HAL
open science

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

Julia Lawall, Julien Brunel, René R. Hansen, Henrik Stuart, Gilles Muller

To cite this version:

Julia Lawall, Julien Brunel, René R. Hansen, Henrik Stuart, Gilles Muller. WYSIWIB: A Declarative Approach to Finding Protocols and Bugs in Linux Code. 2008. hal-00294004

HAL Id: hal-00294004

https://hal.science/hal-00294004v1

Submitted on 8 Jul 2008

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

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

Julia L. Lawall,[1] Julien Brunel,[1] René Rydhof Hansen,[2] Henrik Stuart,[1] Gilles Muller[3]

[1]DIKU, University of Copenhagen, Copenhagen, Denmark
[2]Aalborg University, Aalborg, Denmark
[3]Ecole des Mines de Nantes, Nantes, France
{julia,brunel,hstuart}@diku.dk, rrh@cs.aau.dk, Gilles.Muller@emn.fr

Research report 08/1/INFO, Ecole des Mines de Nantes
July 4, 2008

Abstract

Although a number of approaches to finding bugs in systems code have been proposed, bugs still remain to be found. Current approaches have emphasized scalability more than usability, and as a result it is difficult to relate the results to particular patterns found in the source code and to control the tools to be able to find specific kinds of bugs.

In this paper, we propose a declarative approach based on a control-flow based program search engine. Our approach is WYSIWIB (What You See Is Where It Bugs), since the programmer is able to express specifications for protocol and bug finding using a syntax that is close to that of ordinary C code. Search specifications, called semantic matches, can be easily tailored so as to either eliminate false positives or catch more potential bugs. We introduce our approach by describing three case studies which have allowed us to find 395 bugs.

1. Introduction

In recent years, a multitude of approaches have been proposed to scanning systems code for API protocols and bugs in their usage [3, 4, 6, 8]. 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. Some of these tools have furthermore been successfully commercialized [2]. A weakness of these efforts, however, is in the lack of user interaction with the protocol-finding and bug-finding strategies. 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 are difficult to relate to the code structure. Both of these features make it difficult for the user to understand why something is considered to be part of a protocol or bug, or is overlooked, and thus to identify the inevitable false positives and false negatives.

Based on an extensive study of Linux code, we have observed that many of the API protocols used by Linux code follow a common pattern, related to the purpose of the protocol, such as error handling or managing the allocation of memory. It is thus our belief that taking such patterns into account in the protocol finding and bug finding processes can ease the process of checking the results and make it possible to consider protocols that occur very rarely, and thus are often overlooked by statistics-based approaches. Existing tools, however, either provide no mechanism for the user to influence the protocol-finding or bug-finding process, or only provide mechanisms that are disassociated from the code structure, making it difficult to interpret the results at bug-checking time.

In this paper, we propose a "WYSIWIB" (What You See Is Where It Bugs) approach to protocol and bug finding in Linux code, based on the following steps: 1) describe a class of protocols generically, and apply this description to the Linux source code to collect a set of protocols, 2) instantiate a collection of descriptions of various possible bug patterns for the detected protocols, and 3) apply these descriptions to the Linux source code to collect possible 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 considers how to describe what to search for, while statistics-based approaches consider how to select from the things that are found.

Concretely, we have implemented our approach as collection of tools based on the Coccinelle transformation engine [7]. A key feature of Coccinelle is that specifications are written using a language that is very close to C code, thus easing specification development.

The contributions of this work are:

- We propose a framework for finding protocols in Linux code, iteratively refining them, and using them to find bugs. This framework chiefly builds on the Coccinelle system, but provides some new complementary tools.
- We present three case studies illustrating the use of our approach for finding protocols and bugs. Two of these involve general-purpose classes of protocols. One of them was inspired by a bug fix that was submitted to Linux, illustrating the adaptability of our approach.
- Our case studies find over 3500 potential protocols, with estimated false positive rates ranging from 2% to 55%. Based on these protocols, we have used our framework to find 395 bugs that we have validated.

The rest of this paper is organized as follows. Section 2 reviews the aspects of Coccinelle that are necessary to understand the rest of the paper. Section 3 describes our protocol-finding and bug-finding framework. Sections 4 through 6 each present a case study, illustrating the processing of various kinds of common protocols. Section 7 surveys some of the current limitations of our approach. Finally, Section 8 describes related work and Section 9 concludes.

2. Coccinelle

Coccinelle is a tool for performing control-flow based program searches and transformations in C code [7]. 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 the case study reported in Section 5.

The semantic match shown in Figure 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 named bad_kfree and written in SmPL, and the second with no name 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 based pattern for specifying a match in the case of a SmPL rule, or ordinary Python code in the case of a Python rule. The notation is based on the Linux patch syntax, in that the code patterns have the structure of ordinary C code. In the rest of this section, we describe each of these rules in more detail, and give a brief tour of some of the other features of SmPL.

The rule bad_kfree defines three metavariables: x and E, which represent arbitrary expressions, and p, which represents an arbitrary position in the source program. Once bound, a metavariable must have the same value within the current control-flow path, and thus, for example, the occur-

```
1  @bad_kfree exists@
2  expression x,E;
3  position p;
4  @@
5
6    x = nlmsg_new(...)
7    ... when != x = E
8    kfree_skb@p(x);
9
10 @ script:python @
11 x << bad_kfree.x;
12 p << bad_kfree.p;
13 @@
14 print "line: %s x: %s" % (p[0].line,x)
```

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

rences of x on lines 6, 7, and 8 must all match the same expression. The code pattern in the body of the rule 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. This semantic match uses the operator "`...`" to represent a sequence of terms. In line 8 this represents a sequence of expressions in the argument list of `nlmsg_new` and in line 9 this represents an arbitrary sequence of statements reachable from code matching the call to `nlmsg_new` along any control-flow path. By default such a sequence of statements is quantified over all paths, by the annotation `exists` next to the rule name indicates that for this rule there need exist only one. Many of the remaining features of SmPL are devoted to expressing constraints on these sequences, using the operation `when`. This rule uses `when` (line 7) to indicate that there should be no reassignment of x before reaching the call to `kfree_skb`.

