manually by a programmer. Therefore, iWatcher can associate monitoring function with memory locations. The monitoring function can be program-specific. When an such location is accessed, the associated monitoring function is automatically executed. iWatcher is implemented using a combination of hardware and software support, in particular by changing the architectural support. For example, iWatcher augments L1 and L2 cache lines with flags. There are two flags bits per word in the line. These flags specify what types of accesses to this memory region should be monitored: READONLY, WRITEONLY, or READWRITE. If the read (write)-monitoring bit is set for a word, all loads (stores) to this word automatically trigger the corresponding monitoring function. At a triggering access, the hardware automatically initiates the monitoring function associated with this memory location. iWatcher leverages Thread-Level Speculation (TLS) to reduce overhead for programs with substantial monitoring. TLS is an architectural technique for speculative parallelization of sequential programs. It speeds-up execution by running monitoring-function microthreads in parallel with each other and with the main program. iWatcher is evaluated on applications with various bugs. While the observed overhead is up to 80%, in some cases iWatcher can find bugs with only a 4% overhead

2.5.2.3 Fault Injection

Fault injection is a testing process to cover some particular error-handling code paths by introducing faults that least to the paths. Fault injection can be performed at compile time as well as runtime. The challenging part of this technique is to generate the faults to inject.

Injecting into Source Code In order to inject faults at compile time, the program instruction need to be modified before the program image is loaded and executed. The idea behind this method is to inject errors into the source code or assembly code of the target program to emulate the effect of hardware, software, and transient faults. In this section, we describe few works that inject faults into source code.

A Linux kernel oops occurs when the kernel detects that it an erroneous state. Linux then logs the error and continues its operation with compromised reliability. Yoshimura *et al.* studied the scope of error propagation in the Linux after a kernel oops has occurred [91]. They have classified error propagation into two scopes: (1) *process-local*, if the effect of the error is restricted to the process context that activated it. (1) *kernel-global*, if the error affects the contexts of other processes or global data structures. According to their study, the kernel can recover from error that is process-local while recovery is difficult when the error propagation is kernel-global, as in this case, corrupted global data structures must be stored. They find that the scope of error propagation is mostly process-local, and in case of kernel-global propagation, non-faulty process do not access the inconsistent data. They have used a fault injector [61] on the Linux kernel source code to analyze the scope of error propagation. They have injected 6738 faults into Linux and find that a kernel oops occurs 134 times.

Gu *et al.* used fault injection experiments to study how the Linux kernel responds to transient errors [32]. To generate appropriate workloads, they used UnixBench [5], a benchmark suite to apply workloads to the Linux kernel. Using UnixBench, they were able to profile kernel behavior and identify the most frequently used functions, representing 95% of kernel usage in Linux. Three types of error injection campaigns were conducted on the identified functions: (1) random errors injected in non-branch instructions, (2) random errors injected in conditional branch instructions, and (3) injections to reverse the logic of conditional branches instructions (valid but incorrect conditional branches). More than 35,000 errors were injected into the identified kernel functions within four subsystems: architecture-dependent code (arch), virtual file system interface (fs), the kernel itself, and memory management (mm). Their study show that 95% of system crashes occur because of four major reasons: (1) unable to handle kernel NULL pointers, (2) unable to handle kernel paging requests, (3) invalid opcodes, and (4) general protection faults, e.g., exceeding the segment limit, writing to a read-only code or data segment, or loading a selector with a system descriptor. The overall percentage of error propagation is small (less that 10%). Approximately, 90% of the crashes occur inside the subsystem into which the error was injected. 40% of the crashes occur are within 10 cycles after the injection point. In addition, they studied the severity of crashes. The three categorized level of crashes are: (1) *more severe:* the system needs rebooting and the file system must be completely reformatted, (2) *severe:* the system has to be rebooted and user needs to run a tool to recover the partially corrupted file system, (3) *normal:* the system reboots automatically and the rebooting usually takes less than 4 minutes, depending on the type of machine and the configuration of Linux.

Injecting into Binary Code A fault can be injected into binary code. This technique is more appropriate for users, since it makes it possible to perform Software Fault Injection even when the source code is not available. A important issue concerning injection at binary level is to correctly identify the

programming constructs in a binary program. For example, a function call in the source code may be inlined (i.e. the function call is replaced by the body of called function) by a compiler. In that case, fault injection would not be able to identify the function call, and thus would not be able to inject a *missing function call* fault in that location. One of the most popular Software Fault Injection (SFI) technique for binary code is Generic Software Fault Injection Technique (G-SWFIT) [26].

Cotrineo *et al.* studied the accuracy of fault injection into binary level code [22]. In order to achieve this goal, they have performed two types of fault injection campaigns, one on binary and one on source code and they then compare the results from both campaign for each injection. They have evaluated G-SWIFT by injecting 12 thousand errors into binary code and 18 thousand errors into the corresponding source code. They found three types of results: (1) *Correctly injected faults*, correct faults found in both binary and source code, (2) *Omitted faults*, source code has the fault but not the binary code, and (3) *Spurious faults*, that exists in the binary code but not in the source code. The study shows that several omitted and spurious faults are due to the lack of high-level information in the binary code. In addition their study shows that G-SWIFT can achieve an improved degree of accuracy if the pitfalls are avoided.

LFI (Library Fault Injector) [53] is an automatic tool to generates injection scenarios. and to process of fault scenario preparation and to inject faults LFI does not need library source code and it can work on binary level code. Moreover, LFI presents a simple language to express fault scenarios. The goal of this tool is to make the testing based fault injection technique faster and less human-intensive. LFI has two parts: (1) *a profiler*, that statically analyzes the binary code of the targeted libraries to extract the link information between libraries and the application. For each library, it determines the exported functions, and for each exported function it determines the possible return values. This information is considered as a fault profile. LFI obtains profiles automatically, so testers do not need to be familiar with the internals of the libraries. However, if they have information prior to the generating profiles, they can alter the generate profiles to obtain faster, more accurate results (e.g., by removing functions or faults that are not of interest). Profiler also generats fault injection scenarios. A fault injection scenario describes a sequence of faults to be injected. (2) *a controller* uses the profile information to combine fault profiles with a fault scenario specification to drive the fault injection at the boundary between shared libraries and applications. LFI was evaluated on Linux, Windows and Solaris.

Symbolic execution [15] coupled with fault injection [53], attempts to address these problems by making it possible to activate all execution paths. However, such techniques remain time-consuming, and no form of specification inference is provided. The developer still needs prior knowledge of the various pairs of resource acquisition and release operations.

2.5.3 Improving Error-handling Code

Error-handling is a primary requirement in systems software. Therefore, a language like C that is widely used in systems software must provide fault-free error-handling mechanisms. In this chapter, we first describe some alternative approaches that help to avoid having faults in error-handling mechanism. We then describe few works that find faults in error-handling code.

2.5.3.1 Proposing New Features

Several works propose features to avoid faults in error-handling code. Bruntink *et al.* introduce macros in the exception handling mechanism that help to avoid making mistakes by developers [14]. Weimer

and Necula propose a programming language feature [85]. AMA proposes an approach to pre-allocate the necessary amount of memory for the subsystem before the actual allocation operations are called [75].

Bruntink *et al.* study properties of exception handling in a large industrial C code base [14]. They focus on the error-proneness of exception-handling based on the return code idiom, in which a function returns an error code when exception occurs. They showed that this idiom is omnipresent as well as highly tangled and requires focused and well-thought programming. At first, they studied the different components of the return code idiom to identify which are the most error-prone. The major components are: (1) *Exception Representation*, that defines how an exception is represented, (2) *Exception Raising*, that notifies the raising of an exception, (3) *Handler Determination*, that identifies the exception and determine the associated handler, (4) *Resource cleanup*, that is used to keep the system consistent by releasing allocated resources, and (5) *Exception Interface*, that explicitly specifies the exceptions to raise. According to their study, the most error-prone components are *Exception Raising* and *Handler Determination*. They then built a static analysis tool that detects faults related to those components. They applied their tool to 5 components written in C, developed by ASML, a Dutch company. They found 154 incorrect uses of the return code idiom with a false positive rate of 16%. Their tool found different kinds of faults: (1) function does not return, (2) wrong error variable returned, (3) assigned and logged value mismatch, and (4) unsafe assignment that overwrites the error code in an error variable. Finally, they provided an alternative approach to the return code idiom by introducing macros in the exception handling mechanism that hide some of the implementation details. Using these macros, developers need not write assignments to the error variables explicitly. The macros also help avoid mismatches between the assigned and logged values.

Weimer and Necula present a static data-flow analysis of code including exception-handling code for finding defects in how programs deal with important resources in the presence of exceptional situations [85]. To find defects in they formalize some initial specifications of how a program should acquire and release resources. To find defects in exceptional situations they define a particular fault model to describe what exceptional situations could arise. Their flow-sensitive analysis found over 1300 defects in over 5 million lines of Java code. Their results suggest that improper management of error handling code introduces bugs in a system. They propose a programming language feature, the compensation stack, that keeps track of obligations at run time and ensures that these obligations are discharged. The goal is to help programmers avoid such mistakes.

AMA (Anticipatory Memory Allocation) [75] is an approach to avoid memory allocation failures through a novel combination of static and dynamic techniques. In order to achieve this goal, AMA performs three steps. First, it analyzes all code paths in a Linux kernel subsystem to determine the amount of memory required. Second, based on the results of the first step, the authors manually augmented the kernel code with a call to pre-allocate the necessary amount of memory for the subsystem. Third, AMA redirects all allocation requests to use pre-allocated memory during run-time. This approach ensures that when a memory allocation takes place in the kernel subsystem, it will never fail. AMA was evaluated on the Linux ext2 file system, and for which it was able to avoid memory allocation failures successfully while incurring little space or time overhead.

2.5.3.2 Finding Faults in Error-Handling code

Different types of analysis have been used to detect faults in error-handling code. Some work has focused on error detection and propagation [39, 70], *i.e.*, the correctness of tests and return values, and on fault-injection prioritization with the goal of exercising error recovery code [10].

Static Analysis The return code idiom may require some kind of analysis to find faults in them. Static analysis is one of them. This analysis can examine the code such as error-handling code, that are rarely executed at runtime.

Gunawi *et al.* [33] have studied faults in the detection and propagation of error codes in the context of the return code idiom. They propose *Error Detection and Propagation* (EDP), a static analysis approach that analyzes the flow information of error codes across a code base. EDP creates a function-call graph and does a dataflow analysis on this graph to gather information about propagation of error codes through return values and function parameters. The have detected two types of violations: unsaved and unchecked error codes. An unsaved error code is found when a callee propagates an error code via the return value, but the caller does not save the return value, while an unchecked error code is found when a variable that may contain an error code is neither checked nor used in the future. Gunawi *et al.* have applied EDP to the source code of all file systems and to 3 major storage device drivers (SCSI, IDE, and Software RAID) in Linux. EDP found that 13% of function calls have inconsistently propagated error codes. Rubio-González *et al.* [70] later studied the same problem and proposed a static source code analysis to identify unchecked errors. They categorized the various aspects of the error propagation dataflow problem. First, each failure has its own response. For example an input/output (I/O) error produces an EIO error code while an out of memory error produces an ENOMEM error code. Second, error codes arise in the lower layers of an operating system and propagate upwards through the file system. Third, overwriting a variable that contains an error code before checking the variable's value is a bug. The goal of their work is to detect those instances of an error code that vanish before proper checking is performed. They applied their approach to six file systems and uncover 312 error propagation bugs. Our work is complementary, in that we focus on the contents of blocks of error-handling code, while they focus only on the return values.

Weimer and Necula observed that faults in error-handling code are common in Java and proposed a static analysis, based on a user-provided safety policy, to identify resource-release omission faults [83]. Subsequently, they proposed a specification mining approach that gives more weight to specifications derived from error-handling code [84]. While they found many faults, the specification-mining process has a high rate of false positives.

Fault Injection Banabic and Candea propose a strategy for fault-injection prioritisation to perform run-time checking of error-handling code [10]. The main challenge for the fault injection system is to decide what to inject, where to inject and when to inject. Their technique performs the whole fault injection process automatically, choosing faults, and categorizing the results. Their proposed approach presents a parallel testing system that selects faults to inject by observing the effect of previously injected faults. The process of selecting faults and injecting them continues until a specific target is reached, such as a given level of code coverage, or a time limit. In order to maximize the number of detected bugs in a fixed amount of time they have used a feedback-based algorithm to search for high-impact faults. The main contributions of their work are 1) an algorithm to find high impact faults and 2) techniques for automatic categorization and ranking of faults. They report a handful of faults in MySQL, Apache, and some basic Unix utilities. The reported faults involve omitted tests and duplicated releases.

Error detection and propagation, however, are not enough; a function detecting an error must also undo any of its previous operations that could leave system resources in an inconsistent state. Any *omission* of a resource-release operation in such error-handling code can lead to crashes, deadlocks, and resource leaks. Our approach, presented in Chapter 4, is concerned with find omission of a resource-release operation.

2.6 Survey on Faults in Linux

At the beginning of my PhD, I contributed to the paper *Faults in Linux: Ten Years Later* [67], which was published at ASPLOS 2011. The main goal of this work was to study the different kinds of faults in the latest version of the Linux kernel source code. This work focused on the current quality of the kernel source code. In this work I have contributed to study the different types detected faults and analyze the generated reports to identify the actual faults. My thesis work, *Improving the Quality of Error-Handling Code in Systems Software using Function-Local Information* is motivated by this survey on the quality of Linux source code.

Faults in Linux: Ten Years Later Almost 10 years ago, in 2001, Chou *et al.* published a study of the distribution and lifetime of certain kinds of faults[2] in OS code, focusing mostly on the x86 code in the Linux kernel [19], in versions up to 2.4.1. A major result of their work was that the `drivers` directory contained up to 7 times more of certain kinds of faults than other directories. The ability to collect fault information automatically from such a large code base was revolutionary at the time, and this work has been highly influential. Indeed, their study has been cited over 360 times, according to Google Scholar, and has been followed by the development of a whole series of strategies for automatically finding faults in systems code [7, 46, 74, 78, 86]. The statistics reported by Chou *et al.* have been used for a variety of purposes, including providing evidence that driver code is unreliable [35, 76], and evidence that certain OS subsystems are more reliable than others [24].

Linux, however, has changed substantially since 2001, and thus it is worth examining the continued relevance of Chou *et al.*'s results. In 2001, Linux was a relatively young OS, having first been released only 10 years earlier, and was primarily used by specialists. Today, well-supported Linux distributions are available, targeting servers, embedded systems, and the general public [28, 80]. Linux code is changing rapidly, and only 30% of the Linux 2.6.33 code is more than five years old [20, 21]. Linux now supports 23 architectures, up from 13 in Linux 2001, and the developer base has grown commensurately. The development model has also changed substantially. Until Linux 2.6.0, which was released at the end of 2003, Linux releases were split into stable versions, which were installed by users, and development versions, which accommodated new features. Since Linux 2.6.0 this distinction has disappeared; releases in the 2.6 series occur every three months, and new features are made available whenever they are ready. Finally, a number of fault finding tools have been developed that target Linux code. Patches are regularly submitted for faults found using checkpatch [17], Coccinelle [64], Coverity [23], smatch [78] and sparse [73].

In this work, we transported the experiments of Chou *et al.* to the versions of Linux 2.6, in order to reevaluate their results in the context of the current state of Linux development. Because Chou *et al.*'s fault finding tool and checkers were not released, and their results were released on a local web site but are no longer available, it is impossible to exactly reproduce their results on recent versions of the Linux kernel.[3] To provide a baseline that can be more easily updated as new versions are released, we proposed an experimental protocol based on the open source tools Coccinelle [64], for automatically finding faults in source code, and Herodotos [65], for tracking these faults across multiple versions of a software project. We validated this protocol by replicating Chou *et al.*'s experiments as closely

[2] Chou *et al.* used the terminology "errors." In the software dependability literature [37], however, this term is reserved for incorrect states that occur during execution, rather than faults in the source code, as were investigated by Chou *et al.* and are investigated here.

[3] Chou *et al.*'s work did lead to the development of the commercial tool Coverity, but using it requires signing an agreement not to publish information about its results (`http://scan.coverity.com/policy.html#license`).

as possible on Linux 2.4.1 and then applied our protocol to all versions of Linux 2.6. To ensure the perenity of our work, our tools and results are available in a public archival repository [66].

The contributions of our work are as follows:

- We provided a repeatable methodology for finding faults in Linux code, based on open source tools, and a publicly available archive containing our complete results.
- We showed that the faults kinds considered 10 years ago by Chou *et al.* are still relevant, because such faults are still being introduced and fixed, in both new and existing files. These fault kinds vary in their impact, but we have seen many patches for all of these kinds of faults submitted to the Linux kernel mailing list [48] and have not seen any receive the response that the fault was too trivial to fix.
- We showed that while the rate of introduction of such faults continues to rise, the rate of their elimination is rising slightly faster, resulting in a kernel that is becoming more reliable with respect to these kinds of faults. This is in contrast with previous results for earlier versions of Linux which found that the number of faults was rising with the code size.
- We showed that the rate of the considered fault kinds is falling in the `drivers` directory, which suggests that the work of Chou *et al.* and others has succeeded in directing attention to driver code. The directories `arch` (HAL) and `fs` (file systems) now show a higher fault rate, and thus it may be worthwhile to direct research efforts to the problems of such code.
- We showed that the lifespan of faults in Linux 2.6 is comparable to that observed for previous versions, at slightly under 2 years. Nevertheless, we found that fault kinds that are more likely to have a visible impact during execution have a much shorter average lifespan, of as little as one year.
- Although fault-finding tools are now being used regularly in Linux development, they seem to have only had a small impact on the kinds of faults we consider. Research is thus needed on how such tools can be better integrated into the development process. Our experimental protocol exploited previously collected information about false positives, reducing one of the burdens of tool use, but we proposed that approaches are also needed to automate the fixing of faults, and not just the fault finding process.
- Our study has identified 736 faults in Linux 2.6.33, including RCU faults, some of which have not yet been corrected in the current developer snapshot, `linux-next`. We have submitted a number of patches based on our results.

Chapter 3

Improving Structure of Error Handling Code

Contents

3.1 Summary

The C language does not provide any abstractions for exception handling or other forms of error handling, leaving programmers to devise their own conventions for detecting and handling errors. A typical strategy is to follow each operation that may encounter an error by a conditional that checks for an error result and, if one is found, performs the appropriate cleanup operations before returning

from the function. We refer to this strategy as the *basic strategy*. The basic strategy, however, is error-prone, as it is easy to overlook some cleanup operations that are required, and to forget to update some existing error handling code when the function is extended with new operations that need to be undone in an error case. The error-handling operations free data structures of various complexity, and omitting any of this code when constructing any new error-handling code that becomes needed as the function evolves will lead to memory leaks. Furthermore, there may be substantial code duplication, as the same error handling code may be needed at many places within a function definition.

The Linux coding style guidelines suggest placing error handling code at the end of each function, where it can be reached by `goto`s whenever an error is detected. This coding style has the advantage of putting all of the error-handling code in one place, which eases understanding and maintenance, and reduces code duplication. Nevertheless, this coding style is not always applied. In this chapter, we propose an automatic program transformation that transforms error-handling code into this style. We have applied our transformation to Linux and 6 different C infrastructure software projects, on which it reorganizes the error handling code of over 3000 functions.

3.2 Motivation and Background

In this section, we first illustrate the basic error handling strategy and the `goto`-based error handling strategy, using examples from the Linux kernel source code and Python. We then analyze the frequency of the use of these error-handling strategies in three different systems software projects over a number of versions.

3.2.1 Motivating Examples

Figure 3.1 shows a typical example of error handling code following the basic strategy. Three `if` statements are shown (lines 5, 12, and 21), each checking for a different condition. In each case, if the condition is satisfied, there is a sequence of error handling operations. In two cases (lines 12 and 21), these error handling operations begin by printing a log message specific to the error. This is followed by some new operations, which are then followed by the error handling code from the previous `if`, if any. Each `if` concludes by returning an error indicator that is specific to the error that has occurred (lines 8, 17, and 27). Overall, there is substantial duplication of code. Indeed, the final call to `DPRINT_EXIT` found in each `if` also appears at the normal exit from the function.

Figure 3.2 illustrates a possible reimplementation of this function, using the `goto`-based strategy. The largest sequence of error-handling operations, from the third `if`, has been moved to the end of the function. The `if` branches have each been transformed to perform the operations specific to the given error, namely printing the log message and storing the error indicator in the variable `ret`. Each `if` branch then ends in a `goto` that jumps to the appropriate point in the sequence of error handling operations at the end of the function. This sequence in turn uses `goto` to jump to the original end of the function, to take advantage of the call to `DPRINT_EXIT` that is already available there. The error handling code within the function body is now limited to what is specific to each error case.

To illustrate the full scope of the problem, we next consider an example of error-handling code that is implemented using the basic strategy and that contains memory leaks. The analysis in this chapter does not detect the memory leak. That is addressed in the next chapter but that we argue that the memory leak is less likely to be introduced when the `goto`-based strategy is used.

```
1  static int storvsc_probe(struct device *device) {
2    int ret;
3    ...
4    host_device_ctx->request_pool = kmem_cache_create(...);
5    if (!host_device_ctx->request_pool) {    /*    1    */
6      scsi_host_put(host);
7      DPRINT_EXIT(STORVSC_DRV);
8      return -ENOMEM;
9    }
10   device_info.PortNumber = host->host_no;
11   ret = storvsc_drv_obj->Base.OnDeviceAdd(...);
12   if (ret != 0) {          /*    2    */
13     DPRINT_ERR(STORVSC_DRV, "unable to add ...");
14     kmem_cache_destroy(host_device_ctx->request_pool);
15     scsi_host_put(host);
16     DPRINT_EXIT(STORVSC_DRV);
17     return -1;
18   }
19   ...
20   ret = scsi_add_host(host, device);
21   if (ret != 0) {        /*    3    */
22     DPRINT_ERR(STORVSC_DRV, "unable to add ...");
23     storvsc_drv_obj->Base.OnDeviceRemove(device_obj);
24     kmem_cache_destroy(host_device_ctx->request_pool);
25     scsi_host_put(host);
26     DPRINT_EXIT(STORVSC_DRV);
27     return -1;
28   }
29   scsi_scan_host(host);
30   DPRINT_EXIT(STORVSC_DRV);
31   return ret;
32 }
```

Figure 3.1: Example of the basic error handling strategy (Linux-2.6.34/drivers/staging/hv/storvsc_drv.c)

Figure 3.3 contains an extract of a function from the implementation of Python. This code contains seven blocks of error-handling code, starting on lines 9, 12, 17, 21, 31, 40, and 44. There are two memory leaks among these blocks: release of the resource `it` is omitted in the block on line 18, and release of the resource `bytearray_obj` is omitted in the block on line 45. Both of these bugs have been confirmed by the Python developers.[1] We consider how this code would be written if it used the `goto`-based strategy.

In Figure 3.3, we may observe that the function allocates three resources, `it` (line 8), `bytearray_obj` (line 16), and `item` (line 20), are allocated in the function. These resources are deallocated at different stages in the function. The `decref` of `item` on line 28, the `decref` of `it` on line 39 and the `decref` of `bytearray_obj` on line 46. Therefore, some of the subsequent error-handling blocks do not need to have deallocation operations for some of the resources. To apply the strategy illustrated in Figure 3.2, we should ideally simply add `goto`s that jump to new labels within this error handling code. However, it is also necessary to invert the freeing of `it` and `bytearray_obj` in the blocks of error-handling code, starting on lines 21 and 31, so that we can create a label that only frees `it` before exiting the function. Fortunately the two freeing operations access disjoint data, and so exchanging them is possible. The resulting code, shown in Figure 3.4 has no memory leaks and is resilient to further changes.

[1] http://bugs.python.org/issue13019

```
1  static int storvsc_probe(struct device *device) {
2        int ret;
3        ...
4        host_device_ctx->request_pool = kmem_cache_create(...);
5        if (!host_device_ctx->request_pool) {
6          ret = -ENOMEM;
7          goto out3;
8        }
9        device_info.PortNumber = host->host_no;
10       ret = storvsc_drv_obj->Base.OnDeviceAdd(...);
11       if (ret != 0) {
12         DPRINT_ERR(STORVSC_DRV, "unable to add ...");
13         ret = -1;
14         goto out2;
15       }
16       ...
17       ret = scsi_add_host(host, device);
18       if (ret != 0) {
19         DPRINT_ERR(STORVSC_DRV, "unable to add ...");
20         ret = -1;
21         goto out;
22       }
23       scsi_scan_host(host);
24 out1: DPRINT_EXIT(STORVSC_DRV);
25       return ret;
26 out:  storvsc_drv_obj->Base.OnDeviceRemove(device_obj);
27 out2: kmem_cache_destroy(host_device_ctx->request_pool);
28 out3: scsi_host_put(host);
29       goto out1;
30 }
```

Figure 3.2: Improved version of Figure 3.1

Programmers often extend the definition of functions in the development of a project. Extension may involve allocating new resources and writing new error-handling code. Using the `goto`-based strategy, implementing new error handling code requires simply writing a `goto` to the correct line in this sequence. Adding a new resource allocation requires only adding the corresponding deallocation operation at the correct position in this sequence. In either case, the rest of the function remains correct automatically.

3.2.2 Analysis

To better understand the current state of the error-handling code using basic and `goto`-based strategy in systems software code, we have analyzed the source code of the Linux kernel, which suggests the `goto`-based strategy in its coding style guidelines, and two other kinds of systems software, Python and Wine, which do not provide any suggestions about error-handling code. We have analyzed all three projects over a number of versions. Figure 3.5 shows the number of functions that contain error handling code that use either the basic strategy (red striped bar), the `goto`-based strategy (green/light grey bar), or a mixture of both (blue/dark grey bar). All three figures show that overall, the number of functions that contain error-handling code using the basic, the `goto`-based strategy, and the mix of both strategies is increasing in all of these software projects. Figure 3.6 shows number of functions using different strategies per line of source code in the software projects. Our main concern is those functions, to facilitate their conversion from the basic strategy to the `goto`-based strategy.

We have selected ten Linux versions between Linux-2.0, released in 1996, and a recent version,

```
1  /* python3.2-3.2.2~rc1/Objects/bytearrayobject.c */
2  static PyObject *
3  bytearray_extend(PyByteArrayObject *self, PyObject *arg) {
4    PyObject *it, *item, *bytearray_obj;
5    Py_ssize_t buf_size = 0, len = 0;
6    int value; char *buf;
7    ...
8    it = PyObject_GetIter(arg);
9    if (it == NULL)
10     return NULL;
11   buf_size = _PyObject_LengthHint(arg, 32);
12   if (buf_size == -1) {
13     Py_DECREF(it);
14     return NULL;
15   }
16   bytearray_obj = PyByteArray_FromStringAndSize(NULL, buf_size);
17   if (bytearray_obj == NULL)
18     return NULL;                          /* Omission fault */
19   buf = PyByteArray_AS_STRING(bytearray_obj);
20   while ((item = PyIter_Next(it)) != NULL) {
21     if (! _getbytevalue(item, &value)) {
22       Py_DECREF(item);
23       Py_DECREF(it);
24       Py_DECREF(bytearray_obj);
25       return NULL;
26     }
27     buf[len++] = value;
28     Py_DECREF(item);
29     if (len >= buf_size) {
30       buf_size = len + (len >> 1) + 1;
31       if (PyByteArray_Resize((PyObject *)bytearray_obj, buf_size) < 0) {
32         Py_DECREF(it);
33         Py_DECREF(bytearray_obj);
34         return NULL;
35       }
36       buf = PyByteArray_AS_STRING(bytearray_obj);
37     }
38   }
39   Py_DECREF(it);
40   if (PyByteArray_Resize((PyObject *)bytearray_obj, len) < 0) {
41     Py_DECREF(bytearray_obj);
42     return NULL;
43   }
44   if (bytearray_setslice(self, ..., bytearray_obj) == -1)
45     return NULL;                          /* Omission fault */
46   Py_DECREF(bytearray_obj);
47   Py_RETURN_NONE;
48 }
```

Figure 3.3: Python implementation code containing memory leaks

Linux-3.6, released in 2012. Figure 3.5(a) shows that the number of functions using the basic strategy is increasing more rapidly than the number of functions using the `goto`-based strategy. The number of functions using the `goto`-based strategy or a combination of strategies is increasing gradually. Figure 3.6(a) shows that the number of functions per line of code using the basic strategy is increasing slightly more rapidly then the number of functions per line of code using `goto`-based strategy or the number of functions per line of code using a mix of both strategies in Linux. These figures show the importance of improving the structure of this kind of code.

