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

Suman Saha	Jean-Pierre Lozi	Gaël Thomas	Julia L. Lawall	Gilles Muller
LIP6-Regal	LIP6-Regal	LIP6-Regal	Inria/LIP6-Regal	Inria/LIP6-Regal
Suman.Saha@lip6.fr	Jean-Pierre.Lozi@lip6.fr	Gael.Thomas@lip6.fr	Julia.Lawall@lip6.fr	Gilles.Muller@lip6.fr

***Abstract*—Omitting resource-release operations in systems error handling code can lead to memory leaks, crashes, and deadlocks. Finding omission faults is challenging due to the difficulty of reproducing system errors, the diversity of system resources, and the lack of appropriate abstractions in the C language. To address these issues, numerous approaches have been proposed that globally scan a code base for common resource-release operations. Such macroscopic approaches are notorious for their many false positives, while also leaving many faults undetected.**

We propose a novel microscopic approach to finding resource-release omission faults in systems software. Rather than generalizing from the entire source code, our approach focuses on the error-handling code of each function. Using our tool, Hector, we have found over 370 faults in six systems software projects, including Linux, with a 23% false positive rate. Some of these faults allow an unprivileged malicious user to crash the entire system.

I. 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, 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 [1], 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.

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 [2], [3], [4], [5]. 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. Finally, 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.

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 [3], [6], [7], [8], [9], [10], [11], [12], [13], [14]. 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 [15], 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.

In this paper, we propose an alternative approach that specifically targets 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,[1] that uses heuristics and mostly intraprocedural analysis, including a lightweight intraprocedural alias 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.

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

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 371 resource-release omission faults in the widely used systems software Linux, PHP, Python, Apache, Wine, and PostgreSQL, with a false positive rate of only 23%. 52% of the found 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.

The rest of this paper is organized as follows. Section II presents some examples that motivate our work. Section III presents our fault-finding algorithm, and Section IV describes the design choices taken in the implementation of Hector. Section V evaluates the results obtained by applying Hector to large systems software. Finally, Section VI presents related work and Section VII concludes.

II. Motivation and Background

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 memory region. We then give an overview of error-handling in systems software.

A. Linux resource-release omission faults

We motivate our work using three representative crashes and memory leaks derived from a variety of faults in Linux error-handling code. One of these faults was previously found by a Linux user; in this case, the bug report and Linux commit log contain no evidence that the fault was found using other tools. The other two faults were previously unreported; we have reported them to the appropriate maintainers and provided patches.[2] 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.[3] Fig. 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 unregister operations, according to the acquisitions that have been performed so far. The error-handling code provided with the ACPI resource conflict check on line 10, however, 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;
```

Fig. 1. w83627ehf driver containing an omission fault
(From drivers/hwmon/w83627ehf.c, `sensors_w83627ehf_init`)

Note that the error-handling code starting on line 3 correctly does not release any resources, because none have been successfully acquired at this point. Thus, flow and path sensitivity are necessary to determine what resource-release operations are needed at each point in a function.

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 Fig. 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 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

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

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

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

users on the NFS server. Then, the attacker, connected as an 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 fault has been present since the code was introduced into the Linux kernel in version 2.6.28 (2008), and is still present in Linux 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;
```

Fig. 2. Autofs4 code containing an omission fault
(From fs/autofs4/dev-ioctl.c, `_autofs_dev_ioctl`)

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 Fig. 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. An unprivileged user who mounts a file system from an external disk drive that has been previously formatted with invalid parameters can trigger the fault. 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 allow a 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 Linux 3.6.6.

B. Systems error-handling code

To assess the importance of error-handling code 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. We also study the usage frequency of various resource acquisition and release functions, to estimate the applicability of specification-mining based methods to finding omitted resource releases. Our study primarily focuses on the `drivers`, `sound` (sound drivers), `net` (network protocols), and `fs` (file systems) directories of Linux 2.6.34,[5] but we also consider a selection of other widely used systems software, summarized in Table I.

```
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: ...
```

Fig. 3. ReiserFS code containing an omission fault
(From fs/reiserfs/journal.c, `journal_init`)

TABLE I. CONSIDERED SOFTWARE

Project	(Lines of code)	Version	Description
Linux `drivers`	(4.6MLoC)	2.6.34	Linux device drivers
Linux snd/net/fs	(1.5MLoC)	2.6.34	sound,network and file system
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.