Python rules do not perform any matching against the source program, but may inherit metavariables from other rules, using the notation `rule.`*metaname*, and do some processing of their values. We only use Python for printing out information about the found protocols and bugs. In this example, the Python rule inherits the expression metavariable x holding the allocated value and the position metavariable holding the position of the call to `kfree_skb`. It prints the line on which the call to `kfree_skb` occurs and the text associated with the expression x. The Python interface is a new feature as compared to earlier work on Coccinelle and has been essential to managing the results of our semantic matches. We often use Python to print out information in the form expected by an emacs mode that we have developed, based on the emacs `org` mode, to quickly find and validate reported bug sites.

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 only applies when the pattern matches completely. A rule is applied once for each possible set of values of the inherited metavariables. Besides "`...`", SmPL provides *nests*, `<...`*pattern* `...>`, and *disjunctions*, `(`$pattern_1$ `|` ... `|`$pattern_n$`)`. A nest matches a sequence, like "`...`", but additionally can match zero or more occurrences of *pattern* within the matched sequence. A disjunction matches any of the patterns $pattern_1$ through $pattern_n$. The remaining SmPL operators describe further constraints on the sequences matched by "`...`" or a nest. By default such a sequence is the shortest path between code matching the patterns before and after the sequence operator. The annotation `when any` removes this shortest path restriction, allowing any number of instances of such code to occur within the path as well. Finally, by default, the universal quantification over paths does not require the complete pattern to appear on control-flow paths that end in error handling code (identified, in accordance with typical Linux programming style, as any code that has the form of a conditional ending in a return). The operator `when strict` requires full universal quantification without this relaxation.

Convention Coccinelle includes "`...`" as an operator in the language, but this notation is also convenient for showing where code is omitted in examples. We will by convention reserve "`...`" for the Coccinelle operator and use `[...]` to represent omitted code. Furthermore, we frequently use `[...]` in Python rules, rather than showing the Python code. In such rules, the significant part is the list of inherited metavariables, which indicates what will be printed out.

3. The bug-finding process

There are many sources of information for finding bugs. One can study the bug reports of others, notice a suspicious coding pattern while doing some other work on the code, or learn from sources such as code comments, newsgroups, or research papers 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 search for instances of this pattern throughout the code base, and to explore variants of the pattern as they become apparent.

To motivate our choice of methodology, we begin with a story drawn from our experience in finding and fixing bugs in Linux code. In December 2007, a patch,[1] shown in Figure 2, 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 with the same property. A semantic match for detecting such calls

[1] d30f53aeb31d453a5230f526bea592af07944564 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.

is shown in Figure 3a. It is similar in structure to the semantic match of Figure 1, except for the use of a disjunction on lines 8-12. This disjunction causes the position metavariable p used on line 11 only to record the positions of references to *skb* that do not match the first pattern, *i.e.*, are not on the left hand side of an assignment.

```
--- a/drivers/net/smc911x.c
+++ b/drivers/net/smc911x.c
@@ -1299,9 +1299,9 @@ smc911x_rx_dma_irq(int dma, void *data)
        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 2: A standard patch addressing one instance of the `netif_rx` problem

```
 1  @r exists@
 2  expression skb,e,e1;
 3  position p;
 4  @@
 5
 6   netif_rx(skb);
 7     ... when != skb = e
 8  (
 9    skb = e1
10  |
11    skb@p
12  )
13  @ script:python @
14  skb << r.skb;
15  p << r.p;
16  @@
17  [...]
```

(a)

```
 1  @r exists@
 2  expression skb,e,e1;
 3  position p;
 4  @@
 5
 6  (
 7   netif_rx(skb);
 8  |
 9   netif_rx_ni(skb);
10  )
11    ... when != skb = e
12  (
13    skb = e1
14  |
15    skb@p
16  )
17
18  @ script:python @
19  skb << r.skb;
20  p << r.p;
21  @@
22  [...]
```

(b)

Figure 3: The original (a) and extended (b) semantic matches for finding `netif_rx` problems

This semantic match found some potential bugs. 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 Figure 3b to allow it to match calls to either function. The resulting semantic match found 5 occurrences of this pattern, all of which have been acknowledged as bugs by the Linux developers. So far, our corrections for 4 of these bugs have been accepted into the Linux kernel.[2]

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, we 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 semantic match template either expresses a bug finding rule or collects more information. We explore this approach with respect to the `netif_rx` example in Section 6.

Figure 4 illustrates the use of a framework that we have developed for using Coccinelle in an iterated protocol finding and bug finding process. This framework involves four tools: `Search`, `Instantiate`, `MultiSearch`, and `MakeBugReport`. `Search`, `MultiSearch`, and `MakeBugReport` use Coccinelle in various ways, while `Instantiate` is a separate tool. We describe these tools below:

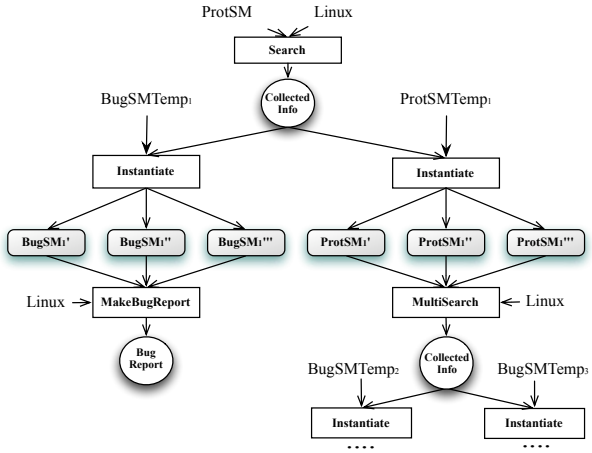

Figure 4: Protocol and bug-finding process

Search The protocol and bug finding process begins when the programmer has an idea of a certain kind of function or collection of functions whose usage protocol may be error prone. He expresses this idea as a semantic match that expresses various properties of the kind of code he is interested in and uses Python to print relevant information about the matched terms. Figure 5 illustrates such a semantic match, which detects any pair of functions f and g. The Python code prints information about these functions in the format expected by the `Instantiate` tool. The first field of this output is a tag that characterizes this match. A semantic match may use multiple such tags to separate various kinds of matched code into categories, as done in the case study 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

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

some property with respect to the protocol. For the semantic match in Figure 5, the output might contain:

