WYSIWIB: A Declarative Approach to Finding API Protocols and Bugs in Linux Code

Julia L. Lawall,[1] Julien Brunel,[1*] Nicolas Palix,[1] René Rydhof Hansen,[2] Henrik Stuart,[1] Gilles Muller[3]
[1]DIKU, University of Copenhagen, Copenhagen, Denmark
[2]Aalborg University, Aalborg, Denmark
[3]EMN/INRIA-Regal, Paris, France
{julia,brunel,npalix,hstuart}@diku.dk, rrh@cs.aau.dk, Gilles.Muller@emn.fr

Abstract

Eliminating OS bugs is essential to ensuring the reliability of infrastructures ranging from embedded systems to servers. Several tools based on static analysis have been proposed for finding bugs in OS code. They have, however, emphasized scalability over usability, making it difficult to focus the tools on specific kinds of bugs and to relate the results to patterns in the source code.

We propose a declarative approach to bug finding in Linux OS code using a control-flow based program search engine. Our approach is WYSIWIB (What You See Is Where It Bugs), since the programmer expresses specifications for bug finding using a syntax close to that of ordinary C code. The key advantage of our approach is that search specifications can be easily tailored, to eliminate false positives or catch more bugs. We present three case studies that have allowed us to find hundreds of potential bugs.

Keywords Protocol finding, bug finding, Linux.

1. Introduction

The Linux operating system (OS) is increasingly used in environments ranging from embedded systems to large servers. The OS used in such environments must be dependable, even for rarely used hardware types and system configurations. An important part of such dependability is the absence of software bugs. Bugs such as invalid pointer dereferences and memory leaks can lead to system crashes, possibly long after the buggy code has been executed.

While Linux is considered to be a reliable OS for common use, there are still situations where a software bug can crash or hang the complete system. One common kind of bug is the invalid use of API functions. In the case of Linux, API functions defined for use within the Linux kernel are associated with a variety of usage protocols, describing for example how error handling should be performed and how resources should be allocated and freed. Finding bugs in the use of these API functions requires first identifying the associated usage protocol and then finding code that violates it. The Linux OS consists of over 19000 files, and provides APIs at many levels, ranging from general purpose APIs for managing memory, locks, files etc., to very specific APIs for a single device family. Identifying the API functions, the protocol for their usage, and code that violates these protocols is thus a daunting task.

In recent years, a number of approaches have been proposed that scan systems code for API protocols and bugs in their usage [3, 4, 6, 11]. A goal of these approaches has been to be highly scalable, and by using techniques such as model checking, statistics, and data mining, it has been possible to apply these approaches to software of millions of lines of code such as the Linux kernel. A weakness of these efforts, however, is that the user can do little to influence the API protocol-finding and bug-finding strategies. API usage protocols are detected using complex heuristics that the user has little control over [3, 6]. Automata-based languages have been proposed for describing code patterns that constitute bugs [3], but these specifications do not follow the code structure. Both of these features make it difficult for the user to understand why a given code fragment is considered to be part of a API usage protocol or bug, or is overlooked. Substantial effort is thus required when using such tools, in order to identify the inevitable false positives and false negatives.

In previous work, we have carried out an extensive study of Linux code, focusing on the impact of API evolutions [10]. In this study, we have observed that many of the Linux API usage protocols follow a common pattern, related to the purpose of the API functions, such as error handling or managing memory allocation. We thus believe that taking such patterns into account in the API protocol finding and

[*]Author's current address: ONERA, Toulouse, France

bug finding processes can ease the checking of the results and make it possible to identify protocols for API functions that are used very rarely, and thus are often overlooked by statistics-based approaches.

In this paper, we propose a "WYSIWIB" (What You See Is Where It Bugs) approach to API protocol and bug finding in Linux code, based on the following steps: 1) describe a class of API protocols generically, and apply this description to the Linux source code to collect a set of possible protocol instances, 2) describe various classes of bugs that can be found in uses of a given kind of API protocol generically, and instantiate these descriptions with respect to the API protocol instances found in step 1, and 3) apply these descriptions to the Linux source code to collect possible API protocol violations. This approach directly exploits the user's knowledge of the source code and guides the validation of the reported bugs based on information explicit in the specification. Our approach is furthermore complementary to statistics-based approaches, in that it describes what to search for, while statistics-based approaches consider how to select from the things that are found.

We have implemented our approach as a collection of tools based on the Coccinelle transformation engine, that we have developed in previous work [9]. A key feature of Coccinelle is that specifications are written using a language that is based on C code and the patch notation [8], which are well known to Linux developers.

The contributions of this work are:

- We propose a new approach to finding API protocols in Linux code, iteratively refining them, and using them to find bugs. This approach chiefly builds on Coccinelle, but provides some new tools.
- We present in detail a case study illustrating the use of our approach for finding API protocols and bugs. This case study involves general-purpose, widely used classes of API protocols, relating to error handling.
- In our experiments, we find over 3000 potential API protocols, with estimated false positive rates of under 15%.
- Based on these API protocols, we have used our approach to find 360 bugs that we have validated. Over 30 of these bugs have subsequently been fixed in Linux, either by ourselves or others.

All of the experiments in this paper are based on a snapshot of Linux dated March 11, 2008,[1] and were run on an HP ProLiant server with two 3GHz quad-core Xeon processors and 16GB memory. Verifying the sites reported in our experiments has involved studying tens of thousands of lines of Linux code. Details about these experiments that could not fit into the paper are available at our website: `http://www.emn.fr/x-info/coccinelle/bugs`.