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. For example, in Figure 2, a block of error-handling code starts on line 6 and includes the code on lines 18-20 at the end of the function. Fig. 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 whether they are consistent. Our examples in Section II-A come from functions containing 7-14 blocks of error-handling code. The fault in the third example was introduced when a function was reorganized, and new error-handling code was introduced, showing the difficulty of maintaining such complex code.

Kinds of errors encountered. The impact of faults in error handling code is determined in part by how often the handled

[5]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.

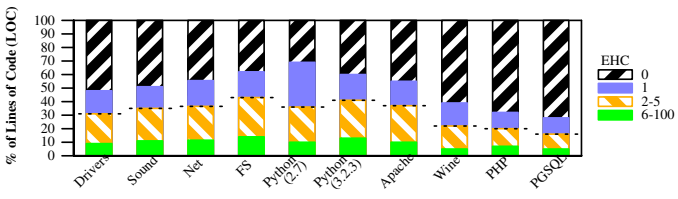

Fig. 4. Percentage of code found within functions that have 0 or more blocks of error-handling code

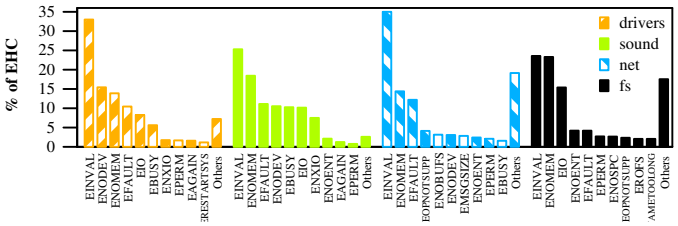

Fig. 5. Distribution of integer error-code return values

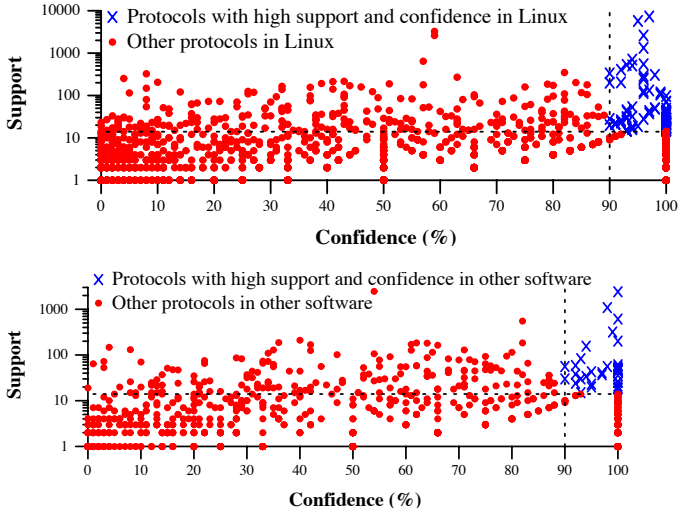

Fig. 6. Support and confidence of the identified protocols

errors occur. It is difficult to automatically determine the source of all the possible errors that may be encountered. Nevertheless, 48% of the blocks of error-handling code in Linux `drivers`, `sound`, `net`, and `fs` return integer error codes, understood by *e.g.* 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. 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 in the case of a system that 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, at an address coming from user level. A malicious application can easily construct an invalid address, making the correctness of the associated error-handling code critical.

Applicability of specification mining. 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 [15], 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 [10], [11], [12], [13], [14], such as the z-ranking used by Engler *et al.* [7]. 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 specification-mining tool PR-Miner [9], for example, which has been applied to Linux code, 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 Section IV for identifying related resource acquisition and release functions, we identify 2747 potential protocols in Linux, and 1051 in the other considered software. Fig. 6 shows the support and confidence of each, as determined by an intraprocedural analysis. Each $\times$ or circle 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 that have lower support and confidence.