```
protocol: START: atomic_dec_and_test, FINISH: spin_unlock_bh
protocol: START: list_add_tail, FINISH: spin_unlock_bh
protocol: START: read_lock_bh, FINISH: read_unlock_bh
[...]
```

```
1  @ r @
2  identifier f,g;
3  @@
4
5  f(...)
6  ...
7  g(...)
8
9  @ script:python@
10 f << r.f;
11 g << r.g;
12 @@
13 print "protocol: START: %s, FINISH: %s" % (f,g)
```

Figure 5: A protocol-finding semantic match, suitable for starting the protocol finding and bug finding process

After having developed the protocol-finding semantic match, the programmer gives it to the `Search` tool, 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 the accuracy of the protocol-finding semantic match. If it contains entries that the programmer knows 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 kind of bug or other needed information, he writes a semantic match template, which is parameterized by the various keys used in reporting the result of the previous step. He then applies the tool `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 given tag.

A semantic match template that is used to search for more information should print output in the form illustrated in Figure 5. A semantic match template that is used to search for bugs should print output in the form recognized by our emacs interface, so that the bugs can be easily validated. An example of a semantic match template that searches for more information is shown in Figure 6. It finds functions *h* that are called at least once, as indicated by the + on the nest brackets, after a call to the previously identified function `FINISH` and before the previously identified function `START`. This semantic match template would be instantiated with respect to each of the pairs of functions identified by `Search`.

```
1  @ r @
2  identifier h;
3  @@
4
5  FINISH(...)
6  <+... h(...) ...+>
7  START(...)
8
9  @ script:python@
10 h << r.h;
12 @@
13 print "protocol: FN1: START FN2: FINISH EXTRA: %s" % h
```

Figure 6: A protocol-finding semantic match, suitable for starting the protocol finding and bug finding process

MultiSearch *and* MakeBugReport `MultiSearch` is used to search for further information based on a collection of instantiated semantic match templates, while `MakeBugReport` is used to search for bugs. `MultiSearch` 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 results in a form suitable for passing to `Instantiate` again. `MakeBugReport` does essentially the same, but collects the result into a bug report for further processing with emacs. In either case, examining the results may cause the programmer to refine the corresponding semantic match template or make him aware of new kinds of bugs that are relevant to the protocol, for which new semantic match templates can be developed.

The next three sections illustrate the use of this process on three case studies, each illustrating a different strategy for finding protocols and bugs. The first case study tries to find protocols by studying function definitions, while the second considers function call sites. The former can be more accurate, because it can analyze the function's precise behavior, but it is not always possible, *e.g.*, in software that depends on libraries for which the source code is not available. The third case study is based on the `netif_rx` example, and illustrates how our approach makes it possible to easily search for specific kinds of bugs as the programmer becomes aware of them. All of the code snippets used to illustrate these case studies have been selected using semantic matches. The bugs we have identified have not yet been validated by the Linux developers. However, our assessment of the number of bugs is based on careful study of the code building on previous experience obtained for patches we have submitted and had accepted, as listed at the Coccinelle web site.[3] All of our experiments are based on a snapshot of Linux dated March 11, 2007[4] and were run on an HP ProLiant server with two 3GHz quad-core Xeon processors and 16GB memory.

[3] `http://www.emn.fr/x-info/coccinelle/`

[4] Git code baadac8b10c5ac15ce3d26b68fa266c8889b163f.

4. Case Study 1: Detecting Inconsistent Error Checks

The C programming language does not provide any built-in exception handling mechanism, and thus applications must define their own protocols for detecting and handling exceptional conditions. In Linux, pointer-typed functions typically return either NULL, a value constructed with ERR_PTR,[5] or both to indicate failure. Code using such functions must then correspondingly follow one of three protocols before dereferencing the result: 1) testing the value for NULL, 2) testing the value for ERR_PTR using the function IS_ERR, or 3) performing both tests. Choosing the wrong protocol amounts to performing either inappropriate tests or insufficient tests and can lead to invalid pointer dereferences that can crash the Linux kernel. In this section, we consider how to classify functions in terms of the kinds of error values they may return and how to detect various kinds of bugs in their use.

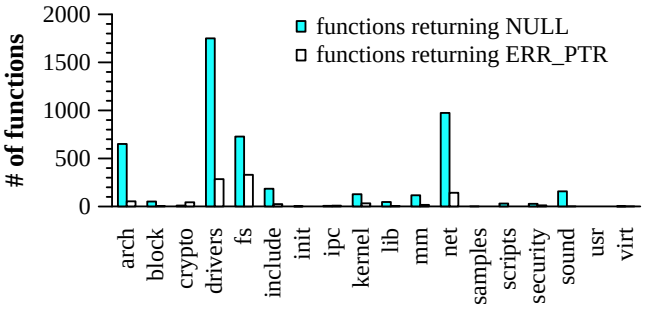

Figure 7: The use of NULL and ERR_PTR in the Linux kernel by directory

4.1 Protocol detection

We detect instances of the above protocols by examining function definitions to infer the usage protocols that they imply. Our goal is to write a protocol finding semantic match that identifies functions in the following categories: Category 1) Functions that indicate an error by returning NULL; Category 2) Functions that indicate an error by returning a value created using ERR_PTR; Category 3) Functions that indicate an error by returning either NULL or ERR_PTR.

To give an overview of the strategies used by our protocol finding semantic match, we consider the Linux functions simple_alloc_urb and clk_get shown in Figure 8. Relevant code is highlighted in italics. As illustrated by lines 7, 16, and 23, a function may explicitly return NULL, ERR_PTR, or a pointer created using &, or 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 ways in which its value is used. For example, in line 9, the conditional test implies that the variable urb is NULL at the point of the return, and in line 14, the dereference of urb means that its value can subsequently be assumed to be a valid pointer. These observations allow us to conclude that the function simple_alloc_urb in Figure 8a is a category 1 function. Similar observations allow us to identify category 2 and 3 functions. For the function clk_get, however, we do not have enough information to classify the function, due to the function call in line 22. In this case, we consider the function to be unknown.

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

(b)

Figure 8: Functions illustrating various ways of returning error codes. These functions are defined in drivers/usb/misc/usbtest.c and arch/powerpc/kernel/clock.c, respectively

We now consider how to write a semantic match that expresses these intuitions. This semantic match is used in the role of "ProtSM" in Figure 4. The semantic match is in four phases, as shown in Figures 9 to 12.

Phase 1: making information explicit As illustrated on line 9 of Figure 8, it is possible to determine whether various variables are NULL or ERR_PTR from the conditional tests that appear in the function. To avoid having to subsequently consider many special cases, this initial phase makes such information explicit, by introducing corresponding assignments under such conditional tests. For example, line 9 of simple_alloc_urb becomes if (!urb) { urb = NULL; return urb; }.

Figure 9 shows the transformation rule for NULL tests. There is a similar rule for IS_ERR tests. These rules make use of the transformation features of Coccinelle, where lines that are annotated with - are removed and lines that are annotated with + are added. These rules furthermore make use of a feature of Coccinelle known as an *isomorphism* [7], *i.e.*, a collection of terms having different forms but essentially the same meaning. Here we take advantage of

[5] For conciseness, we subsequently refer to "a value constructed with ERR_PTR" as just "ERR_PTR".

built-in isomorphisms that convert E == NULL to NULL == E and !E, interchange the arguments of && and ||, and interchange the conditional branches. Nevertheless, SmPL cannot express arbitrary repetitions of operators, and so the pattern does not match cases where the NULL or IS_ERR test is deeply nested within the test expression. Still, we have been able to classify almost 3000 functions into the categories 1, 2, and 3 despite this limitation.

