EHCtor: Detecting Resource-Release Omission Faults in Error-Handling Code for Systems Software

Suman Saha, Jean-Pierre Lozi

ComPAS'2013 : RenPar'21/ SympA'15/ CFSE'9
Grenoble, France, du 16 au 18 janvier 2013

EHCtor: Detecting Resource-Release Omission Faults in Error-Handling Code for Systems Software

Suman Saha and Jean-Pierre Lozi

Abstract
Adequate error-handling code is essential to the reliability of any system. On an error, such code is responsible for releasing acquired resources to restore the system to a viable state. Missing resource-release operations can lead to system crashes, memory leaks and deadlocks. A number of approaches have been proposed to detect such problems, but they mainly target frequently occurring resource-release operations. In this paper, we propose a novel approach to finding resource-release omission faults, focusing on error-handling code. Our approach achieves precision and scalability by exploiting information available within each function definition itself. Using a tool, EHCtor, that we have developed based on this approach, we have found over 370 faults in six different C infrastructure software projects, with a false positive rate 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.

1. Introduction

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 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 [15], and much of systems code is concerned with error detection and handling.

Despite its importance to systems software, error recovery has received insufficient attention in the systems community. Previous work has focused on error detection and propagation [7, 8, 19], *i.e.*, the correctness of tests and return values, and on fault-injection prioritization with the goal of exercising error recovery code [2]. Error detection and propagation, however, are not enough; a function detecting an error must also undo any previous operations that could leave system resources in an inconsistent state. *Omission* of a resource-release operation in such *error-handling code* can lead to crashes, deadlocks, and resource leaks. To our knowledge, detecting these omission faults in error-handling code has not previously been specifically targeted.

Detecting resource-release omission faults is challenging because it requires knowing the set of operations that are required to restore the system to a viable state. Typical systems software relies on a wide range of resources, each associated with many different dedicated operations, making it difficult for any given developer to be familiar with all of them. Approaches have been proposed to automatically identify these operations by some form of *specification mining* [1, 5, 6, 10, 12, 13, 17, 22, 24, 25], in order to find arbitrary faults in function usage. Nevertheless, to reduce the number of false positives, these approaches mostly focus on frequently

occurring functions. In practice, however, operations on specific types of resources may appear in the code only rarely. Existing specification mining-based approaches are thus insufficient for finding many of the faults in error-handling code.

In this paper, we propose a resource-release omission fault finding tool, EHCtor, that targets and exploits the properties of error-handling code (EHC) in C programs. EHCtor finds resource-release omission faults *irrespective* of the number of times the associated acquisition and release functions are used together across the code base, and focuses on the resources themselves, rather than the particular functions that manipulate them. To provide both scalability and precision, EHCtor primarily exploits information available within a single function, specifically the information that can be derived from the function's own error-handling code. *Our key observation is that when one block of error-handling code needs a given resource release operation, nearby error-handling code typically needs the same operation.* Thus, the existing error-handling code within a function can be used as an *exemplar* for the operations that should be performed by other error-handling code in the same function. By exploiting this observation, EHCtor *does not require any fixed or user-provided list of resource-release functions* and *does not depend on the most frequent results obtained by a global scan*, like previous approaches.

EHCtor can be applied to real-sized systems software. We have applied EHCtor to Linux 2.6.34[1] drivers (including sound drivers), networking code, and filesystems, as well as to five widely used open-source systems software projects: PostgreSQL, Apache, Wine, Python, and PHP. 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.

EHCtor finds 371 faults, with an overall false positive rate of 23%, which is below the threshold of 30% that has been found to be acceptable to developers [3]. We find omission faults involving 150 different pairs of resource acquisition and release functions. 52% of these pairs of resource acquisition and release functions are used together in the code fewer than 15 times, making the associated faults unlikely to be detected by previous specification-mining based approaches.

The rest of this paper is organized as follows. Section 2 presents some examples that motivate our work. Section 3 presents the fault-finding algorithm on which EHCtor is based. Section 4 evaluates the results obtained by applying EHCtor to large systems software projects. Finally, Section 5 presents related work and Section 6 concludes.