The rest of this paper is organized as follows. Section 2 reviews the aspects of Coccinelle that are necessary to understand this work. Section 3 describes our API protocol-finding and bug-finding methodology. Section 4 presents a case study, illustrating the processing of a common kind of API protocol. Section 5 presents the results of two other case studies. Section 6 surveys some current limitations of our approach. Finally, Section 7 describes related work and Section 8 concludes.

2. Coccinelle

Coccinelle is a tool for performing control-flow based program searches and transformations in C code [2, 9]. It provides a language, SmPL, for specifying searches and transformations and an engine for performing them. In this paper, we write SmPL code for defining *semantic matches*, which are used for code searching. We present SmPL in terms of a simple semantic match inspired by a case study reported in Section 5.

The semantic match shown in Fig. 1 detects cases where a value allocated using the Linux netlink memory allocation function `nlmsg_new` is deallocated using the generic deallocation function `kfree_skb`. Such a deallocation is undesirable, because the netlink library defines its own deallocation function `nlmsg_free`. The semantic match consists of two rules: the first (lines 1-5) is named bad_kfree and is written in SmPL, and the second (lines 7-8) is written using the SmPL interface to Python. Each rule begins with the declaration of a collection of metavariables, and then follows with either a C-code like pattern specifying a match in the case of a SmPL rule, or ordinary Python code. In the rest of this section, we describe each of these rules in more detail, and present some other features of SmPL.

```
1  @bad_kfree exists@ expression x,E; position p; @@
2
3    x = nlmsg_new(...)
4    ... when != x = E
5    kfree_skb@p(x);
6
7  @ script:python @ x << bad_kfree.x; p << bad_kfree.p; @@
8  print "line: %s x: %s" % (p[0].line,x)
```

Figure 1. A semantic match searching for certain uses of `kfree_skb`

The rule bad_kfree defines three *metavariables* (line 1): x and E, which represent arbitrary expressions, and p, which represents an arbitrary position in the source program. Metavariables are bound by matching the code pattern

[1] baadac8b10c5ac15ce3d26b68fa266c8889b163f in the git repository `http://git.kernel.org/git/?p=linux/kernel/git/torvalds/linux-2.6.git;a=summary`. All subsequent git codes refer to this repository.

against the C source code. The notation `@`*p* binds the position variable *p* to information about the position of the preceding token. Once bound, a metavariable must have the same value within the current control-flow path, and thus, for example, the occurrences of *x* on lines 3, 4, and 5 must all match the same expression. The code pattern (lines 3-5) consists of essentially C code, mixed with some operators to raise the level of abstraction, so that a single semantic match can apply to many code sites. The main abstraction operator is "`...`," representing a sequence of terms. In line 3, "`...`" represents the arguments of `nlmsg_new` and in line 4 "`...`" represents the sequence of statements reachable from a call to `nlmsg_new` along any control-flow path. By default such a sequence of statements is quantified over all paths (*e.g.*, over all branches of a conditional), but the annotation `exists` next to the rule name indicates that for this rule there need be only one. It is also possible to restrict the kinds of sequences that can match "`...`" using the keyword `when`. Line 4 uses `when` to indicate that there should be no reassignment of *x* before reaching the call to `kfree_skb`.

A SmPL rule only applies when it matches the code completely. Fig. 2 shows an extract of C code that uses the function `nlmsg_new`. The rule bad_kfree matches the call to `nlmsg_new` on line 1 and the call to `kfree_skb` on line 9. The metavariable x is bound to the expression `skb`, which occurs in both calls. The metavariable p is bound to various information about the position of `kfree_skb`, such as the file name, line number, etc.

```
1  skb = nlmsg_new(neigh_nlmsg_size(), GFP_ATOMIC);
2  if (skb == NULL)
3    goto errout;
4
5  err = neigh_fill_info(skb, n, 0, 0, type, flags);
6  if (err < 0) {
7    /* -EMSGSIZE implies BUG in neigh_nlmsg_size() */
8    WARN_ON(err == −EMSGSIZE);
9    kfree_skb(skb);
10   goto errout;
11 }
```

Figure 2. Extract of net/core/neighbour.c. Code matched by bad_kfree is in italics.

A Python rule does not match against the source program, but instead inherits metavariables from other rules and does some processing of their values [12]. In this work, we only use Python for printing out information about the found protocols and bugs. Due to space constraints, we do not go into more detail about these rules.

In the more general case, a semantic match can consist of any number of rules, each of which can inherit metavariables from any previous ones. A rule is applied once for each possible set of values of the inherited metavariables. Besides "`...`", SmPL provides *nests*, `<...`*pat* `...>`, and *disjunctions*, `(`pat_1 `| ... |`pat_n`)`. A nest matches a sequence, like "`...`", but additionally can match zero or more occurrences of *pat* within the matched sequence. A nest of the form `<+...`*pat* `...+>` matches one or more occurrences of *pat*. A disjunction matches any of the patterns pat_1 through pat_n. We will present other features of SmPL as needed.

Convention In the examples, we use "`...`" for the Coccinelle operator and `[...]` to represent omitted code. In particular, we frequently replace Python code by `[...]`; in such rules, the significant part is the list of inherited metavariables, which indicates what will be printed.