```
1  @t1 using "likely.iso" disable add_parens @
2  expression *E;
3  identifier def0.f;
4  statement S1, S2;
5  @@
6
7  f(...) {
8  <...
9  (
10   if ((E == NULL | E == NULL&&...))
11 + {
12 +   E = NULL;
13     S1
14 + }
15   else S2
16 |
17   if ((E != NULL | E != NULL||...)) S1
18 + else E = NULL;
19 )
20  ...>
21 }
```

Figure 9: Making NULL values explicit

Phase 2: detecting returns of NULL ***and*** ERR_PTR The second phase detects functions that somewhere return NULL, ERR_PTR, or both. The rule for the NULL case, shown in Figure 10, checks that a function contains either an explicit return of NULL (line 9) or an assignment of some expression to NULL followed by a return of that expression (lines 11-14). It does not handle arbitrary levels of aliasing, but this rarely occurs intraprocedurally in Linux code.

```
1  @returns_null exists@
2  identifier def0.f,fld;
3  expression E, E1;
4  @@
5
6  f(...) {
7    ... when any
8  (
9    return NULL;
10 |
11   E = NULL;
12   ... when != ( E = E1 | E->fld )
13   return E;
14 )
15 }
```

Figure 10: Detecting returns of NULL and ERR_PTR

Phase 3: detecting unknown return values This phase detects cases where the 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 such a function must be in the category unknown. The rules implementing this phase are shown in Figure 11. The first rule uses the position metavariable *p* to mark the locations of all of the assignments where the assigned value is either NULL, ERR_PTR or an explicit pointer (line 9). The second rule detects returns where the argument has been assigned, but not by one of the assignments detected in the first rule, or has not been assigned at all, as would be the case of an explicit function call. In either case, a when clause checks that the returned value is not dereferenced, and is thus not known to be a valid pointer.

```
1  @a depends on !returns_null || !returns_errptr @
2  position p;
3  identifier def0.f;
4  expression E,E1;
5  @@
6
7  f(...) {
8  <...
9  ( E@p = ERR_PTR(...) | E@p = NULL | E@p = &E1 )
10 ...>
11 }
12
13 @b depends on !returns_null || !returns_errptr exists@
14 identifier def0.f, fld;
15 expression E,E1,E2;
16 position p, p1, p2 != a.p;
17 @@
18
19 f@p(...) {
20 (
21   ... when any
22   E@p2 = E1
23   ... when != ( E = E2 | E->fld )
24   return@p1 E;
25 |
26   ... when != ( E = E2 | E->fld )
27   return@p1 E;
28 )
29 }
```

Figure 11: Detecting unknown return values

Phase 4: classifying the functions At this point, we have collected enough information to classify the functions. The rule cat1, shown in Figure 12 considers a function to be in category 1 if every path through the function ends with either a return of NULL, a return of a pointer created with & or a return that was not classified as unknown by phase 3. A second rule, notcat1, which is not shown, ensures that there is not another definition of the function that returns an unknown value, as may occur if #ifdef is used to provide multiple definitions of the function within a single file. Finally, a Python rule prints out the name of any function that satisfies cat1 and does not satisfy notcat1, indicating that the function is in category 1. A similar Python rule, which is not shown, indicates that the function is unknown if it satisfies notcat1.

Similar rules identify functions in category 2. Category 3 functions are those that were found to somewhere return NULL and to somewhere return ERR_PTR in phase 2. Other pointer-typed functions are considered unknown.

Experimental results Figure 13 shows the result of applying Search to the above semantic match, in terms of the number of pointer-typed functions that are classified as into