2. Motivation and Background

We first present two examples of faults in error-handling code found by EHCtor. 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 system software.

2.1. Linux resource-release omission faults

We motivate our work using the examples of crashes and memory leaks derived from faults in Linux error-handling code. One 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 fault was previously unreported; we have reported it to the appropriate maintainers and provided patches. [2] The unreported fault involves rarely used acquisition and release functions that would be unlikely to be taken into account by a specification-mining based approach.

[1] Linux 2.6.34 was released in May 2010. We have taken a somewhat older version to prevent our previous contributions to the Linux kernel from interfering with our results.

[2] http://lkml.org/lkml/2012/4/14/41

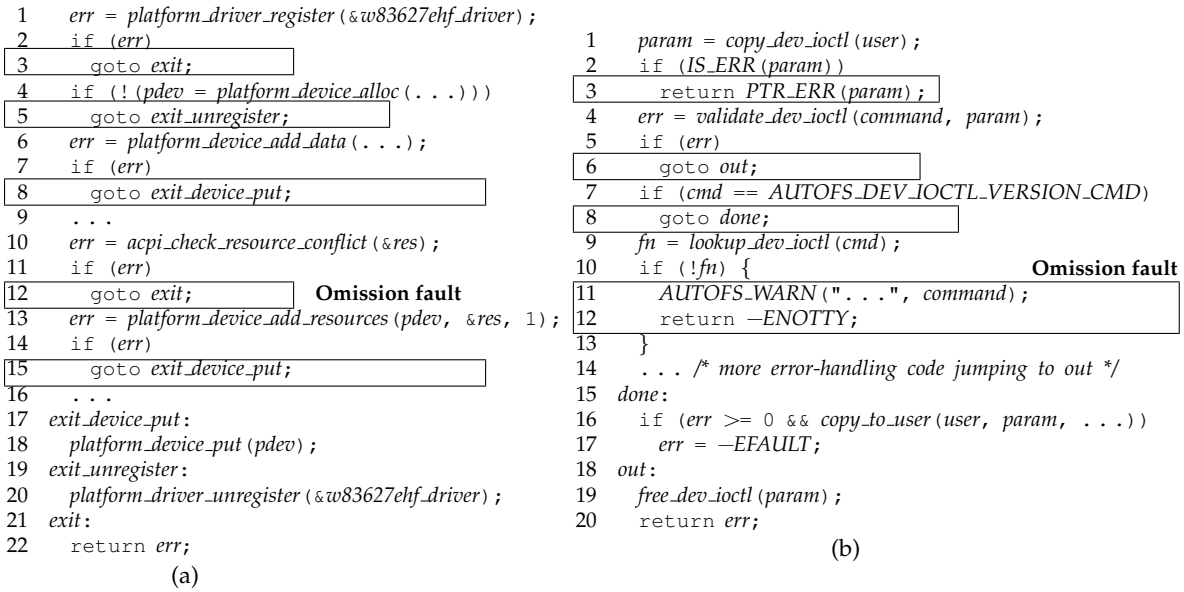


Figure 1: a)Omission fault in w83627ehf driver. b) Omission fault in Autofs4

Crash in handling a resource conflict: In January 2009, a user of the Fedora Rawhide (development) kernel found that installing the w83627ehf driver crashed his machine.[3] Fig. 1(a) shows an extract of the code containing the fault. This extract 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 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 are reported to cause the system to crash.

Memory leak in the handling of invalid user inputs: Using EHCtor, we found a previously unreported memory-release omission fault in the autofs4 IOCTL function. 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 less than 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 non-privileged users on the NFS server. Then, the attacker, connected as a non-priviliged 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.