To understand the design of error-handling code in other kinds of systems software, we have selected ten Python versions between Python-0.9.1, released in 1991 and the latest version, Python

```
1  /* python3.2-3.2.2~rc1/Objects/bytearrayobject.c */
2  static PyObject *
3  bytearray_extend(PyByteArrayObject *self, PyObject *arg) {
4    PyObject *it, *item, *bytearray_obj;
5    Py_ssize_t buf_size = 0, len = 0;
6    int value; char *buf;
7    ...
8    it = PyObject_GetIter(arg);
9    if (it == NULL)
10     return NULL;
11   buf_size = _PyObject_LengthHint(arg, 32);
12   if (buf_size == -1)
13     goto out1;
14   bytearray_obj = PyByteArray_FromStringAndSize(...);
15   if (bytearray_obj == NULL)
16     goto out1;
17   buf = PyByteArray_AS_STRING(bytearray_obj);
18   while ((item = PyIter_Next(it)) != NULL) {
19     if (! _getbytevalue(item, &value))
20       goto out2;
21     buf[len++] = value;
22     Py_DECREF(item);
23     if (len >= buf_size) {
24       buf_size = len + (len >> 1) + 1;
25       if (PyByteArray_Resize(...) < 0)
26         goto out3;
27
28       buf = PyByteArray_AS_STRING(bytearray_obj);
29     }
30   }
31   Py_DECREF(it);
32   if (PyByteArray_Resize((PyObject *)bytearray_obj, len) < 0)
33       goto out4;
34
35   if (bytearray_setslice(self, ..., bytearray_obj) == -1)
36       goto out4;
37
38   Py_DECREF(bytearray_obj);
39   Py_RETURN_NONE;
40
41   out2:
42       Py_DECREF(item);
43   out3:
44       Py_DECREF(bytearray_obj);
45   out1:
46       Py_DECREF(it);
47
48   out:
49       return NULL;
50   out4:
51       Py_DECREF(bytearray_obj);
52       goto out;
53 }
```

Figure 3.4: Improved version of Figure 3.3

3.3.0, released in 2012. We have then selected ten versions of Wine, from version Wine-0.9.22, released in 2006 to the latest version, Wine-1.5.14, released in 2012. Figure 3.5(b) and 3.5(c) show that the error-handling code status in the other systems software is almost same as in the Linux. Python and Wine software has a very large number of functions using the `basic` strategy and these numbers are increasing rapidly. The number of functions using the `goto`-based strategy and the mix of both strategies are increasing very slowly in Python and the increase of these numbers has mostly leveled

off in Wine. In Figure 3.5(b), there is a discontinuity in the graph for Python, because Python 3.0 represents a major revision of the Python language and version 2 and version 3 are being maintained concurrently. Figure 3.6(b) shows that the number of functions per line of code using basic strategy is increasing rapidly in the both series of Python while the number of functions per line of code using the other strategies are always nearly zero. Figure 3.6(c) shows that the number of functions per line of code with different strategies are constant in Wine.

Finally, Figure 3.7 breaks down the above results by subdirectory for Linux-3.6. Most directories have more functions using the basic strategy than the `goto`-based strategy.

3.3 Transformation Algorithm

Our goal is to merge the sequence of statements in each block of error-handling code following the basic strategy, typically a conditional branch ending in a `return`, into a shared sequence of statements at the end of the function, and to replace each error-handling code by a `goto` into this sequence. The algorithm considers one function at a time. The main steps are 1) identify the complete set of error-handling operations found within a given function and collect some other information that is useful in the transformation process, 2) identify operations in this error handling code that can be shared in a sequence at the end of the function, and 3) transform the function definition to move error handling code to the end of the function and insert appropriate `goto`s into this error handling code. These steps are described below, both formally and in terms of examples.

We describe the analysis and transformation rules with respect to a small language, defined according to the grammar shown below. The small language considers the only statements that are assignments, void function calls, conditionals, and `return`. The actual implementation, however, treats the full C language.

Statement t ::= *exp* `=` *exp*`;` | f`(`*exp*`);` | `if (`*exp*`)` $[t]$ (`else` $[t]$)$^?$
Statements $[t]$ ::= $t \ldots t$ (`return` *exp*`;`)$^?$
Program P ::= $[t]$ (l: $[t]$)*

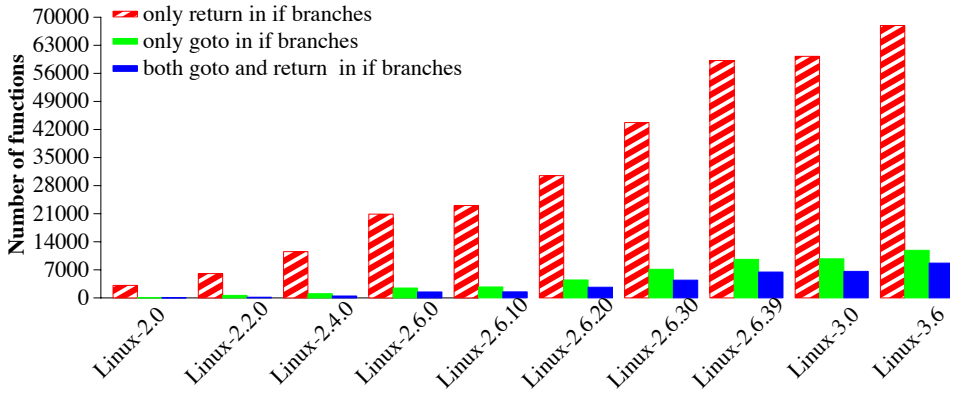
A program in this language is analogous to a function body in C code. In the grammar, *exp* refers to an arbitrary expression, including function calls, f refers to the name of a function, and l refers to a label. To avoid clutter, we omit the braces around the branches of a conditional.

We distinguish two sets of expressions, String and Error. String is the set of strings. Error is the set of expression that may indicate an error, as determined by common Linux coding patterns. Following Linux coding conventions, expressions in Error include `NULL`, negated integers and macros, and expressions of the form `ERR_PTR(`*exp*`)` and `PTR_ERR(`*exp*`)`.[2] Identifiers are also in Error, as they may be initialized to one of these values. A few Linux services, such as ACPI, define their own error conventions. The approach could be extended to take these into account, although in the long term it may be better to refactor that code to use a more standard strategy.

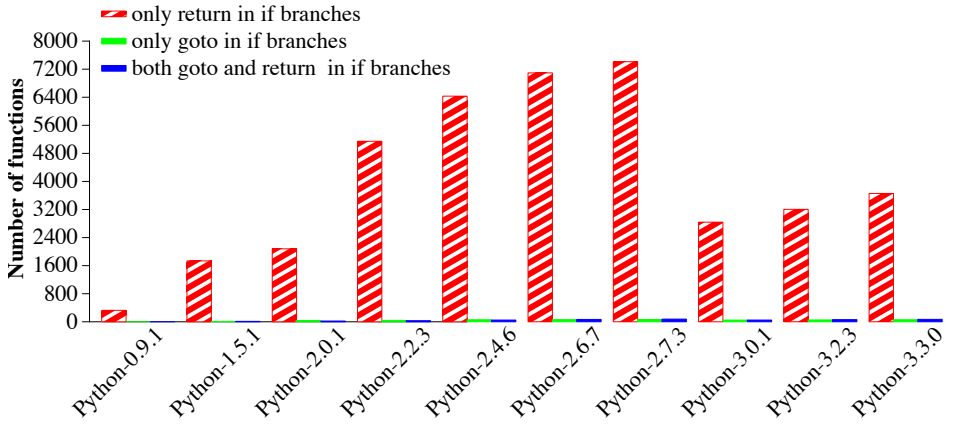
3.3.1 Identifying Error-Handling Code (step 1)

Error handling code is responsible for detecting the failure of an operation, releasing allocated resources to restore the system to a consistent state, and returning an appropriate error indicator to the

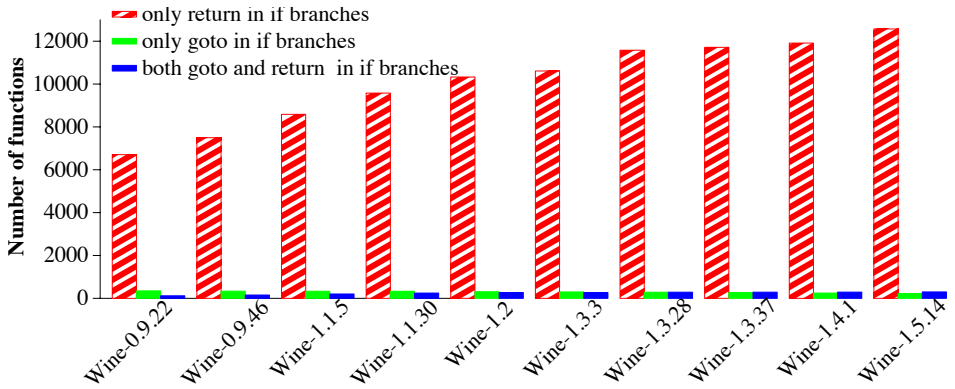
[2] `ERR_PTR(`*exp*`)` and `PTR_ERR(`*exp*`)` coerce an integer error indicator to and from a pointer type, respectively.



(a) Linux

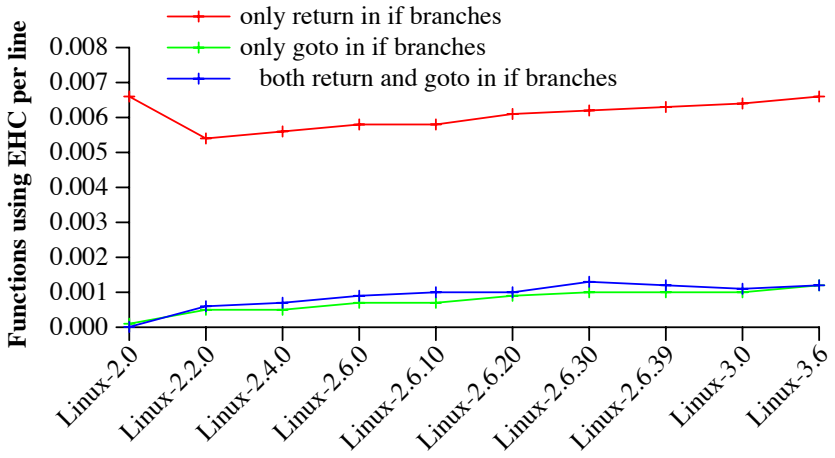


(b) Python

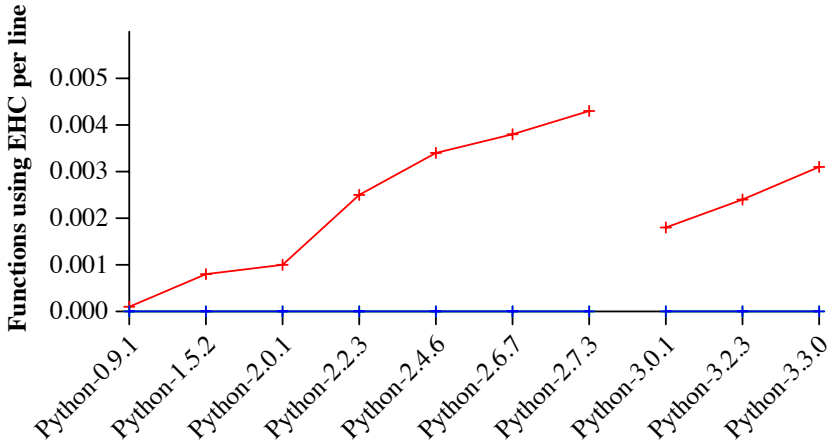


(c) Wine

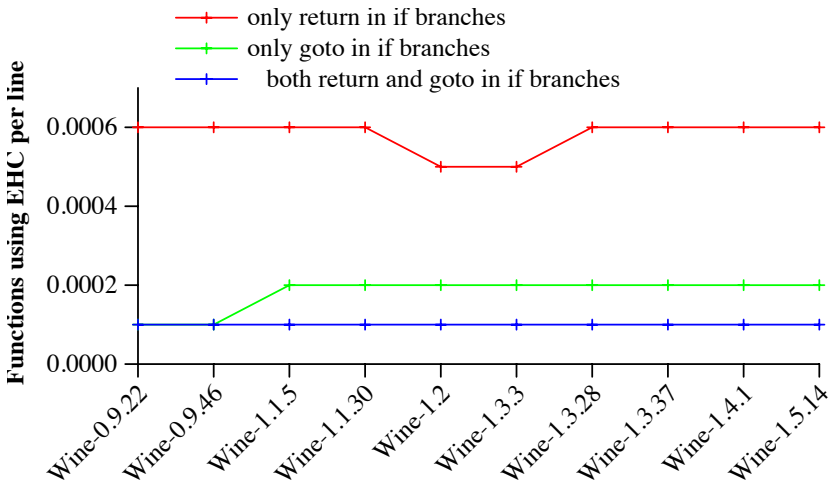
Figure 3.5: Number of functions using only the basic strategy, using the only the `goto`-based strategy, and using a mixture of both.



(a) Linux



(b) Python



(c) Wine

Figure 3.6: Functions using error-handling code (EHC) per line of code, using only the basic strategy, using the only the `goto`-based strategy, and using a mixture of both.

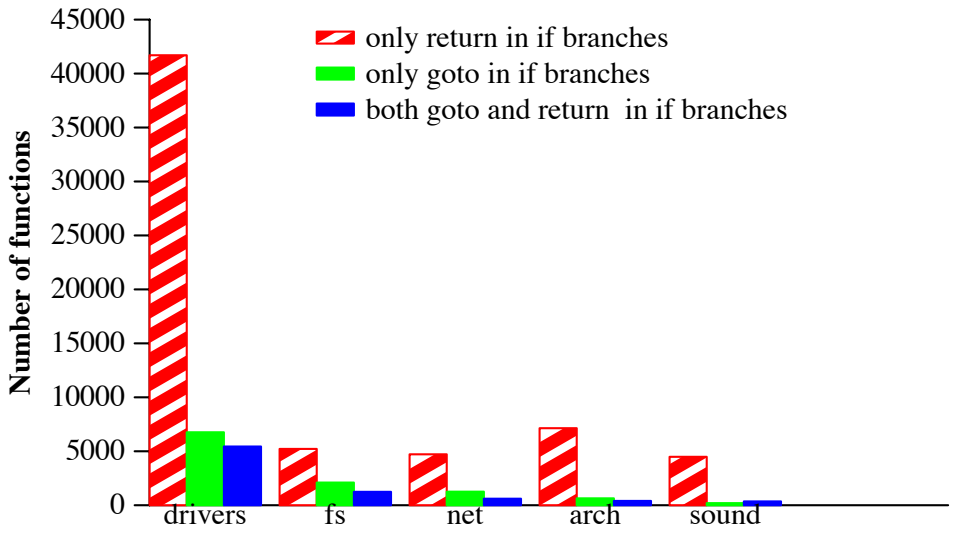


Figure 3.7: Number of Linux 3.6 functions, by directory, using only the basic strategy, using the only the `goto`-based strategy, and using a mixture of both.

calling function. Any operation that may fail must thus be followed by a conditional statement that checks for an error value and performs the appropriate operations.

The C language does not provide any specific abstractions for identifying error-handling code, and thus some heuristics are required. Our heuristics are based on several hypotheses. 1) Whenever an error may occur, it may also not occur, and thus a conditional is required to detect it. 2) When an error occurs, the current computation can either attempt to continue or it can abort. Resource releasing is typically required in the latter case, we call them *state-restoring* operations because they restore the system to a consistent state. 3) When an error occurs, the error-handling code should inform the calling context. For example, an error condition can be indicated via various integer error codes, which are supported by the C standard library function *perror*.

Based on these hypotheses, we identify a block of error-handling code as any conditional branch (possibly containing a *goto*) that ends by either a void return or by returning an error value. Error values are specific to each software project, but typically include `NULL` and various constants, or error-value constructing function calls. For example, in Linux, common error values include negative constants, as illustrated in line 8, 17 and 27 of Figure 3.1. The user must list these error values in a configuration file. This is the only configuration information that is required. Because error values are used for communicating between different parts of the software, they typically change rarely and should be well known to the user. We have developed a tool that proposes a list of possibilities to the user based on the values that are commonly returned in conditional branches.

Our algorithm first focuses on the return value of a conditional branch. Information about the return value is obtained using intraprocedural flow- and path-sensitive constant propagation. This analysis recognizes explicit returns of one of the aforementioned error values, or returns of variables that have been explicitly assigned to one of these values. If the return value is represented as a variable that does not satisfy these properties, or if the function has `void` type, the algorithm analyzes the associated conditional test. In this case, a conditional branch is considered to be a block of error-handling code if the test expression checks for an error value and the branch corresponds to the error

value case.

Table 3.1 shows the kinds of test expressions that are considered to indicate error-handling blocks. Here, `x` is any identifier, `e` is any expression and `C` is a constant (typically defined as a macro).

Table 3.1: Test expressions guarding error-handling blocks

pointer `x`	non-pointer `x`	`e` of any type	
`!x`	`x`	`!e`	`e == !e'`
`x == 0`	`(x = e)`	`IS_ERR(e)`	`e == NULL`
`(x = e) == 0`		`e < 0`	`e == -C`

In Figure 3.3, all of the conditionals that abort the function end by returning `NULL`, and are thus considered to be blocks of error-handling code. Furthermore, all of the conditional tests, on lines 9, 12, 17, 21, 31, 40, 44, also match the patterns shown in Table 3.1.

A conditional that directly returns the result of a function that it calls, other than one of the error functions mentioned above, is not considered to be a block of error handling code, as in this case, the called function is expected to handle any errors.

3.3.1.1 Selecting If Branches (step 1a)

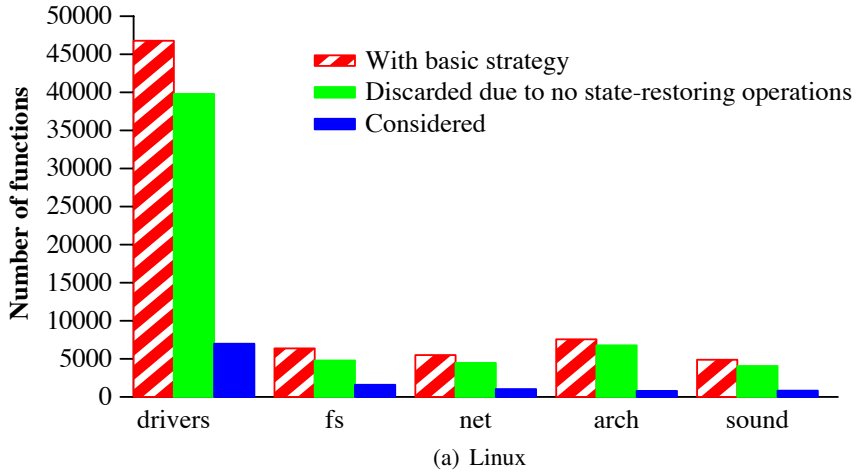
The first step is to select the `if` branches that should potentially be converted from the basic strategy to the `goto`-based strategy. Such `if` branches must at a minimum represent error-handling code. The `if` branch must also not contain other conditionals and must contain at least one function call that is not debugging code, *i.e.*, that does not have a string as an argument.

Figure 3.9 shows a function having multiple `if` branches. The branches labeled 2, 3, 5, and 6 (lines 6, 11, 19, and 25) meet the above conditions and are thus selected for further consideration. On the other hand, the branch labeled 1 (line 4) is not selected because it does not contain any function calls other than the Error call, the branch labeled 4 (line 17) is not selected because it contains another conditional, and the branch labeled 7 (line 31) is not selected because it does not end with a return.

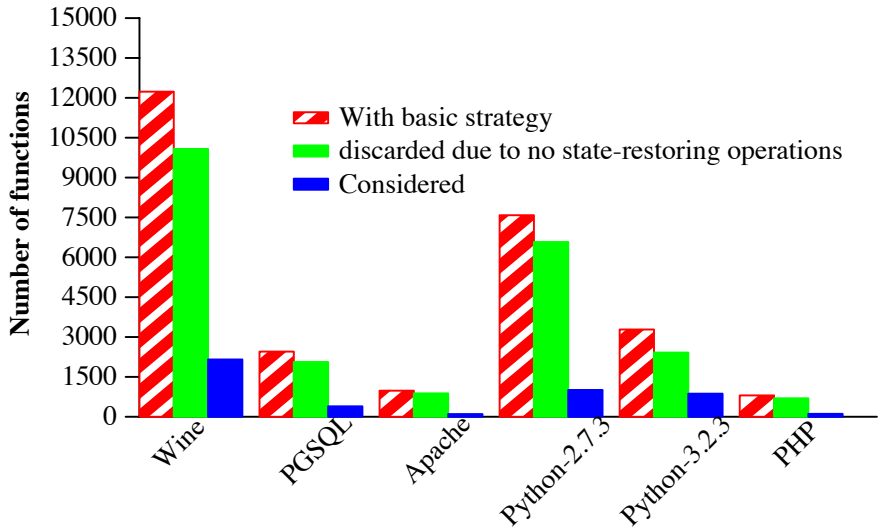
The rule for selecting `if` branches from a program P is formalized as follows. The notation $e_1 \in e_2$, for any terms e_1 and e_2, means that e_1 is a subterm of e_2. The notation "..." means an arbitrary sequence of statements. The result $\mathcal{S}$ of this rule is a set of pairs of a line number and an `if` branch, where the line number is that of the `if` containing the collected branch, obtained using the operator startline. We refer to this set $\mathcal{S}$ as the *branch list*.

$$
\begin{aligned}
\mathcal{S} = \{\langle ln, [t]\rangle \mid\ & \texttt{if (}exp\texttt{)}\ [t] \in P \wedge \\
& [t] \equiv \ldots f(exp') \ \ldots \texttt{return}\ exp''\texttt{;} \wedge \\
& exp' \notin \text{String} \wedge exp'' \in \text{Error} \wedge \\
& \forall exp''', [t]_1, [t]_2 : \texttt{if (}exp'''\texttt{)}\ [t]_1\ (\texttt{else}\ [t]_2)^? \notin [t] \wedge \\
& ln = \text{startline}(\texttt{if (}exp\texttt{)}\ [t])\}
\end{aligned}
$$

Figure 3.8 shows the total number of functions that use the basic strategy or a mixture of the `goto`-based strategy and the basic strategy (red/leftmost bars). The green/middle bars represent the number of functions that are discarded due to the absence of any state-restoring operations in the `if` branches. Thus, this number of functions do not have any basic strategy based `if` branch containing state-restoring operations to be transformed into `goto`-based strategy. Finally, the blue/rightmost bars represent the number of functions that are considered for the transformations process. Figure



(a) Linux



(b) others

Figure 3.8: The red/leftmost bars show the number of functions with basic strategy. The green/middle bars show the number that discarded due to absence of any state-restoring operations. The blue/rightmost bars show the number of the considered functions for the transformation process.

3.8(a) shows that 80% to 90% of the functions in different directories, are discarded before the transformation process. This percentage of the functions do not have any state-restoring operations in the `if` branches that could be moved to label at the end of the functions. In the *fs* directory, 25% of the functions are selected for the transformation. Figure 3.8(b) shows the same information for a range of other systems software, including Wine and Python, considered previously. 80% to 90% of the functions are discarded in different software projects. However, in the Python 3.2.3, 73% of the functions are discarded.

3.3.1.2 Storing the error constants in a variable (step 1b)

An `if` branch implementing the basic strategy can return an error indicator directly or it can store this value in some variable, either before or within the `if` branch. In Linux 3.6 among error-handling

```
1  static int acct_on(char *name) {
2      ...
3    if (IS_ERR(file)) /* 1, not selected */
4      return PTR_ERR(file);
5
6    if (!S_ISREG(file->f_path.dentry->d_inode->i_mode)) { /* 2 */
7      filp_close(file, NULL);
8      return -EACCES;
9    }
10
11   if (!file->f_op->write) { /* 3 */
12     filp_close(file, NULL);
13     return -EIO;
14   }
15
16     ...
17   if (ns->bacct == NULL) { /* 4, not selected */
18     acct = kzalloc(sizeof(struct bsd_acct_struct), GFP_KERNEL);
19     if (acct == NULL) { /* 5 */
20       filp_close(file, NULL);
21       return -ENOMEM;
22     }
23   }
24     ...
25   if (error) { /* 6 */
26     kfree(acct);
27     filp_close(file, NULL);
28     return error;
29   }
30     ...
31   if (ns->bacct == NULL) { /* 7, not selected */
32     ns->bacct = acct;
33     acct = NULL;
34   }
35   return 0;
36 }
```

Figure 3.9: `If`s that do and do not represent error-handling code (Linux-2.6.34/kernel/acct.c)

`if` branches following the basic strategy, the error indicator is returned directly 63% of the time. When different `if` branches return different error indicators, this approach prevents merging the error-handling code. To make this merging possible, our algorithm transforms `if` branches that directly return an error indicator as follows: 1) a new statement is added at the beginning of the `if` branch that stores the current return value in a variable that is common to all selected `if` branches, which we refer to as the *return variable*, and 2) the `return` statement at the end of the `if` branch is transformed to return the value of the return variable.

To carry out this transformation, the algorithm first searches for a variable that is used to return a result somewhere in the function, which can be used as the return variable. Using an existing return variable enables merging return statements and improves readability. If there is no such variable, the algorithm creates a fresh variable for this purpose. If there is more than one such variable, then the first one is chosen. For example, in Figure 3.9, the variable `error` is already used to return the error indicator in the `if` labeled 6 (line 28). Our algorithm thus uses that variable as the return variable. On the other hand, in Figure 3.10 there is no such variable, so the algorithm creates a fresh one.