III. Our Algorithm

The goal of our algorithm is to identify inconsistencies in the releasing of resources in a function's error-handling code. Inconsistencies may be intended, *e.g.,* if the resource has not yet been acquired or has been released in another way, or may represent a fault. The main challenge in designing the algorithm is to distinguish between these cases. Inconsistencies identified as unintended 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 operations to information about 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. A possible implementation of this preprocessing is presented in Section IV-A, but it can be done in any manner.

Our algorithm then works on the (intraprocedural) control-flow graph (CFG) of the provided function definition, annotated with the results of the preprocessing phase. As a running example, we use the code previously shown in Fig. 1, focusing on the resource *pdev*. Fig. 7(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 in Fig. 1. 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 our example (Fig. 7(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, starting from node 4, there are four such paths, shown in Fig. 7(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 Fig. 7(c and e) represent exemplars, because they contain the release operation, while the paths in Fig. 7(b and d) represent candidate faults. In Fig. 7, the exemplar and candidate fault in Fig. 7(c and d), respectively, are marked explicitly. We refer to the resource acquired at the beginning of any such exemplar or candidate fault as the *associated resource*.

The algorithm then compares each candidate fault to each exemplar, starting with the exemplar closest to it in the code, as indicated by the line number, 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 Fig. 7(c) and the

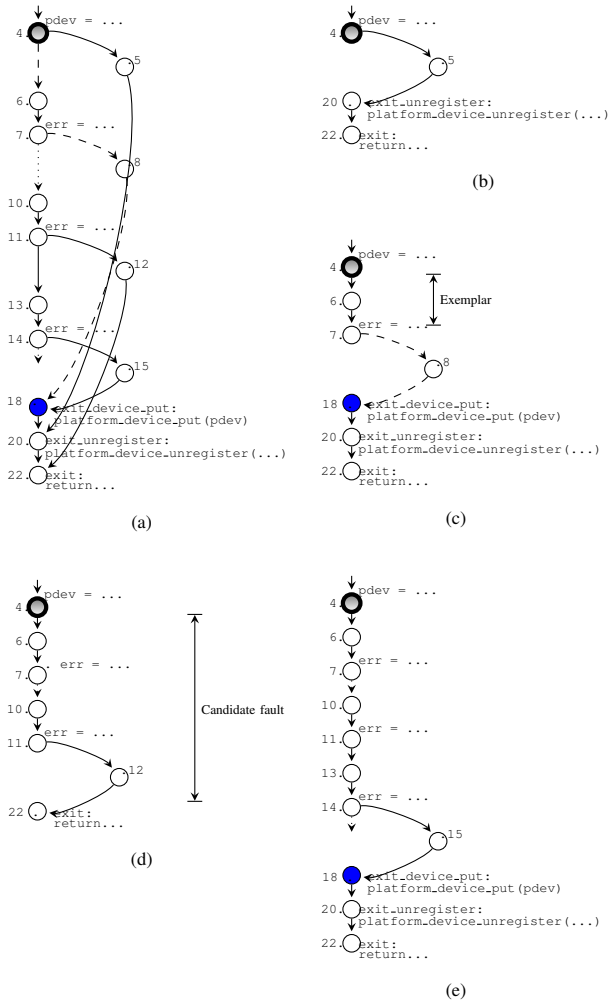

Fig. 7. CFG and paths for Fig. 1

candidate fault in Fig. 7(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 complete set of resource acquisitions reaching the exemplar and the candidate fault both acquire the associated resource in the same way. These acquisitions 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.

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 acquired in the same way. Finally, if a supposed release operation found in the candidate fault also appears in the exemplar, where it is followed 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.

The algorithm applies to our example as follows. The candidate fault shown in Fig. 7(d) satisfies all of the conditions for being reported as a fault. It does not return *pdev* (condition 1), it acquires its associated resource using the same function as the exemplar (Fig. 7(c)) (condition 2), and it does not contain any release of *pdev* (condition 3). 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 Fig. 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.

IV. 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.[6] Creating this implementation requires implementing a preprocessing phase and instantiating the algorithm with various analysis strategies.

A. Preprocessing phase

Preprocessing requires identifying and annotating resource acquisitions, resource releases, and error-handling code. 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.