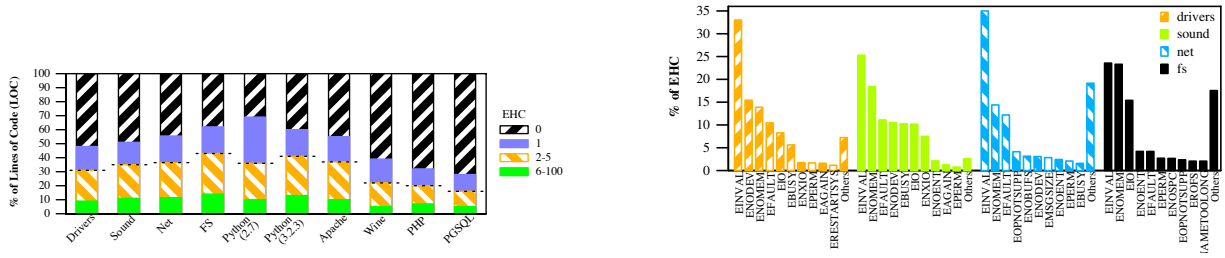
[3] https://bugzilla.redhat.com/show_bug.cgi?id=483208

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

Table 1: Considered software

Project	(Lines of code)	Version	Description
Linux `drivers`	(4.6MLoC)	2.6.34	Linux device drivers
Linux `sound`	(0.4MLoC)	2.6.34	Linux sound drivers
Linux `net`	(0.4MLoC)	2.6.34	Linux networking
Linux `fs`	(0.7MLoC)	2.6.34	Linux file systems
Wine	(2.1MLoC)	1.5.0	Windows emulator
PostgreSQL	(0.6MLoC)	9.1.3	Database
Apache httpd	(0.1MLoC)	2.4.1	HTTP server
Python	(0.4MLoC)	2.7.3	Python runtime
Python	(0.3MLoC)	3.2.3	Python runtime
PHP	(0.6MLoC)	5.4.0	PHP runtime

Both considered versions of Python are in current use.



(a) (b)

Figure 2: a) Percentage of code found within functions that have 0 or more blocks of error-handling code. b) Distribution of integer error-code return values

The fault is located in the error-handling code starting on line 11 in Fig. 1(b). Nearby error-handling code jumps to the end of the function where *param*, which was allocated on line 1, is freed on line 19. The faulty error-handling code, however, just returns `-ENOTTY`, leaking the *param* structure. The 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, 3.4.5.

2.2. Systems error-handling code

The interest of our approach depends on the importance of error handling code in system software. To assess the importance of such in systems software, we consider the amount of code that is found within functions that contain error-handling code and the kinds of errors that are detected. Our study focuses on the main subdirectories of the Linux 2.6.34 kernel as well as a selection of other widely used systems software, as summarized in Table 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. Fig. 2(a) 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 more than one block 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 whether they are consistent.

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 the Linux `drivers`, `sound`, `net`, and `fs` directories, returns integer error codes, understood by the user-level standard library function `perror`, to indicate the error cause. We rely on these error codes to obtain an overview of the reasons for the errors encountered in Linux.
Fig. 2(b) 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 ultimately 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 for most of the subsystems. 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. For `drivers`, the second most common constant is `ENODEV`, which is also common in `sound`. This constant indicates the unavailability of a device, as may be triggered by defective hardware. Another common constant is `EFAULT`, indicating a bad address. This constant 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.

3. Fault Detection and Ranking with EHCtor

The concept behind EHCtor is that *resource-release operations that are needed in one block of error-handling code are likely to be needed in other nearby blocks of error-handling code in the same function*. To create an algorithm based on this intuition, we first need a strategy for identifying error-handling code and resources. The C language does not provide any specific abstractions for these entities, and thus our algorithm begins with a preprocessing phase that identifies them using heuristics. Next, we observe that not every resource-release omission is a fault. It may be that the release is not yet needed, or has already taken place. The algorithm thus includes an analysis that identifies and discards cases where an omitted resource-release operation is indeed not needed. The result of the algorithm is a set of reports consisting of those omissions that represent likely faults. To guide the user in identifying the real faults in the code, we propose a strategy for ranking these reports.