```
@cat1 depends on returns_null && !returns_errptr@
position p, p2 != b.p1;
identifier def0.f;
expression E;
@@

f@p(...) {
  ... when strict
( return NULL; | return &E; | return@p2 E; )
}

@ script:python depends on cat1 && !notcat1@
f << def0.f;
@@
print "category1: FN:%s" % f
```

Figure 12: Classifying the functions

each category. Due to time constraints, we have not been able to verify all of these results. Instead, we have randomly selected 50 functions from each category and studied its definition to determine the validity of the classification. As shown in Figure 13, we find very few false positives. All of the false positives derive from the inadequate interpretation of conditionals. In the case of categories 1 and 2, the false positives are typically in cases where the return value can actually be unknown. All of the category 3 false positives are actually in category 2, since the values involved in the conditional tests imply that a return value of NULL is impossible.

	classified	validated	false positives
category 1	2394	50	1
category 2	480	50	2
category 3	100	50	4
unknown	6940	N/A	N/A

Figure 13: Results of classifying pointer-typed functions

Approaches that are based on data mining or statistics infer protocols from frequently occurring patterns of usage [3, 6]. Figure 14 shows that many of the functions that we have classified are directly called only a few times. Some functions are indicated as being called 0 times because they are only used as the value of a function pointer.

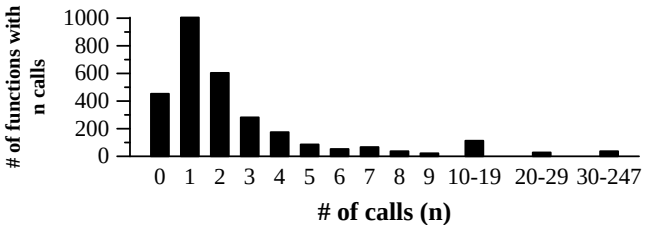

Figure 14: Call frequency for the categorized functions within the Linux kernel source code

4.2 Bug detection

As noted at the beginning of the section, performing inappropriate tests and performing insufficient tests are both undesirable. We thus write bug-finding semantic match templates for each of these cases, and instantiate them with respect to the functions identified in the protocol-finding phase.

Inappropriate tests Figure 15 shows a semantic match template for detecting inappropriate tests for each possible function, FN, in category 1. The rule match detects the case where an expression x is assigned the result of calling a FN (line 6) and then tested using IS_ERR (line 8). 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 detects an initialization of x that is at a different position from any one matched in the first rule (line 15), as required by the constraint on the position metavariable *p1* (line 12). Finally, the Python code, which prints the result, is triggered only if the first rule matches and the second one does not (line 19). Another pair of rules, which is not shown, considers the case where the call appears explicitly in the argument of the IS_ERR test. The semantic match template for category 2 functions is similar, but checks for various kinds of NULL tests rather than an IS_ERR test.

```
1  @match exists@
2  expression x, E;
3  position p1,p2;
4  @@
5
6  x@p1 = FN(...)
7  ... when != x = E
8  IS_ERR(x@p2)
9
10 @other_match exists@
11 expression match.x, E1, E2;
12 position p1!=match.p1,match.p2;
13 @@
14
15 x@p1 = E1
16 ... when != x = E2
17 IS_ERR(x@p2)
18
19 @ script:python depends on !other_match@
20 p1 << match.p1;
21 p2 << match.p2;
22 @@
23 [...]
```

Figure 15: Template for detecting inappropriate tests

Figure 16 shows the result of applying MakeBugReport to this semantic match template, in terms of the number of potential bug sites found and the number of false positives. We consider a potential bug site to be a single function call for which there is at least one inappropriate test. 28 potential bug sites are reported in all. There is only one false positive, which is due to a limitation in Coccinelle's treatment of variable declarations that initialize more than one variable.

	reported sites	bugs	false positives
category 1	2	2	0
category 2	26	25	1

Figure 16: Inappropriate tests bugs

Figure 17 shows an example of a bug found with the semantic match template for category 2 functions. This code comes from the function `aaci_probe` in the file sound/arm/aaci.c. When the call to `aaci_init_card` (line 1) returns `ERR_PTR`, control jumps to the `out` label (line 4). There the result of the call is tested again, this time for being non-`NULL` (line 8). This test succeeds, because an `ERR_PTR` value is always different from `NULL`. The subsequent dereference to access the `card` field (line 9), will then crash the kernel, because `aaci` is an invalid pointer.

```
1   aaci = aaci_init_card(dev);
2   if (IS_ERR(aaci))
3     ret = PTR_ERR(aaci);
4     goto out;
5
6   [...]
7 out:
8   if (aaci)
9     snd_card_free(aaci->card);
```

Figure 17: An inappropriate test found with the template for category 2 functions

Insufficient tests Figures 18 to 20 show extracts of a semantic match template for detecting calls to category 1 functions where there is no `NULL` test before dereferencing the result. The rules for categories 2 and 3 are similar. The semantic match proceeds in three phases.

The first phase, as in the protocol-finding semantic match, reorganizes the code to simplify the subsequent analysis. Here, the reorganization transforms tests by breaking apart conjunctions and disjunctions, as shown for conjunctions in conditionals in Figure 18.

```
1  @and depends on def0 using "../likely.iso" disable and_comm @
2  position xtesta.p;
3  expression E1, E2;
4  statement S1, S2;
5  @@
6
7  - if@p (E1 && E2) S1
8  + if (E1) { if (E2) S1 }
```

Figure 18: Phase 1: Reorganization of conditional tests

The second phase detects cases where a dereference is known to be safe. This can be because the dereference is guarded by a test of the expression for `NULL` (*e.g.*, by a conditional or loop test), or because there is a conditional that tests the expression for `NULL` and then either updates the expression to some other value or aborts in some way (*e.g.*, a `return` or `panic`, but also a `break` or `continue`). Position variables are used to remember the terms that are matched in these cases. Figure 19 shows a simplified version of the rule protected that collects in the metavariable p the positions of dereferences protected by an enclosing `NULL` test (the full rule also considers while and for loops and conditional expressions). The rules updated and aborts, which are not shown, match the remaining conditions.

```
1  @protected using "../likely.iso" @
2  expression def0.x;
3  identifier fld;
4  position p;
5  statement S;
6  @@
7
8  if ((x != NULL | x != NULL &&...)) {<... x@p->fld ...>} else S
```

Figure 19: Phase 2: Detecting dereferences protected by an enclosing conditional

Finally, the third phase, shown in Figure 20, detects and prints the potential bug sites. Starting from an assignment of some expression x to the result of a call to the category 1 function `FN` (line 12), the rule finds any subsequent dereference of the returned value that is not preceded by a reassignment of x and that is not at a position identified by protected. The search furthermore stops at a call to `BUG_ON` that tests for `NULL`, as this aborts the kernel, and at the conditionals identified by the unprotected and aborts rules. Finally, the Python code prints a bug report for each pair of a call to the category 1 function and an identified dereference.

