Improving Error-Handling Code in Systems Software

Suman Saha

LIP6-Regal

1 Introduction

Reliability is essential in systems software. A key element of ensuring reliability is proper handling of error conditions [26]. In general, the role of error handling code is to return the system to a coherent state, typically by undoing recent operations and releasing recently allocated resources. If some of these resource-releasing operations are omitted, the result can be deadlocks and memory leaks. If resource-releasing operations are performed in the wrong order, the result can be invalid data accesses, such as null-pointer dereferences and double frees. These issues are especially critical in the case of operating systems, such as Linux, as an operating system manages many resources over a long period of time, increasing the amount of error conditions that can occur and resource-releasing operations that are needed, and heightening the accumulated impact of any memory leaks.

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 *gotos* 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.

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 code [21]. Missing resource-releasing operations lead to faults in source code. A number of approaches have been proposed to detect such problems [6, 12, 14, 25], but they often have a high rate of false positives, or focus only on commonly used functions. I observe that resource-releasing operations are often found in error-handling code, and that the choice of resource-releasing operation may depend on the context in which it is to be used.

The overall goal of my PhD work is to improve error handling in system's code. I have divided my PhD work into three parts. The first part focuses 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. The second part focuses on finding existing system faults in the error handling code. The third part focuses on fixing the detected system faults in the error handling code.

In the first half of my PhD, I have completed the first part of the PhD work by proposing an approach to improve the structure of error handling code. I have also partially completed the second part of the PhD work by proposing an approach to detect existing resource release omission faults in the error handling code.

In the second half of my PhD, I will extend the fault detection approach to detect concurrency and semantic system faults also. Concurrency faults are those that happen only in a multi-threading environment. They are cause by ill-synchronized operations from multiple threads. Semantic faults are faults that are inconsistent with the original design and the programmers' intention. Finally, in order to complete the third part of my Phd work, I will investigate how to automatically fix all detected system faults in the error handling code.

The rest of this report is organized as follows. Section 2 presents an algorithm to improve the structure if error handling code. This work was published in LCTES'11, in the paper *An Approach to Improving*

the Structure of Error-Handling Code in the Linux Kernel [24]. Section 3 then presents an approach to detect resource-release omission faults. This work has been submitted to PLOS'11 in the paper *Finding Resource-Release Omission Faults in Linux*. Section 4 presents related work. Finally, Section 5 presents several other projects I have contributed during first part of my PhD work and Section 6 concludes.

2 Improving the structure of error-handling code

In C, a typical strategy for implementing error handling code 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. I 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.

One style of programming that can somewhat alleviate these difficulties is to move the resource-releasing operations from the individual error handling conditionals to a single, labelled sequence of resource-releasing operations at the end of the function. I refer to this style of programming as the *goto-based strategy*.

I have defined an automatic program transformation algorithm that converts *basic* strategy error handling into *goto-based* strategy error handling code. I have then implemented my algorithm in OCaml (1300 lines of code) and applied it to the Linux 2.6.34 kernel. The algorithm eliminates almost 6000 lines of duplicate error handling code.

In this section, I first illustrate the basic error handling strategy and the goto-based error handling strategy using examples from the Linux 2.6.34 kernel source code. I then explain the transformation algorithm. Finally, I evaluate the transformation algorithm on the Linux 2.6.34 kernel.

2.1 Motivating Example

Figure 1a 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 shown in red. 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 between the three conditionals. For example, the statements on lines 6 and 7 in the first block of error handling code also appear on lines 15 and 16 in the second block of error handling code and on lines 25 and 26 in the third block of error handling code. The statement on line 14 also appears on line 24. Furthermore, the final call to DPRINT EXIT found in each block of error handling code also appears at the normal exit from the function. The error-handling operations free data structures of various complexity. Omitting any of this code when constructing any new error-handling code that becomes needed as the function evolves will lead to memory leaks.

Figure 1b 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 themselves 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.

2.2 Transformation Algorithm

The goal of the transformation algorithm is to merge the sequence of statements in each error-handling `if` branch into a shared sequence of statements at the end of the function, and to replace each error-handling `if`

```
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      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) {
13     DPRINT_ERR(STORVSC_DRV, "unable to add scsi vsc device");
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) {
22     DPRINT_ERR(STORVSC_DRV, "unable to add scsi host device");
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 }
```

a) *basic* strategy error handling code

```
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 scsi vsc device");
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 scsi host device");
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 }
```

b) Improved version of Figure a with *goto-based* strategy

Figure 1: Example is taken from drivers/staging/hv/storvsc_drv.c

branch by a `goto` into this sequence. The algorithm considers one function at a time. It consists of three steps.