A *resource* is typically represented by a collection of information, and is thus implemented by a pointer to a structure or buffer.[7] 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, we ignore operations that have constant string arguments, as such operations are typically 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 of that resource, according to the above criteria, 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. To account for these cases, we also consider a function call having at most one argument as being a resource acquisition, when the argument, if any, has pointer type and is not involved in an earlier 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 Fig. 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 it 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.

```
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;
```

Fig. 8. Extract of parse_audio_selector_unit (From sound/usb/usbmixer.c)

Hector identifies a *block of error-handling code* as a conditional branch that ends by returning an error value. Information about the return value is obtained using intraprocedural flow- and path-sensitive constant propagation. Error values are specific to each software project, but typically include `NULL` and various constants. In Linux, common error values include negative constants, as illustrated in line 12 of Fig. 2, and calls to `ERR_PTR` and `PTR_ERR`, as illustrated in line 3 of Fig. 2. Currently, the user must list these error values in a configuration file (the only configuration information that the user must provide), but 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. A block of error-handling code might also return no value, or return a variable whose value cannot be determined by the analysis, as illustrated in line 22 of Fig. 1. 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.

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

[7] 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, and `fopen` returns a pointer. The Linux kernel also uses pointers to represent its more primitive file objects.

B. 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, 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. Other possible alias-creating patterns could be added if found to be needed in practice. 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 Fig. 1. We have noted in Section III 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* and that this omission represents a fault. 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. Fig. 7(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 obtained from the successful execution of the resource acquisition on 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 e.g., $y{-}{>}fld = x$. 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.

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.

V. Experimenting with Hector

The goals of our 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, and 5) to understand the scalability of the approach. We evaluate Hector on the large, widely used open-source infrastructure software projects previously described in Table I, amounting to almost 10.5 million lines of C code.

A. Found faults

As shown in Table II, Hector generates a total of 484 reports for all of the projects. We manually investigated all of them and found that 371, 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 study them further in Section V-C.

We first investigate the complementarity of our approach with other approaches. Because we do not have access to implementations of other C code specification mining tools, 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.

TABLE II. 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 snd/net/fs	**92** (66)	**48** (37)	0.0011	40	2	6
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**

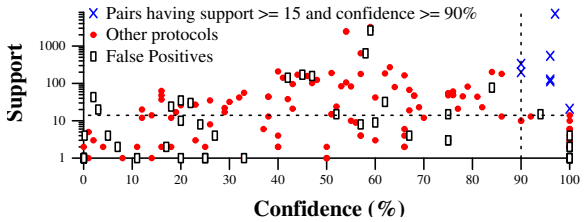

Fig. 9. Support and confidence associated with the protocols in the faults reported by Hector. The dotted lines mark support 15 and confidence 90%.

Comparison to specification mining. In Section II-B, we noted 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. In Fig. 6, we showed that most of the pairs of resource acquisition and release functions identified by the heuristics presented in Section IV-A do not meet the support and confidence thresholds proposed by the specification-mining tool PR-Miner [9]. Here, we focus on the subset of these pairs of resource acquisition and release functions that are associated with the reports generated by Hector.

Fig. 9 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 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 (6%) 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 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 specification-mining-based approaches could drastically increase their number of false positives. Hector finds faults independent of the support and confidence of the protocol.

Fig. 9 also shows as open rectangles the support and confidence for the 55 protocols involved in our 113 false

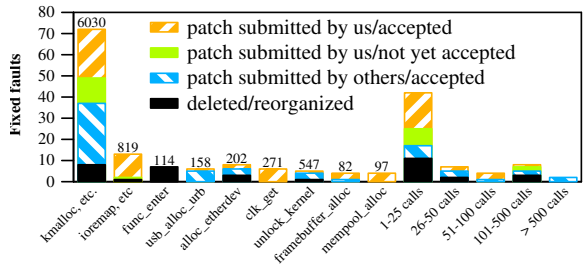

Fig. 10. Fixed or eliminated Linux driver faults. Bars on the left refer to functions associated with 4 or more fixes. These bars are annotated with the support for the corresponding acquisition and release functions. Bars on the right refer to functions with fewer than 4 fixes and varying levels of support.

positives. None of these protocols exceed 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.