Formally, the choice of the return variable is determined by the function rv, defined below. This function takes as arguments the complete program P and the branch list $\mathcal{S}$, defined previously. It returns a pair of the return variable and a new version of the program, which may be augmented with

the declaration of the return variable if no suitable existing variable can be found. In this definition @ is used to concatenate code fragments.

$$\begin{array}{l}\mathsf{rv}(P,\mathcal{S}) = \\ \quad \langle x, P\rangle \qquad\qquad \text{if } \texttt{return } x\texttt{;} \in P \wedge \\ \qquad\qquad\qquad\quad \forall \langle \mathit{ln}, [t]\rangle \in \mathcal{S} : \texttt{return } x\texttt{;} \in [t] \vee x \notin [t] \\ \quad \langle x, \texttt{int } x\texttt{;}@P\rangle \text{ where } x \notin P, \text{otherwise}\end{array}$$

After choosing a return variable, the next step is to transform each selected `if` branch that does not already use that variable in its `return` statement. An assignment of the current return value to the return variable is added at the beginning of each such branch, and the `return` statement is modified to return the return variable at the end. This transformation is safe because the return variable has been chosen such that it is not already used anywhere in the branch. The transformation is performed on elements of the branch list $\mathcal{S}$, producing an extended version of the branch list, $\mathcal{S}_{\text{rv}}$, as follows, where $\langle x, P'\rangle = \mathsf{rv}(P, \mathcal{S})$:

$$\begin{array}{l}\mathcal{S}_{\text{rv}} = \{\langle \mathit{ln}, [t]'\rangle \mid \langle \mathit{ln}, \mathit{ifcode}@\texttt{return } e\texttt{;}\rangle \in \mathcal{S} \wedge \\ \qquad\qquad\qquad [t]' \equiv \mathit{ifcode}@\texttt{return } e\texttt{;} \quad \text{if } e \equiv x \wedge \\ \qquad\qquad\qquad [t]' \equiv x \texttt{ = } e\texttt{;}@\mathit{ifcode}@\texttt{return } x\texttt{;} \text{ otherwise}\}\end{array}$$

3.3.1.3 Creating the label environment (step 1c)

The algorithm next creates a label environment that maps each label to all of the code that can be executed when jumping to that label. This label environment is used subsequently to determine whether the state-restoring code of a given `if` branch matches the code following any existing label. Two kinds of judgements are used. For a program P, the judgement, $\vdash P \rightarrow \mathit{lblenv}$ indicates that the final label environment is *lblenv*. For a sequence of labeled statements $(l' : [t])^*$, a judgement of the form $\vdash (l' : [t])^* \rightarrow \langle [t]', \mathit{lblenv}\rangle$, indicates that $[t]'$ is the sequence of statements at the beginning of $(l' : [t])^*$ that preceding code may fall through to, and *lblenv* is the label environment derived from $(l' : [t])^*$. In this definition, for conciseness, we follow the convention that the first rule that matches is the one that applies. ε is an empty sequence of statements.

$$\frac{\vdash (l : [t]')^* \rightarrow \langle [t]'', \mathit{lblenv}\rangle}{\vdash [t](l : [t]')^* \rightarrow \mathit{lblenv}} \qquad \vdash \varepsilon \rightarrow \langle \varepsilon, \emptyset\rangle$$

$$\frac{\vdash (l' : [t]')^* \rightarrow \langle [t]'', \mathit{lblenv}\rangle \wedge [t]''' = [t] \texttt{ return } e\texttt{;}}{\vdash l : [t] \texttt{ return } e\texttt{; } (l' : [t]')^* \rightarrow \langle [t]''', \{\langle l, [t]'''\rangle\} \cup \mathit{lblenv}\rangle}$$

$$\frac{\vdash (l' : [t]')^* \rightarrow \langle [t]'', \mathit{lblenv}\rangle \wedge [t]''' = [t]@[t]''}{\vdash l : [t] \ \ (l' : [t]')^* \rightarrow \langle [t]''', \{\langle l, [t]'''\rangle\} \cup \mathit{lblenv}\rangle}$$

3.3.2 Partition (step 2a)

The next step is to partition each branch in the branch list computed in step 1a to separate the code that is specific to the given error condition, which we refer to as *if code*, from the potentially sharable state-restoring code, which we refer to as *label code*. In particular, `return` statements are label code, as is any non-debugging function call, *i.e.*, a function call that does not have a string argument. For example, in the first branch of Figure 3.10, the call to `printk` and the assignment statement are

```
1  int dvb_register_device(struct dvb_adapter *adap, ...) {
2  ...
3    if ((id = dvbdev_get_free_id (adap, type)) < 0) {
4        mutex_unlock(&dvbdev_register_lock);
5        *pdvbdev = NULL;
6        printk(KERN_ERR "%s: couldn't find free
7                        device id\n",__func__);
8        return -ENFILE;
9    }
10 ...
11   if (!dvbdev) {
12       mutex_unlock(&dvbdev_register_lock);
13       return -ENOMEM;
14   }
15 ...
16   if (!dvbdevfops) {
17       kfree (dvbdev);
18       mutex_unlock(&dvbdev_register_lock);
19       return -ENOMEM;
20   }
21 ...
22   if (minor == MAX_DVB_MINORS) {
23       kfree(dvbdevfops);
24       kfree(dvbdev);
25       mutex_unlock(&dvbdev_register_lock);
26       return -EINVAL;
27   }
28 ...
29   if (IS_ERR(clsdev)) {
30       printk(KERN_ERR "%s: failed to create
31              device dvb%d.%s%d %(%ld)\n",
32              __func__, adap->num, dnames[type], id,
33              PTR_ERR(clsdev));
34       return PTR_ERR(clsdev);
35   }
36 ...
37   return 0;
38 }
```

Figure 3.10: A function that does not have any variable for storing the error indicator. (Linux-2.6.34/drivers/media/dvb/dvb-core/dvbdev.c)

considered to be specific to the given error condition, and are thus if code. However, in the branch, `mutex_unlock(...)` on line 4 is not specific to the branch. It is common to other branches and it is sharable. Thus, `mutex_unlock(...)` is label code.

The transformation performed in step 3 will leave the if code in the `if` branch and move the label code to the end of the function. If in the original block of error-handling code, label code were to appear before if code, then this transformation process would change the order of the operations. To prevent this, the label code is defined to be the largest suffix of an `if` branch that satisfies the above properties. Given the $\mathcal{S}_{\text{rv}}$ computed above, the result of the partitioning process is a refined branch list:

$$\mathcal{S}_{\text{part}} = \\ \{\langle ln, ifcode, lblcode\rangle \mid \langle ln, [t]\rangle \in \mathcal{S}_{\text{rv}} \wedge \vdash [t] \rightarrow \langle ifcode, lblcode\rangle\}$$

where the judgement $\vdash [t] \rightarrow \langle ifcode, lblcode\rangle$ is defined below. We again follow the convention that the first rule that matches is the one that applies. There is no need for a rule for an `if` statement because an element of the branch list contains no nested conditionals.

$$\frac{\vdash [t] \rightarrow \langle ifcode, lblcode \rangle}{\vdash [t] \texttt{ return } e\texttt{;} \rightarrow \langle ifcode, lblcode@\texttt{return } e\texttt{;} \rangle}$$

$$\frac{\vdash [t] \rightarrow \langle ifcode, lblcode \rangle}{\vdash exp_1 \texttt{ = } exp_2\texttt{;}[t] \rightarrow \langle exp_1 \texttt{ = } exp_2\texttt{;} @ifcode, lblcode \rangle}$$

$$\frac{\vdash [t] \rightarrow \langle ifcode, lblcode \rangle \wedge (exp \in \text{String} \vee ifcode \neq \varepsilon)}{\vdash f\texttt{ (}exp\texttt{) ;}[t] \rightarrow \langle f\texttt{ (}exp\texttt{) ;} @ifcode, lblcode \rangle}$$

$$\frac{\vdash [t] \rightarrow \langle \varepsilon, lblcode \rangle}{\vdash f\texttt{ (}exp\texttt{) ;}[t] \rightarrow \langle \varepsilon, f\texttt{ (}exp\texttt{) ;} @lblcode \rangle} \qquad \vdash \varepsilon \rightarrow \langle \varepsilon, \varepsilon \rangle$$

3.3.3 Filtering (step 2b)

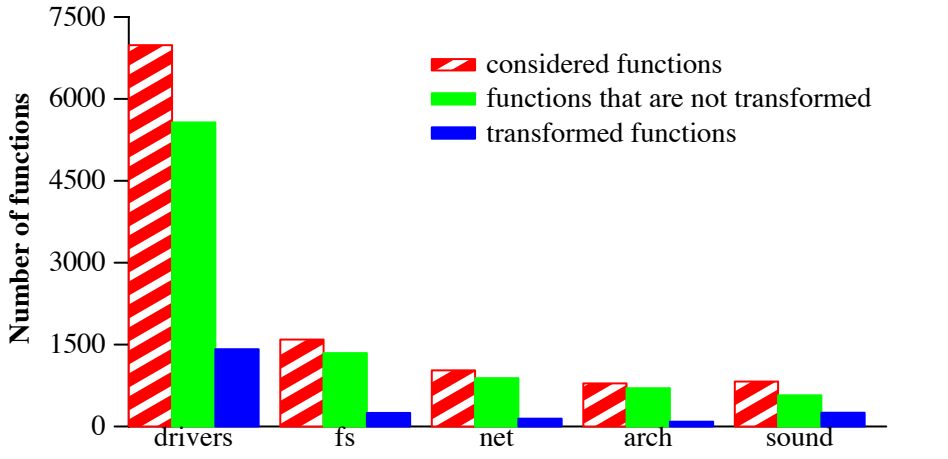
Moving label code to the end of the function can only reduce the code size if part of the label code, including at least one state-restoring operation, is shared with the label code of some other `if` branch or some other existing code at the end of the function. If there is no such shared code, then we consider that the benefit of transforming the code does not outweigh the cost of introducing a `goto`, and thus we remove the entry from the branch list $\mathcal{S}_{\text{part}}$ computed in step 2a. This filtering process is defined as follows, using the program P and the *lblenv* computed above, to produce a filtered branch list $\mathcal{S}_{\text{filter}}$:

$$\begin{array}{l}
\mathcal{S}_{\text{filter}} = \\
\quad \{\langle l, ifcode, lblcode \rangle \mid \\
\quad\ \langle l, ifcode, lblcode \rangle \in \mathcal{S}_{\text{part}} \wedge \\
\quad\ lblcode \equiv \texttt{...}\ \ f\texttt{(}exp_1\texttt{);}\ \texttt{return (}exp_2\texttt{);} \wedge \\
\quad\ (\exists \langle l', ifcode', \texttt{...}\ \ f\texttt{(}exp_1\texttt{);}\ \texttt{return (}exp_2\texttt{);} \rangle \in \mathcal{S}_{\text{part}} : l \neq l' \vee \\
\quad\ \ \exists \langle l', \texttt{...}\ f\texttt{(}exp_1\texttt{);}\ \texttt{return (}exp_2\texttt{);} \rangle \in lblenv \vee \\
\quad\ \ P \equiv \texttt{...}\ f\texttt{(}exp_1\texttt{);}\ \texttt{return (}exp_2\texttt{);}\}
\end{array}$$

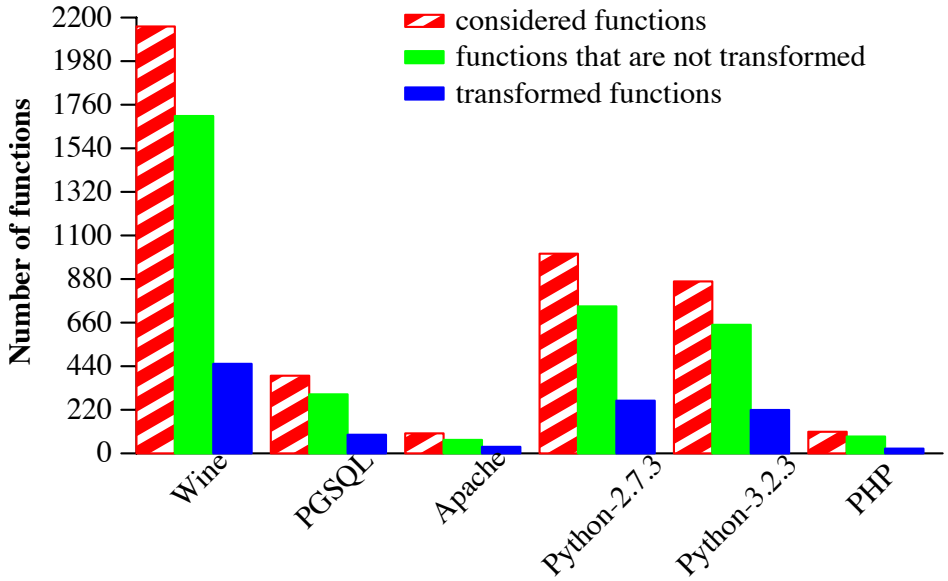
Figure 3.11 shows the total number of functions that were considered for the transformation process (red/leftmost bars), the number of functions that are not considered for transformation (green/middle bars), indicating that there is no shared state-restoring code, and the number of these functions that survive the filtering process (blue/rightmost bars), implying that there is some shared state-restoring code. The Figure 3.11(a) showing that the filtering eliminates two thirds of the selected functions in most directories in the Linux. However, almost half of the selected functions in the `sound` directory are transformed. Figure 3.11(b) also shows the same information for the other software projects. The filtering process eliminates almost two thirds of the selected functions in each of the software projects.

3.3.4 Classification and transformation (step 3)

This step classifies the remaining elements of the branch list, $\mathcal{S}_{\text{filter}}$, according to how difficult they are to transform. On the basis of difficulty, the algorithm chooses the appropriate transformation. We classify the `if` branches into four categories: Simple, Hard, Harder and Hardest. The classification and transformation process iterates over the elements of $\mathcal{S}_{\text{filter}}$ starting with the one with the largest line number. This element typically contains the longest sequence of state-restoring code (*cf* Figure 5.2), undoing all of the operations that have been performed in the function, and thus offers the most opportunity for sharing. We explain the classification and transformation process in terms of an



(a) Linux



(b) others

Figure 3.11: The red/leftmost bars show the number of functions collected in the `if` selection step. The green/middle bars show the number that the filtering step discards. The blue/rightmost bars show the number that the filtering step keeps for transformation.

artificial example that illustrates all of the possible cases. This example is more complex than typical functions.

For a given `if` branch, if the label code is the same as the code already associated with some label in the label environment, then the `if` branch is classified as *Simple*, because no code has to be moved. Instead, it is sufficient to remove the label code from the branch and replace it with a `goto` statement with that label name. An example is the branch labeled 5 in Figure 3.12a (line 20). In this case, the label code is exactly same as the code already at the label `out`.

On the other hand, if an `if` branch's label code is not exactly same as the code at any label, but is the same as a suffix of some existing label's code or is the same as a suffix of the entire function, then the `if` branch is classified as *Hard*. In this case it is also not necessary to move any code, but the

```
1   if(x) {      /* 1, Hardest */
2         kl();
3         gh();
4         return ret;
5   }
6   if(y) {      /* 2, Hardest */
7         ij();
8         cd();
9         return ret;
10  }
11  if(z) {      /* 3, Harder */
12        ef();
13        gh();
14        return ret;
15  }
16  if(k) {      /* 4, Hard */
17        cd();
18        return ret;
19  }
20  if(l) {      /* 5, Simple */
21        ab();
22        cd();
23        return ret;
24  }
25  ...
26  out: ab();
27       cd();
28       return ret;
```

a) Original code

```
1   if(x)
2       goto out4;
3   if(y)
4       goto out3;
5   if(z)
6       goto out2;
7   if(k)
8       goto out1;
9   if(l)
10      goto out;
11  ...
12  out: ab();
13  out1: cd();
14        return ret;
15  out2: ef();
16  out5: gh();
17        return ret;
18  out3: ij();
19        goto out1;
20  out4: kl();
21        goto out5;
```

b) Transformed code

Figure 3.12: Simple, Hard, Harder, Hardest branches

algorithm has to identify a position for a new label. The label code of branch 4 in Figure 3.12a (line 16) matches a suffix of the code at the label `out`. The algorithm thus creates a new label just before the call to `cd` at the end of the function (line 13 in Fugure 3.12b), and replaces branch 4 by a `goto` to the new label (line 8, in Figure 3.12b).

In the third case, an `if` branch's label code matches neither a suffix of any existing label's code nor the code at the original end of the function. Such a branch is classified as *Harder*. For such a branch, the algorithm creates a new label and places it at the end of the function, along with the branch's label code. In the case of a `void` function (outside the scope of our small language), there may be no `return` at the end of the original function. In this case, the algorithm additionally adds `return;` before the new label. For example, after transforming branches 5 and 4 in Figure 3.12a, the code in branch 3 is not a suffix of any existing label's code. So, the algorithm inserts a new label with this code after the `return` statement of the `out` label.

The final category is *Hardest*. In this category, the complete label code is not a suffix of any existing label's code, however a suffix of the label code is the same as a suffix of some existing label's code. This results in two parts of the label code; one that does not match any existing label's code and the other that is a suffix of some existing label's code. The unmatched part can be treated as Harder and the matched part can be treated as though it is Simple or Hard. Branches 2 and 1 of Figure 3.12a (lines 6 and 1) are in the Hardest category. In each case, the first statement of the label code, `ij()` or `kl()`, respectively, is not found in any branch. So these statements are treated as Harder. In each case, the code in the remainder of the branch is identical to or a proper suffix of the code at an existing label. For branch 2 (line 6), the call to `cd` and the `return` are identical to the code at the label introduced for treating branch 4, so this code is treated as Simple. For branch 1 (line 1), the call to

`gh` and the `return` match a suffix of the code at the label introduced when treating branch 3, so this code is treated as Hard.

The complete transformation process selects the element of $\mathcal{S}_{\text{filter}}$ with the largest line number, classifies it according to the rules below, transforms it according to the result of the classification, and then repeats on the next element of $\mathcal{S}_{\text{filter}}$, until all elements have been considered. The label environment *lblenv* is recomputed after each transformation step, according to the rules described in step 1c.

The rules for classifying branches as Simple, Hard, Harder, and Hardest are formalized as shown below for an element of $\mathcal{S}_{\text{filter}}$, $\langle ln, ifcode, lblcode\rangle$. In these rules, $\mathsf{createlbl}()$ creates a new label and $\mathsf{suffix}(a, b)$ is satisfied if a is a suffix of b. A judgement has the form $\vdash_c \langle ln, ifcode, lblcode\rangle \rightarrow \langle ln, ifcode, status\rangle$ where *status* is defined as follows:

$$\begin{array}{rcl} status & = & \mathsf{Simple}(label) \\ & | & \mathsf{Hard}(label, fresh\ label, lblcode) \\ & | & \mathsf{Harder}(fresh\ label, lblcode) \\ & | & \mathsf{Hardest}(fresh\ label, lblcode, status) \end{array}$$

The classification rules are as follows. In each case, the first rule that matches is the one that is applied.

$$\frac{\exists l : \langle l, lblcode\rangle \in lblenv}{\vdash_c \langle ln, ifcode, lblcode\rangle \rightarrow \langle ln, ifcode@\texttt{goto}\ \ l\texttt{;}, \mathsf{Simple}(l)\rangle}$$

$$\frac{\exists \langle l, [t]\rangle \in lblenv : \mathsf{suffix}(lblcode, [t]) \wedge nl = \mathsf{createlbl}()}{\vdash_c \langle ln, ifcode, lblcode\rangle \rightarrow \langle ln, ifcode@\texttt{goto}\ \ nl\texttt{;}, \mathsf{Hard}(l, nl, lblcode)\rangle}$$

$$\frac{\begin{array}{c} lblcode \equiv \ldots\ \ f\ \texttt{(}exp\texttt{)}\ \texttt{;}\ \ \texttt{return}\ \ e\texttt{;} \wedge nl = \mathsf{createlbl}() \wedge \\ (\forall \langle l, [t]\rangle \in lblenv \wedge \mathsf{not}(\mathsf{suffix}(f\ \texttt{(}exp\texttt{)}\ \texttt{;}\ \ \texttt{return}\ \ e\texttt{;}, [t])) \end{array}}{\vdash_c \langle ln, ifcode, lblcode\rangle \rightarrow \langle ln, ifcode@\texttt{goto}\ \ nl\texttt{;}, \mathsf{Harder}(nl, lblcode)\rangle}$$

$$\frac{\begin{array}{l} (\exists [t]_1, [t]_2 : lblcode \equiv [t]_1@[t]_2 \wedge \\ \quad (\forall \langle l, [t]\rangle \in lblenv : \forall [t]'_1 \neq \varepsilon : \\ \qquad \mathsf{suffix}([t]'_1, [t]_1) \Rightarrow \mathsf{not}(\mathsf{suffix}([t]'_1@[t]_2, [t]))) \wedge \\ nl = \mathsf{createlbl}() \wedge \\ \vdash_c \langle -1, [t]_1, [t]_2\rangle \rightarrow \langle -1, [t]''_1, status\rangle \end{array}}{\vdash_c \langle ln, ifcode, lblcode\rangle \rightarrow \langle ln, ifcode@\texttt{goto}\ \ nl\texttt{;}, \mathsf{Hardest}(nl, [t]_1, status)\rangle}$$

The transformation is then in two parts, considering first the label code and then the if code. Given a classified triple $\langle ln, ifcode, status\rangle$, we first transform the program P to reposition the label code as indicated by *status*. This part of the transformation uses judgements of the form $status \vdash_{\text{lbl}} P \rightarrow P'$, producing a new program P'.

$$\mathsf{Simple}(l) \vdash_{\text{lbl}} P \to P$$

$$\mathsf{Hard}(l, nl, lblcode) \vdash_{\text{lbl}} \ldots l : [t] \ @ \ lblcode \ldots \to \\ \ldots l : [t] \ @ \ nl : lblcode \ldots$$

$$\mathsf{Harder}(nl, lblcode) \vdash_{\text{lbl}} \ldots \texttt{return } e\texttt{;} \to \ldots \texttt{return } e\texttt{; } nl \texttt{ : } \quad lblcode$$

$$\mathsf{Hardest}(nl, [t]_1, \mathsf{Simple}(l)) \vdash_{\text{lbl}} \ldots \texttt{return } e\texttt{; } l \texttt{ : } \quad [t]_2 \ldots \to \\ \ldots \texttt{return } e\texttt{; } nl \texttt{ : } \quad [t]_1 \ @ \ l \texttt{ : } \quad [t]_2 \ldots$$

$$\mathsf{Hardest}(nl, [t], \mathsf{Simple}(l)) \vdash_{\text{lbl}} \ldots \texttt{return } e\texttt{; } \to \\ \ldots \texttt{return } e\texttt{; } nl \texttt{ : } \quad [t] \ @ \ \texttt{goto } l$$

$$\mathsf{Hardest}(nl, [t]_1, \mathsf{Hard}(l, nl', [t]_2)) \vdash_{\text{lbl}} \ldots \ l : [t] @ [t]_2 \ldots \texttt{return } e\texttt{; } \to \\ \ldots \ l : [t] @ nl' : [t]_2 \ldots \texttt{return } e\texttt{; } nl \texttt{ : } \quad [t]_1 \ @ \ \texttt{goto } l$$

Finally, given a classified triple $\langle ln, ifcode, status \rangle$ and the program P' produced by the above rules, we adjust the corresponding `if` statement in the program to use the new if code. This part of the transformation uses judgements of the form $status \vdash_{\text{if}} P \to P'$, producing a new program P'. This new program P' is then used on the next iteration, to treat the next element of $\mathcal{S}_{\text{filter}}$.

$$\frac{\mathsf{startline}(\texttt{if } (exp) \ [t]) = ln}{\langle ln, ifcode, status \rangle \vdash_{\text{if}} \ldots \texttt{ if } (exp) \ [t] \ \ldots \to \\ \ldots \texttt{ if } (exp) \ ifcode \ \ldots}$$

3.4 Evaluation

The algorithm, excluding the parser, has been implemented as 1300 lines of OCaml code. For the parser, we have reused the parser developed for the program transformation system Coccinelle [63, 64]. This parser does not require first executing the C preprocessor, and thus all Linux code, for all possible architectures and configurations, can be treated. In this section, we present the results of applying our tool to five directories (`drivers`, `fs`, `net`, `arch`, and `sound`) in Linux 3.6 kernel source code as well as to five widely used open-source systems software projects: PostgreSQL, Apache, Wine, Python, and PHP on one core of an 8-code 3GHz machine with 16GB memory.

The goals of the evaluation of our transformation algorithm are 1) analyze a real example to understand the effects of the transformation to the systems code, 2) to assess the impact of the filtering process presented in Section 3.3.3, 3) to understand the different types of the branches in the considered systems software, 4) to understand the effect of creating assignments, `gotos` and labels to the systems code, and 5) to understand the contributions of the transformation process in code sharing.

3.4.1 An example from the Linux kernel.

Figure 3.13 shows an extract of code from Linux 3.6 using the basic strategy, and Figure 3.14 shows the result of the transformation. The transformation starts with the branch labelled 4 (line 34). This

```
1  static int download_fw(struct edgeport_serial *serial) {
2    ...
3    if (serial->product_info.TiMode == TI_MODE_DOWNLOAD) {
4       ...
5    }
6    else if ((start_address = get_descriptor_addr(serial,
7       I2C_DESC_TYPE_FIRMWARE_BLANK, rom_desc)) != 0) {
8       ...
9       if (!vheader) { /* 1 */
10         dev_err(dev, "%s - out of memory.\n", __func__);
11         kfree(header);
12         kfree(rom_desc);
13         kfree(ti_manuf_desc);
14         return -ENOMEM;
15      }
16      ...
17      if (status) { /* 2 */
18         kfree(vheader);
19         kfree(header);
20         kfree(rom_desc);
21         kfree(ti_manuf_desc);
22         return status;
23      }
24      ...
25      if (status) { /* 3 */
26         dbg("%s - can't read header back", __func__);
27         kfree(vheader);
28         kfree(header);
29         kfree(rom_desc);
30         kfree(ti_manuf_desc);
31         return status;
32      }
33      ...
34      if (status) { /* 4 */
35         dev_err(dev,"%s - UMPC_COPY_DNLD_TO_I2C failed\n",...);
36         kfree(rom_desc);
37         kfree(ti_manuf_desc);
38         return status;
39      }
40   }
41   ...
42   stayinbootmode:
43        dbg("%s - STAYING IN BOOT MODE", __func__);
44        serial->product_info.TiMode = TI_MODE_BOOT;
45        return 0;
46 }
```

Figure 3.13: Example for transformation. (Linux-3.6/drivers/usb/serial/io_ti.c)

branch has no code in common with the only label that is available, and so it is classified as Harder. Its code is moved to the end of the function, with the label `out` (Figure 3.14, line 37). Next, the branch labelled 3 (line 26) is considered. This branch contains a superset of the operations at the label `out`, and so it is classified as Hardest. Because the label `out` is at this point immediately preceded by `return 0` and the code at the label `out` is a suffix of the code in branch 3, the new label `out1` can be placed just before `out`. Next, we consider the branch labelled 2 (line 18). This branch has the same state-restoring operations as the branch labelled 3, and thus the branch labelled 2 is considered to Simple and reuses the label `out1`. Finally, the branch labelled 1 (line 10) contains a suffix of this code, implying that it is classified as Hard. The label `out2` is introduced in transforming this branch. Overall, all of the `if`s of the function are transformed, and most are reduced to a `goto` and possibly

```
1  static int download_fw(struct edgeport_serial *serial) {
2    ...
3    if (serial->product_info.TiMode == TI_MODE_DOWNLOAD) {
4       ...
5    }
6    else if ((start_address = get_descriptor_addr(serial,
7       I2C_DESC_TYPE_FIRMWARE_BLANK, rom_desc)) != 0) {
8       ...
9       if (!vheader) {
10         status = -ENOMEM;
11         dev_err(dev, "%s - out of memory.\n", __func__);
12         goto out2;
13      }
14      ...
15      if (status)
16         goto out1;
17      ...
18      if (status) {
19         dbg("%s - can't read header back", __func__);
20         goto out1;
21      }
22      ...
23      if (status) {
24         dev_err(dev,"%s - UMPC_COPY_DNLD_TO_I2C failed\n",...);
25         goto out;
26      }
27   }
28   ...
29   stayinbootmode:
30         dbg("%s - STAYING IN BOOT MODE", __func__);
31         serial->product_info.TiMode = TI_MODE_BOOT;
32         return 0;
33   out1:
34         kfree(vheader);
35   out2:
36         kfree(header);
37   out:
38         kfree(rom_desc);
39         kfree(ti_manuf_desc);
40         return status;
41 }
```

Figure 3.14: Transformed version of Figure 3.13.

a debugging statement.

3.4.2 The impact of filtering.

As was shown in Figure 3.11, for many functions that use the basic strategy, all of the branches are filtered, and thus the function is not transformed by our algorithm. Furthermore, due to the filtering, only a subset of the error handling code within a function may be transformed. Table 3.2 shows the number of the considered functions using the basic strategy, quantifies the reasons why functions are not transformed, and indicates the number of functions that are partially affected and unaffected by the filtering process. Between 14% and 49% of the functions are fully or partially transformed in different applications.