3. The Bug-Finding Process

There are many sources of information for finding bugs. One can study bug reports [7], notice a suspicious coding pattern while doing some other work on the code, or learn from sources such as code comments [13] or newsgroups about coding protocols that programmers may sometimes overlook. The problem then is to turn this awareness of the potential for bugs into actual bug detection. Typically, bug finding depends highly on chance, as the person who is aware of the protocol must be looking at the specific code containing the bug and have the protocol in mind at that time. The goal of this work is to enable a programmer who becomes aware of a potential pattern of bugs in protocol usage to easily and systematically search for instances of this pattern throughout the code base, and to explore variants of the pattern as they become apparent.

3.1. An experience in bug finding

To motivate our approach, we begin with a real story drawn from our experience in finding bugs in Linux code. In December 2007, a patch,[2] shown in Fig. 3, was submitted to Linux. This patch was based on the observation that the function `netif_rx` could free its argument, and that thus it was not safe to refer to its argument after calling the function. The developer had found and fixed the problem in one file. We saw this patch and wondered whether there could be other calls to the same function that have the same property. We thus created a semantic match for detecting such calls, shown in Fig. 4a. This semantic match matches a call to `netif_rx` followed by a dereference of the argument.

This semantic match found 2 bugs in Linux 2.6.24-rc5. In the process of validating them, however, we found that the function `netif_rx_ni` also had the same property. We thus augmented the semantic match as shown in Fig. 4b to allow it to match calls to either function (lines 6-10). The resulting semantic match found 5 occurrences of this pattern that have been acknowledged as bugs by the Linux developers. Our

[2]Git code d30f53aeb31d453a5230f526bea592af07944564.

```
--- a/drivers/net/smc911x.c
+++ b/drivers/net/smc911x.c
@@ -1299,9 +1299,9 @@
   PRINT_PKT(skb->data, skb->len);
   dev->last_rx = jiffies;
   skb->protocol = eth_type_trans(skb, dev);
-  netif_rx(skb);
   dev->stats.rx_packets++;
   dev->stats.rx_bytes += skb->len;
+  netif_rx(skb);

   spin_lock_irqsave(&lp->lock, flags);
   pkts = (SMC_GET_RX_FIFO_INF()&RX_FIFO_INF_RXSUSED_)>>16;
```

Figure 3. A standard patch fixing one instance of the `netif_rx` problem

```
@r exists@
expression skb,e,e1;
position p;
identifier fld;
@@

 netif_rx(skb)
 ... when != skb = e
 skb@p->fld

@ script:python @
skb << r.skb; p << r.p;
@@
[...]
```

(a: original semantic match)

```
@r exists@
expression skb,e,e1;
position p;
identifier fld;
@@

(
 netif_rx(skb)
|
 netif_rx_ni(skb)
)
 ... when != skb = e
 skb@p->fld

@ script:python @
skb << r.skb; p << r.p;
@@
[...]
```

(b: extended semantic match)

Figure 4. Semantic matches finding `netif_rx` problems

corrections for 4 of these bugs have been accepted into the Linux kernel.[3] The function containing the remaining bug site is no longer part of Linux.

This episode clearly highlights how a flexible searching tool such as Coccinelle can find bug patterns more efficiently and completely than a human programmer. But still, it does not go far enough. Rather than accidentally finding another function that follows the same protocol as one that caused a bug, one would like to be able to find all of the functions that follow that protocol, and then create a bug finding rule for each of them. Indeed, it can be useful to iterate this process, instantiating a collection of semantic match templates according to a set of protocols, where each template either expresses a bug finding rule or collects more information.

3.2. Tools

We have developed a collection of tools based on Coccinelle to support an iterated protocol finding and bug finding process. This process, illustrated in Fig. 5, uses four tools: `Search`, `Instantiate`, `MakeBugReport`, and `MultiSearch`. `Search`, `MakeBugReport`, and `MultiSearch` use Coccinelle in various ways, while `Instantiate` is separate. We describe these tools below.

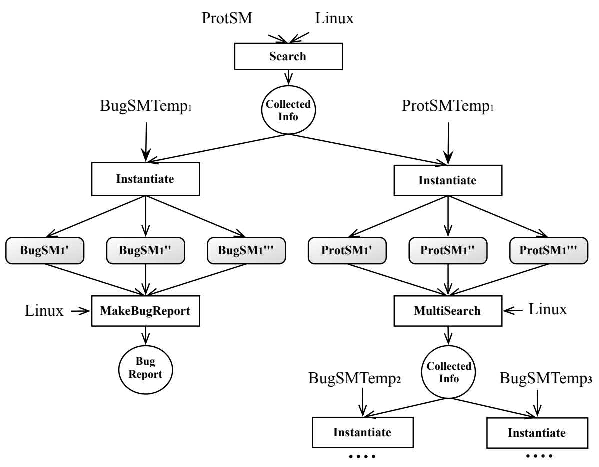


Figure 5. Protocol and bug-finding process

Search The protocol and bug finding process begins when the programmer has an idea of a certain kind of function or collection of functions, such as `netif_rx`, whose usage may be error prone. He expresses this idea as a protocol-finding semantic match ProtSM that expresses various properties of the kind of code he is interested in and uses Python to print relevant information about the matched terms. Fig. 6 illustrates such a semantic match, which detects any function that, like `netif_rx`, may use `kfree_skb` to free its argument without having reassigned it first. The Python code prints information about these functions in the format expected by `Instantiate`, which consumes the result of `Search`. The first field of this output is a *tag* that characterizes this match. Multiple tags may be used to separate various kinds of matched code into categories, as required in Section 4. The remainder of the output is a sequence of key-value pairs. By convention, we write the key in capital letters. The value is typically some part of the matched code, often the name of a matched function that was found to have some property with respect to the protocol. For the semantic match in Fig. 6, the output might contain, *e.g.*:

```
kfree_skb: FN:handle_ing TY:struct sk_buff *
kfree_skb: FN:netif_rx TY:struct sk_buff *
kfree_skb: FN:dev_queue_xmit TY:struct sk_buff *
```