1. 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. I identify error-handling code as an `if` branch that ends by returning an error number, *NULL* or *ERR_PTR()*.

2. The second step is to identify operations in this error handling code that can be shared in a sequence at the end of the function. If there is no such shared code, then I consider that the benefit of transforming the code does not outweigh the cost of introducing a goto.

3. The final step is to transform the function definition to move error-handling code to the end of the function and insert appropriate `gotos` into each error handling `if` branch. I refer to the code that need to be moved as *label code*. This step classifies the `if` branches, according to how difficult they are to transform: Simple, Hard, Harder and Hardest. On the basis of difficulty, the algorithm chooses the appropriate transformation. An `if` branch is classified as Simple if the label code is the same as the code in an existing label, because no code has to be moved. An `if` branch is classified as Hard if the 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. A branch is classified as Harder if an `if` branch's label code does not match a suffix of any existing label's code or the code at the original end of the function. The Hardest category is a combination of Harder and either Simple or Hard. Due to limited of spaces, I could not explain the transformation algorithm with the real example. However, the algorithm is properly explained in my LCTES'11 paper [24].

2.3 Evaluation

The algorithm, excluding the parser, has been implemented as 1300 lines of OCaml code. For the parser, I have reused the parser developed for the program transformation system Coccinelle [19, 18]. This section presents the results of applying the tool to the source code of the Linux 2.6.34 kernel (released May 2010). Linux 2.6.34 contains over 8 million lines of C code, as calculated using SLOCCount [27], and processing this code using the tool takes approximately 25 minutes on one core of an 8-code 3GHz machine with 16GB memory.

Branch 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. Table 1 shows that the number of labels introduced is significantly lower than the number of branches, reflecting a good number of Simple branches. 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. Finally, the number of return variable initializations is also much lower than the number of branches, indicating that the need to introduce a return variable to abstract over error indicators is not a major burden.

	Branches	Label	Goto	Assignment
number	4164	2791	4214	1905
avg per function		1.54	2.33	1.61

Table 1: Total number and average number of labels, gotos, and assignment statements created in the transformed Linux-2.6.34 functions.

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, these should ideally be many Simple and Hard branches as these cases introducing code sharing, and these branches should also contain a good number of state-restoring operations that can be shared. On the other hand, Harder branches, and to some extent Hardest branches, simply move existing code. Table 2 shows the number of merged and moved lines of code. Over 45% more code is merged than moved.

	Merged	Moved	Merged/Moved	Total Transformed
number	5819	3546	1.64	9365
avg per function	3.22	1.96		5.18

Table 2: Total number and average number of lines that are merged, moved, and transformed in the transformed functions.

3 Finding resource-release omission faults

The approach presented in Section 2 will help to reduce the number of system faults occurring in the error handling code in the future. However, the existing error handling code has a number of different kinds of system faults. One of them is resource-release omission faults. A challenge in detecting resource-release omission faults is to identify the set of expected resource-releasing operations. One approach that has extensively been explored is to identify pairs of functions that first allocate and then release some resource, and then to scan the code base for occurrences of an allocation without a corresponding resource-releasing operation [6, 12, 14, 25]. For some kinds of resources, however, the choice of resource-releasing operation is context-sensitive, depending on the phase within an initialization process at which a release is needed or

whether the current file has defined a specialized resource-releasing function. When the choice of resource-releasing operation depends on the context, fault-finding rules relating an allocation function to e.g., its most common resource-releasing operation, will lead to false positives in cases where the context indicates that a different resource-releasing operation is required. On the other hand, ignoring cases where the set of possible resource-releasing functions appears to contain more than one element will lead to false negatives. I am not aware of existing approaches that have addressed these issues.

In this section, I propose an approach to find resource-releasing omission faults in the Linux operating system, building on the observation that many Linux functions need to deal with multiple possible failures, and thus appropriate error handling code is often nearby. To exploit this nearby error handling code, the approach first collects a list of calls to probable resource-releasing operations, from all error handling code in a given function. Based on this information, it then searches for error handling code that has illegal omission of resource-release operations. The reliance on a list of the resource-releasing operations used anywhere in the functions error-handling code implies that the approach naturally takes into account context-sensitive constraints on the choice of resource-releasing operations. Furthermore, unlike statistics-based approaches, the approach is independent of the frequency of use of the resource-releasing operations across the code base.