Filtering discards an `if` branch either 1) because its label code contains only a `return` statement or 2) because its label code ends with state restoring code that is not shared with that of any other label code. Columns 6 to 9 of Table 3.2 show the number of functions for which all of the `if` branches are

	Directory	Functions	Functions that are transformed			Functions that are not transformed			
			Total	Full processing	Partial processing	Strict	Single if	Unordered	No sharing
Linux	*drivers*	6984	1415(20%)	1234(18%)	181(2%)	2596(36%)	2900(42%)	59(1%)	114(2%)
	fs	1592	248(16%)	184(12%)	64(4%)	709(45%)	597(38%)	7(1%)	31(2%)
	net	1028	143(14%)	111(11%)	32(3%)	389(38%)	472(46%)	11(1%)	13(2%)
	arch	789	383(49%)	78(10%)	305(39%)	0(0%)	384(49%)	3(0%)	19(2%)
	sound	823	252(31%)	221(27%)	31(4%)	132(16%)	420(51%)	8(1%)	11(1%)
	total	**11216**	**2441(22%)**	**1828(16%)**	**318(6%)**	**3726(33%)**	**4782(43%)**	**88(1%)**	**188(2%)**
Wine		2155	452(21%)	321(18%)	131(7%)	365(17%)	1131(52%)	14(1%)	193(9%)
PostgreSQL		392	94(24%)	65(26%)	29(12%)	147(38%)	144(37%)	2(1%)	5(1%)
Apache		101	33(33%)	26(30%)	7(8%)	14(14%)	51(50%)	0(0%)	2(2%)
Python-2.7.3		1008	266(26%)	212(23%)	54(6%)	95(9%)	602(60%)	17(2%)	28(3%)
Python-3.3.0		868	219(25%)	165(21%)	54(7%)	84(10%)	525(60%)	15(2%)	25(3%)
PHP		109	24(22%)	16(29%)	8(19%)	26(24%)	46(42%)	3(3%)	10(9%)
	Grand total	**15849**	**3529(22%)**	**2633(17%)**	**896(5%)**	**4457(28%)**	**7272(46%)**	**139(1%)**	**451(3%)**

Table 3.2: Number of functions of Linux-2.6.34 that can and cannot be transformed.

filtered due to these reasons. Label code may contain only a return statement due to the requirement in the partitioning process that state-restoring code only be added to the label code when there are no subsequent assignments or debugging statements. If an assignments or debugging code occurs just before the `return`, then the label code will not contain any state-restoring operations. The number of functions discarded for this reason is shown in the column "Strict". The lack of shared state-restoring code may occur because a function has only one error-handling `if` ("Single if"), because there are shared state-restoring operations but they appear in the wrong order ("Unordered"), or because there are multiple `if` branches with state-restoring operations but none are shared ("No sharing"). The latter two cases may indicate bugs, and are thus reported to the user for further inspection.

All of the error-handling code in a function may be transformed, or some of the `if` branches may be filtered and thus only a portion of the error-handling code is transformed. Table 3.2 shows that, depending on the applications, between 10 and 30% of the functions are fully transformed and 2 to 39% more are partially transformed. Overall, 22% of the functions that contain some error-handling code structured according to the basic strategy are transformed. Table 3.2 shows that, 28% of the functions are not transformed because of *strict* reason. 46% of the functions are not transformed because there is only one `if` branch in the function. 1% of the functions are not transformed because state-restoring operations are in the wrong order in the branches. 3% of the functions are not transformed state-restoring operations in the branches are not shared. In total, depending on the applications, between 51% and 86% of the functions are not transformed.

3.4.3 Branch classification.

Transforming a Simple branch replaces the branch's label code by a single `goto`, which is a best case in terms of the reduction in code size. Transforming a Hard branch achieves similar improvement, as it requires only adding a new label. Both the Simple and Hard cases may also introduce an assignment for the return variable. Transforming a Harder `if` branch in itself increases code size, because the label code is copied to the end of the function as is, and a `goto`, a label and possibly an assignment must be introduced. Nevertheless, the filtering process guarantees that at least part of the copied code for a Harder branch is shared with another branch. Finally, transformation of a Hardest branch copies some code and introduces an extra `goto`, but part of its code is again shared with some other branch.

Table 3.3 shows the number of `if` branches in each category in the various applications, as well as the percentage in each category as compared to the total number of transformed `if` branches in the given application. In many of the applications, the percentages in all of the categories are roughly similar. The most significant exception is Apache. Apache has a higher proportion of simple and Hardest harder branch than the other software and lower proportion of Hard and Hardest simple. There are other exceptions to the proportions of classifications of branches in the applications. PHP has higher proportions of Harder, and a lower proportion of Hardest simple and Hardest harder. In the Linux, `drivers`, arch and `sound` have a higher proportion of Simple branches. `fs` has a relatively high proportion of Hardest harder branches, while `sound` has a very low proportion of Hard branches. `sound` is a collection of sound-card drivers that were split off from the `drivers` directory early in the Linux 2.5 series, and it may be that they have followed a different development pattern than the rest of the code.

		Simple(%)	Hard(%)	Harder(%)	Hardest Simple(%)	Hardest Hard(%)	Hardest Harder (%)	Total(%)
Linux	*drivers*	1074(34.5%)	573(18.4%)	814(26.2%)	330(10.6%)	20(0.6%)	298(9.6%)	3109(100.0%)
	fs	112(24.9%)	71(15.8%)	111(24.7%)	27(6.0%)	6(1.3%)	123(27.3%)	450(100.0%)
	net	81(29.9%)	47(17.3%)	76(28.0%)	16(5.9%)	5(1.8%)	46(17.0%)	271(100.0%)
	arch	58(32.0%)	33(18.2%)	52(28.7%)	18(10.0%)	3(1.7%)	17(9.4%)	181(100.0%)
	sound	367(45.1%)	32(3.9%)	206(25.3%)	55(6.8%)	1(0.1%)	152(18.7%)	813(100.0%)
	total	**1692(35.1%)**	**756(15.7%)**	**1259(26.1%)**	**446(9.2%)**	**35(0.7%)**	**636(13.2%)**	4824(100.0%)
Wine		293(29.3%)	126(12.6%)	316(31.6%)	92(9.2%)	7(0.7%)	166(16.6%)	1000(100.0%)
PostgreSQL		111(40.8%)	39(14.3%)	75(27.6%)	21(7.7%)	1(0.4%)	25(9.2%)	272(100.0%)
Apache		52(42.2%)	7(5.7%)	29(23.8%)	1(0.8%)	1(0.8%)	32(26.2%)	122(100.0%)
Python-2.7.3		200(33.3%)	101(16.8%)	191(31.8%)	55(9.2%)	4(0.7%)	49(8.2%)	600(100.0%)
Python-3.2.3		159(31.1%)	93(18.2%)	158(30.9%)	45(8.8%)	3(0.6%)	54(10.5%)	512(100.0%)
PHP		20(35.7%)	8(14.3%)	21(37.5%)	1(1.8%)	2(3.6%)	4(7.1%)	56(100.0%)
	Grand total	**2527(34.2%)**	**1130(15.3%)**	**2049(27.7%)**	**661(8.9%)**	**53(0.7%)**	**966(13.1%)**	7386(100.0%)

Table 3.3: Number and percentage of Simple, Hard, Harder and Hardest branches among the transformed branches in Linux-2.6.34. Hardest branches have a suffix of their state-restoring code that is Simple or Hard.

3.4.4 Branch transformation.

Transformation process creates labels, `gotos`, and return variable initialization statements while converting basic strategy into `goto`-based strategy. Table 3.4 shows the number of transformed functions and the number of labels, `gotos`, and return variable initialization statements ("assignment" column) introduced by the transformation. The transformation of a given branch introduces at most two labels and two gotos (Hardest case), and at most one assignment. A Simple branch, however, introduces no labels, and a Hard or Harder branch introduces only one. The information in this table thus presents another perspective on the number of branches in the various categories (Table 3.3).

Table 3.4 shows that the number of labels introduced is always significantly lower than the number of branches, reflecting a good number of Simple branches. Apache has a particularly high ratio of branches to labels created, reflecting the high rate of Simple branches in this case. The number of `gotos` introduced is overall slightly higher than the number of branches, because each branch introduces at least one `goto` and Hardest branches may introduce two. The number of branches requiring two `gotos` is however small (*cf*, Hardest-Hard case in Table 3.3). Finally, the number of return variable initializations is also much lower than the number of branches, and for many applications it is lower than the number of labels, indicating that the need to introduce a return variable to abstract over error indicators is not a major burden.

		Transformed functions	Branch	Label (Avg)	Goto (Avg)	Assignment (Avg)
Linux	*drivers*	1415	3109	2055 (1.45)	3129 (2.21)	2787 (1.97)
	fs	248	450	344 (1.38)	456 (1.84)	448 (1.81)
	net	143	271	195 (1.36)	276 (1.93)	270 (1.89)
	arch	88	181	126 (1.43)	184 (2.09)	143 (1.63)
	sound	252	813	447 (1.77)	814 (3.23)	577 (2.29)
	total	**2146**	**4824**	**3167 (1.47)**	**4859(2.26)**	**4227 (1.97)**
Wine		452	1000	714 (1.58)	1007 (2.23)	863 (1.91)
PostgreSQL		94	272	162 (1.72)	273 (2.91)	176(1.88)
Apache		33	122	71 (2.15)	123 (3.73)	90(2.73)
Python-2.7.3		266	600	404 (1.52)	604 (2.27)	481 (1.81)
Python-3.3.0		219	512	356 (1.63)	515 (2.35)	422(1.92)
PHP		24	56	38 (1.58)	58 (2.42)	43 (1.83)
Grand total		**5380**	**7386**	**4912 (1.52)**	**7439(2.30)**	**6305 (1.95)**

Table 3.4: Total number and average number of labels, gotos, assignments created in the transformed functions.

3.4.5 Code sharing.

The goal of the approach is to cause state-restoring code to be shared, to improve robustness in the face of maintenance and to reduce code size. Thus, we should ideally not just have many Simple and Hard branches, but these branches should also contain a good number of state-restoring operations that can then be shared. On the other hand, Harder branches, and to some extent Hardest branches, simply move existing code. Table 3.5 shows the number of merged and moved lines of code. In most applications, at least 45% more code is merged than moved.

Another perspective on the same information is to consider how many functions have only Simple and Hard branches, implying that the state-restoring operations are already available at the end of the

		Merged (Avg)	Moved (Avg)	Merged/Moved	Transformed (Avg)
Linux	*drivers*	4416 (3.1)	2584 (1.8)	1.7	7000 (4.9)
	fs	591 (2.4)	462 (1.9)	1.3	1053 (4.2)
	net	350 (2.4)	248 (1.7)	1.4	598 (4.2)
	arch	243 (2.8)	156 (1.8)	1.6	399 (4.5)
	sound	1062 (4.2)	656 (2.6)	1.6	1718 (6.8)
	total	**6662 (3.1)**	**4106 (1.9)**	**1.6**	**10768(5.0)**
Wine		1218 (2.7)	958 (2.1)	1.3	2176 (4.8)
PostgreSQL		391 (4.2)	240 (2.6)	1.6	631 (6.7)
Apache		156 (4.7)	109 (3.3)	1.4	265 (8.0)
Python-2.7.3		789 (3.0)	553 (2.1)	1.4	1342 (5.1)
Python-3.3.0		677 (3.1)	477 (2.2)	1.4	1154 (5.3)
PHP		69 (2.9)	56 (2.3)	1.2	125 (5.2)
Grand total		**9962 (3.1)**	**6499 (2.0)**	**1.5**	**16461(5.1)**

Table 3.5: Total number and average number of lines that are merged, moved, and transformed in the transformed functions.

		Harder = 0 and Hardest = 0	Harder = 1 and Hardest = 0	Harder >1 and Hardest = 0	Harder $\geq$ 0 and Hardest $\geq$ 1
Linux	*drivers*	446	688	54	227
	fs	52	150	23	23
	net	33	84	11	15
	arch	26	41	4	17
	sound	31	161	13	47
	total	**588**	**1124**	**105**	**329**
Wine		84	253	42	73
PostgreSQL		24	52	6	12
Apache		5	23	4	1
Python-2.7.3		62	151	7	46
Python-3.3.0		49	121	9	40
PHP		6	12	3	3
Grand total		**818**	**1736**	**176**	**504**

Table 3.6: The number of functions with various numbers of Harder and Hardest branches.

function, how many have one Harder branch and then only Simple and Hard branches, implying that all of the state-restoring code is being shared, and how many have multiple Harder branches or have Hardest branches, implying the need for extra blocks of state-restoring code at the end of the function. As shown in Table 3.6, across all systems software, most functions have only one Harder branch, and then the rest of the branches are Simple or Hard. A large number of functions also have only Simple and Hard branches, providing the best case for reduction in code size. Finally, only a few functions have multiple Harder branches or have Hardest branches. These results show that overall the systems code is well suited to the use of the `goto`-based strategy.

3.5 Conclusion

This chapter has focused on the structuring of error handling code in the Linux kernel. The Linux kernel coding style guidelines advocate organizing such code using labels and `gotos`, but a substantial part of the Linux kernel source code as well as other systems code still do not follow this strategy.

We have proposed an automatic transformation that converts error-handling code that is dispersed and duplicated throughout the body of a function into error-handling code that uses the `goto`-based strategy. We have found that our transformation applies to many functions across the systems source code, and that it identifies many opportunities for code sharing.

Chapter 4

Finding Faults in Error Handling Code

Contents

4.1 Summary

Adequate error-handling code is essential to the reliability of any systems software. On an error, such code is responsible for releasing acquired resources to restore the system to a viable state. Omitting such operations leads not only to memory leaks, but also to system crashes and deadlocks. This is a continual problem in ensuring the robustness of system software. Finding such faults is very challenging due to the difficulty of systematically reproducing system errors and the diversity of system resources and their associated resource release operations. To address these issues, over 10 years of research has focused on macroscopic approaches that globally scan a code base for common resource-release operations. Such approaches are notorious for their high rates of false positives, while at the same time, in practice, they leave many faults undetected.

In this chapter, we observe that resource-release operations are often found in error-handling code, and that the choice of resource-release operation may depend on the context in which it is to be used. We propose a novel microscopic approach to finding resource-release faults in systems software, taking into account such software's diversity of resource types and resource-release operations. Rather than generalizing from the results of a complete scan of the source code, our approach achieves precision and scalability by focusing on the error-handling code of each function. Using a tool, Hector, that we have developed based on this approach, we have found over 485 faults in 19 different C systems software projects, including Linux, Python, and Apache, with a false positive rate of 23%, well below the 30% that has been reported in a survey by Coverity to be acceptable to developers. Some of these faults are exploitable by an unprivileged malicious user, making it possible to crash the entire system.

4.2 Motivation

We first present some faults in error-handling code that have been found using Hector. These examples reveal that faults in error-handling code can have an impact that goes beyond just the loss of a few bytes due to an unreleased resource. We then give an overview of error-handling in systems software.

4.2.1 Linux resource-release omission faults

We motivate our work using four representative crashes and memory leaks derived from a variety of faults in Linux error-handling code. Two of these faults were previously found by users; in these cases, the Linux commit logs contain no evidence that the faults were found using other tools. The other two faults were previously unreported; we have reported them to the appropriate maintainers and provided patches.[1] The unreported faults involve rarely used acquisition and release functions that would be unlikely to be reported by existing specification-mining based approaches.

Crash following a resource conflict In January 2009, a user of the Fedora Rawhide (development) kernel found that installing the w83627ehf driver crashed his machine.[2] Figure 4.1 shows an extract of the faulty code. It performs a series of operations, on lines 1, 4, 6, 10, and 13, that may encounter an error. If an error is detected, the function branches to the error-handling code (boxed) on lines 3, 5, 8, 12 and 15, respectively. In the first three cases, the error-handling code correctly jumps to labels at the end of the function that execute an increasing sequence of device unregister operations, according

[1] http://lkml.org/lkml/2012/4/14/41, http://lkml.org/lkml/2012/5/3/230

[2] https://bugzilla.redhat.com/show_bug.cgi?id=483208

to the resource acquisitions that have been performed so far. The error-handling code provided with the ACPI resource conflict check on line 10, however, is faulty, as it jumps to the last label in the function, which just returns the error code. The device remains registered even though it does not exist, and subsequent operations by the kernel on the non-existent device cause the system to crash.

```
 1    err = platform_driver_register(&w83627ehf_driver);
 2    if (err)
 3      goto exit;
 4    if (!(pdev = platform_device_alloc(...)))
 5      goto exit_unregister;
 6    err = platform_device_add_data(...);
 7    if (err)
 8      goto exit_device_put;
 9    ...
10    err = acpi_check_resource_conflict(&res);
11    if (err)
12      goto exit;          Omission fault
13    err = platform_device_add_resources(pdev, &res, 1);
14    if (err)
15      goto exit_device_put;
16    ...
17  exit_device_put:
18    platform_device_put(pdev);
19  exit_unregister:
20    platform_driver_unregister(&w83627ehf_driver);
21  exit:
22    return err;
```

(From drivers/hwmon/w83627ehf.c, `sensors_w83627ehf_init`)

Figure 4.1: w83627ehf driver containing an omission fault

Note that the error-handling code starting on line 3 correctly does not release any resources. Thus, flow and path sensitivity are necessary to determine when release operations are needed.

The fault was introduced a few weeks before it was detected, when the initialization functions of 14 `hwmon` drivers were updated to check whether the resources required by the device conflicted with those used by ACPI. In three of these drivers, the newly added error-handling code jumped to the wrong label, causing all of the resource-release operations to be skipped.

Crash following the detection of an incorrectly configured driver In December 2008, a user of a Cyclades serial board reported that when the board was used in a particular configuration, installing it would fail, and then the board could not be installed again until the machine was rebooted.[3] Subsequently accessing the driver via the `/proc` interface would also crash the machine. The problem was traced to error-handling code that did not release any of the locally allocated resources.

The code associated with this fault is similar to that of the previous one: as execution proceeds within the initialization function `cy_pci_probe`, error-handling code is implemented as jumps to an increasingly complex sequence of resource-release operations. The error-handling code containing the fault, however, just returns with an error value, and thus none of the needed cleanup actions are performed.

This fault was introduced in a major reorganization of the code in May 2007, that added some new resource acquisitions and corresponding error-handling code containing resource releases. The faulty block of error-handling code was not updated to use the error-handling code required by its new context.

[3] https://bugzilla.kernel.org/show_bug.cgi?id=12137

Memory leak in the handling of invalid user inputs Using Hector, we found a previously unreported memory-release omission fault in the autofs4 IOCTL function. As shown in Figure 4.2, the error-handling code starting on line 11 does not release the resource *param* that was previously released in the error-handling code starting on lines 6 and 8. Using a 9-line program, we were able to repeatedly invoke the IOCTL function with an invalid command argument, and use up almost all of the 2GB of memory on our test machine in under one minute. This fault is exploitable by an unprivileged user who has obtained the `CAP_MKNOD` capability. We have verified that an unprivileged user can obtain this capability using a previously reported NFS security vulnerability.[4] Using this vulnerability, an attacker, having usurped the IP address of an NFS client, is able to create an autofs4 device file accessible to unprivileged users on the NFS server. Then, the attacker, connected as a unprivileged user on each NFS client machine, can exploit the autofs4 fault to exhaust all the memory of each client machine by issuing invalid IOCTL calls, preventing other programs from allocating memory and causing them to fail in unpredictable ways. Reclaiming the lost memory requires rebooting each affected machine.

The resource-release omission has been present since the code was introduced into the Linux kernel in version 2.6.28 (2008), and is still present in the most recent version, 3.6.6.

```
 1   param = copy_dev_ioctl(user);
 2   if (IS_ERR(param))
 3     return PTR_ERR(param);
 4   err = validate_dev_ioctl(command, param);
 5   if (err)
 6     goto out;
 7   if (cmd == AUTOFS_DEV_IOCTL_VERSION_CMD)
 8     goto done;
 9   fn = lookup_dev_ioctl(cmd);
10   if (!fn) {                                  Omission fault
11     AUTOFS_WARN("...", command);
12     return -ENOTTY;
13   }
14   ... /* more error-handling code jumping to out */
15 done:
16   if (err >= 0 && copy_to_user(user, param, ...))
17     err = -EFAULT;
18 out:
19   free_dev_ioctl(param);
20   return err;
```

(From fs/autofs4/dev-ioctl.c, `_autofs_dev_ioctl`)

Figure 4.2: Autofs4 code containing an omission fault

Memory leak in the handling of an invalid file system Using Hector, we found a previously unreported memory-release omission fault in the initialization of the ReiserFS file system journal. The omission occurs when there is an attempt to mount the file system and some parameters stored within the file system are found to be invalid. As shown in Figure 4.3, the error-handling code starting on line 16 does not release `bhjh` that was previously released in the error-handling code starting on line 9. The fault can be triggered by an unprivileged user, if such a user mounts a file system from an external disk drive that has been previously formatted with invalid parameters. On a modern Linux distribution, such a file system is normally mounted using autofs, which imposes a delay between file-system mounts, thus limiting the possible damage. Older systems, however, may be configured to

[4] http://lwn.net/Articles/328594/

allow the user to mount such a file system directly. In the latter case, as an unprivileged user, we were able to use up almost all of the 2GB of memory on our test machine within an hour, by repeatedly mounting the file system.

The fault was introduced in Linux 2.6.24 (2008), and is still present in the most recent version, 3.6.6.

```
 1  bhjh = journal_bread(sb, ...);
 2  if (!bhjh) {
 3    reiserfs_warning(sb, ...);
 4    goto free_and_return;
 5  }
 6  jh = (struct reiserfs_journal_header *)(bhjh->b_data);
 7  if (is_reiserfs_jr(rs)
 8      && (le32_to_cpu(...) != sb_jp_journal_magic(rs))) {
 9    reiserfs_warning(sb, ...);
10    brelse(bhjh);
11    goto free_and_return;
12  }
13  journal->j_trans_max = le32_to_cpu(...);
14  ...
15  if (check_advise_trans_params(sb, journal) != 0)
16    goto free_and_return;                          Omission fault
17  journal->j_default_max_commit_age = journal->j_max_commit_age;
18  ...
19  brelse(bhjh);
20  ...
21  free_and_return: ...
```

(From fs/reiserfs/journal.c, `journal_init`)

Figure 4.3: ReiserFS code containing an omission fault

4.3 Systems error-handling code

We have already described in Chapter 2 how the number of functions with error-handling code is increasing versions by versions of the applications. To better understand the current state of error-handling code in systems software, we analyze these functions in more detail. In this Chapter, we consider the amount of code that is found within functions that contain error-handling code, the kinds of functions that contain error-handling code, and the kinds of errors that are handled. Our study primarily focuses on the `drivers`, `sound` (sound drivers), `net` (network protocols), and `fs` (file systems) directories of Linux 2.6.34, but we also consider a selection of other widely used systems software, previously summarized in Table 2.1.

4.3.1 Amount of code containing error-handling code

We define a *block of error-handling code* as the code executed from when a test for an error is found to be true up to the point of returning from the containing function. The block may include *goto*s. Figure 4.4 shows the percentage of code found within functions that contain zero, one, or more blocks of error-handling code. Depending on the project, 28%-69% of the code is within functions that contain at least one block of error-handling code and 16%-43% of the code is within functions that contain multiple blocks of error-handling code (shown below the horizontal dashed lines). The latter functions are of particular interest, because in such functions, it is possible to identify resource-release omission faults by comparing the various blocks of error-handling code to each other and determining

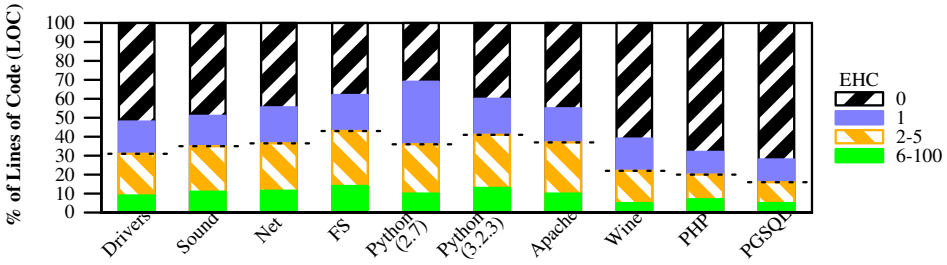


Figure 4.4: Percentage of code found within functions that have 0 or more blocks of error-handling code

whether they are consistent. Our examples in Section 4.2.1 come from functions containing 7-14 blocks of error-handling code. The fault in the fourth example was introduced when a function was reorganized, and new error-handling code was introduced, showing the difficulty of maintaining such complex code.

4.3.2 Role of code containing error-handling code

The role of the functions containing error-handling code determines how often these functions are likely to be executed, and thus the impact that any faults in their error-handling code are likely to have. Determining the role of every function that contains error-handling code in a large code base is impractical. In the case of Linux, we can obtain a partial view by exploiting the fact that many kinds of services, including drivers, network protocols and file systems, are represented in the kernel as statically declared structures, containing various callback functions. The names of the fields in these structures then suggest the role of the functions stored within them. Table 4.1 shows the average number of blocks of error-handling code in the functions stored in the fields related to basic system interactions in the most common of these structures.

Initialization functions, stored in fields named *e.g.*, `probe` and `attach`, typically contain many blocks of error-handling code. Faults in such functions can crash the machine or make it impossible to use a device, as illustrated by our examples in Section 4.2.1. Nevertheless, such faults are unlikely to lead to major memory leaks, even in the hands of a malicious user, because initialization functions are often executed only once and the initialization process may require root privilege. More frequently executed functions include IOCTL functions, as illustrated by our third example, and read/write functions. IOCTL functions on average also contain several blocks of error-handling code, while read and write functions contain fewer. The latter two kinds of functions can typically be executed many times by an unprivileged user, and thus faults in such code can open the system to attacks. Finally, error-handling code is rare in close and shutdown functions, as such operations rarely allocate resources and are expected to always be successful.

4.3.3 Kinds of errors encountered

The impact of faults in error handling code is determined in part by how often the handled errors occur. It is difficult to automatically determine the source of all the possible errors that may be encountered. Nevertheless, 48% of the error-handling code in Linux `drivers`, `sound`, `net`, and `fs`, returns integer error codes, understood by the user-level standard library function `perror`, to

Table 4.1: Average (and maximum) number of blocks of EHC in the most common kinds of callback functions

Structure (Static instances)	Direct-ories	probe, etc.	open	ioctl	read	write	remove, close etc.
file_operations (761)	DSNF	–	< 1 (10)	3.0 (62)	1.8 (12)	2.4 (57)	–
platform_driver (727)	DSN	4.6 (19)	–	–	–	–	< 1 (2)
pci_driver (557)	DS	5.6 (25)	–	–	–	–	< 1 (2)
net_device_ops (378)	DN	< 1 (2)	1.9 (11)	2.2 (24)	–	–	< 1 (2)
usb_driver (231)	DS	3.9 (21)	–	2.5 (5)	–	–	< 1 (2)
i2c_driver (210)	DS	3.6 (13)	–	–	–	–	< 1 (2)
seq_operations (123)	DNF	< 1 (4)	–	–	–	–	0 (0)
fb_ops (122)	D	–	< 1 (2)	4.6 (32)	2.0 (3)	2.6 (6)	–

D = drivers, S = sound, N = net, F = fs

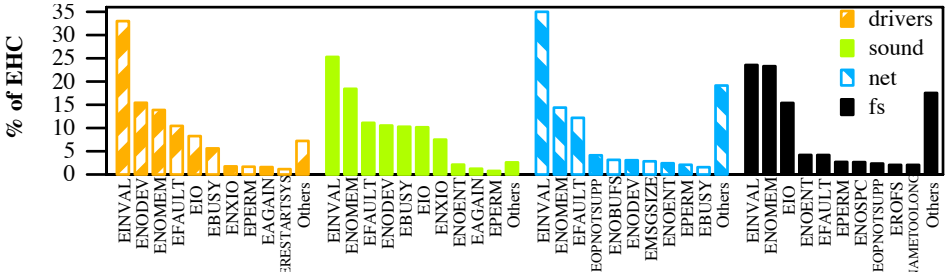


Figure 4.5: Distribution of integer error-code return values

indicate the error cause. We rely on these error codes to obtain an overview of the reasons for the errors encountered in Linux.