After having developed the protocol-finding semantic match, the programmer gives it to `Search`, which uses Coccinelle to apply it to each file of the Linux kernel. The results are collected in a single output file. The programmer may inspect this result to assess its accuracy. If it contains

[3]Git codes 299f590f26da9764f20e905879f0090552ff2e86, 505a41d43c24345f3fa77ddab152d1f82dd8264d, and 9b3efc0133a807070dbd21254102995b65969965.

```
@ kfree exists @ identifier f,x; type T; expression E; @@

f(...,T x,...) { ... when != x = E
  kfree_skb(x); ... }

@ script:python @ f << kfree.f; t << kfree.T; @@
print "kfree_skb: FN:%s TY:%s" % (f,t)

```

Figure 6. A protocol-finding semantic match, to start the protocol and bug finding process

entries that do not correspond to valid protocols or it is missing some protocols that the programmer is otherwise aware of, then he can refine the semantic match to eliminate these false positives and false negatives.

Instantiate Having obtained information about a collection of protocols from the initial protocol-finding semantic match, the programmer then considers what kinds of bugs are relevant or what other information might be needed to find bugs. For each case, he writes a *semantic match template*, which is a semantic match that is parameterized by the various keys used in reporting the result of the previous step. He then applies `Instantiate` to the semantic match template, the result of the initial protocol-finding semantic match, and a tag. The result is a collection of semantic matches, one for each element of the result of the previous step that is associated with the tag.

Fig. 7 illustrates a semantic match template that is used to find bugs. It performs the same search as the previous one, but this time an occurrence of a function h is considered to be a bug. The information printed by such a semantic match should be in the form expected by an emacs mode that we have developed, based on the emacs `org` mode.[4] Our emacs mode allows one to jump directly from a bug report to the relevant fragment of code in the Linux source tree. Key words can be highlighted in color, according to their purpose within the semantic match. This emacs mode makes it significantly easier to validate the reported potential bugs.

```
@ r @ TY E; expression E1; position p,p1; @@

FN(...,E@p,...)
...
( E = E1 | E@p1 )

@ script:python @ p << r.p; p1 << r.p1; @@
cocci.print_main(p)
cocci.print_secs("ref",p1)
```

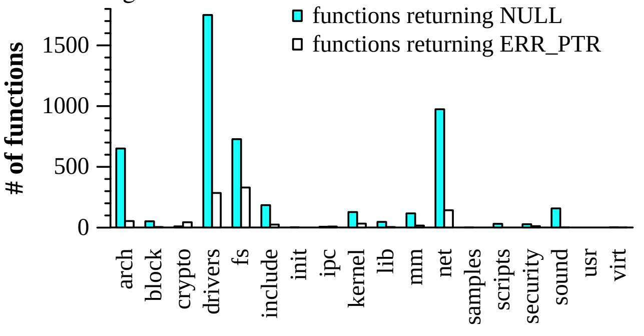
Figure 7. A semantic match template for bug finding

Semantic match templates can also be constructed to search for further information. In this case, the template prints its result using the key-value pair notation of Fig. 6.

MakeBugReport and MultiSearch `MakeBugReport` is used to search for bugs based on a collection of instantiated semantic match templates. It takes as input a collection of semantic matches produced by `Instantiate`, uses Coccinelle to apply each of them to the Linux kernel, and collects the result into a bug report for further processing with emacs. `MultiSearch` is similar, for semantic match templates that search for further information. In either case, examining the results may cause the programmer to refine the corresponding semantic match template or reveal new kinds of bugs that are relevant to the protocol, for which new semantic match templates can be developed.

4. Case Study: Inconsistent Error Checks

We now present a case study illustrating our methodology. This case study focuses on error handling. The C programming language does not provide any built-in error handling mechanism, and thus applications must define their own protocols for detecting and handling exceptional conditions. In Linux OS code, pointer-typed functions typically return either `NULL`, a value constructed with `ERR_PTR`,[5] or both to indicate failure. The frequency of use of each of these strategies across the various directories of the Linux kernel is shown in the graph below (in some directories, these strategies are too rarely used to be visible in the graph). Code calling a function that may fail must then perform appropriate tests before dereferencing the result. Performing insufficient or inappropriate tests can lead to invalid pointer dereferences that can crash the Linux kernel. We thus consider how to classify functions in terms of the error checking protocols that callers must follow and how to find various kinds of bugs in their use.



4.1. Protocol detection

For protocol detection, we use a semantic match that separates functions into the following categories: 1) Func-

[4]http://orgmode.org/

[5]`ERR_PTR` is a macro that coerces an error flag to a pointer. For conciseness, we refer a value constructed with `ERR_PTR` as just "`ERR_PTR`".

tions that indicate an error only by returning NULL; 2) Functions that indicate an error only by returning ERR_PTR; 3) Functions that may indicate an error using either NULL or ERR_PTR; 4) Functions that always return a valid pointer (*i.e.*, a pointer that can safely be dereferenced).