3.1. Detecting faults

Preprocessing phase: This phase recognizes a *block of error-handling code* as a conditional that ends, possibly after one or more gotos, 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 12 of Fig. 1(b), and calls to the functions `ERR_PTR` and `PTR_ERR`, as illustrated in line 3 of Fig. 1(b). Information about these error values must be provided by the user in a configuration file. Because error values are used for communicating between different parts of the software, they typically change rarely and are well known. A block of error-handling code may also return no value, or return a variable whose value is not statically apparent, as illustrated in line 22 of Figure 1(a). In this case, the conditional is considered to

be a block of error-handling code if the test expression checks for an error value. A conditional that directly returns the result of a function that it calls, other than one of the error functions mentioned above, is never considered to be a block of error handling code, as in this case, the called function must handle any errors.

A *resource* is typically represented by a collection of information, and is thus implemented by a pointer to a structure or buffer. Resource acquisition and release are typically complex operations, and are thus implemented by function calls. The algorithm recognizes an acquisition as a function call that returns a pointer-typed value, either directly or via a reference argument (`&x`). It recognizes a release as the last operation on a resource in a given execution path. A release operation should have only one acquired resource among its arguments, which is assumed to be the released resource. We furthermore require that a release operation have no string argument, as string arguments are typically used in debugging code. To improve accuracy, within the file containing the analyzed function, we identify resource-release operations interprocedurally. A function call that has an acquired resource as an argument and whose definition contains a release operation for that resource is also considered to be a release operation.

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 when the resource is encapsulated in the function itself. To account for these cases, we also consider a function call having at most one argument where the argument has pointer type and is not involved in an earlier resource acquisition as being a resource acquisition. The corresponding release is similar, but must occur in a block of error-handling code.

Not every pointer-typed value is a resource, not every function that returns a pointer-typed value represents an acquisition, and it is not always the case that the last operation on a pointer-typed value in a given function releases that value. To reduce the number of false positives, *i.e.*, values that represent resources that must be released but are not, our algorithm requires all of these properties: a resource is a pointer-typed value that is acquired somewhere within the function and is released in some subsequent block of error-handling code.

Analysis phase: Omission of a resource-release may represent a fault, or it may be legitimate, *e.g.*, because the resource is *not yet acquired*, has *already been released*, or has been passed to another function that has the *side-effect* of releasing the resource on success, on failure, or both. The analysis phase distinguishes between these cases. In presenting the analysis phase, we refer to a block of error-handling code that releases a given resource as an *exemplar* for the release of that resource and a block of error-handling code that does not release that resource as a *candidate fault*. This phase then uses the exemplars to determine whether each candidate fault is a real fault.

The analysis phase considers that a resource-release may *not yet be required* if a dataflow analysis shows that a possible value of the resource reaching the candidate fault is different from any possible value reaching any exemplar. Consider the block of error-handling code starting on line 5 in Fig. 1(a), which is missing the release of `pdev`. Exemplars for the release of `pdev` are found in the block of error-handling code starting on line 8 and the block of error-handling code starting on line 15. The exemplars and the candidate fault are all reachable from the definition of `pdev` in line 4. For the exemplars, however, `pdev` is known to be non-`NULL`, based on the conditional test enclosing the initialization, while in the case of the candidate fault, it is known to be `NULL`. Since these possible values are incompatible, the algorithm determines that it has no information about whether releasing `pdev` is necessary at the point of the candidate fault, *i.e.*, the release is *not yet* known to be *acquired*, and no fault is reported. It is indeed not a fault.