```
1  @unprotected using "../likely.iso" exists@
2  expression def0.x;
3  identifier fld, fld1;
4  position p != protected.p;
5  position call.p1;
6  position any updated.px;
7  position any aborts.ab;
8  statement S,S1;
9  expression E;
10 @@
11
12   x@p1 = FN(...)
13   ... when != ( x = E | x->fld1 )
14 (
15   BUG_ON((x == NULL | x == NULL||...));
16 |
17   if@ab (...) S1 else S
18 |
19   if@px (...) S1 else S
20 |
21   x@p->fld
22 )
23
24 @ script:python @
25 p << unprotected.p; // position of ref
26 p1 << call.p1;      // position of call
27 fld << unprotected.fld;
28 @@
29 [...]
```

Figure 20: Phase 3: Detecting unprotected dereferences

Figure 21 shows the result of applying `MakeBugReport` to this semantic match template, in terms of the number of potential bug sites found and the number of false positives. We consider a potential bug site to be a single function call for which there are one or more dereferences of the result without a previous required check. 535 potential bugs are reported in all, of which over 90% are for category 1 functions. For category 1, we have verified 250 potential bug sites, due to time limitations. For the other categories, we have verified all reported bugs. In all there are 42 false positives, with again most being for category 1. The reasons for

the false positives vary. The most common are that a dereference occurred at the destination of a goto, causing the semantic match not to detect that it was protected by a prior NULL test, and that the NULL test was present, but embedded in the definition of some other called function. The former issue could be addressed by enhancing the semantic match to check whether code at the destination of a goto is only reachable from code where the test has been performed. Addressing the latter issue in general would require an interprocedural analysis.

	reported sites	validated sites	bugs	false positives
category 1	496	250	215	35
category 2	20	20	16	4
category 3	19	19	16	3

Figure 21: Insufficient tests bugs

Figure 22 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 1), which defined in the same file, calls the generic memory allocation function `kzalloc` and returns NULL if the memory allocation fails. In this case, the dereference in line 5 will crash the kernel.

```
1 ver_packet = alloc_ctrl_packet(
2    sizeof(struct ipw_setup_get_version_query_packet),
3    ADDR_SETUP_PROT, TL_PROTOCOLID_SETUP,
4    TL_SETUP_SIGNO_GET_VERSION_QRY);
5 ver_packet->header.length =
6    sizeof(struct tl_setup_get_version_qry);
```

Figure 22: An insufficient tests bug in the function ipw_send_setup_packet in the file drivers/char/pcmcia/ipwireless/hardware.c

5. Case Study 2: Detecting 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. Such functions may also be used to manage reference counts. The goal of this case study is to detect functions that allocate and deallocate resources, and bugs in their usage.

5.1 Protocol detection

We consider allocation and deallocation of resources that are represented as pointer values. Allocation and deallocation of such resources are carried out by function calls. The challenge is to distinguish allocation and deallocation functions from functions that manipulate the data in other ways, such as accessing its fields or storing it more permanently, *e.g.*, in a global variable. In this case study, we try to characterize a pair of allocation and deallocation functions based on how they are used.

Our strategy is based on the following observations. When a function calls an allocation function, it typically tests the returned resource for validity and then saves or deallocates this result. In particular, when an error occurs in such a function, the function typically deallocates the resource before returning. Indeed, deallocation must be done if the resource is only stored in a local variable. This pattern is illustrated by the code fragment shown in Figure 23. Line 10 uses the function `alloc_tty_driver` to allocate a `tty_driver` resource, and the result is stored in the local variable `drv`. The result is checked for validity in the next line. The result remains in a local variable until the conditional on line 14, which checks for an error condition. If an error has occurred, the conditional frees the resource `drv` on line 15 using the function `put_tty_driver` and returns an error value, `-1`.

```
 1  static int capinc_tty_init(void)
 2  {
 3    struct tty_driver *drv;
 4
 5    if (capi_ttyminors > CAPINC_MAX_PORTS)
 6      capi_ttyminors = CAPINC_MAX_PORTS;
 7    if (capi_ttyminors <= 0)
 8      capi_ttyminors = CAPINC_NR_PORTS;
 9
10    drv = alloc_tty_driver(capi_ttyminors);
11    if (!drv)
12      return -ENOMEM;
13    ... // initializations of various fields of dev
14    if (tty_register_driver(drv)) {
15      put_tty_driver(drv);
16      printk(KERN_ERR "Couldn't register capi_nc driver\n");
17      return -1;
18    }
19    capinc_tty_driver = drv;
20    return 0;
21  }
```

Figure 23: Extract of drivers/isdn/capi/capi.c. Code relevant to the allocation-deallocation pattern is shown in italics.

A semantic match based on these observations is shown in Figure 24. In this semantic match, the metavariable E represents the local variable that contains the allocated value, the metavariable f represents the allocation function, and the metavariable g represents the deallocation function. The metavariable f is specified not to be one of the basic memory allocation functions, `kmalloc`, `kzalloc`, `kcalloc` (line 2), because it is well-known that memory allocated with these functions should be deallocated using `kfree`. Lines 9-13 of the semantic match establish the pattern of an allocation. These lines check for a local variable declaration (line 9), then any subsequent assignment of that variable to the result of a call to a function f (line 11), and finally a test whether the initialized value of the local variable is NULL (line 13), which is assumed to be a validity test on the result. Lines 18-25 of the semantic match search for the pattern of a deallocation under an error condition. This has the form of a conditional (line 18) containing a call to a function g (line 22) that takes the local variable as some argument, followed by a return (line 24), with no intervening use of the local variable

(line 23). The pattern furthermore specifies that within the conditional and before the call to the deallocation function, the local variable cannot be updated (line 19) stored in the value of some other expression (line 20), or passed to another function (line 21). The latter condition serves to avoid the case where this other function stores the value in some more permanent way, such as adding it to a global queue. Finally, the same conditions are placed on the code between the allocation code and the deallocation code. Any other code, however, can appear, including other conditionals (line 17).