To motivate the strategies used by our protocol finding semantic match, we consider the Linux functions simple_alloc_urb and clk_get shown in Fig. 8. Relevant code is highlighted in italics. As illustrated by lines 8 and 25, a function may explicitly return NULL or ERR_PTR, or, as illustrated in lines 14 and 17, it may store such values in some variable and then return the value of that variable. Often, however, the return value is derived from a more complex expression, typically a function call, about which we have no direct information. Nevertheless, it may be possible to infer some information about such an expression from the uses of its value. For example, in line 10, the conditional test implies that the variable urb is NULL at the point of the return, and in line 15, the dereference of urb means that its value can subsequently be assumed to be a valid pointer. These observations allow us to conclude that simple_alloc_urb is in category 1. Similar observations allow us to identify functions in the other categories. For clk_get, however, we do not have enough information to classify the function, due to the function call in line 24. In this case, we classify the function as unknown.

```
 1  // from drivers/usb/misc/usbtest.c
 2  static struct urb *simple_alloc_urb (
 3    struct usb_device *udev,
 4    int pipe, unsigned long bytes)
 5  {
 6    struct urb *urb;
 7
 8    if (bytes < 0) return NULL; // explicit null
 9    urb = usb_alloc_urb (0, GFP_KERNEL);
10    if (!urb) return urb; // null inferred from test
11    ...
12    if (!urb->transfer_buffer) {
13      usb_free_urb (urb);
14      urb = NULL;
15    } else memset (urb->transfer_buffer, 0, bytes);
16    // null or valid pointer inferred from assignment or dereference
17    return urb;
18  }
19
20  // from arch/powerpc/kernel/clock.c
21  struct clk *clk_get(struct device *dev, const char *id)
22  {
23    if (clk_functions.clk_get)
24      return clk_functions.clk_get(dev, id);
25    return ERR_PTR(-ENOSYS);
26  }
```

Figure 8. Functions returning error codes.

We now consider how to write a semantic match that exploits these intuitions. It is used in the role of "ProtSM" at the root of Fig. 5. The semantic match first rewrites the program to make the information implied by NULL and IS_ERR tests explicit. For example, line 10 of Fig. 8 becomes if(!urb) {urb = NULL; return urb;}. The rest of the semantic match is in three phases, of which the main rules are shown in Figs. 9 to 11.

```
 1  @rn exists@
 2  identifier relevant.f,fld; position prelevant.pf,ret_null,e_null;
 3  expression E,E1;
 4  @@
 5
 6  f@pf(...) { ... when any
 7  (
 8    return@ret_null NULL;
 9  |
10    E@e_null = NULL
11    ... when != ( E=E1 | E->fld )
12    return@ret_null E;
13  )
14  }
```

Figure 9. Finding NULL returns

Phase 1: finding known return values The first phase finds functions that somewhere return NULL, ERR_PTR, or a valid pointer. The rule for the NULL case, shown in Fig. 9, checks that a function contains either an explicit return of NULL (line 8) or an assignment of some expression to NULL followed by a return of that expression (lines 10-12). The position of such a return is saved in the variable ret_null for reference in subsequent rules. The annotation **when any** in line 6 allows any code to appear between the beginning of the function and the matched return or assignment; without this annotation, the "..." would match the shortest path from *e.g.* the beginning of the function body to the first assignment matching $E = \mathrm{NULL}$, whereas we want the assignment that is closest to the return.

Phase 2: finding unknown return values This phase finds cases where a return value is derived from an expression that is not NULL, ERR_PTR or an explicit pointer, and that is never dereferenced. We have no information about such expressions, and so a function returning such a value must be in the category unknown.

Fig. 10 shows the rule br implementing this phase. This rule considers two cases, represented by the disjunction on lines 8-16. The first disjunct (lines 9-12) matches the case where the returned expression is initialized in an assignment that does not match one of the assignments found in rules rn (Fig. 9), re, or rp (the analogues for ERR_PTR and valid pointers), and this assignment is not followed by a dereference or a reassignment of the returned value. The second disjunct (lines 14-15) matches the case where the returned expression is a variable that is never dereferenced or initialized. Such a variable is assumed to be global or a parameter, that is initialized to an unknown value. In both cases, the position of the returned expression is recorded in the variable bad_return to indicate that its value is unknown.

```
@br exists@
identifier relevant.f,E3!={NULL},fld; position prelevant.pf;
position p2 != {rn.e_null,re.e_err,rp.e_ptr}, bad_return;
expression E,E1,E2;
@@

f@pf(...) {
(
  ... when any
  E@p2 = E1
  ... when != ( E->fld | E = E2)
  return E@bad_return;
|
  ... when != ( E3->fld | E3 = E2 ) // parameter/global variable case
  return E3@bad_return;
)
}
```

Figure 10. Finding unknown return values

Phase 3: classifying the functions The four rules shown in Fig. 11 use the information collected in phases 1 and 2 to classify a function as category 1. Similar rules identify functions in category 2. Category 3 functions are those that were found to return NULL somewhere and ERR_PTR somewhere in phase 1. Category 4 functions are those that were found to always return a valid pointer in phase 1. Other pointer-typed functions are considered unknown.

We consider only the rules for category 1 in detail. The rule precat1 identifies functions where the position of every return has been stored in phase 1 in either the variable ret_null or the variable ret_ptr. For each return it furthermore checks that the position of its argument was not also stored in the variable bad_return in phase 2. Next, the rule cat1 checks that at least one return NULL; or assignment to NULL is under a conditional. We have found that in other functions, the return of NULL is typically only there to satisfy the type checker of the C compiler, and is dead code in practice. Finally, the last rule prints out the name of each function that satisfies cat1.