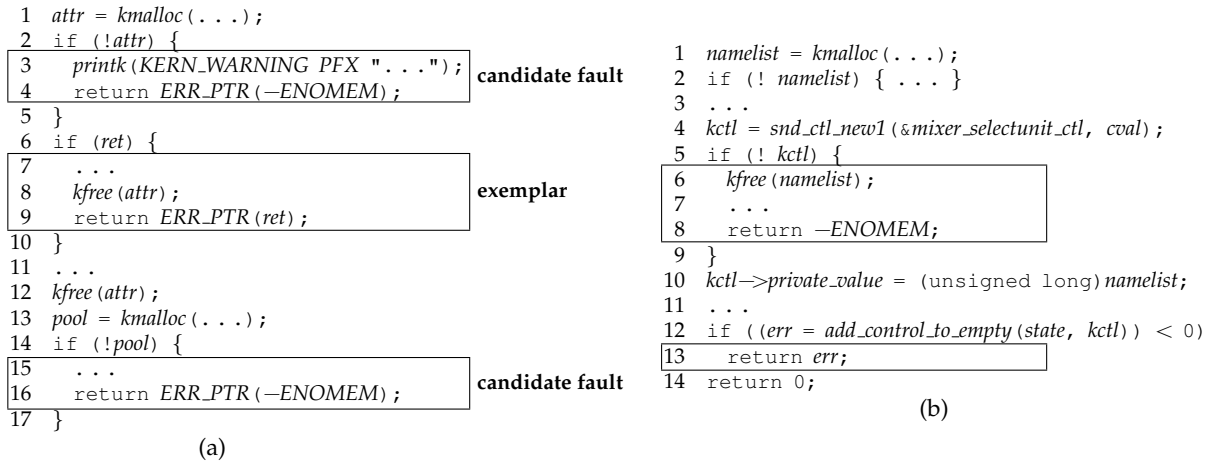


Figure 3: a) Extract of ib_create_fmr_pool b) Extract of parse_audio_selector_unit.

The analysis phase considers that the resource may *already be released* when the candidate fault is preceded by a function call that satisfies the constraints for being a release of the resource and that does not appear in any execution path leading to an exemplar. In Fig. 3(a), the block of error-handling code (starting on line 7) releases `attr` while the second block of error-handling code (starting on line 15) does not. In both cases, the only reaching definition of `attr` is the one on line 1. However, preceding the execution of the error-handling code starting on line 15, there is a call to `kfree` on `attr` in line 12. This call has the form of a resource-release operation, in that the resource is the only pointer-typed argument and there is no reference to `attr` in the execution path to the error-handling code after this call. Furthermore, this call does not appear in the execution path to the exemplar, implying that there is no information that a resource-release operation is necessary after this call. Note that these considerations are independent of the fact that `kfree` is a well-known resource-release function, and that it is the resource-release function used by the exemplar. The algorithm thus determines that it has no information about whether a resource release is still necessary, *i.e.*, the resource may *already be released*, and does not report the fault. The omission is indeed not a fault.

Finally, we consider the case where the resource is *released as a side-effect* of some other operation. In Fig. 3(b), the resource `namelist` is allocated on line 1, and stored within a field of the resource `kctl` on line 10. The resource `namelist` is freed in the error-handling code starting on line 6, but is not freed in the error-handling code on line 13. The analysis phase observes, however, that `kctl`, from which `namelist` is reachable, is not used on either success or failure after the call to the non-local function `add_control_to_empty` on line 12. It thus assumes that `add_control_to_empty` has freed both `kctl` and `namelist`, and thus does not report a fault. `add_control_to_empty` does indeed free both `kctl` and `namelist`, and thus this omission is not a fault.

As this example illustrates, our analysis does take into account aliases such as `kctl` that are constructed within a control-flow path, and assumes that if a resource is released then all resources that are reachable from it are released as well.

3.2. Ranking the fault reports

To help the user of a fault-finding tool focus on the reports that are the most likely to represent real faults, a standard approach is to rank the reports in some way. We propose a novel ranking

strategy that reflects the properties of error-handling code.
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 Control flow graph (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 know that a release has been necessary, and 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 resource-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 we know that the resource has previously reached a state in which it should be released, but may be released already.
For example, in Fig. 1(a), 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. Experimenting with EHCtor

The goals of our experiments with EHCtor are 1) to determine its success in finding faults, 2) to compare the results obtained using our algorithm with those of related approaches, 3) to understand the reason for any false positives. EHCtor consists of around 3500 lines of OCaml code, excluding the code for the parser and abstract syntax, which we have borrowed from the open-source transformation tool Coccinelle.[5] We evaluate EHCtor on the large, widely used open-source software projects described in Table 1. For this, we have analyzed 10.5 million lines of C code, in total. Our tests have been carried out on one core of a 8-core 3GHz machine with 16GB memory. Analyzing *e.g.*, Linux `drivers` (4.6 MLOC) takes around 3 hours.