In this section, I first illustrate omission faults in error handling code, using an example from *drivers* directory of the Linux 2.6.34 kernel. I then propose an algorithm to detect resource-release omission faults in the error handling code. Finally, I evaluate the algorithm on the *drivers* directory of Linux 2.6.34 kernel.

3.1 Motivating Example

In this section, I illustrate an example that contains omitted resource-release operation in error handling code. In the code shown in Figure 2, at line 3, a resource is allocated by calling the function `wl1251_alloc_hw`, and the result is stored in the variable `hw`. The error handling code on lines 7 and 8 and the error handling code on lines 17 and 18 both jump to the label `out_free`, which calls `ieee80211_free_hw` at line 21 to release `hw`. However, the error handling code in the middle of the function, on lines 12 and 13, does not have any operation that releases `hw`. This omission may lead to a memory leak. I note that the function `wl1251_alloc_hw` is only used twice in the Linux kernel, once with this resource-releasing operation and once without. Therefore, statistics-based approaches [6, 14], which use some parameters (e.g.; the number of occurrences of the protocol) to identify omitted resource-releasing operations, are not likely to detect this fault.

```
1  static int __devinit wl1251_spi_probe(struct spi_device *spi) {
2    ...
3    hw = wl1251_alloc_hw();
4    if (IS_ERR(hw)) return PTR_ERR(hw);
5    ...
6    if (ret < 0) {
7      ...
8      goto out_free;
9    }
10   ...
11   if (!wl->set_power) {
12     ...
13     return -ENODEV;
14   }
15   ...
16   if (ret < 0) {
17     ...
18     goto out_free;
19   }
20   ...
21   out_free: ieee80211_free_hw(hw);
22           return ret;
23 }
```

Figure 2: Example of an omission fault that may cause a memory leak. (net/wireless/wl12xx/wl1251 spi.c)

3.2 Detecting resource-release omission faults

I have designed an algorithm to detect resource-release omission faults. The algorithm first globally analyzes a function's error handling code to identify its resource-releasing operations, and then finds omissions of these operations in the individual blocks of error-handling code. Concretely, the algorithm performs the following steps:

1. Collect the complete set of resource-releasing operations used by a given function in its error-handling

code.

2. Compare each block of error-handling code within the function with the set of collected resource-releasing operations to determine whether any resource-releasing operations are omitted. I refer to this set of omitted resource-releasing operations as the *candidate set* for the given block of error-handling code.

3. Analyze the omitted resource-releasing operations in candidate set using heuristics to determine whether they represent probable omission faults, taking into account the context in which the omission occurs. The algorithm uses the following heuristics to identify cases in which a given resource-releasing operation is not actually needed:

   (a) The variables used to describe the released resource are undefined or have a different definition at the point of the error-handling code than at the point of the occurrence of the element of the candidate set.

   (b) The released resource is returned by the error-handling code.

   (c) The resource is released in an alternate way.

3.3 Results

As shown in Table 3, the tool generates a total of 126 reports within 78 functions of the *drivers* directory. I have manually investigated all of these reports and found that 103 represent actual faults, which come from 65 different functions. The assessment of the faults is mainly based on my own understanding of the code but patches based on some of the reports have been submitted to the maintainers of the affected code, and these patches have been accepted.

	Total reports	Faults	FP	TODO
Fault	126	103	20	3
Function	78	65	10	3

Table 3: Total number of Faults, False Positives (FP), and TODO

I also found 20 false positives within 10 functions, amounting to 16% of the total number of reports. A false positive rate of under 30% has been found to be acceptable in practice [1], and indeed the absolute number of false positives is not large. I am still investigating 3 reports from 3 functions, due to insufficient expertise in the associated APIs.