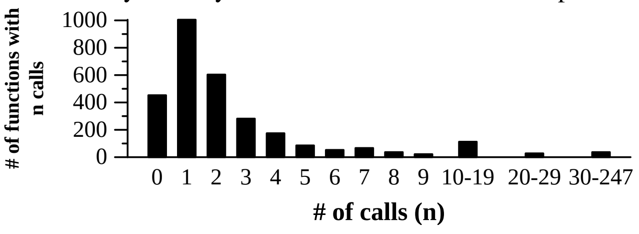
Experimental results The table below shows the result of applying Search to the above semantic match, in terms of the number of pointer-typed functions that are classified into each category. In each case except unknown, we have manually checked the classification of every function. The few false positives derive from the inadequate interpretation of the values tested by conditionals. In the case of category 1, the false positives are typically in cases where the return value can actually be unknown. For category 3, most of the false positives are actually in category 2, since the values of conditional tests imply that a NULL return value is impossible.

	classified	false positives
category 1	1640	9
category 2	478	1
category 3	112	9
category 4	623	5
unknown	7123	N/A

```
@precat1 depends on rn && !re@
identifier relevant.f; position prelevant.pf;
position any rn.ret_null; position any rp.ret_ptr;
position p != br.bad_return; expression E;
@@

f@pf(...) { ... when strict
 ( return@ret_null E@p; | return@ret_ptr E@p; )
}

@cat1 depends on precat1 exists@
identifier relevant.f; position prelevant.pf;
position any rn.ret_null, rn.e_null; expression E; statement S;
@@

f@pf(...) {
 <+...
 if(...) { <+... ( return@ret_null E; | E@e_null = NULL ) ...+> }
 else S
 ...+>
}

@ script:python depends on cat1 @
f << relevant.f; pf << relevant.pf;
@@
print "category1: FN:%s" % f
```

Figure 11. Classifying the functions

Approaches based on data mining or statistics infer protocols from frequently occurring patterns of usage [3, 6]. The graph below shows that many of the category 1-3 functions that we have classified are only rarely called directly, and thus would be likely to be missed by data-mining based approaches. Some functions are indicated as never being called because they are only used as the value of a function pointer.



4.2. Bug finding

Both inappropriate and insufficient error-handling tests are undesirable. We write bug-finding semantic match templates for each case, and instantiate them with respect to the functions identified in the protocol-finding phase.

Inappropriate tests Fig. 12 shows a semantic match template to be used in the role of BugSMTemp for finding inappropriate tests for each category 1 function, FN. The rule match detects the case where an expression *x* is assigned the result of calling FN (line 3) and then tested using IS_ERR (line 5). This check, however, is not sufficient, because there may be some other value of this expression that can reach the test, via another execution path, and it might be legitimate to test this other value for IS_ERR. The second rule, other_match, detects an initialization of *x* that is at a different position from any one matched in the first rule (line 11), as required by the constraint on the position metavariable *p1* (line 8). Finally, the Python code, which prints the result, is triggered only if the first rule matches and the second one does not (line 15). The semantic match template for category 2 functions is similar, but checks for `NULL` tests.

```
1  @match exists@ expression x, E; position p1,p2; @@
2
3  x@p1 = FN(...)
4  ... when != x = E
5  IS_ERR(x@p2)
6
7  @other_match exists@
8  expression match.x, E1, E2; position p1!=match.p1,match.p2;
9  @@
10
11 x@p1 = E1
12 ... when != x = E2
13 IS_ERR(x@p2)
14
15 @ script:python depends on !other_match@
16 p1 << match.p1; p2 << match.p2; @@
17 [...]
```

Figure 12. Finding inappropriate tests

The table below shows the result of applying `MakeBug-Report` to this semantic match template, in terms of the number of potential bug sites found and the number of false positives. A potential bug site is a single function call for which there is at least one inappropriate test match. 28 potential bug sites are reported in all. There are 7 false positives for category 2; one is due to a limitation in Coccinelle's treatment of variable declarations that initialize more than one variable, three are due to confusion between multiple functions with the same name but different return types, and the remainder are due to inadequate interpretation of values. For category 4, 16 of the false positives are due to a confusion between a category 4 function and a non-category 4 macro with the same name. All of the bugs for categories 1 and 2 have been corrected, either by ourselves or others.

	reported sites	bugs	false positives
category 1	2	2	0
category 2	26	19	7
category 4	44	23	21

The code below illustrates a bug found with the semantic match template for category 2 functions. When the call on line 2 returns `ERR_PTR`, control jumps to the `out` label (line 5). There the result of the call is retested, but for being non-`NULL` (line 6). This test succeeds, because `ERR_PTR` is different from `NULL`. The subsequent dereference (line 6), then crashes the kernel, because `aaci` is an invalid pointer.

```
1  // aaci_probe in sound/arm/aaci.c
2    aaci = aaci_init_card(dev);
3    if (IS_ERR(aaci)) { ret = PTR_ERR(aaci); goto out; }
4    [...]
5  out:
6    if (aaci) snd_card_free(aaci->card);
```

Insufficient tests We have also developed semantic match templates to find insufficient tests, such as when there is no `NULL` test between a call to a category 1 function and a dereference of the returned value. These semantic match templates are available on our website (*cf.* Section 1).