4.1. Found faults

As shown in Table 2, EHCtor generates a total of 484 reports for all projects. We manually investigated all reports and found that 371 of them, from 247 different functions, represent actual faults. These faults occur in the use of 150 pairs of resource acquisition and release operations. There are 113 false positives. We examine the reasons for these false positives in Section 4.2.

Table 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
Total	**484 (314)**	**371 (247)**	0.0018	**343**	**9**	**19**

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

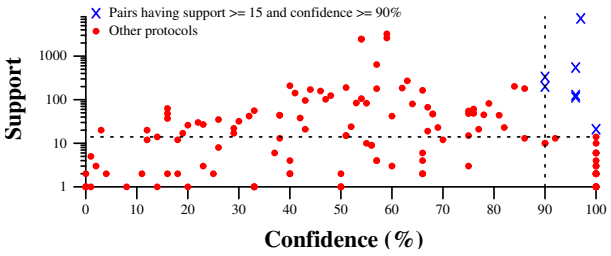


Figure 4: Support and confidence associated with the functions found in the faults reported by our algorithm. The dotted lines mark support 15 and confidence 90%.

Comparison to specification mining

Specification mining approaches detect sets or sequences of functions that are commonly used together and that represent the required *protocol* for carrying out a particular task. Such approaches typically suffer from a high rate of false positives [11], and thus use some form of pruning and ranking to make the most likely specifications the most apparent to the user. Common metrics used in pruning and ranking include *support* and *confidence*, or variants thereof [5, 13, 17, 22, 24, 25]. Support 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 tool PR-Miner, for example, which has been applied to Linux code [12], 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%

Fig. 4 shows the support and confidence for the protocols involved in our identified faults. The $\times$s and circles represent the 150 pairs of resource acquisition and release operations associated with the 371 faults identified by EHCtor, and the y-axis indicates support, while the x-axis indicates confidence. Protocols associated with 52% of the faults found by EHCtor have support less than 15, and protocols associated with 86% of the faults found by EHCtor 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 (6%) of the 371 faults found by EHCtor, implying that 94% of the faults found by EHCtor would be overlooked when using these thresholds. Indeed, the well-known Linux protocol *kmalloc/kfree*, for which we find 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 data-mining-based approaches could drastically increase their number of false positives. EHCtor finds faults independent of the support and confidence of the protocol.

4.2. False positives

Figure 5(a) shows the number of false positives among the reports generated by EHCtor and the reasons why these reports are considered to be 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 [3]. 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 newly acquired resources (Not alloc, 26%), or for recognizing existing resource releases, whether via an alias (Via alias, 29%) or via a non-local call (Non-local call frees, 12%), or releases performed in the caller of the considered function rather than within the function itself (Caller frees, 13%).

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`, four of the six false positives are due to error-handling code

		FP	Reasons					
			Not EHC	Not alloc	Via alias	Non-local call frees	Caller frees	Other
drivers	**293**	**56**	3	16	11	13	8	5
sound	**32**	**13**	0	0	13	0	0	0
net	**13**	**6**	0	0	0	0	1	5
fs	**47**	**25**	0	7	6	1	6	5
Python(2.7)	**17**	**4**	0	0	3	0	0	1
Python(3.2.3)	**22**	**2**	0	1	0	0	0	1
Apache	**5**	**2**	1	0	0	0	0	1
Wine	**31**	**1**	0	1	0	0	0	0
PHP	**16**	**3**	0	3	0	0	0	0
PGSQL	**8**	**1**	0	1	0	0	0	0
Total	**484**	**113 (23%)**	**4**	**29**	**33**	**14**	**15**	**18**