Comparison with data-mining strategies Data-mining based approaches to identifying pairs of related allocation and resource-releasing operations and other similar protocols typically use thresholds defined in terms of *support* (the number of occurrences of the protocol) and *confidence* (the number of occurrences of some relevant information that match an expected pattern vs. the number that do not) to reduce the number of false positives. The data-mining-based protocol-finding tool PR-Miner [14], for example, only reports on sets of functions that occur together at least 15 times, with a confidence of at least 90%. I have evaluated the identified faults with respect to these parameters, as shown in Figure 3. The red triangles and blue circles represent the 30 pairs of allocation and resource-releasing operations associated with the 103 identified faults. The y-axis indicates support, while the

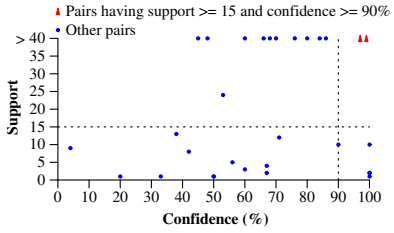

Figure 3: The red triangle dots represent the pairs that have atleast 15 min_supports and 90% confidence. The green dots represent the rest of the pairs

x-axis indicates confidence. The figure shows that only two pairs, marked as red triangles, have support greater than 15 and confidence greater than 90%. These two pairs are associated with only 10 of the 103 faults found by the approach. On the other hand, reducing the support or confidence thresholds used by data-mining-based approaches could drastically increase their number of false positives.

Context-sensitivity I observe that for some kinds of resources, the choice of resource-releasing operation may depend on the context in which releasing the resource is needed. Searching for one type of resource-releasing operation when the context indicates that another one should be used results in false positives. In order to reduce the number of false positives, the tool must be aware of the context in which error-handling code appears.

The tool found 331 allocations of resources for which at least one resource-releasing operation seems to be omitted. I refer to these as *candidate resources*. There are some cases, mentioned in Section 3.2, where a resource-releasing operation is not needed. Say, a resource-releasing operation seems to be omitted, but another resource-releasing operation is used instead. 22.4% of the 331 candidate resources are released within a single function by two different operations while 4.2% of the candidate resources are released by three different operations. A resource can be referenced via another pointer. Any resource-releasing operation on that pointer may release the resource as well. 5.2% of the candidate resources are released in this manner. Finally, a resource is released by an intervening function when this function fails to perform a specific task. 15.7% of the candidate resources are released in this manner. Moreover, 14.8% are released by a call to some other function that is defined in the same file. The tool takes these issues into account and does not generate reports in these cases.

4 Related Work

My work is to improve error handling code in system by transforming *basic*-strategy error handling into *goto-based* strategy error handling. and finding faults in error handling. This section briefly describes some existing works those are related to refactoring programming code or finding faults from system code.

Refactoring programming code In existing work, a number of studies have classified the kinds of exceptions that can occur, the kind of exception handling that is required, and the kinds of exception handling abstractions that are provided by current programming languages, as well as proposing new exception handling abstractions [4, 9, 10]. I consider exceptions that simply abort the subcomputation, and my proposed improvement to exception handling stays within the constructs available in C. My work addresses the issues of readability and uniformity, which these studies have identified as critical.

Bruntink *et al.* study properties of exception handling in a large industrial C code base [3]. They focus on the error-proneness of the exception handling mechanism based on the return code idiom and logging of exceptional conditions. They showed this idiom is omnipresent as well as highly tangled and requires focused and well-thought programming. At first, they characterize the return code idiom in terms of the existing model for exception handling mechanisms. Based on the characterization, they define a fault model for exception handling. This model identify when a fault occurs and what happens when a fault occurs. Based on the fault model they then develop a static analyzer to detect violations to the return code idiom in the source code. Finally, they provide an alternative approach to write exception handling code by hiding some of the implementation details. My work does not provide any alternative approach to write error handling code. It does use existing *goto-based* strategy to improve the structure of error handling code. This is always better to improve the existing technique rather than introducing new one.

Filho *et al.* present a technique to transform the exception handling code of a Java program into an aspect [7], providing modularity and reuse. 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 [17]. Bruntink performs a similar study on C code, using hypothetical `try` and `catch` constructs [2]. C does not provide any such abstractions. My work in Section 2 is concerned with improving the structure

of the error handling code that is present, potentially helping the user find problems in the usage of the state-restoring operations, rather than ensuring that all error conditions are checked for.