The table below shows the result of applying `MakeBug-Report` to the semantic match templates for finding insufficient tests, in terms of the number of potential bug sites found and the number of false positives. A potential bug site is a single function call for which the result is somewhere dereferenced without a prior required check. 233 potential bug sites are reported in all, of which most are for category 1 functions. We have manually verified all of the reported bugs. In all there are 72 false positives, with again most being for category 1. The causes for the false positives vary. A common cause is that some other value is tested, and the result of this test implies that the value returned by the category 1 function cannot be `NULL`. This could be addressed by integrating a dataflow analysis into Coccinelle. Another common cause is that a previous `NULL` test uses a subsystem-local abort operator, such as `FAIL`, and the semantic match does not recognize this as breaking the control flow. Information about such operators could be collected using our protocol-finding strategy. Correcting these bugs often requires creating error-handling code, which can be non-trivial. Thus, we have submitted few patches to Linux based on these bugs reports. Nevertheless, over half of the category 2 bugs have been fixed, by ourselves or others.

	reported sites	bugs	false positives
category 1	201	139	62
category 2	21	17	4
category 3	11	5	6

The code below shows an example of a typical bug found with the semantic match template for category 1 functions. This code calls the function `alloc_ctrl_packet` (line 2), which calls the generic memory allocation function `kzalloc` and returns `NULL` if the memory allocation fails. In this case, the dereference in line 6 will crash the kernel.

```
1  // ipw_send_setup_packet in drivers/char/pcmcia/ipwireless/hardware.c
2  ver_packet = alloc_ctrl_packet(
3     sizeof(struct ipw_setup_get_version_query_packet),
4     ADDR_SETUP_PROT, TL_PROTOCOLID_SETUP,
5     TL_SETUP_SIGNO_GET_VERSION_QRY);
6  ver_packet->header.length =
7     sizeof(struct tl_setup_get_version_qry);
```

5. Other Experiments

We now present our results for two other case studies: one on the use of functions involved in resource allocation and another generalizing the `netif_rx` example.

Allocation and deallocation functions Linux code contains many functions that allocate and deallocate resources. Often these are wrappers around generic allocation functions, such as `kmalloc`, that additionally perform service-specific initializations. In this case study we identify such functions, and find bugs in their usage.

In contrast to the protocol-finding semantic match presented in Section 4, which considers function definitions, here the protocol-finding semantic match focuses on a specific pattern of function calls, namely a first call that returns a pointer typed value that is compared to `NULL`, to detect failure of the allocation, and then a second call that deallocates the value before the enclosing function returns an error value. This strategy can detect allocation functions that do not call a known allocation function, such as `kmalloc`, directly. Nevertheless, it can lead to false positives, as accessor functions and functions that search for an element within a list can be used in a similar way. We thus refine the semantic match to consider only cases where the same deallocation function is used consistently within a given file and where the fields of the allocated value are always initialized before being referenced. This strategy identifies 304 pairs of allocation and deallocation functions.

Due to the complexity of many allocation and deallocation functions and the lack of relevant documentation, we have not checked all of the returned pairs. Instead, we have focused on those that cause bugs to be reported in the bug-finding phase. Of these, 39 are false positives. We expect that the actual number of false positives should be similar, as an invalid protocol is not respected by most call sites, and thus leads to bug reports.

The bug finding semantic match template is similar to the protocol finding semantic match, except that it reports a bug whenever the allocated data is not stored or freed before leaving the function with an error value. As shown in the table below, this semantic match reports 261 possible bugs, of which we have judged that 141 are real bugs, and 120 are false positives. We have begun submitting corresponding patches to the Linux community. Many of the false positives are due to cases in which one of a small set of functions is used to either store a certain type of allocated data in some way or to deallocate it, implying that no deallocation is needed before returning. While the generic bug finding rule does not give good results for this type of data, it could easily be extended to take into account these cases.

	reported sites	bugs	false positives
alloc/dealloc	261	141	120

Bug detection inspired by the netif example The third case study is motivated by the example presented in Section 3. The essential feature of the bug that was originally found was that the function `netif_rx` could free its argument, and that thus it was not safe to refer to its argument after calling the function. Because use after free is a general problem, we write a semantic match to find functions that possibly or definitely free some argument, and then a semantic match template to find calls to such function that are followed by a dereference of that argument.

The table below summarizes the number of bugs and false positives found. Most of the false positives are where the identified function `FN` decrements a reference count and possibly frees its argument. At some calls to `FN`, the reference count is known to be greater than 1, and thus the decrement cannot cause the argument to be freed.

	reported sites	bugs	false positives
guaranteed free	10	5	5
possible free	22	9	13

6. Current Limitations

Coccinelle was originally designed to perform program transformations. Although our approach has found many bugs in Linux code, our case studies have also revealed some limitations of Coccinelle for protocol and bug finding. These include the lack of an interprocedural analysis, the lack of a dataflow analysis, and the need to make some kinds of inferred information explicit in the code.

In our experience, semantic matches to find bugs often start out simple, *e.g.*, with the idea that a call to an allocation function should be followed by a call to a deallocation function along all paths. They must then be refined, to eliminate any identified false positives and false negatives. These refinements may entail working around the above limitations, such as making the propagation of values explicit by matching both variable initializations and variable uses (*cf.*, Section 4), to get around the lack of a built-in dataflow analysis.