FP = False positives

(a)

	Reports			Faults			FP		
	H	LI	LF	H	LI	LF	H	LI	LF
drivers	68	127	98	62	116	59	6	11	39
sound	11	8	13	11	8	0	0	0	13
net	6	3	4	2	3	2	4	0	2
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	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
PGSQL	1	5	2	1	4	2	0	1	0
Total	**119**	**205**	**160**	**105**	**181**	**85**	**14**	**24**	**75**

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

(b)

Figure 5: (a) False Positives (b) Report ranking

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.

To help the user navigate among the reports, we have proposed a ranking strategy (Section 3.2). 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 Figure 5(a) 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 Figure 5(a)). Figure 5(b) 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.

5. Related Work

While EHCtor targets specifically omission faults in error-handling code software, several approaches have been proposed to detect omission faults more generally in infrastructure software [5, 6, 10, 12, 17, 22, 24, 25]. One heavily explored technique is to use data mining to extract implicit programming rules from source code and then to use static analysis to detect faults based on those programing rules. We first present a few of the proposed variants.

Engler *et al.* use static analysis to automatically extract programming rules from source code, based on user-defined templates [5]. Ranking calculated in terms of support and confidence is used to highlight the most probable rules. PR-Miner uses frequent itemset mining to extract programming rules, without using templates [12]. Results are pruned and ranked according to support and confidence. Kremenek *et al.* use factor graphs in automatically inferring specifications directly from programs [9]. Ramanathan *et al.* integrate mining within a path-sensitive dataflow framework to identify potential preconditions for invocation of a function [18]. In each of these cases, the identified specifications can be used to find faults in code. EHCtor 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.

Weimer and Necula observed that faults in error-handling code are common in Java, and pro-

posed a static analysis, to identify resource-release omission faults [23]. Subsequently, they proposed a specification mining approach that gives more weight to specifications derived from error-handling code [24]. While they found many faults, the specification-mining process has a high rate of false positives. To reduce the rate of false positives, Le Goues and Weimer integrated extra information such as author expertise [11]. This approach, however, also reduced the number of found faults. Furthermore, statistics are still used, so rarely used resource-release functions may be overlooked. Sundararaman *et al.* also focus on faults in error-handling code, by trying to avoid the execution of error-handling code in the first place, through the definition of an alternate memory allocator [21]. Gunawi *et al.* [7] and Rubio-González *et al.* [19] have studied faults in the detection and propagation of error values. 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. In previous work [20], we considered the use of local information to find omitted error-handling code, but the approach was evaluated only on Linux drivers, and found substantially fewer faults than found by EHCtor. Banabic and Candea propose a strategy for fault-injection prioritisation to perform run-time checking of error-handling code [2]. The reported faults involve omitted tests and duplicated releases, while EHCtor focuses on release omissions.

Another approach to detect faults is to monitor program execution. A dynamic analysis tool such as Valgrind [16] only reports on faults in code that is actually executed. Forcing the execution of all error-handling code would require developing an elaborate testing framework, potentially involving multiple kinds of hardware, depending on the application. Symbolic execution [4] coupled with fault injection [14], 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.

6. Conclusion

In this paper, we have shown that error-handling code is a significant source of faults in systems code, and that such code 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 EHCtor, and applied it to find 371 faults in Linux and five other systems software projects. A limitation of our approach is the need for at least one exemplar of a given resource-release operation in the given 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.

Bibliography

1. Ammons (G.), Bodík (R.) and Larus (J. R.). – Mining specifications. *In : POPL 2002*.
2. Banabic (R.) and Candea (G.). – Fast black-box testing of system recovery code. *In : EuroSys 2012*.
3. Bessey (A.), Block (K.), Chelf (B.), Chou (A.), Fulton (B.), Hallem (S.), Henri-Gros (C.), Kamsky (A.), McPeak (S.) and Engler (D.). – A few billion lines of code later: using static analysis to find bugs in the real world. *Commun. ACM*, vol. 53, février 2010.