Figure 4.5 shows the percentage of the considered blocks of error-handling code that involve the various constants used in each of the Linux `drivers`, `sound`, `net`, and `fs` directories, focusing on the top 10 such constants used in each case. The errors associated with these values differ in their source and likelihood. `EINVAL` is the most common value throughout and indicates that the function has received invalid arguments. These arguments may depend on values received from applications or hardware, allowing invalid values from the user level or from hardware malfunctions to trigger a fault. `ENOMEM`, indicating insufficient memory, is the next most common value in most cases. Running out of kernel memory is unlikely, except in the case of low-memory embedded systems or unless the system is already under a memory-leak based attack, and thus faults in such blocks of error-handling code are unlikely to be triggered in an otherwise well-programmed system. For `drivers`, the second most common constant is `ENODEV`, which is also common in `sound`. `ENODEV` indicates the unavailability of a device, as may be triggered by defective hardware. Another common constant is `EFAULT`, indicating a bad address. `EFAULT` is commonly used by functions copying data to or from user space, where an address comes from user level. A malicious application can easily construct an invalid address, making the correctness of the associated error-handling code critical. Finally, a common value, especially for `fs`, is `EIO`, indicating an I/O error. Such an error can again be triggered by defective hardware, such as a disk or other device.

4.4 Algorithm

The goal of our algorithm is to identify inconsistencies in the releasing of resources in a function's error-handling code. Such inconsistencies are reported as faults. The algorithm is *microscopic* in

that it is primarily based on intraprocedural information. It is made resistant to false positives in the information about resource acquisition and release operations by following a strategy of correlating information about acquisition and release operations within each analyzed function.

The input to our algorithm is a function definition where some statements have been already annotated as being resource acquisitions or releases. These annotations are performed by a *preprocessing phase*, which is orthogonal to our algorithm. The preprocessing phase must also annotate each acquisition or release with an expression representing the affected resource, and annotate some basic blocks as being the start of a block of error-handling code. This preprocessing can be done in any manner; we present an implementation in Section 4.5.1. Our algorithm then works on the control-flow graph (CFG) of the provided function definition, annotated with the results of the preprocessing phase. The CFG is interprocedural, but only takes into account functions defined in the same file, and does not unfold recursive calls. Each node in the CFG contains a single atomic statement (assignment, function call, etc.), or the header of a control-structure, such as a loop or a conditional. An edge connects such a term to one that may be executed immediately after it. As a running example, we use the code shown in Figure 4.1, focussing on the resource *pdev*. Figure 4.6(a) shows a portion of this code's CFG, starting from line 4, where *pdev* is first initialized. Nodes are numbered according to the corresponding line numbers. A branch to the right enters error-handling code.

Given the annotated CFG, the first step of the algorithm connects resource releases in error-handling code to the resource acquisitions that can reach them. This is done by what amounts to an intraprocedural live-variable analysis, in which acquisitions are considered to be definitions and releases in error-handling code are considered to be the only uses. In the example (Figure 4.6(a)), the release of *pdev* on line 18 (solid node), which is part of error-handling code, is found to be live at the acquisition of *pdev* on line 4 (shaded node), by following in reverse the dashed edges.

Next, for each acquisition that is found to have at least one "live" release, the algorithm walks forwards through the function's (CFG), collecting each possible subset of the CFG nodes that represents a path from the acquisition to any block of error-handling code. For our example, there are four such paths, shown in Figure 4.6(b-e). The resulting set of paths is then divided into a set of *exemplars*, which for some resource contain both an acquisition of the resource and a release of the resource in error-handling code, and a set of *candidate faults*, which contain an acquisition but no corresponding release in error-handling code (annotated releases prior to the error-handling code are possible). Exemplars are truncated just before the block of error-handling code. In our example, the paths in Figure 4.6(c and e) represent exemplars, because they contain the release operation, while the paths in Figure 4.6(b and d) represent candidate faults. We refer to the resource acquired at the beginning of any such path as the *associated resource*.

The algorithm then compares each candidate fault to the exemplar closest to it in the code, to determine whether the exemplar provides evidence that the candidate fault should release its associated resource in its error-handling code. In our example, we consider the exemplar in Figure 4.6(c) and the candidate fault in Figure 4.6(d). A fault report is generated for the candidate fault if the following conditions all hold:

1. The candidate fault does not return the resource.
2. The candidate fault and the exemplar both allocate their associated resource in the same way. These allocations may, but need not, occur at the same line of code.
3. Any operation in the candidate fault prior to the error-handling code that is annotated as a release of the associated resource also occurs in the exemplar.

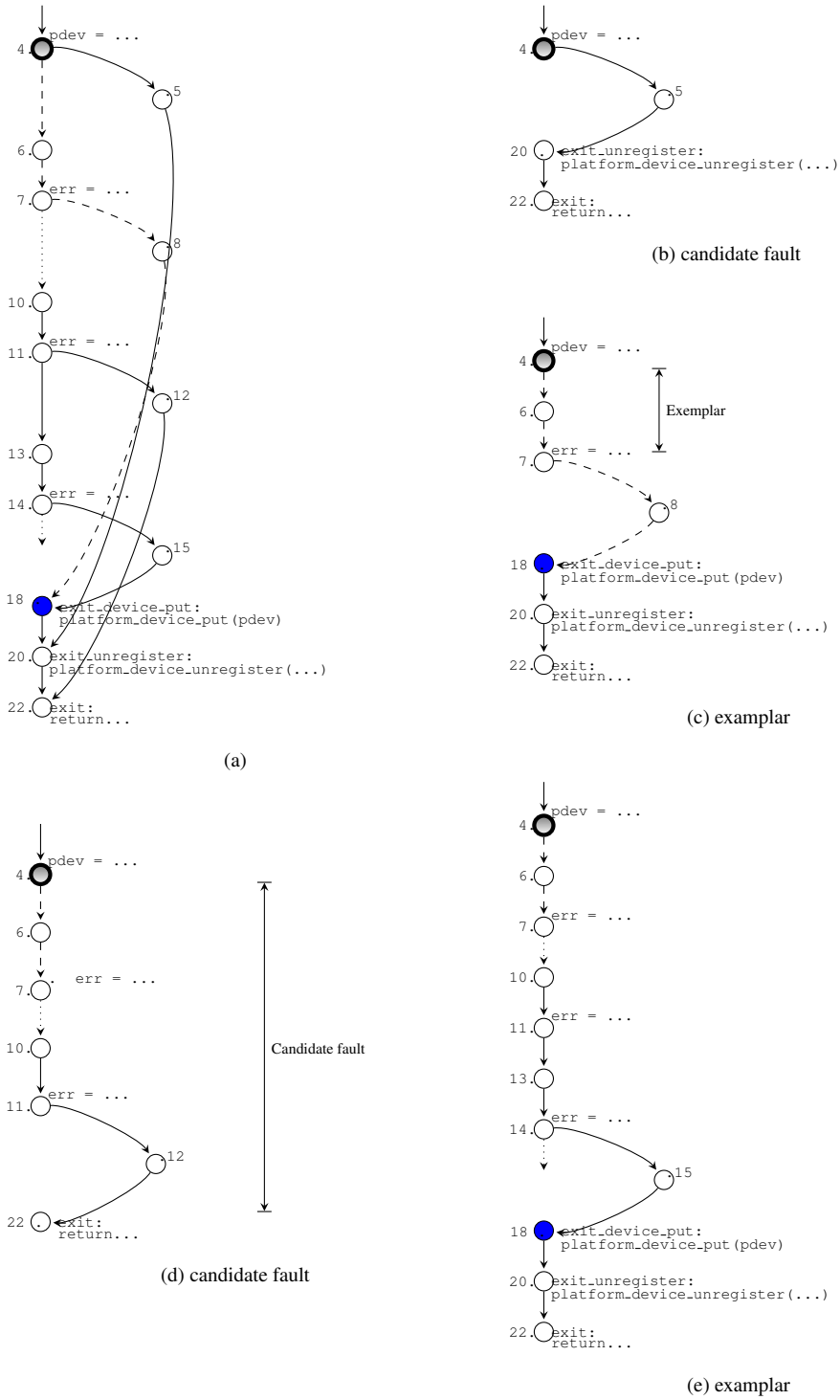


Figure 4.6: CFG and paths for Figure 4.1

These conditions are motivated as follows. If the candidate fault returns the resource (condition 1), then the resource should not be released, and indeed the block at the end of the candidate fault

is probably not really error-handling code. Condition 2 results from the observation that we only have evidence that the resources associated with the candidate fault and the exemplar should be released in the same way if they were allocated in the same way. Finally, if a supposed release operation is followed in the exemplar by another release of the same resource in error-handling code, then the supposed release operation does not really perform a release (condition 3). The set of generated reports is then returned as the output of the algorithm. In our example, the candidate fault (Figure 4.6(d)) does not return *pdev*, it allocates its associated resource using the same function as the exemplar (Figure 4.6(c)), and it does not contain any release of *pdev*. Thus, the omission of the release of *pdev* in the block of error-handling code starting on line 12 is correctly reported as a fault.

As a second example, consider the code in Figure 4.3 and the acquisition of *bhjh* on line 1. One path from the acquisition leads through the error-handling code starting on line 9. This error-handling code releases *bhjh* using *brelse*, and so the path is considered to be an exemplar. Suppose that another path from the acquisition leads through the call to *brelse* on line 19 to a later block of error-handling that does not release *bhjh*. This path would be considered to be a candidate fault. However, it meets only the first two of the conditions for reporting a fault; it does not satisfy the third condition because it contains a release of *bhjh* that does not appear in the (truncated) exemplar. The algorithm correctly concludes that the call to *brelse* annotated as a release on line 19 is an actual release of *bhjh*, and thus no further release is needed.

4.5 Implementation

We have validated our algorithm by implementing a tool, Hector. Hector consists of around 3500 lines of OCaml code, excluding the C parser and abstract syntax, which we have borrowed from the open-source C-code transformation tool Coccinelle.[5] Creating this implementation requires implementing a preprocessing phase and instantiating the algorithm with various analysis strategies.

4.5.1 Preprocessing phase

Preprocessing requires identifying and annotating error-handling code, resource acquisitions, and resource releases. Due to the nature of the C language, this must necessarily be done using heuristics. Our heuristics mostly rely on intraprocedural information, making the implementation highly scalable. We have already described how error-handling code is identified in Chapter 3. Here we discuss present process of annotating resource acquisitions and release.

A *resource* is typically represented by a collection of information, and is thus implemented by a pointer to a structure or buffer.[6] Resource acquisition and release are typically complex operations, and are thus implemented by function calls. Hector recognizes an acquisition as a function call that returns a pointer-typed value, either directly or via a reference argument ($\&x$), and recognizes a release as the last operation on a resource in a path in the CFG. The result of a release should not be tested, as release operations do not normally report error codes. Finally, the preprocessors ignore operations that have constant string arguments, as such operations are typically debugging code. A resource can be

[5] http://coccinelle.lip6.fr/

[6] File descriptors, as obtained by `open`, are an exception, being represented as integers, and thus Hector does not detect file descriptor release omissions. `open` is, however, now rarely used, in favor of the more modern `fopen`, which provides richer functionalities. `fopen` returns a pointer. The Linux kernel also uses pointers to represent its more primitive file objects.

released by some locally defined operation other than the omitted resource-releasing operation either before or inside the error-handling code. To improve accuracy, within the file containing the analyzed function, the preprocessors identify resource-release operations interprocedurally. A function call that has an acquired resource as an argument and whose definition contains a release of that resource, according to the above criteria, is also considered to be a release operation.

In Figure 4.7, the error-handling code on lines 2-4 uses `kfree(fw)` to release `fw`. However, the subsequent error-handling code, on lines 7-8, uses `free_fw` to release `fw`. The function `free_fw` is defined in the same file and its definition contains `kfree(fw)` to release the resource `fw`. Therefore, the preprocessors identify `free_fw` as a resource-release operation interprocedurally. Thus, `free_fw` operation is also used to release `fw` in line 11 prior to the error-handling code on line 14, and thus no call to `kfree(fw)` is needed in either case.

```
1   if (...) {
2     ...
3     kfree(fw);
4     return −ENOMEM;
5   }
6   if (...) {
7     free_fw(fw);
8     return −EFAULT;
9   }
10  ...
11  free_fw(fw);
12  ...
13     if (...)
14       return err;
```

Figure 4.7: Release of a resource via a locally defined function (to be more like the others). (Extract of vx_hwdep_dsp_load, from Linux-2.6.34/sound/drivers/vx/vx_hwdep.c)

Some kinds of resources, notably locks, are not acquired and released according to the above patterns, but instead using a function that takes the resource as an argument, or even takes no arguments. To account for these cases, we also consider a function call having at most one argument, where the argument, if any, has pointer type and is not involved in an earlier resource acquisition, as being a resource acquisition. The corresponding release operation must occur in a block of error-handling code and must include the same argument value, if any, as verified by checking that the corresponding arguments have the same set of reaching definitions.

Finally, in some cases a resource is released as a side-effect of another operation. In Figure 4.8, the resource *kctl* is acquired on line 4. On line 12, *kctl* is passed to the function *add_control_to_empty*, which is the last operation on *kctl* before the return on line 13. This call would not normally be considered a release, because its value is tested. Nevertheless, *kctl* is never again referenced on any execution path following this call, neither on the success nor the failure of the test, and thus the call is considered to either release *kctl* or store it in some way that makes a subsequent release in error-handling code unnecessary. The latter is indeed the behavior of this function.

4.5.2 Instantiation of the algorithm

The algorithm needs to connect resource-release operations to the corresponding possible resource acquisitions, and then to collect the paths in which an acquired resource is live. For connecting the operations, Hector uses a backwards dataflow analysis that takes into account alias information. Concretely,

```
1  namelist = kmalloc(...);
2  if (! namelist) { ... }
3  ...
4  kctl = snd_ctl_new1(&mixer_selectunit_ctl, cval);
5  if (! kctl) {
6    kfree(namelist);
7    ...
8    return -ENOMEM;
9  }
10 kctl->private_value = (unsigned long)namelist;
11 ...
12 if ((err = add_control_to_empty(state, kctl)) < 0)
13   return err;
14 return 0;
```

Figure 4.8: Release a resource via a side-effect of another operation.
(Extract of parse_audio_selector_unit, from Linux-2.6.34/sound/usb/usbmixer.c)

the alias analysis considers statements of the form $y = x$, $y{-}{>}fld = x$, and $y = f(\ldots, x, \ldots)$ as creating a possible alias from y to x. For collecting the paths, Hector uses a forward path-sensitive dataflow analysis, again taking into account alias information. In both cases, the analyses are flow sensitive and purely intraprocedural.

The need for path sensitivity is illustrated by the use of *pdev* in Figure 4.1. It was noted in Section 4.4 that the execution path starting with line 4 and passing through the block of error-handling code starting on line 12 is missing a release of *pdev*. The execution path starting on line 4 and passing through the block of error-handling code starting on line 5 is likewise missing a release of *pdev* (cf. Figure 4.6(b)). However, the path-sensitivity of the path collection process implies that the latter path is not reported as a fault, because it includes a successful test that *pdev* is null, implying that its value is different from the one (line 4) for which a release is needed.

The need for alias analysis arises when an execution path beginning with an acquisition of some resource x contains an assignment such as $y{-}{>}fld = x$ that makes x reachable from some other resource y. Alias information makes the path collection process aware that x may either be released directly or be released via a release of y, thus allowing a path that contains either resource release to be considered to be an exemplar. In Figure 4.9, on line 6, the structure `pr` is stored in the structure `device`. The subsequent error-handling code on lines 9-10 releases `pr` using `device` on line 14 rather than explicitly calling `kfree`.

```
1   if (...) {
2     kfree(pr);
3     return ;
4   }
5   ...
6   device->driver_data = pr;
7   ...
8   if (...) {
9     ...
10    goto err_remove_fs;
11  }
12  ...
13  err_remove_fs:
14    acpi_processor_remove_fs(device);
```

Figure 4.9: Release a resource via other pointer.
(Extract of acpi_processor_add, from Linux-2.6.34/drivers/acpi/processor_driver.c)

Finally, the need for flow sensitivity arises when a resource is acquired and released more than once within a single function. This is often the case of locking in systems code.

4.5.3 Formal Description of the Algorithm

Now, we formally describe our algorithm to detect resource-release omission faults.

A *path* in a CFG is a sequence of edges $\{e_1, \ldots, e_m\}$ such that for each $i < m$, the target of e_i is the same as the source of e_{i+1}. For a given path p, p_0 is the first node in the path and p_i is the node that is the target of the edge e_i.

The following definitions relate to properties of the nodes and paths in a CFG:

- $\mathsf{code}(n)$ is the set of subterms of the term found at node n.
- $\mathsf{calls}(x, n)$ is the set of elements of $\mathsf{code}(n)$ that have the form of a function call with x as one of the arguments.
- $\mathsf{aliases}(x, n)$ is the set of variables that may become aliases of x at node n. Concretely, $\mathsf{aliases}(x, n) = \{y \mid y = x \in \mathsf{code}(n) \vee y\text{->}\mathit{fld} = x \in \mathsf{code}(n) \vee y = f(\ldots, x, \ldots) \in \mathsf{code}(n)\}$.
- $\mathsf{paths}(n_1, n_2, V_{write})$ is the set of paths from node n_1 to node n_2 such that the target node of each edge in the path does not write the variables in V_{write}.
- $\mathsf{paths3}(n_1, n_2, n_3, V_{write}) =$
  $\{p \mid p \in \mathsf{paths}(n_1, n_3, V_{write}) \wedge n_2 \in p\} \cup$
  $\{p \mid \mathsf{paths}(n_2, n_1, \emptyset) \neq \emptyset \wedge p \in \mathsf{paths}(n_1, n_3, V_{write})\}$

$\mathsf{paths3}(n_1, n_2, n_3, V_{write})$ contains both paths from n_1 to n_3 that pass through n_2 and paths from n_1 to n_3 that are reachable from n_2. This allows paths3 to capture cases where a path crosses into a block of error-handling code, whose associated test is represented by the node n_2, and the case where the entire path is found inside the block of error-handling code itself.

The following definitions relate to a given resource in the candidate set of a given block of error-handling code:

- x is the variable pointing to the omitted resource.
- n_{fault} is the node containing the return at the *bottom* of the block of error handling code.
- $test_{fault}$ is the node containing the test expression of the conditional whose branch is the block of error handling code.
- $n_{exemplar}$ is the node at the *top* of the block of error handling code that is serving as the exemplar for the suspected release omission.

For example, consider the block of error-handling code starting on line 12 of Figure 4.1 and the element `pdev` of its candidate set. One possible exemplar for the release of `pdev` is the block of error-handling code starting on line 8. For convenience, we again assume that the node numbers in the CFG are the same as the line numbers in the code. In this case, x is `pdev`, n_{fault} is 21, $test_{fault}$ is 11, and $n_{exemplar}$ is 8.

Finally, the following definitions relate to the interaction between nodes and paths in the CFG and the released resource x.

- A-fault is the set of nodes that contain acquisitions of x and reach n_{fault} through $test_{fault}$.
  A-fault $= \{d \mid \exists exp : x \ = \ exp \in \mathsf{code}(d) \wedge \mathsf{paths3}(d, n_{test}, n_{fault}, \{x\})$
- A-exemplar is the set of nodes that contain acquisitions of x and reach $n_{exemplar}$.
  A-exemplar $= \{d \mid \exists exp : x \ = \ exp \in \mathsf{code}(d) \wedge \mathsf{paths}(d, n_{exemplar}, \{x\})$
- D-fault is the set of nodes that contain any form of definition of x and reach n_{fault} through $test_{fault}$.
- D-exemplar is the set of nodes that contain any from of definition of x and reach $n_{exemplar}$.
- $\mathsf{Vref}(x, p)$ is the set of aliases for x, including x itself, at the end of path p.
  $\mathsf{Vref}(x, p) = \{x\} \cup \{y \mid \exists p_i : y \in \mathsf{aliases}(x, p_i) \wedge \forall j > i : \forall exp : y \ = \ exp \notin \mathsf{code}(p_j)\}$
- $\mathsf{Release}(x, p)$ is the set containing the node within the path p containing the last function call having an element of $\mathsf{Vref}(x, p)$ as an argument, such that there is no further reference to any element of $\mathsf{Vref}(x, p)$ within p. If there is no such node, $\mathsf{Release}(x, p)$ is empty.
  $\mathsf{Release}(x, p) = \{p_i \mid \exists y \in \mathsf{Vref}(x, p) : \mathsf{calls}(y, p_i) \neq \emptyset \wedge \forall j > i : \mathsf{Vref}(x, p) \cap \mathsf{code}(p_j) \neq \emptyset\}$

For example, for the block of error-handling code starting on line 12 of Figure 4.6(d), we consider the released resource `pdev` and the nodes $test_{fault}$, *i.e.*, 11, and n_{fault}, *i.e.*, 22, in computing A-fault. The only acquisition of `pdev` that reaches node 22 through node 11 is the one on line 4. This acquisition reaches $n_{exemplar}$, *i.e.*, 8, in Figure 4.6(c) as well. Thus, both A-fault and A-exemplar are $\{4\}$. Between the acquisition of `pdev` on line 4, $test_{fault}$ on line 11, and n_{fault} on line 22, there is no function call having `pdev` as an argument, and thus the set Release is empty.

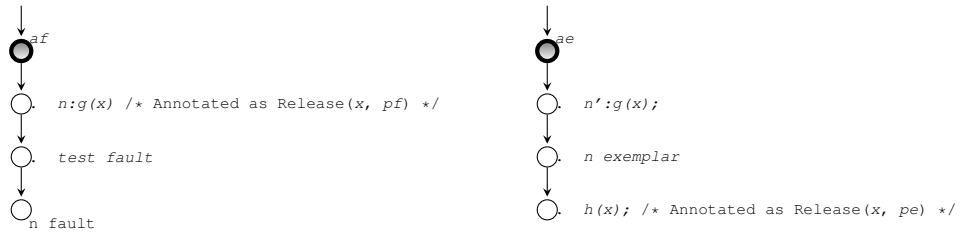
Using these definitions, for a given cadidate fault, represented by the nodes $test_{fault}$ and n_{fault}, and for each exemplar, represented by the node $n_{exemplar}$, we classify the candidate fault as Fault or Unknown, as described by the rules below. These rules are considered in order until one of them produces a result. A candidate fault is reported to the user as a fault if it is classified as Fault for *any* exemplar. Otherwise, it is considered to be a false positive, and is not reported to the user. These rules are formalized here.

- If $x \in \mathsf{code}(n_{fault})$, then Unknown.

  The first condition to be a fault report in our algorithm in Section 4.4 is that a candidate fault does not return the resource. If x is used to compute the return value, it cannot be freed. If this case is selected for any exemplar, it is selected for all exemplars, and so the algorithm will detect that this report is a false positive. This case is not illustrated by our examples.

- If $\{\mathsf{code}(n) \mid n \in \text{D-fault}\} \neq \{\mathsf{code}(n) \mid n \in \text{D-exemplar}\}$, then Unknown.

  The second condition to be a fault report in our algorithm is that the candidate fault and the exemplar both allocate their associated resource in the same way. If the sets of acquisitions reaching the candidate fault and the exemplar are different, then the exemplar provides no information about whether the resource should be released. The analysis of acquisitions takes into



(a) candidate fault (*Pf*)

(b) exemplar (*Pe*)

Figure 4.10: An annotated resource-releasing operation in the candidate fault appears in the exemplar before the actual releasing operation.

account not only explicit acquisitions, but also information that can be inferred from test expressions, via path sensitivity. For example, in Figure 4.1, at line 4, a resource is allocated by calling `platform_device_alloc`, and the result is stored in the variable `pdev`. From the analysis of the complete function, we find that `pdev` is not released in the block of error-handling code starting on line 5. However, the enclosing conditional tests whether `pdev` is NULL, and thus within the error-handling code we consider that the value of `pdev` is to NULL, as identified by the test, not at the result of the call to `platform_device_alloc`. Therefore, the omission of the release of `pdev` is not considered to be a fault.

- If the following condition is satisfied, then Fault.

  $\exists a_f \in$ A-fault : $\exists p_f \in \mathsf{paths3}(a_f, test_{fault}, n_{fault}, \{x\})$:
  $\exists a_e \in$ A-exemplar : $\exists p_e \in \mathsf{paths}(a_e, n_{exemplar}, \{x\})$:
  $\forall n \in \text{Release}(x, p_f)$:
  $\exists y \in (\mathsf{Vref}(x, p_f) \cap \mathsf{Vref}(x, p_e))$: $\exists n' \in p_e$:
  $\mathsf{calls}(y, n) \cap \mathsf{calls}(y, n') \neq \emptyset \wedge \mathsf{paths}(n', n_{exemplar}, \{y\}) \neq \emptyset$

  Figure 4.10 illustrates this condition, which corresponds to the third condition of our algorithm, as presented in Section 4.4. In this case a call that was annotated by the preprocessing phase as a resource-releasing operation with respect to n_{fault} also appears in an execution path that ends in $n_{exemplar}$. Thus, the call must not actually release the resource, because in $n_{exemplar}$ it is followed by another resource-release operation. There is no subsequent reference to x on the path to n_{fault}, by the definition of Release, and thus this path contains a resource-release omission fault.

  This case is also illustrated by the real example in Figure 4.6. There, we have A-fault $= \{4\}$, $n_{fault} = 22$, and Release $= \emptyset$. For any $a \in$ A-fault and $p \in \mathsf{paths}(a, n_{fault}, \{x\}, \emptyset)$, both of which are nonempty, $\text{Release}(x, p) = \emptyset$, and thus the omission of a release of `pdev` here is correctly considered to be a fault. In Figure 4.10(a) the annotated release operation, `g(x)` in the candidate fault also appear before annotated release operation `h(x)` in the examplar (Figure 4.10(b). Thus, `g(x)` in the candidate fault will be not considered as releasing operation..

- If the following condition is satisfied, then Fault.

  $\exists a \in$ A-fault : $\exists p \in \mathsf{paths3}(a, test_{fault}, n_{fault}, \{x\})$:
  $\exists n \in \text{Release}(x, p)$: $\mathsf{paths}(n, test_{fault}, \{x\}) \neq \emptyset \wedge$
  $\exists n'$: $\mathsf{paths}(n, n', \{x\}) \neq \emptyset \wedge x \in \mathsf{code}(n')$

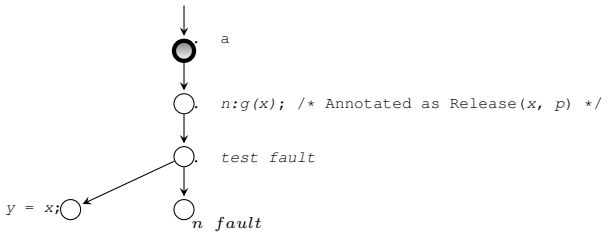


Figure 4.11: An annotated resource-releasing operation followed by an operation on the released resource.

Intuitively, if a call at n that appears to be a resource-releasing operation with respect to n_{fault} is followed along any path by a reference to the released resource, then it is considered that the call at n did not release the resource. There is thus no resource releasing operation for x in the path to n_{fault}, and thus there is a fault. In Figure 4.11 the annotated release operation, `g(x)` is followed along a path by an operation, `y = x;` on the released resource `x`. Therefore, the annotated release operation will not be as an actual releasing operation.

- Otherwise, Unknown.

4.6 Ranking reports

To help the user focus on the reports that are the most likely to represent real faults, a standard approach, taken by *e.g.,* Engler *et al.* [27], PR-Miner [46], Le Goues and Weimer [44], *etc*, is to rank the reports in some way. Previous approaches have ranked reports based on some combination of support and confidence. We propose a novel ranking strategy that reflects the properties of error-handling code.

The main cause for false positives in our approach is the failure of the heuristics in the preprocessing phase. The preprocessing phase may identify a block of code as error-handling code when actually it represents a success condition, consider a call to be an acquisition even though it does not acquire any resources, or overlook a call as a release even though it does release the resource. We propose a strategy for ranking the reports as *high* or *low* confidence that takes these possibilities into account.

