Hunting bugs with Coccinelle

Department of Computer Science, University of Copenhagen

Henrik Stuart

8th August 2008

Abstract

Software bugs are an ever increasing liability as we become more dependent on software. While many solutions have been produced to find bugs, there is still ample room for improvement. In this thesis we have used the source-to-source transformation engine for the C programming language, Coccinelle, by extending it with reporting facilities and static analysis prototyping capabilities using Python that integrate with the OCaml code of Coccinelle. Using the prototyping capabilities, we have developed patterns for matching stack-based buffer overflows and use-after-free bugs. We have furthermore developed an alternative control flow graph representation for Coccinelle in an effort to decrease the number of false positives when we search for use-after-free bugs, and we have implemented a generalised constant propagation algorithm to estimate value ranges for program variables. We have run our bug patterns on several code-bases ranging from 30,000 lines of source code up to over 5.5 million lines of source code and found bugs in all of the code-bases. While our patterns only provide a first step towards making Coccinelle into a general-purpose bug hunting tool, they have successfully shown that Coccinelle has the potential to compete with many of the currently available bug finding tools.

Resumé

I takt med at vi bliver mere afhængige af software jo større et problem bliver programfejl. Selvom der er lavet mange løsninger til at finde programfejl, så er der stadig rig mulighed for at lave forbedringer. Vi har benyttet Coccinelle, et kildeteksttransformeringsprogram til C-programmeringssproget, og udvidet det med funktionalitet til at rapportere fejl og med funktionalitet til at prototype statiske analyser ved at integrere Python med den eksisterende OCaml-kode som Coccinelle er skrevet i. Ved at bruge prototype-funktionaliteterne har vi udviklet søgemønstre til at finde stak-baserede buffer-overløb og *use-after-free*-fejl. Vi har ydermere udviklet en alternativ repræsentation af *control flow graphs* i Coccinelle for at begrænse antallet af falske positiver ved søgning efter *use-after-free* fejl, og vi har implementeret *generalised constant propagation* til at beregne de mulige værdier en program-variabel kan have på kørselstidspunktet. Vi har afviklet vores søgemønstre på kildetekster til flere programmer som indeholder fra 30.000 linjers kildetekst til over 5,5 millioner linjers kildetekst, og vi har fundet fejl i samtlige programmer. Selvom vores søgemønstre kun er det første skridt til at bruge Coccinelle som et generelt anvendeligt fejlfindingsværktøj, så har de vist, at Coccinelle har potentiale til at konkurrere på lige fod med mange af de fejlfindingsværktøjer som er tilgængelige i dag.

To Ida who always brings the sunshine

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

Software has permeated our lives to a degree where we are increasingly dependent on it. This dependency comes with a cost that we pay when software malfunctions. For end users the cost may be nothing more than a slight nuisance when their media player crashes during their favourite television show, but for a company, the halted flow of traffic to their website can mean millions of euros in losses, and for critical software, the malfunction of electronically controlled car brakes could result in the ultimate cost, the loss of human life.

Despite the fact that there has been an increased focus on testing with various unit test tools, and the existence of several analysis tools that can find possible bugs in software, there is still an overwhelming amount of reported vulnerabilities in commercial and open source software alike, ranging from benign issues that the local user has to initiate, to vulnerabilities where malicious attackers can remotely crash a system or assume complete control of it.

One of the contributing factors to the infrequent use of analysis tools is that they are often hard to use and require a serious investment of time into understanding the underlying theory of their functionality. Furthermore, they may often only be suitable for a single purpose and not allow the user to dictate or extend its functionality.

In this thesis we will use the source-to-source transformation tool Coccinelle to find faults in software by using its existing source-code matching functionality and by extending it with static analysis features.

The following sections will describe Coccinelle and give a brief overview of program analysis.

1.1 Coccinelle

Maintenance frequently touches many components in a software program, and in some cases changes in a core component may require changes in all the program parts that use this component—so-called *evolution* and *collateral evolution*. Coccinelle has been born out of a study of collateral evolutions in the Linux kernel [Padioleau et al., 2006c] where changes to core systems need to propagate correctly not only to the thousands of drivers in the Linux kernel source code tree, but also to all the proprietary drivers. Propagating such changes is an error-prone process where most of the know-how is left in the hands of the kernel maintainer. To date this has mostly been done manually, leaving many subtle bugs in driver code for many subsequent versions of the Linux kernel [Padioleau et al., 2006c].