4. Bucur (S.), Ureche (V.), Zamfir (C.) and Candea (G.). – Parallel symbolic execution for automated real-world software testing. *In : EuroSys 2011*.
5. Engler (D. R.), Chen (D. Y.), Chou (A.) and Chelf (B.). – Bugs as deviant behavior: A general approach to inferring errors in systems code. *In : SOSP 2001*.
6. Gabel (M.) and Su (Z.). – Javert: Fully automatic mining of general temporal properties from dynamic traces. *In : FSE 2008*.
7. Gunawi (H. S.), Rubio-González (C.), Arpaci-Dusseau (A. C.), Arpaci-Dusseau (R. H.) and Liblit (B.). – EIO: Error handling is occasionally correct. *In : FAST 2008*.
8. Kadav (A.), Renzelmann (M. J.) and Swift (M. M.). – Tolerating hardware device failures in software. *In : SOSP 2009*.
9. Kremenek (T.), Twohey (P.), Back (G.), Ng (A.) and Engler (D.). – From uncertainty to belief: Inferring the specification within. *In : OSDI 2006*.
10. Lawall (J. L.), Brunel (J.), Hansen (R. R.), Stuart (H.), Muller (G.) and Palix (N.). – WYSIWIB: A declarative approach to finding protocols and bugs in Linux code. *In : DSN 2009*.
11. Le Goues (C.) and Weimer (W.). – Specification mining with few false positives. *In : TACAS 2009*.
12. Li (Z.) and Zhou (Y.). – PR-Miner: automatically extracting implicit programming rules and detecting violations in large software code. *In : ESEC/FSE 2005*.
13. Lo (D.), Khoo (S.-C.) and Liu (C.). – Mining temporal rules for software maintenance. *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 20, 2008.
14. Marinescu (P. D.) and Candea (G.). – Lfi: A practical and general library-level fault injector. *In : DSN 2009*.
15. Melliar-Smith (P. M.) and Randell (B.). – Software reliability: The role of programmed exception handling. *In : ACM Conference on Language Design for Reliable Software*.
16. Nethercote (N.) and Seward (J.). – Valgrind: a framework for heavyweight dynamic binary instrumentation. *In : PLDI 2007*.
17. Nguyen (T. T.), Nguyen (H. A.), Pham (N. H.), Al-Kofahi (J. M.) and Nguyen (T. N.). – Graph-based mining of multiple object usage patterns. *In : ESEC-FSE 2009*.
18. Ramanathan (M. K.), Grama (A.) and Jagannathan (S.). – Path-sensitive inference of function precedence protocols. *In : ICSE 2007*.
19. Rubio-González (C.), Gunawi (H. S.), Liblit (B.), Arpaci-Dusseau (R. H.) and Arpaci-Dusseau (A. C.). – Error propagation analysis for file systems. *In : PLDI 2009*.
20. Saha (S.), Lawall (J. L.) and Muller (G.). – Finding resource-release omission faults in Linux. *In : 6th PLOS Workshop*.
21. Sundararaman (S.), Zhang (Y.), Subramanian (S.), Arpaci-Dusseau (A. C.) and Arpaci-Dusseau (R. H.). – Making the common case the only case with anticipatory memory allocation. *In : FAST 2011*.
22. Wasylkowski (A.), Zeller (A.) and Lindig (C.). – Detecting object usage anomalies. *In : ESEC-FSE 2007*.
23. Weimer (W.) and Necula (G. C.). – Finding and preventing run-time error handling mistakes. *In : OOPSLA 2004*.
24. Weimer (W.) and Necula (G. C.). – Mining temporal specifications for error detection. *In : TACAS 2005*.
25. Yang (J.), Evans (D.), Bhardwaj (D.), Bhat (T.) and Das (M.). – Perracotta: Mining temporal API rules from imperfect traces. *In : ICSE 2006*.