The ranking strategy gives a fault report a *high* rank when the block of error-handling code containing the fault is both preceded in the CFG by a block of error-handing code that releases the resource and followed in the CFG by a release of the resource, whether or not in error-handling code. The former indicates that a release has been necessary, while the latter indicates that the resource has not yet been released. The ranking strategy gives a report that is only somewhere followed in the CFG by a block of error-handling code releasing the resource a *low-initial* rank, as it is possible that in this case a release is not yet needed. Finally, the ranking strategy gives a *low-final* rank to a report that is somewhere preceded in the CFG by a block of error-handling code that releases the resource but is not followed in the CFG by any release of the resource, as in this case the resource may have been released in some undetected way already.

For example, in Figure 4.1, the faulty block of error-handling code starting on line 12 does not

release `pdev`, while the preceding and following blocks of error-handling code, starting on lines 8 and 15 respectively, do release this resource. The fault is thus ranked *high*.

4.7 Experimenting with Hector

The goals of the experiments with Hector are 1) to determine its success in finding faults in systems code, 2) to compare the results obtained with those of related approaches, 3) to assess the potential impact of the identified faults, 4) to understand the reason for any false positives and false negatives, 5) to understand the benefits of analysis strategies and 6) to understand the scalability of the approach. We evaluate Hector on the large, widely used open-source infrastructure software projects previously described in Table 2.1.

4.7.1 Found faults

As shown in Table 4.2, Hector generates a total of 624 reports for all of the projects. We manually investigated all of them and found that 485, from 296 different functions, represent actual faults. There are 139 false positives. We study them further in Section 4.7.5.

Table 4.2: Faults and containing functions (Fns)

	Reports (Fns)	**Faults** (Fns)	**Faults** per EHC	**Impact**		
				Resource leak	Dead lock	Debug
Linux `drivers`	**293** (180)	**237** (152)	0.0026	217	7	13
Linux `sound`	**32** (19)	**19** (13)	0.0018	16	0	3
Linux `net`	**13** (13)	**7** (7)	0.0005	7	0	0
Linux `fs`	**47** (34)	**22** (17)	0.0012	17	2	3
Python (2.7)	**17** (13)	**13** (11)	0.0007	13	0	0
Python (3.2.3)	**22** (13)	**20** (12)	0.0023	20	0	0
Apache httpd	**5** (5)	**3** (3)	0.0012	3	0	0
Wine	**31** (19)	**30** (18)	0.0009	30	0	0
PHP	**16** (13)	**13** (10)	0.0053	13	0	0
PostgreSQL	**8** (5)	**7** (4)	0.0010	7	0	0
Samba4	**13** (9)	**12** (8)	0.0009	12	0	0
ALSA-driver	**12** (10)	**10** (8)	0.0095	10	0	0
wise	**13** (1)	**13** (1)	0.0032	13	0	0
libvirt	**4** (3)	**2** (1)	0.0002	2	0	0
GlusterFS	**18** (7)	**17** (7)	0.0029	17	0	0
QuteCom	**7** (3)	**7** (3)	0.0021	7	0	0
wpa_supplicant	**6** (1)	**6** (1)	0.0010	6	0	0
RedHat	**9** (6)	**9** (6)	0.0023	9	0	0
hostapd	**6** (1)	**6** (1)	0.0012	6	0	0
LibEtPan	**23** (10)	**10** (4)	0.0036	10	0	0
FreeRADIUS	**12** (4)	**8** (3)	0.0045	8	0	0
ALSA-lib	**13** (6)	**11** (5)	0.0045	11	0	0
IPsec	**4** (2)	**3** (1)	0.0017	3	0	0
Total	**624 (377)**	**485 (296)**		**457**	**9**	**19**

Figure 4.12 shows the number of functions with various numbers of blocks of error-handling code that contain at least one fault. We have already seen, in Section 4.2, that larger numbers of blocks of error-handling code complicate code maintenance. Indeed, the number of faulty functions tends to increase as there are more blocks of error-handling code. Above four blocks of error-handling code the number of faults found gradually declines, however, there are also fewer of such functions. For example, for the functions with more than 25 blocks of error-handling code, Hector finds only three

faults, but there are only 74 such functions in the considered directories, and thus the fault rate is actually quite high in this case.

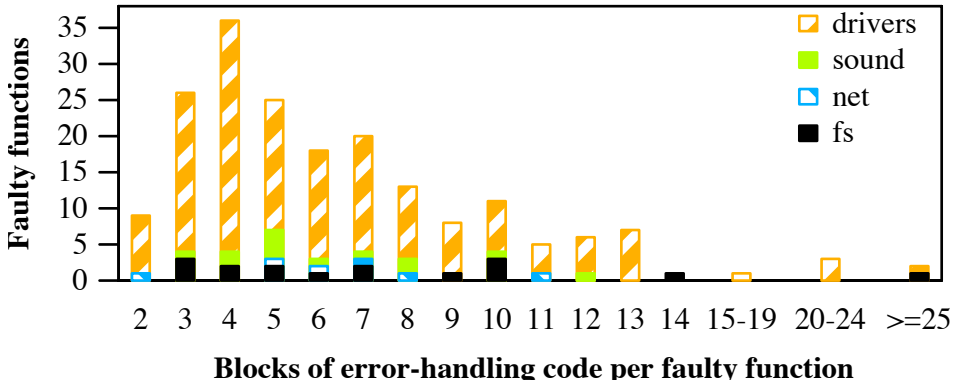


Figure 4.12: Complexity of faulty functions

We first investigate the complementarity of our approach with other approaches. Because the implementation of other C code specification mining tools are not publicly available, we first assess our results in terms of the strategies and thresholds used in previous work. We then consider how many of the faults detected by Hector have been found and fixed in practice in Linux code.

4.7.2 Comparison to specification mining.

In Chapter 1, we observed that specification mining approaches often rely on thresholds defined in terms of support (the number of times the protocol is followed across the code base) and confidence (the percentage of occurrences of a portion of the protocol that satisfy the complete protocol) to reduce the number of false positives. Figure 5.5 showed that most of the pairs of resource acquisition and release functions identified by the heuristics presented in Section 4.5.1 do not meet the support and confidence thresholds proposed by the specification-mining tool PR-Miner [46]. Here, we focus on the subset of these pairs of resource acquisition and release functions that are associated with the reports generated by Hector.

Figure 4.13 shows the support and confidence for the protocols involved in our identified faults in Linux, Python, Apache, PostgreSQL, Apache PHP, and Wine. The $\times$s and circles represent the 150 pairs of resource acquisition and release operations associated with the 371 faults in these systems software, identified by Hector. Protocols associated with 52% of the faults found by Hector have support less than 15, and protocols associated with 86% of the faults found by Hector have confidence less than 90%. Indeed, only 7 pairs, marked as $\times$, have support greater than or equal to 15 and confidence greater than or equal to 90%. These 7 pairs are associated with only 23 (7%) of the 371 faults found by Hector, implying that 94% of the faults found by Hector would be overlooked when using these thresholds. Indeed, the well-known Linux protocol *kmalloc*/*kfree*, for which Hector finds 28 faults, only has confidence of 59%, as many of the functions that call *kmalloc* have no reason to also call *kfree*. On the other hand, reducing the support or confidence thresholds used by specification-mining-based approaches could drastically increase their number of false positives. Hector finds faults independent of the support and confidence of the protocol.

Figure 4.13 also shows as open rectangles the support and confidence for the 55 protocols involved in the 113 false positives generated by Hector for these systems software. None of these protocols ex-

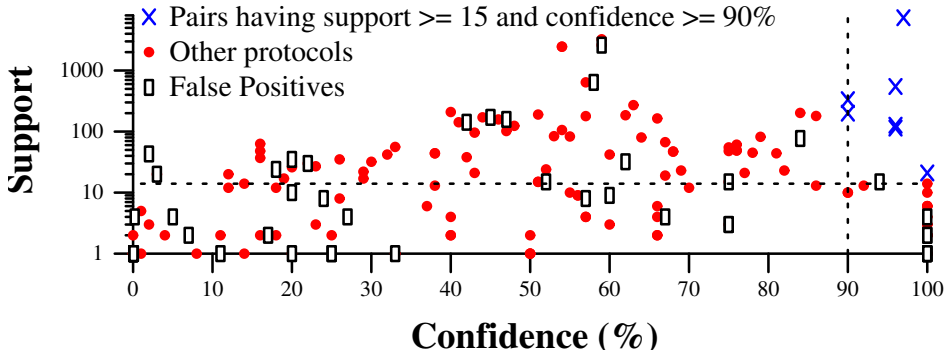


Figure 4.13: Support and confidence associated with the protocols in the faults reported by Hector. The dotted lines mark support 15 and confidence 90%.

ceed the thresholds of support 15 and confidence 90%, showing the reasonableness of these thresholds in a setting where false positives are very likely. Otherwise, these protocols show a distribution similar to that of protocols for which there are faults, with some having high support or high confidence. These results suggest that support and confidence are not very helpful in assessing these cases.

4.7.3 Comparison to faults fixed in Linux.

Linux 2.6.34 was released in May 2010, and thus some of the faults identified by Hector have subsequently been fixed or otherwise eliminated by other developers. We have furthermore submitted patches for many of the faults detected by Hector, for Linux and for other software. Figure 4.14 summarizes the status of the 187 faults in `drivers` that have been fixed or otherwise eliminated since the release of Linux 2.6.34. The fixes include patches that we have submitted and have been accepted (74), patches that we have submitted but have not yet been accepted (23), patches that have been submitted by others and have been accepted (55), and faults that have disappeared due to reorganization or elimination of the code (36). The faults in the third category were primarily identified manually by developers, and thus the involved functions may have low support.

72 of the faults fixed by ourselves or others involve the common memory allocation functions *kmalloc*, *kzalloc*, and *kcalloc*. Because these functions and the corresponding release function, *kfree*, are well known, such faults could be found using fault-finding tools such as Coccinelle, smatch, and sparse,[7] that are configurable with respect to a priori known protocols. These tools are regularly applied to the Linux kernel, and thus the fact that such faults remain suggests a lack of attention to the affected files by tool users or lack of attention to the submitted patches by the associated maintainers. For the remaining functions, only 30% of the faults have been found and fixed by others. This shows that the strategies Hector uses are complementary to existing maintenance approaches. While many of these functions are used less often, within the implementation of a given service, a function with few overall call sites may be even more important than widely used generic functions, such as *kmalloc*. Indeed, omitting a single *kfree* typically results in the loss of only a few bytes, while an omission fault associated with a more specialized function, e.g., one that unregisters a device from the kernel, can lead to serious errors such as resource unavailability and kernel crashes, as illustrated in Section 4.2.1.

[7] http://coccinelle.lip6.fr, http://smatch.sourceforge.net/, https://sparse.wiki.kernel.org/index.php/Main_Page

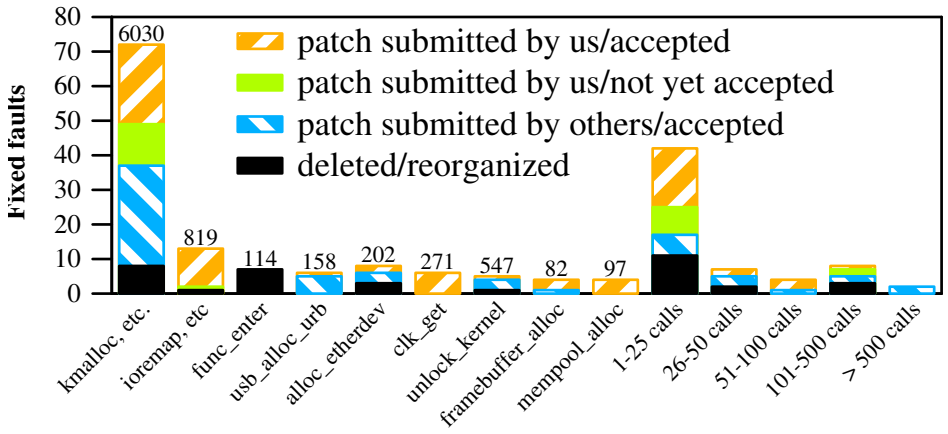


Figure 4.14: Fixed or eliminated Linux driver faults. Bars on the left refer to functions associated with 4 or more fixes. These bars are annotated with the support for the corresponding acquisition and release functions. Bars on the right refer to functions with fewer than 4 fixes and varying levels of support.

4.7.4 Impact of the detected faults

As illustrated in Section 4.2.1, the kinds of faults Hector detects can lead to crashes, memory exhaustion or deadlocks. Faults can also involve omitted debugging operations, which do not themselves cause a system crash, but can complicate the process of debugging other errors, particularly those that are difficult to reproduce.

Faults in Linux. We first focus on Linux, as this is the most critical and long-running of the considered software projects. The impact of a fault in error-handling code depends on the probability that the function containing the fault will be executed, the likelihood that the associated error will occur, and the nature of the omitted operation. Table 4.3 classifies the faults that we have found according to these properties. Linux kernel functions vary in the degree of privilege required to cause them to be executed and the number of times they are likely to be executed in normal system usage, with read/write functions being executed the most often and requiring the least privilege, and initialization functions being executed the least often and frequently requiring the greatest privilege. We furthermore distinguish between *static* initialization functions, which are only executed during the boot, and *dynamic* initialization functions, for e.g., hotpluggable devices that can be loaded and unloaded many times within the lifetime of a system. The errors handled range from a lack of memory, which should be rare in a correctly dimensioned system, to invalid arguments from the user level, which are completely under user control. Finally, we classify faults according to the effect the fault may have: a memory leak (Leak), a deadlock (Lock), or inconsistent debugging logs (Debug).

We first consider the faults in terms of the properties of the containing function. Almost 40% of the faults found in Linux code are in dynamic initialization functions, and this ratio reaches almost 50% if static initialization functions are included. Indeed, Kadav and Swift have found that initialization functions make up 30-50% of the code of many kinds of drivers [40]. 12 of the faults occur in read/write functions, which users typically invoke repeatedly. A third of these faults depend in some way on a file structure, which may depend on user-level requests. Most of the rest of the faults depend only on internal structures, making it less likely that specific user actions can trigger the fault.

Next, we consider the faults in terms of the reason for the handled error. Over half of the faults (No

Table 4.3: Impact of faults found in Linux

		Lack of memory	Transient errors	No device or address	Invalid user value	Total
Read/write	Leak	2	2	6	0	**10**
	Lock	0	0	0	0	**0**
	Debug	0	0	0	2	**2**
Ioctl	Leak	12	3	16	5	**36**
	Lock	0	0	0	1	**1**
	Debug	0	0	1	2	**3**
Open	Leak	16	9	46	1	**72**
	Lock	1	1	5	0	**7**
	Debug	1	1	8	1	**11**
Dynamic init	Leak	48	5	49	7	**109**
	Lock	0	0	0	0	**0**
	Debug	0	0	2	1	**3**
Static init	Leak	12	2	14	2	**30**
	Lock	0	0	0	1	**1**
	Debug	0	0	0	0	**0**
Total	**Leak**	**90**	**21**	**131**	**15**	**257**
	Lock	**1**	**1**	**5**	**2**	**9**
	Debug	**1**	**1**	**11**	**6**	**19**

device or address) are found in the handling of errors related to invalid arguments and non-existent devices, represented by constants such as `EINVAL`. Such faults may arise from invalid user requests or unavailable or malfunctioning devices. 23 of the faults are found in the handling of errors related to invalid values received from the user level (`EFAULT`), such as invalid addresses for copying data to or from the kernel, which are easy for the user to construct.

Finally, we consider the effect of the faults. 9 involve omitted unlock operations, thus introducing potential deadlocks. Among the faults that have the most potential impact, in 1 case, the error can be caused by an invalid user-level value, provided via an ioctl, while in 4 other cases the error is caused by the inability to access a resource such as a file, the identity of which may ultimately depend on user-level requests. These faults may thus be exploitable by a determined attacker. In two other cases, the error derives from malfunctioning hardware; such errors may be more difficult for attacker to exploit, but can result in the inability to access related resources. Finally, over 90% of the faults cause memory leaks. Of these, 88% are in functions that can be iterated, and of these 5% are in read/write functions that can be iterated by an unprivileged user.

These results generalize the examples presented in Section 4.2, showing that faults in error-handling code can potentially have a significant impact on the reliability of systems software.

Faults in other software. To have a broader view of the potential impact of faults in error-handling code, we have also studied the impact of the faults found by Hector in the PHP and Python language runtimes. Out of the 13 faults Hector finds in the PHP runtime, 11 are located in PHP functions that are called by at least 14 API functions (i.e., functions that are exposed directly to PHP developers). Several of the associated blocks of error-handling code are triggered by bad argument values or malformed input files (images, in particular, in the *gd2* module). These blocks of error-handling code expose PHP applications to memory leaks. Moreover, since PHP is commonly used as a web scripting language, an attacker could potentially provide faulty arguments to a remote PHP script or upload malformed files in order to trigger memory leaks on a remote server. Indeed, 7 of the memory leaks detected by Hector pertain to persistent memory (i.e., memory that is never released as long as the web server runs). For Python, 8 of the 33 faults found in Python code are in three Python 3.2.3 API functions. These functions either are new since Python 2.7.2 or have been completely reimplemented. Most of the remaining faults are in initialization functions or in functions stored in

Python modules. Python manages internal data structures using reference counts, and almost all of the faults involve omission of a reference count decrement operation.

For PHP, we have designed a possible attack that exploits a fault in the function *_xmlwriter_get_valid_file_path()*. We wrote a PHP script that calls this function via the PHP runtime function *xmlwriter_open_uri()* a hundred million times with a faulty argument that triggers the bug. Running this PHP script on an apache2 web server results in an `apache2` process that uses up all of the available RAM of a 4GB server. An attacker could use this fault in two ways. First, if he has the ability to upload PHP files to the server in a directory where they are interpreted by Apache, he can upload our script and access it remotely to use up all memory. Second, if he finds a PHP script on the server that uses *xmlwriter_open_uri()* with an argument that is passed in via an HTML form, he can fetch the page millions of times with a faulty argument until all of the memory of the server is exhausted.

4.7.5 False positives

Table 4.4 shows the number of false positives among the reports generated by Hector and the reasons why these reports are false positives. The overall false positive rate is 23%, which is below the threshold of 30% that has has been found to be the limit of what is acceptable to developers [12]. The reasons for the false positives vary, including failure of the heuristics for distinguishing error-handling code from successful completion of a function (Not EHC, 4%), failure of the heuristics for identifying acquired resources (Not alloc, 22%), or for recognizing existing releases, whether via an alias (Via alias, 36%) or via a non-local call (Non-local call frees, 14%), or releases performed in the caller of the considered function rather than in the function itself (Caller frees, 11%).

Table 4.4: False Positives

		FP	**Reasons**					
	Reports	(Rate, Fns)	Not EHC	Not alloc	Via alias	Non-local call frees	Caller frees	Other
Linux `drivers`	**293**	**56** (19%,34)	3	16	11	13	8	5
Linux `sound`	**32**	**13** (41%,6)	0	0	13	0	0	0
Linux `net`	**13**	**6** (46%,6)	0	0	0	0	1	5
Linux `fs`	**47**	**25** (53%,17)	0	7	6	1	6	5
Python (2.7)	**17**	**4** (24%,2)	0	0	3	0	0	1
Python (3.2.3)	**22**	**2** (9%,2)	0	1	0	0	0	1
Apache httpd	**5**	**2** (20%,2)	1	0	0	0	0	1
Wine	**31**	**1** (3%,1)	0	1	0	0	0	0
PHP	**16**	**3** (19%,3)	0	3	0	0	0	0
PostgreSQL	**8**	**1** (12%,1)	0	1	0	0	0	0
Samba4	**13**	**1** (1%,1)	0	0	1	0	0	0
ALSA-driver	**12**	**2** (17%,2)	0	0	0	2	0	0
wise	**13**	**0** (0%,0)	0	0	0	0	0	0
libvirt	**4**	**2** (50%,2)	2	0	0	0	0	0
GlusterFS	**18**	**1** (6%,1)	0	0	1	0	0	0
QuteCom	**7**	**0** (0%,0)	0	0	0	0	0	0
wpa_supplicant	**6**	**0** (0%,0)	0	0	0	0	0	0
RedHat	**9**	**0** (0%,0)	0	0	0	0	0	0
hostapd	**6**	**0** (0%,0)	0	0	0	0	0	0
LibEtPan	**23**	**13** (57%,6)	0	0	13	0	0	0
FreeRADIUS	**12**	**4** (33%,1)	0	0	0	4	0	0
ALSA-lib	**13**	**2** (15%,1)	0	0	2	0	0	0
IPsec	**4**	**1** (25%,1)	0	1	0	0	0	0
Total	**624**	**139 (23%,89)**	**6**	**30**	**50**	**20**	**15**	**18**

FP = False positives, Rate = FP/Reports, Fns = Containing functions

The Linux `sound`, `net`, and `fs` directories all have false positive rates higher than 30%. All of the `sound` false positives come from the use of a single function that creates an alias via which the resource is released. The affected functions all show the same pattern, making these false positives easy to spot. For `net`, 4 of the 6 false positives are due to error-handling code related to timeouts, in which case it is not necessary to release all of the resources. Again, the affected functions have a similar structure. Finally, the `fs` faults are more varied, and thus more difficult to identify. Still, there are fewer than 50 `fs` reports in all, making the identification of false positives tractable by a filesystem expert.

For the software other than Linux, the false positive rate positives varies considerably. Wise, QuteCom, Wpa_supplicant, Hostapd, and RedHat have no false positives. Libvirt and LibEtPan have false positive rates of 50% or higher. Libvirt has only 4 reports, including only 2 false positives, meaning that assessing the set of reports is not time-consuming. LibEtPan uses a pattern of nested data structures that the alias information collected by Hector is not sufficient to take into account. Nevertheless, many of the false positives have a similar structure and we found them easy to evaluate.

To help the user navigate among the reports, we have proposed a ranking strategy (Section 4.6). The ranking strategy is motivated by the common cases of false positives, where a candidate fault occurs before a release is actually needed (*low-initial* rank, corresponding to the Not alloc cases in Table 4.4) and where a candidate fault occurs after the resource has already been released (*low-final* rank, corresponding to the Via alias and Non-local call free cases in Table 4.4). Table 4.5 shows the total number of *high*, *low-initial* and *low-final* ranked reports. Few false positives are *high*. The user may thus study the high ranked reports first, to get an overall understanding of the use of resources in the software, and then consider the low ranked reports, taking into account the acquired intuitions.

Table 4.5: Report ranking

	Reports			Faults			FP		
	High	Low-Initial	Low-Final	High	Low-Initial	Low-Final	High	Low-Initial	Low-Final
Linux `drivers`	68	127	98	62	116	59	6	11	39
Linux `sound`	11	8	13	11	8	0	0	0	13
Linux `net`	6	3	4	2	3	2	4	0	2
Linux `fs`	12	20	15	8	12	2	4	8	13
Python (2.7)	4	7	6	4	7	2	0	0	4
Python (3.2.3)	3	6	13	3	5	12	0	1	1
Apache httpd	2	1	2	2	1	0	0	0	2
Wine	10	15	6	10	15	5	0	0	1
PHP	2	13	1	2	10	1	0	3	0
PostgreSQL	1	5	2	1	4	2	0	1	0
Samba4	3	2	8	3	2	7	0	0	1
ALSA-driver	5	4	3	5	4	1	0	0	2
wise	13	0	0	13	0	0	0	0	0
libvirt	0	1	3	0	0	2	0	1	1
GlusterFS	4	4	10	4	4	9	0	0	1
QuteCom	3	0	4	3	0	4	0	0	0
wpa_supplicant	0	2	4	0	2	4	0	0	0
RedHat	2	7	0	2	7	0	0	0	0
hostapd	0	2	4	0	2	4	0	0	0
LibEtPan	6	4	13	6	4	0	0	0	13
FreeRADIUS	2	0	10	2	0	6	0	0	4
ALSA-lib	5	0	8	5	0	6	0	0	2
IPsec	0	1	3	0	0	3	0	1	0
Total	**162**	**232**	**230**	**148**	**206**	**131**	**14**	**26**	**99**

(H = High, LI = Low-initial, LF = Low-final)

Table 4.6: Faults, false positives, and false negatives, for kmalloc, kzalloc, and kcalloc

	Coccinelle			Hector		
	Faults	FP	FN	Faults	FP	FN
Linux `drivers`	38	28 (42%)	70 (65%)	86	10 (10%)	22 (20%)
Linux `sound`	2	6 (75%)	6 (75%)	7	13 (65%)	1 (13%)
Linux `net`	4	5 (56%)	1 (20%)	1	1 (50%)	4 (80%)
Linux `fs`	1	8 (89%)	1 (50%)	1	7 (88%)	1 (50%)

4.7.6 False negatives

Hector requires an exemplar of the release of a resource before it can detect that a release of that resource is somewhere omitted. This exemplar permits Hector to find faults without precise information about resource acquisition and release functions. However, without an exemplar, no fault can be detected, resulting in false negatives. Other potential reasons for false negatives are analogous to the reasons for false positives, e.g., failing to recognize a call that represents an acquisition, and considering a call to be a release operation when the called function does not perform a release.

Estimating the rate of false negatives is difficult, because it requires complete knowledge of the set of faults in a system. Indeed, we know of no other fault-finding tools for systems code for which false negatives have been investigated. To reduce the amount of code to study while being able to estimate the effect of the need for an exemplar, we focus on the Linux kernel functions, *kmalloc*, *kzalloc*, and *kcalloc*, for which Figure 4.14 showed that faults are common. To further reduce the amount of code to consider, we focus on cases where the acquired resource is stored in a local variable and is not passed to another function or stored in another location before reaching the error-handling code; these restrictions imply that there is a high probability that the resource must be released before the variable referencing it goes out of scope. we have implemented this strategy using the open-source tool Coccinelle [64]. Coccinelle does not implement a specific fault-finding policy, but instead makes it possible to search for patterns involving properties of paths within a function's CFG.

Table 4.6 shows the rate of detected faults in the use of *kmalloc*, *kzalloc*, and *kcalloc* and the rate of false positives, for the Coccinelle rule and for Hector. From this information, we compute a lower bound on the number and rate of false negatives by comparing the set of faults found by each approach to the complete set of faults found by either approach. While Hector has a high rate of false negatives, the absolute numbers involved are small. Almost all of the false negatives are due to the lack of an exemplar. There are only three cases, all in a single function, where there is a failure of the preprocessing heuristics, as a call is considered to be a release when it is not. Furthermore, the Coccinelle rule also has a high rate of false negatives, because of the restrictions noted above to avoid false positives. These restrictions are indeed only partially successful, because the rate of false positives is up to 89%, and is consistently higher than that of Hector.

4.7.7 The benefits of the analysis features

The instantiation of our algorithm provides the following features 1) identifying the candidate faults that use different resource release operations that may or may not be defined in the same file, instead of the releasing operation found in the exemplar, 2) identifying operations whose side effects release the resources, and 3) identifying operations that release the resource via another pointer.

These features are illustrated by the examples in Figures 4.7, 4.8, and 4.9 respectively. In this section, we evaluate in more detail the impact of the analysis, taking into account the presence of alternate

Table 4.7: Discard candidate faults using different analysis features

	analyzable candidate faults	multiple releasing operations that may or may not be locally defined	release by side effect of other operations	release via another pointers
Linux `drivers`	567	220	2	52
Linux `sound`	193	157	0	4
Linux `net`	33	11	1	8
Linux `fs`	235	84	14	90
Wine	42	11	0	0
PostgreSQL	14	2	0	2
Apache httpd	5	0	0	0
Python (2.7.3)	52	16	0	19
Python (3.2.3)	65	22	0	21
PHP	20	4	0	0
Samba	114	78	2	21
ALSA-driver	170	153	0	5
wise	17	4	0	0
libvirt	8	2	0	2
GlusterFS	27	5	1	3
QuteCom	9	2	0	0
wpa_supplicant	18	11	0	1
RedHat	13	4	0	0
hostapd	18	11	0	1
LibEtPan	75	1	0	51
FreeRADIUS	76	64	0	0
ALSA-lib	23	8	0	2
IPsec	4	0	0	0
Total	**1798**	**870 (48.4%)**	**20 (1.1%)**	**282 (15.7%)**

resource-releasing operations based on the scenarios in Figures 4.7, 4.8, and 4.9. On the considered software, we analyze how often these features discard candidate faults avoiding false positives.