Comparison to faults fixed in Linux. Linux 2.6.34 was released in May 2010, and thus some of the faults we have identified 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. Fig. 10 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,[8] 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 II-A.

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

B. Impact of the detected faults

As illustrated in Section II-A, the kinds of faults we detect 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 III 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).

TABLE III. 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**

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 [16]. 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 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 an 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 II, 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 directly exposed 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.

C. False positives

Table IV 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 [17]. 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, 26%), or for recognizing existing releases, whether via an alias (Via alias, 29%) or via a non-local call (Non-local call frees, 12%), or unawareness of releases performed in the caller of the considered function rather than in the function itself (Caller frees, 13%).

TABLE IV. FALSE POSITIVES

	Reports	FP (Rate, Fns)	Reasons					
			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 snd/net/fs	**92**	**44** (47%,29)	0	7	19	0	7	10
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
Total	**484**	**113 (23%,74)**	**4**	**29**	**33**	**14**	**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.

D. 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,

TABLE V. 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%)

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. Rather than trying to identify all of the faults in our considered software, we compare the results of Hector with an alternate fault-finding approach that does not rely on exemplars. To reduce the amount of code to study, we focus on resource-release omission faults involving resources acquired using the basic Linux kernel memory allocation functions, *kmalloc*, *kzalloc*, and *kcalloc*, for which Fig. 10 showed that faults are common. We furthermore 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, and thus reduce the rate of false positives. We have implemented this strategy using the open-source tool Coccinelle [18]. Coccinelle does not implement a specific fault-finding policy, but instead makes it possible to specify patterns that are used to search for code fragments that exhibit certain properties within the paths of a function's CFG.

Table V shows the rate of detected resource-release omission 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.

E. 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 3 hours. Over all the considered projects, the processing time, excluding the parsing time, ranges from 0.0002 s/LOC (seconds per line of code) to 0.0068 s/LOC. Apache, which is the smallest project (0.1 MLOC), and Linux `drivers`, which is the largest, have essentially the same processing time per line, at 0.0019 s/LOC, showing the scalability of the approach.

VI. RELATED WORK

Our most closely related work is that of Weimer and Necula on specification mining for fault finding [13], which also focuses on error-handling code. They target user-level programs written in Java, which provides specific abstractions for exceptions, while we target systems code written in C, where error-handling code is ad hoc. They search for pairs of functions a and b, where the a functions may, but need not, correspond to our acquisition operations, and the b functions correspond to our release operations. For a given pair of functions a and b, they require the existence of what amounts to an exemplar and what amounts to a candidate fault, but do not require the exemplar and candidate fault to come from the same function. Thus, their mining process can be thrown off by local variations in API usage protocols. In practice, on almost 1 million lines of Java code, from 9 different projects, almost all of their mined specifications are false positives, reaching a false positive rate of 90%. To reduce the rate of false positives, Le Goues and Weimer integrate extra information such as author expertise [15], but doing so also reduces the number of found faults. Furthermore, results are ranked according to statistics, so rarely used release functions may be overlooked.

Sundararaman *et al.* also focus on faults in error-handling code, by simply trying to avoid the need to execute error-handling code, through the definition of an alternate memory allocator [19]. We have seen in Section II-B that systems code can encounter other kinds of errors, such as defective devices and bad user-level values, which the approach of Sundararaman *et al.* cannot address. Resource Acquisition Is Initialization (RAII) is a resource management technique originating in C++ that exploits the ability to associate a variable with cleanup code, which is executed when the variable goes out of scope [20]. RAII eliminates the need for resource releases in exception handlers, but has the side effect that resources are also released on a normal function exit. The latter is too constrained for systems code, where allocated resources must persist over multiple requests by applications or hardware.

Engler *et al.* use static analysis to automatically extract programming rules from source code, based on user-defined templates [7]. Ranking calculated in terms of support and confidence is used to highlight the most probable rules. The approach can also use "must beliefs" derived from the user's knowledge of the semantics of the code, rather than statistics. Such must beliefs are not available in our setting, where there is a very wide range of resource acquisition and release operations. PR-Miner uses frequent itemset mining to extract programming rules, without using templates [9]. Results are pruned and ranked according to support and confidence. MUVI applies a similar strategy to find missing locking operations [21]. Kremenek *et al.* use factor graphs in inferring specifications directly from programs [22]. Ramanathan *et al.* integrate mining within a path-sensitive dataflow framework to identify potential preconditions for invocation of a function [23]. In each of these cases, the identified specifications can be used to find faults in code. Hector 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.