We have begun the development of a simple dataflow analysis that can be used to propagate ranges of integer values, as is needed for detecting array bounds errors [12]. Nevertheless, such program analyses are necessarily approximate. If the user does not understand the limitations of the analysis, he may find it more difficult to see how to refine the semantic match to improve the result. Thus, there is a tradeoff between the current approach in which the action of the semantic match is expressed explicitly in its implementation, and a more implicit approach based on a range of more or less powerful program analyses. We plan to investigate this tradeoff in future work.

7. Related Work

Engler *et al.* [3] initiated the idea of using a checker that is neither sound nor complete to provide the scalability needed to find bugs in systems code. The checker uses rules expressed as automata, that have a structure quite different from C code. Engler *et al.* [4] also proposed to search for protocols in the form of pairs of functions that occur together frequently. Their approach tends to find a very large number of candidate protocols, on which statistics are used to select the most likely. There is no opportunity for the user to interject his understanding of the code structure. Later work uses automata to characterize the behaviors of typical classes of protocols, and then the user can participate in assigning specific functions to roles in such an automaton [5]. But the specifications remain distant from the source code. This work is not publicly available for comparison with related projects.

Li and Zhou use data mining to collect sets of terms that often occur together, and thus identify a number of complex protocols in Linux and other open source systems [6]. Ramanathan *et al.* show that including path sensitivity in this process significantly improves precision [11]. These tools are not publicly available. Coccinelle semantic matches also take into account control-flow paths. While the protocols we have detected in the case studies in this paper involve essentially only two operations, more complex protocols could be detected by writing more complex semantic matches. In contrast to a data mining based approach, in our approach the programmer must be aware of the basic structure of the protocol, but we can exploit this property to ease the protocol and bug validation process.

Weimer and Necula [14] propose an approach similar to that of Engler *et al.* [4], but they focus on execution paths that include error-handling code, which gives them both a smaller set of potential protocols and a smaller set of false positives. In our second case study, we have also found it useful to focus on the kinds of code that occur in error-handling. Our use of this code is tailored via the SmPL code to the class of protocol being considered, rather than following a fixed strategy as in Weimer and Necula's work.

The Static Driver Verifier (SDV) [1], developed at Microsoft, uses model checking to prove that certain bugs do not occur in systems code. Specifications amount to automata describing invalid behaviors. These automata are expressed in a C-like notation, but do not follow the structure of the code to be processed. Because SDV gives a guarantee of correctness, rather than just finding potential bugs, it is more expensive than the aforementioned approaches. It is thus better suited to being applied to individual drivers rather than to an entire operating system.

8. Conclusion and Future Work

In this paper, we have presented a framework for searching for protocols and bugs in Linux code. A principal goal of this framework is to allow users to quickly and easily interject their understanding of the code structure into the protocol and bug finding process. Our framework is based on the use of the Coccinelle transformation tool that provides a specification language that is close to C code. We have complemented Coccinelle with a collection of tools for managing a series of searches that allow a uniform approach to protocol and bug finding, based on strategies encoded by the user. We are currently submitting patches based on our results to the Linux developer community.

Coccinelle is unique among the tools used for protocol and bug finding that we know of in that it supports program transformation, via *semantic patches*. This makes it possible to specify not only how to find bugs but also how to fix them. For some of our examples, such as replacing the use of a generic deallocation function by a specific one, the change is very systematic and developing a semantic patch is straightforward. More work is necessary, however, to identify larger classes of bugs that can be fixed automatically.

Availability Coccinelle is available at `http://www.emn.fr/x-info/coccinelle/`

References

[1] T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. K. Rajamani, and A. Ustuner. Thorough static analysis of device drivers. In *Eurosys*, pages 73–85, Leuven, Belgium, Apr. 2006.

[2] J. Brunel, D. Doligez, R. R. Hansen, J. Lawall, and G. Muller. A foundation for flow-based program matching using temporal logic and model checking. In *36th Principles of Programming Languages (POPL)*, pages 114–126, Savannah, GA, USA, Jan. 2009.

[3] D. R. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *OSDI*, pages 1–16, San Diego, CA, USA, Oct. 2000.

[4] 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*, pages 57–72, Banff, Canada, Oct. 2001.

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

[6] Z. Li and Y. Zhou. PR-Miner: automatically extracting implicit programming rules and detecting violations in large software code. In *10th European Software Engineering Conference*, pages 306–315, Lisbon, Portugal, 2005.

[7] Lkml: The Linux kernel mailing list. `http://www.tux.org/lkml/`.

[8] D. MacKenzie, P. Eggert, and R. Stallman. *Comparing and Merging Files With Gnu Diff and Patch*. Network Theory Ltd, Jan. 2003.

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

[10] Y. Padioleau, J. L. Lawall, and G. Muller. Understanding collateral evolution in Linux device drivers. In *Eurosys*, pages 59–71, Leuven, Belgium, Apr. 2006.

[11] M. K. Ramanathan, A. Grama, and S. Jagannathan. Path-sensitive inference of function precedence protocols. In *29th*

International Conference on Software Engineering, pages 240–250, Minneapolis, MN, USA, 2007.

[12] H. Stuart. Hunting bugs with Coccinelle. Master's thesis, University of Copenhagen, Copenhagen, Denmark, Aug. 2008.

[13] L. Tan, D. Yuan, G. Krishna, and Y. Zhou. /*icomment: bugs or bad comments*/. In *SOSP*, pages 145–158, Stevenson, WA, USA, Oct. 2007.

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