On all 19 considered software projects, Hector found 380K candidate faults, each being a possible execution path that contains a resource acquisition operation but no corresponding release operation in error-handling code. However, by applying the first two conditions presented in Section 4.4 to the candidate faults, Hector discards 378K (99.5%) candidate faults, leaving only 1798 (0.5%) candidate faults. We call these candidate faults the *analyzable candidate faults*. Table 4.7 shows the number of analyzable candidate faults and the number of analyzable candidate faults that are eliminated in the analysis phases.

In the example shown in Figure 4.7, different execution paths use different resource release operations. These releasing operations may or may not be defined in the same file. A specification mining approach that takes into account only the specification that has the highest rank, would give a false positive in at least one of these cases. Table 4.7 shows that 48.4% of the analyzable candidate faults use different releasing operations to release the resource than the releasing operations in the exemplars. Hector identifies all of these cases and discard these candidate faults to avoid false positives.

In the example shown in Figure 4.8, a resource is released by an intervening function when this function fails on some task. Hector identifies this scenario and removes the candidate faults from consideration. Table 4.7 shows that 1.1% of the analyzable candidate faults use such operations to

Table 4.8: Scalability of Hector

	Time (seconds)	Lines of code(LoC)	Time/LoC
Linux `drivers`	18342	4.6MLoC	0.0051
Linux `sound`	1649	0.4MLoC	0.0038
Linux `net`	1428	0.4MLoC	0.0036
Linux `fs`	10503	0.7MLoC	0.015
Wine	1381	2.1MLoC	0.0007
PostgreSQL	750	0.6MLoC	0.0013
Apache httpd	2224	0.1MLoC	0.0153
Python (2.7.3)	1306	0.4MLoC	0.0033
Python (3.2.3)	1885	0.3MLoC	0.0063
PHP	114	0.6MLoC	0.0002
Samba	10308	496KLoC	0.0208
ALSA-driver	2155	474KLoC	0.0045
wise	335	276KLoC	0.0012
libvirt	3445	224KLoC	0.0154
GlusterFS	1799	193KLoC	0.0093
QuteCom	190	188KLoC	0.0010
wpa_supplicant	635	160KLoC	0.004
RedHat	1333	151KLoC	0.0089
hostapd	440	134KLoC	0.0033
LibEtPan	47	94KLoC	0.0005
FreeRADIUS	1252	82KLoC	0.0153
ALSA-lib	733	76KLoC	0.0097
IPsec	2076	62KLoC	0.0335

release the resource.

A resource can be made accessible via another pointer, and can be released via this pointer in a candidate fault, as illustrated in Figure 4.9. Hector analyzes such pointers to determine whether the pointer is used to released the resource. If so, Hector discards those candidate faults to avoid false positives. Table 4.7 shows that 15.7% of the candidate faults use other pointers to release the resources.

Finally, Hector generates reports for 37.8% of the candidate faults.

In the Section 4.7.6, we have already discussed that Hector may produce false negatives.

4.7.8 Scalability

We carried out our tests on one core of a 8-core 3GHz Intel Xeon with 16GB RAM. Analyzing Linux `drivers`, which is the largest considered project (4.6 MLOC), takes around 5 hours. Table 4.8 shows the processing time per line of code for each considered project. Over all the considered projects, the processing time, excluding the parsing time, ranges from 0.0003 s/LOC (seconds per line of code) to 0.0335 s/LOC. Linux `drivers`, which is the largest project (4.6MLoC), and, IPSec, which is the smallest (62KLoC), have processing time per line of code 0.0051 s/LOC and 0.0335 s/LOC respectively, showing the scalability of the approach. Figure 4.15 shows the different processing time per line of code of different software projects. In this figure, the software projects are organized according to their total number of lines of code. The smallest project comes first.

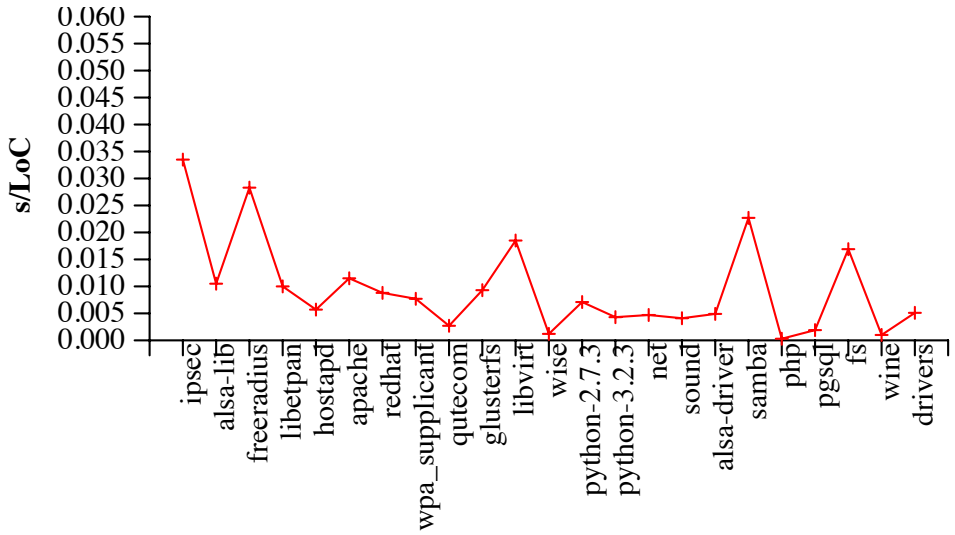


Figure 4.15: Analyzing time (in seconds) per line of code

4.7.9 Threats to validity

The validity of our study depends on the representativeness of the software chosen and the accuracy of our labeling of the reports. For the software, we have chosen projects that are widely used or that have many C functions and a relatively high density of error handling code. These characteristics are not found in all C software, but do serve to define the kinds of software targeted by our approach. For the labeling of the reports, we have validated some of our results by submitting patches to the relevant developer communities, as illustrated for Linux in Section 4.7.1.

4.8 Conclusion

In this chapter, we have defined an algorithm for finding resource-release omission faults that takes local information into account. Our algorithm finds a number of probable faults in different kinds of systems software with a low rate of false positives. We have shown that taking local information into account significantly reduces the number of false positives. Moreover, the algorithm ranks the generated reports to draw the user's attention to the more probable fault instances.

Chapter 5

Conclusion and Future Work

Contents

Our studies on error-handling code in this dissertation have shown that the number of functions with error-handling code in widely used systems software is increasing version by version. Moreover, we have found that writing error-handling code in a dispersed way creates duplicate code that is difficult for developers to maintain. As a result, programmers often make mistakes while writing new error-handling code. The motivation of this dissertation is to improve error handling code in systems software by refactoring such code and by finding faults in it. We have first focused on improving the structure of error-handling code with the goal of helping to reduce the number of system faults that may occur in error handling code in future. We have then focused on finding existing faults in systems error-handling code.

5.1 Conclusion

The first part of this dissertation has focused on the refactoring of error-handling code in systems software. The Linux kernel coding style guidelines advocate organizing such code using labels and `gotos`, but systems software frequently does not follow this strategy. We have proposed an automatic transformation that converts error-handling code that is dispersed and duplicated throughout the body of a function such that it uses the single labelled sequence of state-restoring operations at the end of

the function. We have found that our transformation applies to many functions across the systems software, and that it identifies many opportunities for code sharing. The approach was actually motivated by Linux, but we have found that the approach is applicable to other systems software. That is, in the other software, the `goto` style is sometimes but not always used.

Next, we have shown that error-handling code is a substantial source of faults in systems code, and that such faults can have a significant impact on system reliability. We have presented a novel approach to finding faults in error-handling code of systems software that uses a function's existing error-handling code as an exemplar of the operations that are required. By focusing on one function at a time, while taking into account a small amount of interprocedural information from other functions defined in the same file, we obtain a fault-finding algorithm that is precise and scalable. We have implemented our approach as the tool Hector, and applied it to find 485 faults in Linux and 18 other systems software projects.

5.2 Limitations and Future Work

5.2.1 Relax the need for exemplars

A limitation of our approach to finding faults in error-handling is the need for at least one exemplar of a given resource-release operation in the considered function. In future work, we will consider whether it is possible to relax this requirement, *e.g.*, to find exemplars in other functions in the same file, or in functions that appear to play the same role in the implementations of related services.

5.2.2 Other memory related bugs

In our work, we have focused on finding missing resource-releasing operations that may lead to memory leaks and deadlocks. However, we can use our approach to find other types of memory related bugs. For example, the algorithm keeps track of NULL values, but does not fully use this information, e.g., to find probable NULL pointer dereferences. We will consider the benefit of such checks in future work. The algorithm also can be used to find double free bugs in error-handling code. Our algorithm identifies any releasing operation on a resource and keeps the information for further analysis. Therefore, it can easily detect if a freed resource is used in an operation for freeing again.

5.2.3 Fixing bugs

Another direction of future work is to consider how to automatically fix the faults, based on the information in the exemplar, or based on the history of the software as a whole, taking into account how similar faults have been fixed in other parts of the software over time.

We have a preliminary idea to automatically fix the detected omission faults. Recall that *high* ranked fault reports are in Section 4.6. The ranking strategy gives a fault report a *high* rank when the block of error-handling code containing the fault is both preceded in the CFG by a block of error-handing code that releases the resource and followed in the CFG by a release of the resource, whether or not in error-handling code. In this case, we propose to insert the omitted We target only the *high* ranked fault reports and propose to insert omitted resource-releasing operations into the error-handling code. Doing so requires determining: 1) what resource releasing operation to use, if the function uses

multiple releasing operations for a particular resource, and 2) where to insert the resource releasing operation into the existing error-handling code.

In order to insert the appropriate releasing operation, the algorithm uses the releasing operation in the preceding error-handling code. The following error-handling code may contain an alternate releasing operation to release the resource which might not be the appropriate operation to be inserted. In order to insert the omitted resource-releasing operation at the right place in the error-handling code, the algorithm uses the information found in the following block of error-handling code. Usually, the number of resource-releasing operations needed in error-handling code increases as execution progresses through a function, as more and more resources are allocated that need to be released. The later blocks of error-handling code thus provide more information about the relationships between the complete set of resource-releasing operations than the earlier ones. The algorithm inserts the omitted releasing operation into the error-handling code after the last operation that accesses the resource's associated variable and the operations that appear in the following error-handling code prior to the omitted releasing operation.

```
1  ...
2  x = kmalloc(...);
3  ...
4  if(!y) {
5    kfree(x);
6    return -ENOMEM;
7  }
8  ...
9  if(!z) {
10   put(x,y);
11   kfree(y);
12   return -ENOMEM;
13 }
14 ...
15 ptr->name = x;
16 ...
17 if(!m) {
18   kfree(z);
19   put(y,x);
20   kfree(y);
21   free(ptr);
22   return -ENOMEM;
23 }
```

a. Code containing an omitted resource-release fault

```
1  ...
2  x = kmalloc(...);
3  ...
4  if(!y) {
5    kfree(x);
6    return -ENOMEM;
7  }
8  ...
9  if(!z) {
10   put(x,y);
11   kfree(y);
12   kfree(x);
13   return -ENOMEM;
14 }
15 ...
16 ptr->name = x;
17 ...
18 if(!m) {
19   kfree(z);
20   put(y,x);
21   kfree(y);
22   free(ptr);
23   return -ENOMEM;
24 }
```

b. The same code, after fixing the fault

Figure 5.1: Example of fixing a resource-release omission fault

In practice, simple examples among the high-ranked fault reports, and thus we have made up an example in Figure 5.1 to show the algorithm in its full generality. In Figure 5.1a, a resource is allocated by calling the function `kmalloc`, and the result is stored in the variable `x` (line 2). The error-handling code on lines 5-6 then calls `kfree(x)` to release `x`. The variable `x` is stored in the structure `ptr` on line 15. The error-handling code on lines 18-22 calls `free(ptr)` on line 21 to release `x` via `ptr`. However, the error-handling code in the middle of the function, on lines 10-12, does not have any operation that releases `x`. This omission fault receives rank *high* and is selected to be fixed. The call `free(ptr)` cannot be used to release the resource in the error-handling code on lines 10-12, because the resource only becomes referenced by `ptr` on line 15, after the block of error-handling code in lines 10-12. Therefore, `free(ptr)` is not the appropriate resource-releasing operation to be inserted in the error-handling code on lines 10-12. Thus, the algorithm inserts the omitted resource-

releasing operation `kfree(x)` into the block of error-handling code on lines 10-12 after `kfree(y)` on line 11. This placement is based on the information in the subsequent block of error-handling code on line 18-22, that says `x` is released using `free(ptr)` on line 21 after `kfree(y)`. However the preceding error-handling code on lines 5-6 does not have any information about `kfree(y)` that can not help the algorithm to decide whether the omitted resource-releasing operation after `kfree(y)`. Figure 5.1b shows the function after fixing the omitted the resource-releasing fault. The new call to `kfree(x)` is on line 12, in the error-handling code on lines 10-13.

5.2.4 Finding shared variables

A data-race is a well-known type of concurrency bug. A data-race occurs when two different threads in a given program can simultaneously access a shared variable, with at least one of the accesses being a write operation. A number of works have been proposed to detect data-races in concurrent programs [18, 36, 58, 92]. The most critical step in detecting data-races is to automatically identify shared variables [41]. Incorrect identification of shared variables leads to a high rate of false positives in data-race reports. Kahlon *et el.* studied the accuracy of the identification of shared variables [41]. They propose a static analysis approach to identify shared variables, and then use the identified shared variables to detect data-races. Their shared variable detection routine is based on the precise that all shared variables are either global variable of threads, aliases thereof, pointers passed as parameters to API functions or escape variables.

We will consider how the use of local information can be applied to identify shared variables. An ad hoc synchronization is a synchronization that is implemented in an ad hoc way. That does not follow any modularize manner like synchronization using `lock/unlock` or any user-defined function calls. Detecting ad hoc synchronization is another challenging problem while detecting concurrency bugs. The accuracy of identifying ad hoc synchronization helps to reduce the rate of false positives and discovers new concurrency bugs [89]. The accuracy on identifying of shared variables can help to identify ad hoc synchronization accurately.

5.2.5 Bugs in web applications

Finally, we plan to apply the expertise acquired on C code to propose an approach for detecting bugs in error-handling code in PHP programs. Compilers for web languages such as PHP do not provide strong type-checking or static detection of mistakes such as the use of undeclared variables. Therefore, Web Applications are more defect-prone than Desktop applications [79]. Semantic bugs in PHP code may lead to generating incorrect HTML code, degrading the user experience and potentially crashing the user's browser.[1] Because two languages are involved, PHP and HTML, finding such bugs is especially challenging.

Figure 5.2 illustrates the kinds of problems that can arise. This example is taken from the work of *Artzi et al.* on finding bugs in dynamic web applications [9]. In this example the error-handling code on lines 5-6 terminates the PHP program without executing the code on line 9 that prints the HTML footer. This generates a malformed HTML page.

Currently, few program validation tools target PHP. One used in Facebook is Pfff [3], a set of tools and APIs, that provides a lint-like bug finder to find generic bugs in PHP code. Nevertheless,

[1] https://bugzilla.mozilla.org/show_bug.cgi?id=209095 and https://bugzilla.mozilla.org/show_bug.cgi?id=320459

```
1  ...
2  make_header();// print HTML header
3  ...
4  if($_GET['page2'] == 1337) {
5     require('printReportCards.php');
6     die();// terminate the PHP program
7  }
8  ...
9  make_footer();// print HTML footer
10 ...
```

Figure 5.2: Bug in the PHP error-handling code

this technique does not target semantic bugs. Some semantic bugs can be found by testing. But it is difficult to force tests into all error handling code, making semantic bugs in such code hard to find.

The goal of the bug-finding tool for PHP programs is to find semantic bugs in error-handling code, taking into account context-sensitivity. To exploit the information contained in error-handling code, the algorithm will first collect information about the structure and contents of all the error-handling code in a given PHP file. Based on this information, it will then search for error-handling code that is missing some important information or contains some incorrect information. Finally, it will use some heuristics to determine whether these omissions or incorrect information are legitimate or represent actual bugs.

5.3 Summary of Contributions

The contributions of this thesis are as follows:

- An algorithm to transform error-handling from the *basic strategy* to the *goto-based strategy*. This transformation can reduce duplicated code and make it easier for a developer to maintain the error-handling code, thus, it helps developers avoid introducing new bugs into error-handling code.
- 22% of the conditionals containing state-restoring error-handling code have been successfully converted from the basic strategy to the `goto`-based strategy.
- The concept of extracting specifications using local information to a single function definition instead of using a global scan of the code base.
- A collection of heuristics for identifying error-handling code in C systems software. These heuristics address the issue that the C language does not provide any error-handling abstractions.
- An algorithm for finding faults in error-handling code, implemented as a tool, Hector.
- 485 faults identified by Hector in the error-handling code of 19 system software projects, including 285 faults found in Linux.
- 97 patches submitted to the system developers of Linux. Among them 74 patches have been accepted and 23 have not yet been accepted.
- Examples of crashes and malicious attacks that are enabled by the omission of resource-releasing operations in the error-handling code.

Appendix: Résumé de la thèse

N'importe quel système logiciel peut rencontrer des erreurs, telles que des requêtes inapropriées de la part d'applications supportées, ou des comportements inattendus de materiel ne fonctionnant pas correctement ou étant mal configurés. Si le système d'un logiciel, tel que son système d'exploitation, l'exécutif d'un langage de programmation, ou un serveur web, ne récupère pas sur ces erreurs correctement, elles peuvent conduire à un echec plus serieux tel qu'un arrêt brutal ou une vulnérabilité à une attaque par un utilisateur malicieux. Ainsi, une recupération sur erreurs correcte est essentielle quand un système supporte des services critiques ou déstinés à s'exécuter logntemps. En effet, la capacité à récupérer des erreurs a longtemps été vue comme la pierre angulaire de la robustesse des systèmes [55], et beaucoup de code de systèmes sont concernés par la détection et la gestion d'erreurs. Par exemple, 48% du code des drivers de Linux 2.6.34 se trouve dans des fonctions qui gèrent au moins une erreur. [2]

Le code système est écrit en C, qui contrairement aux langages de programmation modernes comme Java, ne fournissent aucune abstraction spécifique pour la gestion des ressources ou la gestion d'erreurs. Le code de gestion d'erreurs est responsable de la détection de l'échec d'une opération, libérant les ressources pour restaurer le système dans un état cohérent, et retournant un indicateur d'erreur à la fonction appellante. N'importe quelle opération qui peut échouer doit être suivie par une branche conditionelle qui vérifie si une valeur d'erreur est retournée et effectue l'opération appropriée.

La Figure 5.3 montre un exemple de code de gestion d'erreurs. Au début de l'extrait de code, il y a trois expressions conditionelles aux lignes 5, 9, et 14 qui testent differentes conditions. Dans chaque cas, si une erreur est detectée, la branche conditionelle appelle d'abord `unlock_kernel` puis retourne un indicateur d'erreur. Les opérations de gestion d'erreur libèrent des structures de différentes complexités, et ommetre une partie de ce code lors de la construction de n'importe quel code de gestion d'erreur, ce qui deviens obligatoire quand la fonction évolue, conduira à des fuites mémoires.

```
1  {
2   ...
3   lock_kernel();
4   ...
5   if (!autofs_oz_mode(sbi)) {
6     unlock_kernel();
7     return -EACCES;
8   }
9   if (autofs_hash_lookup(dh, &dentry->d_name)) {
10    unlock_kernel();
11    return -EEXIST;
12  }
13  n = find_first_zero_bit(sbi->symlink_bitmap, AUTOFS_MAX_SYMLINKS);
14  if (n >= AUTOFS_MAX_SYMLINKS) {
15    unlock_kernel();
16    return -ENOSPC;
17  }
18  ...
19  d_instantiate(dentry, inode);
20  unlock_kernel();
21  return 0;
22 }
```

Figure 5.3: Code de traitement d'exception (Linux-2.6.34/fs/autofs/root.c)

[2] Linux 2.6.34 a été publié 2010. Nous nous concentrons sur une version de Linux datant de quelques années pour éviter que nos contributions au noyau Linux des premières étapes du développement de Hector n'interfèrent avec nos résultats

Restructuration de code: Une stratégie typique pour l'implémentation d'un code de gestion d'erreurs dans un système logiciel est montré en Figure 5.3.

La stratégie consiste à faire suivre chaque opération qui peut rencontrer une erreur par un bloc conditionel qui vérifie si une erreur s'est produite, et, le cas échéant, effectue les opérations de nettoyage appropriées avant de sortir de la fonction. Nous appelons cette stratégie *stratégie basique*. Cette stratégie, cependant, est sujette à erreurs, car il est facile de négliger certaines opérations de nettoyage qui sont requises, et d'oublier de mettre à jour certains codes de gestion d'erreurs existants quand la fonction est étendue avec de nouvelles opérations qui ont besoin d'être défaites dans un cas d'erreur. De plus, il peut y avoir de la duplication de code, du fait que le même code de gestion d'erreurs peut être necessaire à plusieurs endroits dans une même fonction.

Pour illustrer ces problèmes, considérons encore la Figure 5.3. Les trois bloc conditionels de gestion d'erreurs appellent la même fonction: `unlock_kernel`. Si le protocole pour utiliser ces fonctions change, le code doit être adapté dans chaque cas. De plus, négliger l'appel à `unlock_kernel` est une erreur qui va potentiellement conduire à un interblocage. Si d'autres allocations de ressources sont ajoutées à la fonction et le code de gestion d'erreur n'est pas mis à jour correctement, il va y avoir d'autre type de fuites de ressources.

Un style de programmation qui peut alléger ces difficultés est de déplacer les opérations de restauration d'état des blocs conditionnels de gestion d'erreurs individuels à une sequence d'opérations de restauration d'état à la fin de la fonction, atteignable grâce à une étiquette. Nous appellons ce style de programmation la strategie basé sur `goto`. Dans cette stratégie, chaque bloc conditionel de gestion d'erreur effectue uniquement les opérations qui sont spécifiques à la condition de l'erreur identifiée, tel que journaliser l'erreur ou stocker son l'indicateur dans une variable. Ensuite, un `goto` saute à la bonne position dans une suite d'opération de restauration d'état à la fin de la fonction. Cette approche place toutes les opérations de restauration d'état dans une zone facilement identifiable. Si le corps de la fonction est modifié de telle sorte qu'il y a des nouvelles conditions d'erreur possibles, les codes de gestion d'erreurs associés auront seulement besoin d'effectuer un saut au bon endroit dans cette suite. Si des nouvelles opérations de changement d'état sont ajoutées dans le corps de la fonction, les opérations de restauration d'état correspondantes ont seulement à être ajoutées à cette séquence. Enfin, la duplication de code est en grande partie limitée à l'introduction de `goto`, quelque soit la complexité du processus de gestion d'erreurs.

Nous considérons comment ce code serait écrit si il utilisait la stratégie basé sur `goto`. Sur la figure 5.3, nous observons que tous les codes de gestion d'erreurs appellent la même fonction, `unlock_kernel`. Nous observons aussi qu'il y a un autre appel de cette fonction à la ligne 20, à la fin de la fonction. Par conséquent, le programmeur peut utiliser cet appel au lieu d'écrire le même appel dans chaque bloc de code de gestion d'erreurs, en ajoutant une étiquette juste avant la ligne 20 et un saut de tous les blocs de code de gestion d'erreurs vers cette étiquette. Cette transformation, cependant, n'est pas suffisante pour obtenire une implémentation correcte. Une des difficultés de la conversion du code en Figure 5.3 pour utiliser la stratégie basée sur des `goto` est que trois blocs de gestion d'erreurs retournent differents indicateurs d'erreurs. Ainsi, il est aussi nécessaire d'assigner tous les indicateurs d'erreurs dans une seule variable. Ensuite il devient possible de fusionner tous les blocs de code de gestion d'erreurs en un seul.

La version améliorée de l'exemple donné en Figure 5.3 est montrée en Figure 5.4. Cette version déclare une variable `ret` à la ligne 3 au début de la fonction. Ensuite, elle utilise cette variable pour stocker l'indicateur d'erreur à la ligne 7, 11 et 16, dans chaque bloc de code de gestion d'erreurs. L'exemple utilise aussi cette variable `ret` pour stocker la valeur de retour de la fonction, `0`, dans le cas ou aucune erreur n'est détéctée, à la ligne 20. Une nouvelle étiquette `out` est ajoutée ligne 22 et

des gotos sont ajoutés à l'interieur des blocs de code de gestion d'erreurs pour sauter vers la nouvelle étiquette. Finalement, la variable ret est retourné à la ligne 24.

```
1  {
2   ...
3   int ret;
4   lock_kernel();
5   ...
6   if (!autofs_oz_mode(sbi)) {
7     ret = -EACCES;
8     goto out;
9   }
10  if (autofs_hash_lookup(dh, &dentry->d_name)) {
11    ret = -EEXIST;
12    goto out;
13  }
14  n = find_first_zero_bit(sbi->symlink_bitmap,AUTOFS_MAX_SYMLINKS);
15  if (n >= AUTOFS_MAX_SYMLINKS) {
16    ret = -ENOSPC;
17    goto out;
18  }
19  ...
20  ret = 0;
21  d_instantiate(dentry, inode);
22  out:
23    unlock_kernel();
24    return ret;
25  }
```

Figure 5.4: Version améliorée de l'Exemple 5.3

Actuellement, plusieurs fonctions dans les logiciels systèmes utilisent les stratégies basées sur goto. Cette stratégie est aussi recommendée par la documentation du noyau Linux.[3] Néanmoins, un grand nombre de fonctions utilisent encore la stratégie basique, et un grand nombre de bugs ont été découverts dans ce type de code. Par exemple, dans les patches de corrections de bug appliqués au noyau Linux 2.6.20 après sa publication, nous avons trouvé qu'à peu près un tier (12/32) de ceux dont l'unique effet est d'ajouter un appel à kfree ou à une fonction de déverouillage telle que spin_unlock, le bug est dans du code qui utilise la stratégie basique. La plupart des 32 autres bugs n'étaient pas dans du code de gestion d'erreurs. Nous avons obtenu des résultats similaires (6/20) dans l'ensemble de patches de Linux 2.6.34. [4]

Afin d'améliorer la structure des codes de gestion d'erreurs dans les systèmes logiciels, notre première contribution est un algorithme transformant le code de gestion d'erreur implémenté selon la stratégie basique afin qu'il suive la stratégie basé sur goto. Cet algorithme fusionne les opérations de réstauration d'état trouvées dans chaque bloc conditionel en une séquence d'opérations de réstauration d'état à la fin de la fonction. Nous avons implémenté cet algorithme dans un outil, que nous avons appliqué à cinq répertoires (drivers, fs, net, arch, et sound) dans le code source de Linux 3.6 ainsi qu'à cinq projets open-source largement utilisés : PostgreSQL, Apache, Wine, Python, and PHP. Cet outil convertit 22% des blocs conditionels contenant des codes de gestion d'erreurs de remise en état pouvant etre fusionnées en un seul, de la stratégie de base à la stratégie basée sur des goto.

[3] Linux-2.6.34/Documentation/CodingStyle, Chapter 7.

[4] Les patches de Linux 2.6.20 proviennent de git://git.kernel.org/pub/scm/linux/kernel/git/stable/linux-2.6.20.y.git en utilisant la commande "git log -p v2.6.20..". Les patches de Linux 2.6.34 proviennent de git://git.kernel.org/pub/scm/-linux/kernel/git/next/linux-next.git en utilisant la commande "git log -p v2.6.33..v2.6.34".