Coccinelle consists of three parts. The most visible part of Coccinelle is the domain-specific language SmPL (Semantic Patch Language) that allows one to express evolutions using a syntax that is familiar to Linux kernel developers—SmPL programs, or rather semantic patches, are subsequently compiled to a formula expressed in computational tree logic with existentially quantified program variables, CTL-VW [Padioleau et al., 2006a, Brunel et al., 2008]. As part of SmPL there is also an isomorphism mechanism that allows the user to express what C constructs should be considered equivalent, e.g. `x == NULL` is equivalent to `NULL == x`. The second, and also very important part of Coccinelle, is the custom C parser that parses C programs without expanding preprocessor macros—this is done in an effort to keep the familiarity of the *diff* and *patch* workflow for kernel developers so that evolutions can also be performed on preprocessor macros. When the C source code is parsed, the C parser generates both a modifiable abstract syntax tree that the transformations are performed on, and a control flow graph.[1] Finally, the last part is the behind-the-scenes model checker that matches the generated CTL-VW formula against the control flow graph. Based on the matches the model checker finds, the transformations are applied to the abstract syntax tree, which is then unparsed to create the transformed source code. All this is illustrated in Figure 1.1, which is adapted from Padioleau et al. [2006a, Figure 4].

Apart from using Coccinelle as an aid in describing evolutionary changes, its code matching capabilities can also be used for finding bugs [Stuart et al., 2007, Lawall et al., 2008]. In this section we will describe the features of SmPL, focusing on the features needed to find bugs. The rest of the section is structured as follows: §1.1.1 will describe the code transformation features, §1.1.2 will illustrate the different patterns for matching code, §1.1.3 will explain the isomorphism features, §1.1.4 will explain the different ways to alter the way that CTL-VW code is generated, and §1.1.5 will describe how to chain together multiple rules to perform more complex matches.

1.1.1 Transforming code using SmPL

To understand how semantic patches work, we must first understand what a regular patch is. If we look at the source code example in Listing 1.1 and we want to replace all uses of `f` with uses of `g` then we must do this manually. Once we have finished this process, we may generate a *diff* file that shows the differences between the original state and the new state. The diff file is frequently called a patch due to the program commonly used to apply diff files to existing source code. An example diff file that changes uses of `f` to uses of `g` in Listing 1.1 can be seen in Listing 1.2. Line 1 indicates the original source file and line 2 the revised source file. Lines 5 and 10 indicate that the use of `f` is to be removed, and lines 6 and 11 indicate to add a use of `g`. Using the *patch* utility to update a system can be error-prone as it hinges on the diligence of the programmer making the changes to identify all places that a change should be made. It has been shown that for larger systems in particular the programmer may frequently miss such places [Padioleau et al., 2006c].

[1]The control flow graph will be described in more detail in §3.3.