The tool Coverity,[9] based on the research of Engler *et al.* [7], [**?**], includes rules for identifying memory leaks as well as other rules that are able to identify errors within error-handling code. We have collected and categorized the entire set of patches accepted into the Linux kernel between April 2005 and April 2013 that mention Coverity.[10] Out of 523 such patches, only 109 (21%) relate to error-handling code. Of these, 64 involve one or more missing occurrences of `kfree` and 16 more involve missing or duplicate occurrences of some other function containing "free" in its name. 3 patches involve functions whose name contains the substring "lock" and 3 involve functions whose name contains the substring "put". 14 involve unnecessary error-handling operations rather than omitted operations, and are detected as null pointer dereferences. The remaining 6 patches involve a variety of other functions and conditions. Hector has made it possible to find more than twice as many faults, involving a more diverse set of functions, within just one Linux version. While we do not know the version of Coverity used by the Linux developers, nor the strategies used by the Linux developers to decide which reported faults to fix, these results suggest that our work is complementary to the strategies used by the Coverity tool.

Wu *et al.* identify resource acquisition and release operations in Java code by interprocedural analysis of method definitions [24], ultimately relying on a list of known release operations. Ravitch et al. take a similar strategy for C code [25]. These approaches could be used in an alternative implementation of the preprocessing phase of our algorithm. Our proposed implementation is mostly intraprocedural and does not require advance knowledge of any resource-release functions; the latter is an advantage for Linux, which manages a wide range of types of resources and does not rely on standard libraries. The analyses required are furthermore less costly, as interprocedural analysis is limited to a single file.

Gunawi *et al.* [26] and Rubio-González et al. [27] 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. Banabic and Candea propose a strategy for fault-injection prioritisation to perform run-time checking of error-handling code [29]. The reported faults involve omitted tests and duplicated releases, while Hector focuses on release omissions.

Another approach to detect faults is to monitor program execution. A dynamic analysis tool such as Valgrind [30] only reports on real faults that can occur in real executions, and is insensitive to procedure-call boundaries. Thus, it may find some faults that involve interprocedural dependencies and cannot be found by Hector. On the other hand, such a tool can only find faults in the code that is actually executed, given the available test cases. 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 [31] coupled with fault injection [32], 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. Thus, the developer still needs precise prior knowledge of the various pairs of resource acquisition and release operations.

Some other works use static analysis to find faults in Linux code. Chou et al. [2] and Palix *et al.* [4] use patterns to automatically find simple faults such as null pointer dereferences. Their techniques are not sufficient to find arbitrary resource-release omissions in error-handling code because they do not infer protocols. The rule INull, originally developed by Chou *et al.* and which is also part of the static analysis tool Coverity, checks for the dereference of a value that is subsequently tested for being NULL. Like our work, INull also relies on function-local consistency information, comprising the dereference and the NULL tests. Nevertheless, the case addressed by INull is simpler than that of resource-release omissions, because the identification an operation as a NULL test or as a dereference is unambiguous, drastically reducing the possibility of false positives. In another form of consistency analysis, Tan et al. [33] find faults by comparing code with its expected behavior, described in comments. Comments have been useful in assessing the faults reported by Hector, and it could be interesting to combine the two approaches.

VII. Conclusion

In this paper, 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 371 faults in Linux and 5 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. 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. Finally, we will consider how the use of local information can be applied to other program analysis problems, such as identifying shared variables.

[9] http://scan.coverity.com/

[10] https://git.kernel.org/cgit/linux/kernel/git/next/linux-next.git/log/?id=refs/tags/next-20130412

References

[1] P. M. Melliar-Smith and B. Randell, "Software reliability: The role of programmed exception handling," in *ACM Conference on Language Design for Reliable Software*, 77.

[2] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler, "An empirical study of operating systems errors," in *SOSP'01*.

[3] 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 *DSN'09*.