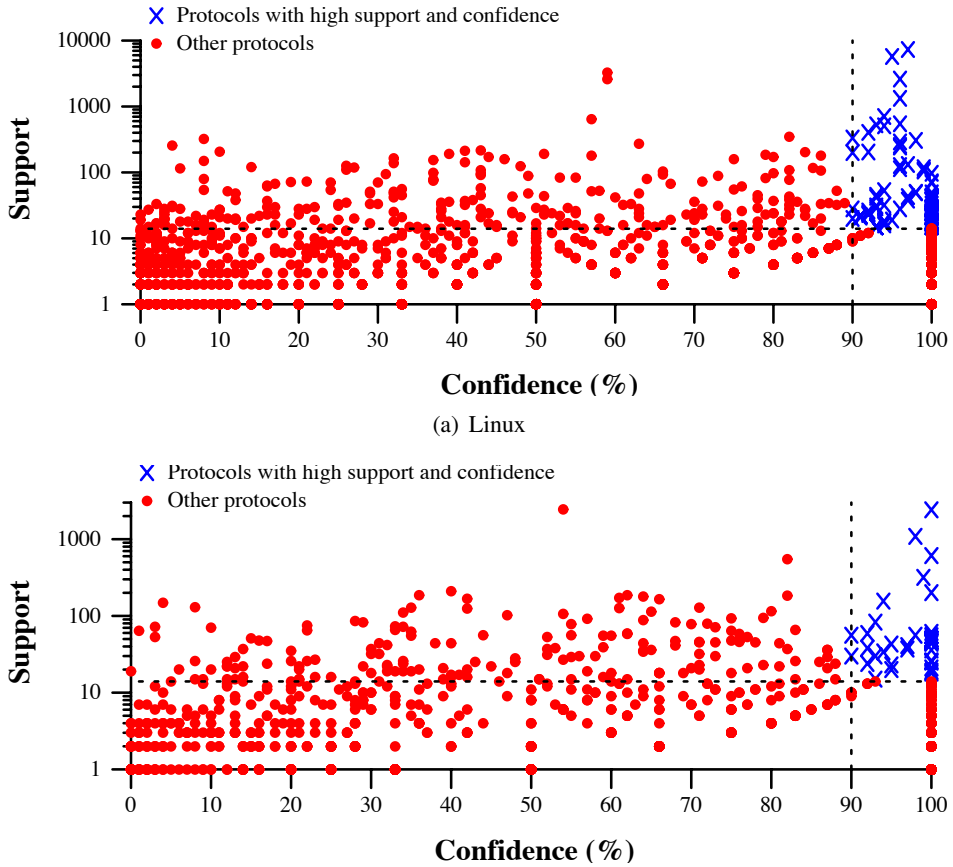
Amélioration la qualité d'un code de gestion d'erreur: Même quand le code de gestion d'erreurs est structuré comme recommendé par les conventions de code de Linux, la gestion de la libération des ressources allouées reste un problème critique pour la garantie de la robustesse du système. [67].

Une partie critique de la récupération sur erreurs est de libérer toutes les ressources que l'erreur a rendu incohérentes ou non-nécessaires. Ommetre une libération necessaire peut mener à des arrêts brutaux, des interblocages et des fuites de ressources. Les fautes d'omission de libération de ressources sont des cas particulier du problème de vérification que les protocoles d'utilisation des API sont respéctés. Ce problème a fait l'objet d'un nombre de recherche conséquent [19, 43, 67, 83]. Un défi, cependant, est d'identifier les opérations de libération de ressources qui sont requises. En effet, le code des systèmes manipule differents types de ressources, chacuns associés avec leur propres opérations dédiées, rendant difficile pour un développer donné de devenir familière avec elles. De plus, le protocole pour libérer un type donné de ressource peut varier d'un sous-système à l'autre, et peut même varier à l'interieur d'une fonction, selon l'état de la ressource.

Dans le contexte du problème général de vérification de l'utilisation des APIs, un nombre élevé de travaux ont proposé de compléter un outils de recherche de fautes avec une phase préliminaire de *fouille de spécifications* afin de trouver des ensembles d'opérations qui pourraient être présentes ensembles dans le code [8, 27, 31, 43, 46, 49, 62, 82, 84, 90]. Ces approches suivent une stratégie macroscopique, identifiant les ensembles communs d'opérations par un balayage globale de la base de code, ou d'un historique d'exécution suffisament gros. En pratique, cependant, de tels balayages rapportent beaucoup de faux-positifs [44], qui, à leur tour, conduisent à beaucoup de faux-positifs parmis les fautes trouvées. Afin de réduire le taux de faux positifs, l'approche de la fouille de spécifications limite les résultats rapportés aux opérations les plus fréquentes. Les spécifications résultantes, cependant, sont insuffisantes pour trouver des fautes d'omissions de libération de ressouce dans les fonctions peu utilisées, ce qui est fréquent dans du code système.

L'approche de la fouille de spécifications détecte les ensembles dans lequels les séquences de fonctions qui sont communément utilisées ensembles et nécessaires pour représenter les protocoles requis pour effectuer une tache particulière. De telles approches souffrent en générale d'un haut taux de faux positifs, [44], et ainsi utilisent des formes d'élaguage et de classement pour rendre les spécifications qui correspondent le mieux les plus apparentes pour l'utilisateur. Les métriques communes incluent le support et la confiance, ou des variantes [49, 62, 82, 84, 90], comme par exemple le "classement-z" utilisé par Engler *et al.* [27]. Le support est le nombre de fois que le protocol est respécté à travers la base de code, alors que la confiance est le pourcentage d'occurences d'une partie du protocole qui satisfait le protocole complet. L'outil de fouille de spécification PR-Miner, par exemple, qui a été appliqué au code de Linux [46], a été évalué avec un seuil qui a pour conséquence d'élaguer les rapports de fautes dans lesquelles le protocol associé n'a pas un support d'au moins 15 et une confiance d'au moins 90%.

En utilisant les heuristiques que nous présenterons au Chapitre 4 pour identifier les ressources liées aux fonctions d'aquisition et de libération, nous identifions 2747 protocoles potentiels dans Linux, et 1051 dans les autres logiciels considérés (Wine, PostgreSQL, Appache, Python, and PHP). La Figure 5.5 montre le support et la confiance de chacun, déterminés par une analyse interprocédurale. Chaque point ou $\times$ sur cette figure représente un ou plusieurs protocoles avec les même valeurs de support et de confiance. Pour Linux, seulement 3% des protocoles ont à la fois un support de 15 ou plus, et une confiance de 90% ou plus. 88% ont un support en dessous de 15 et 58% ont une confiance en dessous de 90%. Pour chacun des autres logiciels, 3% des protocoles ont à la fois un support de 15 ou plus, et une confiance de 90% ou plus. 81% ont un support en dessous de 15 et 68% ont une confiance en dessous de 90%. Les distributions sont ainsi toutes similaires au niveau du noyau et du



(a) Linux

(b) Other Software (Wine, PostgreSQL, Appache, Python, and PHP)

Figure 5.5: Support et confiance des protocoles identifiés

niveau utilisateur. Les fautes dans l'utilisation de presque tous ces protocoles seraient négligées dans une apprche avec fouille de spécifications utilisant ces seuils. Réduire ces seuils pourrait augmenter significativement le nombre de faux positifs. Il y a ainsi un besoin pour une approche de détection de fautes capable trouver des fautes dans l'utilisation de protocoles ayant un support et une confiance plus basse.

Dans cette thèse, nous proposons une approche alternative qui cible spécifiquement les proprietés de code de gestion d'erreurs (CGE) dans les systèmes logiciels écrits en C. Nous observons que *quand un bloc de code de gestion d'erreurs a besoin d'une opération de libération de ressources donnée, les codes de gestion d'erreurs proches ont généralement besoin de la même opération.* En se basant sur cette observation, nous proposons un algorithme de recherche *microscopic* de fautes d'omission de libération de ressources, basé sur une analyse de flux et de chemins en grande partie interprocedurale, qui cible et exploite les propriétés de code de gestion d'erreurs. Notre algorithme est resistant aux faux positifs dans l'ensemble des opérations d'aquisition et de libération de ressources, résultant en un faible taux de faux positifs dans les rapports de fautes, et passe très bien à l'échelle. Il trouve des fautes d'omission de libération de ressources *quel que soit* le nombre de fois que les opérations d'aquisition et de libération associées sont utilisées ensemble à travers la base de code, et est independant de la stratégie pour les identifier. Il se concentre sur l'endroit où une libération de ressources est nécessaire, en se basant sur les informations trouvées dans la même fonction, et n'est

pas géné par des informations dérivés d'une autre partie du système. Comme preuve de concepte, nous fournissons une implémentation, Hector,[5] qui utilisent des heuristiques et des analyses en grande partie interprocedurales pour identifier des opérations de libération de ressource. Hector *ne nécéssite aucune liste fixé ou fournie par l'utiisateur d'opérations de libération de ressource* et *ne dépend pas des résultats les plus fréquents obtenus par un balayage global*, mais qui atteint toujours un faible taux de faux positifs.

Les principales contributions de notre travail sont:

- Nous mettons en évidence le fait que les fautes d'omissions de libération de ressources dans les codes de gestion d'erreurs sont un problème important, qui peut conduire à des arrêts brutaux, l'indisponibilité de ressources, et l'épuisement de la mémoire. La plupart des codes de gestion d'erreurs sont rarement exécutés, rendant les fautes difficiles à trouver par les tests.

- Nous montrons que les outils existants pour trouver les fautes dans du code système ne sont pas capables de trouver la majorité de ces fautes. Ces outils se reposent en effet sur la fréquence d'utilisation des fonctions pour réduire le nombre de faux positifs.

- Nous proposons un algorithme de détection de fautes d'omission de libération de ressources basé sur l'observation que le motif de code trouvé dans une fonction peut fournir une idée des besoins dans le reste du code à l'interieur de la même fonction. Les applications de cette approche sont illustrées par le fait que dans les systèmes logiciels considérés, jusqu'à 43% du code est dans des fonctions qui contiennent de multiples blocs de code de gestion d'erreurs.

- En utilisant Hector, nous trouvons 485 fautes d'omission de ressources dans 19 systèmes logiciels, avec un taux de faux positif de 23%.

- Parmis les 485 fautes, il y a 371 fautes d'omission de libération de ressources dans les systèmes très utilisés Linux, PHP, Python, Apache, Wine, and PostgreSQL. 52% des 371 fautes impliquent des couples d'aquisition et de libération de ressources qui sont utilisés ensembles moins de 15 fois dans le code, la faute associée étant peu encline à être détectée par les aproches par fouilles de données décrite précedemment. Nous avons soumis des patches basés sur beaucoup de nos résultats aux développeurs des sytèmes concernés, et ces patches ont été acceptés ou sont en attente d'évaluation.

- Nous avons trouvé que 257 des 285 fautes trouvées dans Linux causent des fuites de mémoires, et 9 conduisent à des interblocages.

- Un grand nombre des fautes de Linux que nous avons trouvé sont du code non-initialisé, affectant par exemple l'installation d'un préiphérique branché à chaud ou le montage d'un système de fichier. D'autres sont trouvées dans des fonction exécutées plus fréquemment, telles que les fonctions IOCTL ou les fonctions read/write. Les fautes détectées par Hector dans les codes de gestion d'erreurs peuvent en pratique causer des arrêt brutaux ou l'indisponibilité de certain périphériques, quand le système a une vue incohérente de l'état des ressources. Il peuvent aussi conduirent à des interblocages et à des fuites de mémoire, si le code fautif peut être itéré. Ces conditions peuvent rendre le système vulnérable à des attaques malicieuses.

[5] Les trois premières lettres de "Hector" sont une permutation de "EHC."

Plan de la thèse : Le but général de cette dissertation est d'améliorer la gestion des erreurs dans le code système. Nous avons divisé ce travail en deux parties. La première partie se concentre sur l'amélioration des structures de code de gestion d'erreurs dans le but d'aider à réduire le nombre de fautes pouvant etre présentes dans le code de gestion d'erreurs dans le futur. La seconde partie se concentre sur la recherche de fautes existantes dans le code de gestion d'erreurs des systèmes logiciels.

Le manuscrit est organisé de la manière suivante:

- **Chapitre 2 : Contexte.** Ce chapitre est composé de plusieurs sections. D'abord, il décrit brievement differents types de bugs et ceux que nous ciblons. Deuxièmement, il décrit la terminologie utilisée dans nos approches. Troisèmement, il décrit brievement les systèmes logiciels que nous utilisons pour évaluer les outils proposés, et ensuite décrit l'état actuel de la gestion d'erreurs dans les logiciels considérés en quatrième section. La cinquième section décrit l'état de l'art et le compare avec notre travail. La dernière section décrit aussi ma contribution à un papier sur une étude des fautes dans Linux qui a été publié à [ASPLOS11].

- **Chapitre 3 : Amélioration de la structure des code de gestion d'erreurs.** Ce chapitre présente d'abord un algorithme pour améliorer la structure des codes de gestion d'erreurs dans un système logiciel et décrit l'implémentation de cet algorithme. *Ce travail a été publié à [LCTES11].*

- **Chapitre 4 : Recherche de fautes dans les codes de gestion d'erreurs.** Ce chapitre présente un algorithme pour trouver les fautes d'omission de libération de ressources dans les codes de gestion d'erreurs et l'implémentation de cet algorithme dans l'outil Hector. *Une version préliminaire de ce travail a été publié à [PLOS11, Operating System Review (OSR'11)]. Ce travail a été publié à [DSN13].*

- **Chapitre 5 : Conclusion et travaux futurs.** Pour conclure la thèse, nous fournissons un résumé des leçons apprises lors de notre travail. Nous fournissons aussi des directions pour les travaux futurs dans ce domaine.

Bibliography

[1] 2007 desktop linux market survey: http://desktoplinux.com.

[2] Information about the apache web server. (n.d.). web hosting services, vps servers and domain names by ntc hosting. retrieved november 12, 2012, from http://ntchosting.com/apache-web-server.html.

[3] Pfff: Php frontend for fun, http://github.com/facebook/pfff.

[4] Php and perl crashing the enterprise party: http://news.cnet.com.

[5] Unixbench: www.tux.org/pub/nux/benchmarks/system/unixbench.

[6] Wikipedia: http://wikipedia.org.

[7] Alex Aiken, Suhabe Bugrara, Isil Dillig, Thomas Dillig, Brian Hackett, and Peter Hawkins. An overview of the Saturn project. pages 43–48.

[8] Glenn Ammons, Rastislav Bodík, and James R. Larus. Mining specifications. In *POPL 2002*, pages 4–16.

[9] Shay Artzi, Adam Kiezun, Julian Dolby, Frank Tip, Danny Dig, Amit Paradkar, and Michael D. Ernst. Finding bugs in dynamic web applications. In *ISSTA 08*.

[10] Radu Banabic and George Candea. Fast black-box testing of system recovery code. In *EuroSys 2012*, pages 281–294.

[11] A. Baratloo, N. Singh, and T. Tsai. Transparent run-time defense against stack smashing attack. In *USENIX Annual Technical Conference*, 2000.

[12] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. A few billion lines of code later: using static analysis to find bugs in the real world. *Commun. ACM*, 53:66–75, February 2010.

[13] Magiel Bruntink. Reengineering idiomatic exception handling in legacy C code. In *12th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 133–142, Athens, Greece, April 2008.

[14] Magiel Bruntink, Arie van Deursen, and Tom Tourwé. Discovering faults in idiom-based exception handling. In *28th International Conference on Software Engineering (ICSE)*, pages 242–251, Shanghai, China, May 2006.

[15] Stefan Bucur, Vlad Ureche, Cristian Zamfir, and George Candea. Parallel symbolic execution for automated real-world software testing. In *EuroSys 2011*, pages 183–198.

[16] CDT/User/FAQ – Eclipsepedia, 2010. http://wiki.eclipse.org/CDT/User/FAQ.

[17] Checkpatch. http://www.codemonkey.org.uk/projects/checkpatch/.

[18] J. D. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, and M. Sridharan. Eficient and precise datarace detection for multithreaded object-oriented programs. In *PLDI*, 2002.

[19] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. An empirical study of operating systems errors. In *SOSP 2001*, pages 73–88.

[20] Jonathan Corbet. The age of kernel code in various subsystems, February 2010. http://lwn.net/Articles/374622/.

[21] Jonathan Corbet. How old is our kernel?, February 2010. http://lwn.net/Articles/374574/.

[22] D. Cotroneo, A. Lanzaro, R. Natella, and R. Barbosa. Experimental analysis of binary-level software fault injection in complex software. In *IEEE 9th European Dependable Computing Conferene (EDCC'12)*, pages 162–172, May 2012.

[23] Static source code analysis, static analysis, software quality tools by Coverity Inc. http://www.coverity.com/, 2008.

[24] Alex Depoutovitch and Michael Stumm. Otherworld – giving applications a chance to survive OS kernel crashes. In *ACM EuroSys*, pages 181–194, Paris, France, April 2010.

[25] A. Dinning and E. Schonberg. Detecting access anomalies in programs with critical sections. In *ACM/ONR Workshop on Parallel and Distributed Debugging (AOWPDD)*, 1991.

[26] J. Duraes and H. Madeira. Emulation of software faults: A field data study and a practical approach. In *IEEE Transactions on Software Engineering*, volume 32, 2006.

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

[28] Fedora project, 2010. http://fedoraproject.org/.

[29] Fernando Castor Filho, Cecília M. F. Rubira, Raquel de A. Maranhão Ferreira, and Alessandro Garcia. Aspectizing exception handling: A quantitative study. In *Advanced Topics in Exception Handling Techniques*, volume 4119 of *Lecture Notes in Computer Science*, pages 255–274. Springer, 2006.

[30] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison Wesley, 1999.

[31] Mark Gabel and Zhendong Su. Javert: Fully automatic mining of general temporal properties from dynamic traces. In *FSE 2008*, pages 339–349.

[32] W. Gu, Z. Kalbarczyk, K. Ravishankar, and Z. Yang. Characterization of linux kernel behavior under errors. In *Dependable Systems and Networks (DSN'03)*, pages 459–468, June 2003.

[33] Haryadi S. Gunawi, Cindy Rubio-González, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Ben Liblit. EIO: Error handling is occasionally correct. In *FAST 2008*, pages 207–222.

[34] R. Hasting and B. Joyce. Purify: Fast detection of memory leaks and access errors. In *USENIX Winter 1992 Technical Conference*, pages 125–136, 1992.

[35] Jorrit N. Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S. Tanenbaum. Fault isolation for device drivers. In *2009 IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 33–42, Estoril, Portugal, June 2009.

[36] M. Herlihy and J. E. B. Moss. Transactional memory: architectural support for lock-free data structures. In *ISCA*, 1993.

[37] IEEE std 982.2-1988 IEEE guide for the use of IEEE standard dictionary of measures to produce reliable software, 1988.

[38] R. Jones and P. Kelly. Backwards-compatible bounds checking for arrays and pointers in c programs. In *International Workshop on Automatic Debugging*, pages 13–26, 1997.

[39] Asim Kadav, Matthew J. Renzelmann, and Michael M. Swift. Tolerating hardware device failures in software. In *SOSP 2009*, pages 59–72.

[40] Asim Kadav and Michael M. Swift. Understanding modern device drivers. In *ASPLOS 2012*, pages 87–98.

[41] Vineet Kahlon, Yu Yang, Sriram Sankaranarayanan, and Aarti Gupta. Fast and accurate static data-race detection for concurrent programs. In *19th international conference on Computer aided verification (CAV'07)*, pages 226–239, Berlin, Germany, July 2007.

[42] Ted Kremenek, Paul Twohey, Godmar Back, Andrew Ng, and Dawson Engler. From uncertainty to belief: Inferring the specification within. In *OSDI 2006*, pages 161–176.

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

[44] Claire Le Goues and Westley Weimer. Specification mining with few false positives. In *TACAS 2009*, pages 292–306.

[45] Zhenmin Li, , Shan Lu, Suvda Myagmar, and Yuanyuan Zhou. CP-Miner: a tool for finding copy-paste and related bugs in operating system code. In *6th USENIX conference on Operating systems design and implementation,*.

[46] Zhenmin Li and Yuanyuan Zhou. PR-Miner: automatically extracting implicit programming rules and detecting violations in large software code. In *ESEC/FSE 2005*, pages 306–315.

[47] Martin Lippert and Cristina Videira Lopes. A study on exception detection and handling using aspect-oriented programming. In *ICSE'00*, pages 418–427, Limerick, Ireland, March 2000.

[48] Lkml: The Linux kernel mailing list. http://lkml.org/.

[49] David Lo, Siau-Cheng Khoo, and Chao Liu. Mining temporal rules for software maintenance. *Journal of Software Maintenance and Evolution: Research and Practice*, 20:227–247, 2008.

[50] Shan Lu, Zhenmin Li, Feng Qin, Lin Tan, Pin Zhou, and Yuanyuan Zhou. Bugbench: A benchmark for evaluation bug detection tools. In *Bugs'05*.

[51] Shan Lu, Soyeon Park, Chongfeng Hu, Xiao Ma, Weihang Jiang, Zhenmin Li, Raluca A. Popa, and Yuanyuan Zhou. MUVI: automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs. In *SOSP 2007*, pages 103–116.

[52] E. Marcus and H. Stern. Blueprints for high availability. In *John Wiley and Sons, 2000*.

[53] Paul Dan Marinescu and George Candea. LFI: A practical and general library-level fault injector. In *DSN 2009*, pages 379–388.

[54] Bill McCloskey and Eric Brewer. ASTEC: a new approach to refactoring C. In *ESEC/FSE-13*, pages 21–30, Lisbon, Portugal, 2005.

[55] P. M. Melliar-Smith and Brian Randell. Software reliability: The role of programmed exception handling. In *ACM Conference on Language Design for Reliable Software*, pages 95–100, 1977.

[56] MISRA. *Guidelines for the use of the C language in critical systems*. MIRA Limited, 2004.

[57] M. Mortensen and S. Ghosh. Refactoring idiomatic exception handling in C++: Throwing and catching exceptions with aspects. In *Industry Track of the International Conference on Aspect-Oriented Software Development (AOSD)*, Vancouver, Canada, March 2007.

[58] M. Naik, A. Aiken, and J. Whaley. Effective static race detection for java. In *PLDI*, 2006.

[59] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *PLDI 2007*, pages 89–100.

[60] R. H. B. Netzer and B. P. Miller. Improving the accuracy of data race detection. In *PPoPP*, 1991.

[61] W. T. Ng and P. M. Chen. The systematic improvement of fault tolerance in the rio file cache. In *29th Symposium on Fault-Tolerant Computing (FTCS '99)*, pages 76–83, October 1999.

[62] Tung Thanh Nguyen, Hoan Anh Nguyen, Nam H. Pham, Jafar M. Al-Kofahi, and Tien N. Nguyen. Graph-based mining of multiple object usage patterns. In *ESEC-FSE 2009*, pages 383–392.

[63] Yoann Padioleau. Parsing C/C++ code without pre-processing. In *Compiler Construction (CC'09)*, pages 109–125, York, UK, March 2009.

[64] Yoann Padioleau, Julia Lawall, René Rydhof Hansen, and Gilles Muller. Documenting and automating collateral evolutions in Linux device drivers. In *EuroSys 2008*, pages 247–260.

[65] Nicolas Palix, Julia Lawall, and Gilles Muller. Tracking code patterns over multiple software versions with Herodotos. pages 169–180, Rennes and Saint Malo, France, March 2010.

[66] Nicolas Palix, Suman Saha, Gaël Thomas, Christophe Calvès, Julia Lawall, and Gilles Muller. Database of Faults in Linux: Ten Years Later, August 2010. http://hal.inria.fr/docs/00/50/92/56/ANNEX/10years.sql.pg_dump.

[67] Nicolas Palix, Gaël Thomas, Suman Saha, Christophe Calvès, Julia Lawall, and Gilles Muller. Faults in Linux: ten years later. In *ASPLOS 2011*, pages 305–318.

[68] F. Qin, S. Lu, and Y. Zhou. Safemem: Exploiting ecc-memory for detecting memory leaks and memory corruption during production runs. In *11th International Symposium on High-Performance Computer Architecture (HPCA'05)*, 2005.

[69] Murali Krishna Ramanathan, Ananth Grama, and Suresh Jagannathan. Path-sensitive inference of function precedence protocols. In *ICSE 2007*, pages 240–250.

[70] Cindy Rubio-González, Haryadi S. Gunawi, Ben Liblit, Remzi H. Arpaci-Dusseau, and Andrea C. Arpaci-Dusseau. Error propagation analysis for file systems. In *PLDI 2009*, pages 270–280.

[71] O. Ruwase and M. S. Lam. A practical dynamic buffer overflow detector. In *11th Annual Network and Distributed System Security Symposium (NDSS)*, pages 159–169, 2004.

[72] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. In *ACM TOCS*, 1997.

[73] D. Searls. Sparse, Linus & the Lunatics, November 2004. Available at http://www.linuxjournal.com/article/7272.

[74] Sparse. https://sparse.wiki.kernel.org/index.php/Main_Page.

[75] S. Sundararaman, Y. Zhang, S. Subramanian, A. C. A. Dusseau, and R. H. A. Dusseau. Making the common case the only case with anticipatory memory allocation. In *9th USENIX conference on File and stroage technologies*, pages 17–17, 2011.

[76] Michael M. Swift, Muthukaruppan Annamalai, Brian N. Bershad, and Henry M. Levy. Recovering device drivers. *ACM Transactions on Computer Systems*, 24(4):333–360, 2006.

[77] Lin Tan, Ding Yuan, Gopal Krishna, and Yuanyuan Zhou. /*icomment: bugs or bad comments?*/. In *SOSP 2007*, pages 145–158.

[78] The Kernel Janitors. Smatch, the source matcher, 2010. Available at http://smatch.sourceforge.net.

[79] Marco Torchiano, Filippo Ricca, and Alessandro Marchetto. Are web applications more defect-prone than desktop applications? In *STTT 2011*.

[80] Ubuntu, 2010. http://www.ubuntu.com/.

[81] B. Vasundhara and K. V. Chalapati Rao. Improving software modularity using aop. *International Journal of Computer Science and Informatics (IJCSI)*, 2:191–195, 2010.

[82] Andrzej Wasylkowski, Andreas Zeller, and Christian Lindig. Detecting object usage anomalies. In *ESEC-FSE 2007*, pages 35–44.

[83] Westley Weimer and George C. Necula. Finding and preventing run-time error handling mistakes. In *OOPSLA 2004*, pages 419–431.

[84] Westley Weimer and George C. Necula. Mining temporal specifications for error detection. In *TACAS 2005*, pages 461–476.

[85] Westley Weimer and George C. Necula. Exceptional situations and program reliability. *ACM Transactions on Programming Languages and Systems*, 30(2), 2008.

[86] David Wheeler. Flawfinder home page. Web page: http://www.dwheeler.com/flawfinder/, October 2006.

[87] David A. Wheeler. SLOCCount. http://www.dwheeler.com/sloccount/.

[88] Qian Wu, Guangtai Liang, Qianxiang Wang, Tao Xie, and Hong Mei. Iterative mining of resource-releasing specifications. In *ASE 2011*, pages 233–242.

[89] Weiwei Xiong, Soyeon Park, Jiaqi Zhang, Yuanyuan Zhou, and Zhiqiang Ma. Ad hoc synchronization considered harmful. In *9th USENIX conference on Operating systems design and implementation*, pages 1–8, Vancouver, Canada, October 2010.

[90] Jinlin Yang, David Evans, Deepali Bhardwaj, Thirumalesh Bhat, and Manuvir Das. Perracotta: Mining temporal API rules from imperfect traces. In *ICSE 2006*, pages 282–291.

[91] T. Yoshimura, H. Yamada, and K. Kono. Is linux kernel oops useful or not? In *8th Workshop on Hot Topics in System Dependability (HotDep'12)*, October 2012.

[92] Y. Yu, T. Rodeheffer, and W. Chen. Efficient detection of data race conditions via adaptive tracking. In *SOSP*, 2005.

[93] P. Zhou, F. Qin, W. Liu, Y. Zhou, and J. Torrellas. iWatcher: Efficient architecture support for software debugging. In *31st International Symposium on Computer Architecture (ISCA)*, pages 224–237, 2004.