```
 1  @alloc exists@
 2  identifier E,f != { kmalloc,kcalloc,kzalloc },g,h1,h2;
 3  expression E1,E2;
 4  int ret!=0;
 5  type T;
 6  statement S;
 7  @@
 8
 9    T *E;
10    ... when any
11    E = f(...);
12    ... when != E
13    if (E == NULL) S
14    ... when != ( E = E1; | E1 = E; | h1(...,E,...) )
15        when any
16    if (...) {
17      ... when != ( E = E2; | E2 = E; | h2(...,E,...) )
18      g(...,E,...);
19      ... when != E
20      return ret;
21    }
23
24  @ script:python @
25  f << alloc.f; // identifier
26  g << alloc.g; // identifier
27  @@
28  [...]
```

Figure 24: A simplified semantic match for detecting allocation-deallocation protocols

This semantic match finds 376 pairs of allocation and deallocation functions, but it overlooks a number of others that do not match the specified code patterns in some way. For example, an allocation can be described, as specified by this semantic match, in three separate statements (declaration, allocation, and validation) but some of these steps can also be merged, *i.e.*, the allocation can be part of the declaration, or it can appear in the test expression of the conditional performing the validation. Furthermore, the test expression of this conditional can be more complicated, using the boolean operators && and ||. We can also allow for a subsequent test using IS_ERR, which has the form of a function call but is known not to save its argument. Finally, in the deallocation part, the function g may return a value, and the entire function may return a non-integer value, such as NULL, or no result at all.

Our complete semantic match addressing the above issues is about 20 lines longer than the one shown. It finds 602 pairs of allocation and deallocation functions, including all of the ones found by the simplified version. We have studied 50 randomly selected pairs of identified allocation and deallocation functions and found that 37 represent valid protocols and 13 are false positives. The most common reason for a false positive is that the function identified as an allocation function performs an access rather than an allocation, and so the last function that uses the accessed data within an error path is inappropriately identified as a deallocation function.

Even the generalized semantic match is not able to detect all allocation/deallocation protocols, because some such functions may never be used in the context required by the semantic match. For example, an allocation function may always be used in a context where its result is immediately stored in a structure field or it may never be used in a function that needs to handle an error. The NULL test on the result of the allocation function might always be omitted, or it might be expressed in a more convoluted way that is not detected by the semantic match. All of these constraints were built into the semantic match to try to reduce the number of false positives while still finding many protocols. Nevertheless, the programmer could experiment with relaxing them.

This semantic match has focused on allocation functions that return pointer-typed values where failure of the allocation is represented as NULL. It could easily be adapted to the case where the allocation function returns ERR_PTR to indicate failure, or where the allocated information is represented as an integer rather than a pointer.

5.2 Bug detection

The bugs we search for have the form of a call to an allocation function with no subsequent call to a deallocation function. As in the protocol-finding semantic match, we focus on values stored in local variables and error-handling code, to reduce the number of false positives.

Searching for missing deallocations A simplified version of the bug-finding semantic match template is shown in Figure 25. It follows the protocol-finding semantic match quite closely, except that the calls to the deallocation function are replaced by constraints that no call to the deallocation function appears. Finally, as compared to the protocol-detecting semantic match, we also remove the constraint that the value is not passed to another function, as our goal now is to find as many bugs as possible.

The semantic match template from which this simplified version has been constructed finds some bugs, as shown in the first line of Figure 26, but it also finds about as many false positives. In some cases, the value is passed to an intermediate functions that either saves or deallocates it. In particular, many Linux services define a cleanup function that deallocates a number of resources that the service may have allocated. In other cases, the deallocation is under a conditional that is more complex that the NULL test checked for in the rule in Figure 24.

To address this issue, we can exploit the ease of modifying SmPL code to create more constrained semantic match templates that match fewer bugs, but also fewer false posi-

```
@bug exists@
type T;
identifier E;
expression E1,E2;
statement S,S1,S2;
int ret != 0;
position a,p;
@@

  T *E;
  ... when any
  E = ALLOC@a(...);
  ... when != E
  if (E == NULL) S
  ... when any
      when != ( E = E1 | E1 = E | FREE(...,E,...) )
      when != if (E != NULL) { ... FREE(...,E,...) ... }
(
   if (E == NULL) S1 else S2
|
   if (...) {
     ... when != ( E = E2 | E2 = E | FREE(...,E,...) )
         when != if (E != NULL) { ... FREE(...,E,...) ... }
       return@p ret;
   }
)

@ script:python @
a << bug.a;
p << bug.p;
@@
[...]
```

Figure 25: A simplified template for detecting missing deallocations

	reported sites	validated sites	bugs	false positives
Full detection	470	180	81	99
Subsequent deallocation required	158	100	50	50
Allocated data not passed to an intervening function	189	61	36	25
Both restrictions	76	43	27	15

Figure 26: Missing deallocation bugs

tives. One can then concentrate on a smaller, simpler bug report in which the matched code has more commonality thus making the bugs easier to verify. After having thoroughly studied these simpler bug reports, one can return to the original bug report and consider the remaining cases, having gained more experience about the structure of the affected code. We have considered three variants on the semantic match of Figure 24: one where we require that there is a deallocation of the same value after the conditional in which a bug seems to occur, one in which we reintroduce the constraint that the value is not passed to any function between the allcation and the conditional in which the bug seems to occur, and one that combines the two. Other variants are possible.

Figure 26 also shows the result of projecting the information about bugs and false obtained for the full semantic match template onto the bug reports generated by the more restricted versions. The first constraint has no significant impact on the ratio of bugs to false positives, but imposing the second constraint or combining the two substantially reduces the number of false positives, while still making it possible to find some bugs.

A special case Finally, we have created an extremely constrained semantic match template, which was motivated by our study of the original reported bugs. In some cases, a specific allocation function and deallocation function are defined for a particular type of value. The allocation function may, for example, allocate memory using `kmalloc` and initialize some fields, but in some cases the deallocation does nothing specific at all, and simply calls a generic deallocation function such as `kfree`. Because the effect is the same, code may use the generic function rather than the specific one. This, however, represents a breaking of the abstraction boundary, and can lead to problems later, if is becomes necessary to augment the deallocation function. We have thus written a semantic match to find deallocation functions that simply call some other function with the same argument, and then a semantic match to correct the bug by replacing calls to these functions by calls to the specific variant that corresponds to the allocation function that is used. In this case, we use the tool `MultiSearch` to iterate the protocol-finding process. The semantic match template identifying the relevant deallocation functions is shown in Figure 27. The semantic match template finding the bugs is a generalization of the semantic match used to present Coccinelle in Section 2. This semantic match template can furthermore potentially find bugs that the previously presented ones do not, because it does not restrict the search for the deallocation function to error conditions.