[4] N. Palix, G. Thomas, S. Saha, C. Calvès, J. Lawall, and G. Muller, "Faults in Linux: ten years later," in *ASPLOS'11*.

[5] W. Weimer and G. C. Necula, "Finding and preventing run-time error handling mistakes," in *OOPSLA'04*.

[6] G. Ammons, R. Bodík, and J. R. Larus, "Mining specifications," in *POPL'02*.

[7] 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 *SOSP'01*.

[8] M. Gabel and Z. Su, "Javert: Fully automatic mining of general temporal properties from dynamic traces," in *FSE'08*.

[9] Z. Li and Y. Zhou, "PR-Miner: automatically extracting implicit programming rules and detecting violations in large software code," in *ESEC/FSE'05*.

[10] D. Lo, S.-C. Khoo, and C. Liu, "Mining temporal rules for software maintenance," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 20, 2008.

[11] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen, "Graph-based mining of multiple object usage patterns," in *ESEC-FSE'09*.

[12] A. Wasylkowski, A. Zeller, and C. Lindig, "Detecting object usage anomalies," in *ESEC-FSE'07*.

[13] W. Weimer and G. C. Necula, "Mining temporal specifications for error detection," in *TACAS'05*.

[14] J. Yang, D. Evans, D. Bhardwaj, T. Bhat, and M. Das, "Perracotta: Mining temporal API rules from imperfect traces," in *ICSE'06*.

[15] C. Le Goues and W. Weimer, "Specification mining with few false positives," in *TACAS'09*.

[16] A. Kadav and M. M. Swift, "Understanding modern device drivers," in *ASPLOS'12*.

[17] 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*, vol. 53, Feb. 2010.

[18] Y. Padioleau, J. Lawall, R. R. Hansen, and G. Muller, "Documenting and automating collateral evolutions in Linux device drivers," in *EuroSys'08*.

[19] S. Sundararaman, Y. Zhang, S. Subramanian, A. Arpaci-Dusseau, and R. Arpaci-Dusseau, "Making the common case the only case with anticipatory memory allocation," in *FAST'11*.

[20] B. Stroustrup, *Exception Safety: Concepts and Techniques*. LNCS, 2001, vol. 2022.

[21] S. Lu, S. Park, C. Hu, X. Ma, W. Jiang, Z. Li, R. A. Popa, and Y. Zhou, "MUVI: automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs," in *SOSP'07*.

[22] T. Kremenek, P. Twohey, G. Back, A. Ng, and D. Engler, "From uncertainty to belief: Inferring the specification within," in *OSDI'06*.

[23] M. Ramanathan, A. Grama, and S. Jagannathan, "Path-sensitive inference of function precedence protocols," in *ICSE'07*.

[24] D. R. Engler, B. Chelf, A. Chou, and S. Hallem, "Checking system rules using system-specific, programmer-written compiler extensions," in *OSDI'00*.

[25] Q. Wu, G. Liang, Q. Wang, T. Xie, and H. Mei, "Iterative mining of resource-releasing specifications," in *ASE'11*.

[26] T. Ravitch, S. Jackson, E. Aderhold, and B. Liblit, "Automatic generation of library bindings using static analysis," in *PLDI'09*.

[27] H. S. Gunawi, C. Rubio-González, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and B. Liblit, "EIO: Error handling is occasionally correct," in *FAST'08*.

[28] C. Rubio-González, H. S. Gunawi, B. Liblit, R. H. Arpaci-Dusseau, and A. C. Arpaci-Dusseau, "Error propagation analysis for file systems," in *PLDI'09*.

[29] R. Banabic and G. Candea, "Fast black-box testing of system recovery code," in *EuroSys'12*.

[30] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," in *PLDI'07*.

[31] S. Bucur, V. Ureche, C. Zamfir, and G. Candea, "Parallel symbolic execution for automated real-world software testing," in *EuroSys'11*.

[32] P. D. Marinescu and G. Candea, "LFI: A practical and general library-level fault injector," in *DSN'09*.

[33] L. Tan, D. Yuan, G. Krishna, and Y. Zhou, "/*icomment: bugs or bad comments?*/," in *SOSP'07*.