My transformation can be considered to be a form of refactoring, since it changes the structure, but not the semantics of the code [8]. Few tools support refactoring C code. Eclipse provides the CDT development environment for C and C++ code, but the support for refactoring seems to be incomplete [5]. McCloskey and Brewer propose Asfact for refactoring C code [15]. They describe that macros in C program are difficult to analyze and are often error-prone. Therefore, for a replacement of macro language, they define a new syntactic macro language, that addresses the most important deficiencies of the preprocessors and that eliminates many of the errors that the macro language introduces. They then provide an approach that translates CPP macros, include files, and preprocessor conditionals into semantically equivalent declarations in the proposed language. Their idea completely eliminates the C preprocessor from the refactoring process. It permits complex transformation to be applied directly to source code, without an initial preprocessing step.

Improve quality of programming code Weimer and Necula present a static data flow analysis on exception handling code for finding bugs in how programs deal with important resources in the presence of exceptional situations [26]. To find defects in programs they formalize s ome 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 to help programmers avoid such mistakes. I do not propose new programming language features, but instead show how to restructure error-handling code to make it less error-prone.

MISRA is a software development standard [16] for the C programming language developed by MISRA (Motor Industry Software Reliability Association) The use of *goto-based* strategy to write error handling code is restricted in MISRA C. In this work, they describe that goto can indicate badly constructed and incomprehensible logic, making testing difficult. Despite this, the use of goto has several benefits which have been described in Section 2. 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.

Finding faults in system code 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. [6] and Li et al. [14] both propose variations of this approach.

Engler *et al.* [6] use static analysis to automatically extract programming rules from source code without prior knowledge of the system. They define six templates checkers according to the extracted programing rules. They then use those templates to find contradictions of those templates in the source code. Any contradiction implies the existence of an error in the code. Finally, they use a statistical analysis to rank each error by the probability of its rules. 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.

Li *et al.* [14] propose a method called PR-Miner that uses a data mining technique called frequent itemset mining to efficiently extract implicit programming rules from large software code. 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. Based on the programming rules generated, PR-Miner can find bugs by detecting violations to these rules. The main idea again is that the programming rules usually hold for most cases and violations happen only occasionally. PR-Miner then prunes the false violations using inter-procedural analysis. After PR-Miner detects rule violations and prunes false positives, it ranks all remaining violations and reports them to programmers. It ranks the violations based on the confidence (the number of occurrences of some relevant information that match an expected pattern vs. the number that do not) of the violated rules.

Kremenek et al. [11] present a framework based on factor graphs for automatically inferring specifications directly from programs. Ramanathan et al. [23] integrate mining within a path-sensitive dataflow framework to define potential preconditions of a procedure. Le Goues and Weimer [13] integrate extra information about nonfunctional code characteristics such as churn and author expertise. In each case, these specifications can be used to find faults in the source code. Palix *et al.* have used Coccinelle to conduct a study of faults in versions of Linux and several other open source projects released between 2005 and 2009 [20]. They proposed Herodotos, in order to correlate the faults between releases.

My approach in Section 3 is completely different from these approaches in that it relies entirely on local information rather than a global analysis of the software. As compared to other approaches, my approach may result in false negatives, when error handling code is omitted and there is no relevant code nearby. But, as I have shown, it can also find faults in the use of protocols that are likely to be overlooked or given a low rank by other approaches.

5 Other Work as Part of the PhD

At the beginning of my PhD, I contributed to paper *Faults in Linux: Ten Years Later* [22], subsequently published at ASPLOS 2011. This work transported the experiments on Linux versions 1.0 through 2.4.1 of Chou *et al.* [6] about faults in system code to the more recent Linux versions 2.6.0 to 2.6.33, released between late 2003 and early 2010. The main goal of this work was to find the answers to the questions : What has been the impact of these changes on code quality? Are drivers still a major problem? I contributed to the analysis of the generated bug reports used in this work.

Currently, I am involved with a work that proposes an approach to automatically constructing a debugging interface for the Linux kernel from the definitions of the functions that are exported by the kernel. This approach is based on static analysis of the in-kernel code, to infer the preconditions of each exported function, making it possible to construct an interface containing a wrapper for each identified function. In this work, I have measured the prevalence of the different types of safety holes (code fragments in function definitions that can cause the kernel to crash or hang if the function is used improperly) in different versions of Linux kernel code.