```
@r@
identifier f,x;
type T;
@@
FREE(T x) { f(x); }

@ script:python @
f << r.f;
@@
print "oneline: AL:ALLOC FR:FREE FN:%s" % f
```

Figure 27: A template for detecting a deallocation function that simply calls another deallocation function

The semantic match template detecting deallocation functions that simply call some other function on the same argument detected 18 such deallocation functions, associated with in all 24 allocation functions. Bugs are reported in the use of three of the deallocation functions, `framebuffer_release`, `nlmsg_free`, and `vfree`, for a total of 27 bug reports, all but two of which are in the use of `nlmsg_free`. The bug in the use of `vfree` is a false positive, as it is part of a conditional that tests whether `vfree` or `kfree` should be used. Checking these bugs took only a few minutes. These results clearly show the benefits of being able to create semantic matches that check for very specific

conditions, as there is the potential for a high rate of found bugs and validation of these bugs can be very easy.

6. Case Study 3: 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 number of bugs and false positives found are summarized in Figure 28. Most of the false positives are where the identified function `FN` decrements a reference count and possibly frees its argument. At some calls to such a function `FN`, the referece count is known to be greater than 1, and thus the decrement cannot cause the value to be freed.

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

Figure 28: Reference after call to a freeing function

7. Current Limitations

Coccinelle was originally designed for performing program transformations. Although our approach has found a large number of bugs in Linux code, our case studies have also revealed some limitations of Coccinelle for protocol and bug finding. These limitations include the lack of an interprocedural analysis, the lack of a dataflow analysis, the inability to interpret arbitrarily complex conditional texts, and the need to make some kinds of inferred information explicit in the code. As a result, semantic matches can be complex. Furthermore, they tend to become more complex as they are refined, to eliminate false negatives and false positives.

Some of these limitations may be essential for transformation, where the user would like to retain very tight control over the conditions under which the transformation takes place. For protocol and bug finding, however, it may be useful to consider whether a weaker coupling between the code fragments in the semantic match and the elements of the matched code could make the approach easier to use while maintaining or reducing the rate of false positives.

8. 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. Enlger *et al.* also proposed [4] to search for protocols in the form of pairs of functions that occur together frequently. The former is based on checking rules expressed as automata, that have a structure quite different from C code. The latter 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.

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 [8]. 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 [10] propose an approach similar to that of Engler *et al.* [4], but they focus on protocols and bugs that occur in error paths, which gives them both a much smaller set of potential protocols and a much smaller set of false positives. In our second case study, we have also found it useful to exploit the kinds of code that occur in error paths. Our use of error paths is also 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.

In previous work [9], we have used Coccinelle to reexpress the bug-finding automata presented in the work of Engler *et al.* [3]. That paper did not consider protocol finding. Furthermore, it only considered a few very generic functions, such as `kmalloc` and `kfree`, for which there appear to be few remaining usage errors in Linux today.

9. 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.

Coccinelle is unique among the tools used for protocol and bug finding that we know of in that it supports program transformation. 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 creating a semantic patch is straightforward. More work is necessary, however, to identify larger classes of bugs that can be fixed automatically.

Availability The source code for the semantic matches used in our experiments is available at the following URL:

`http://www.diku.dk/~julia/bugs`

References

[1] BALL, T., BOUNIMOVA, E., COOK, B., LEVIN, V., LICHTENBERG, J., MCGARVEY, C., ONDRUSEK, B., RAJAMANI, S. K., AND USTUNER, A. Thorough static analysis of device drivers. In *EuroSys 2006* (Apr. 2006), pp. 73–85.

[2] Coverity. http://www.coverity.com/, 2008.

[3] ENGLER, D. R., CHELF, B., CHOU, A., AND HALLEM, S. Checking system rules using system-specific, programmer-written compiler extensions. In *SOSP* (Oct. 2000), pp. 1–16.

[4] ENGLER, D. R., CHEN, D. Y., CHOU, A., AND CHELF, B. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *SOSP* (Oct. 2001), pp. 57–72.

[5] KREMENEK, T., TWOHEY, P., BACK, G., NG, A., AND ENGLER, D. From uncertainty to belief: Inferring the specification within. In *OSDI* (Nov. 2006), pp. 161–176.

[6] LI, Z., AND ZHOU, Y. PR-Miner: automatically extracting implicit programming rules and detecting violations in large software code. In *ESEC/FSE* (2005), pp. 306–315.

[7] PADIOLEAU, Y., LAWALL, J., HANSEN, R. R., AND MULLER, G. Documenting and automating collateral evolutions in Linux device drivers. In *Eurosys 2008* (Mar. 2008), pp. 247–260.

[8] RAMANATHAN, M. K., GRAMA, A., AND JAGANNATHAN, S. Path-sensitive inference of function precedence protocols. In *ICSE* (2007), pp. 240–250.

[9] STUART, H., HANSEN, R. R., LAWALL, J., ANDERSEN, J., PADIOLEAU, Y., AND MULLER, G. Towards easing the diagnosis of bugs in OS code. In *PLOS* (Oct. 2007).

[10] WEIMER, W., AND NECULA, G. C. Mining temporal specifications for error detection. In *TACAS* (Apr. 2005), pp. 461–476.