6 Conclusion

The first part of my PhD work has been described in this report. In this work, I have proposed two approaches to improve the error handling in system. The first approach is an automatic transformation that converts error-handling code that is dispersed and duplicated throughout the body of a function such that it uses the `goto`-based strategy. I have found that the transformation applies to many functions across the Linux kernel, and that it identifies many opportunities for code sharing. The second approach focused context-sensitivity constraints on the choice of resource-releasing operations, and used this as a guideline for finding of omitted resource-releasing operations in the error handling code of the Linux kernel. The proposed approach finds a number of probable faults in error handling code with only a small number of false positives. I have shown in particular that taking context-sensitivity into account significantly reduces the number of false positives.

In the second part of the PhD work, I plan to extend my work in several dimensions. First, I will extent my faults detection approach that will detect concurrency and semantic system faults also. Second, I will investigate how to automatically fix all detected system faults in the error handling code.

References

[1] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler. A few billion lines of code later: using static analysis to find bugs in the real world. *Commun. ACM*, 53:66–75, Feb. 2010.

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

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

[4] P. A. Buhr and W. Y. R. Mok. Advanced exception handling mechanisms. *IEEE Trans. Software Eng.*, 26(9):820–836, 2000.

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

[6] D. R. Engler, D. Y. Chen, A. Chou, and B. Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *Proceedings of the 18th ACM Symposium on Operating System Principles*, pages 57–72, Banff, Canada, Oct. 2001.

[7] F. C. Filho, C. M. F. Rubira, R. de A. Maranhão Ferreira, and A. 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.

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

[9] A. F. Garcia, C. M. F. Rubira, A. B. Romanovsky, and J. Xu. A comparative study of exception handling mechanisms for building dependable object-oriented software. *Journal of Systems and Software*, 59(2):197–222, 2001.

[10] J. B. Goodenough. Exception handling: Issues and a proposed notation. *Commun. ACM*, 18(12):683–696, 1975.

[11] T. Kremenek, P. Twohey, G. Back, A. Ng, and D. Engler. From uncertainty to belief: Inferring the specification within. In *OSDI*, pages 161–176, Nov. 2006.

[12] J. L. Lawall, J. Brunel, R. R. Hansen, H. Stuart, G. Muller, and N. Palix. WYSIWIB: A declarative approach to finding protocols and bugs in Linux code. In *The 39th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 43–52, Estoril, Portugal, June 2009.

[13] C. Le Goues and W. Weimer. Specification mining with few false positives. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 5505 of *Lecture Notes in Computer Science*, pages 292–306, York, UK, Mar. 2009.

[14] Z. Li and Y. Zhou. PR-Miner: automatically extracting implicit programming rules and detecting violations in large software code. In *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 306–315, Lisbon, Portugal, Sept. 2005.

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

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

[17] 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, Mar. 2007.

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

[19] Y. Padioleau, J. Lawall, R. R. Hansen, and G. Muller. Documenting and automating collateral evolutions in Linux device drivers. In *EuroSys 2008*, pages 247–260, Glasgow, Scotland, Mar. 2008.

[20] N. Palix, J. Lawall, and G. Muller. Tracking code patterns over multiple software versions with Herodotos. In *Proc. of the ACM International Conference on Aspect-Oriented Software Development, AOSD'10*, pages 169–180, Rennes and Saint Malo, France, Mar. 2010.

[21] N. Palix, G. Thomas, S. Saha, C. Calvès, J. Lawall, and G. Muller. Faults in Linux: Ten years later. In *Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Newport Beach, CA, USA, Mar. 2011.

[22] N. Palix, G. Thomas, S. Saha, C. Calvès, J. Lawall, and G. Muller. Faults in Linux: Ten years later. In *Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2011)*, pages 305–318, Newport Beach, CA, USA, Mar. 2011.

[23] M. K. Ramanathan, A. Grama, and S. Jagannathan. Path-sensitive inference of function precedence protocols. In *29th International Conference on Software Engineering*, pages 240–250, Minneapolis, MN, USA, May 2007.

[24] S. Saha, J. Lawall, and G. Muller. An approach to improving the structure of error-handling code in the linux kernel. In *ACM SIGPLAN/SIGBED Conference on Languages, Compilers, Tools and Theory for Embedded Systems (LCTES)*, Chicago, IL, USA, Apr. 2011.

[25] W. Weimer and G. C. Necula. Mining temporal specifications for error detection. In *Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 3440 of *Lecture Notes in Computer Science*, pages 461–476, Edinburgh, UK, Apr. 2005.

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

[27] D. A. Wheeler. SLOCCount. `http://www.dwheeler.com/sloccount/`.