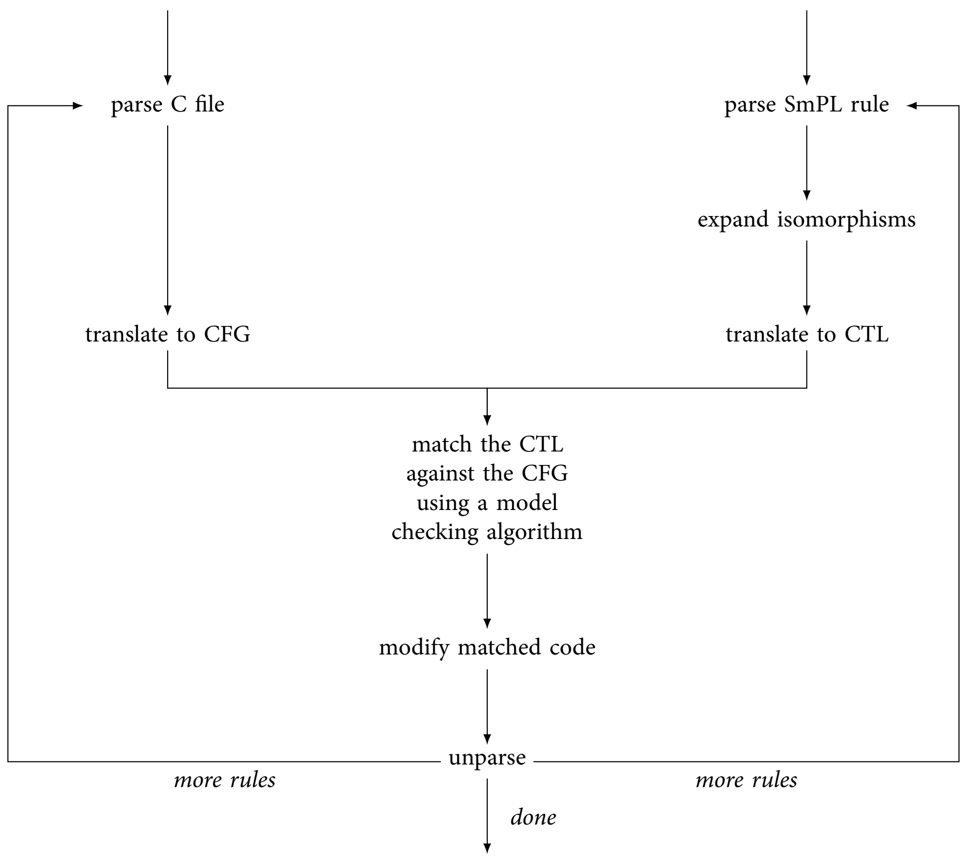

Figure 1.1: The workings of Coccinelle

```
void foo() {
  f();
}

void bar() {
  f();
}
```

Listing 1.1: C functions calling `f()`

```
--- a/foo.c 2008-08-05 09:15:44.000000000 +0200
+++ b/foo.c 2008-08-05 09:16:09.000000000 +0200
@@ -1,7 +1,7 @@
 void foo() {
- f();
+ g();
 }

 void bar() {
- f();
+ g();
 }
```

Listing 1.2: Diff file for replacing uses of `f` with uses of `g` in Listing 1.1

At the very basic level semantic patches work almost like regular patches, as illustrated in Listing 1.3, where all calls to `f` is replaced with calls to `g`. The difference to the regular *patch* utility is that the semantic patch can replace the function call in all files regardless of its location, whereas the regular *patch* utility only would be able to replace `f` with `g` in a specific file and in a specific context. This alone gives Coccinelle a benefit over the program *patch*.

However, semantic patches affords us a great deal more control in what we match. This is done using meta-variables that allows us to abstract several things of the abstract syntax tree including types, expressions, statements, and identifiers. As shown in Listing 1.4 we can state that no matter what argument `f` is called with, it should be replaced with `g` with the same argument. Since a function argument is an expression [ISO/IEC 9899:1990, ISO/IEC 9899:1999], we use an expression meta-variable `E`. This allows us to easily replace both `f(usb->buffer)` and `f(data)` with corresponding calls to `g`—something that would have required specific, manual replacements by a developer at every location where `f` is used, if he was using *patch* instead.

SmPL also allows us to create semantic patches with more complex patterns. Consider e.g. Listing 1.5 where we replace `f` with `g` inside all `while` loops when we are in a then-branch of an `if`, and replace `h` with `g` in the else-branch. This illustrates the case where special-purpose functions `f` and `h` are replaced with a more general function `g`. The '`...`' construct is used to say that zero or more control flow graph nodes may occur between two constructs, or that the contents are not important for the patch like the conditional expression for both the `while` and `if`.

We can also create semantic patches that allow us to update parts of an expression as illustrated in Listing 1.6. This replaces any expression on the form `x + y` with `2 + y`. While being nonsensical, we can use this in general to add new parameters to functions, replace single arguments in function calls or restructure conditionals where one part of the conditional must be removed.

The last type of meta-variable we will briefly discuss is the `position` meta-variable that will be most useful when reporting bugs. Other bound meta-variables do not

```
@@ @@
- f();
+ g();
```

Listing 1.3: Simple SmPL patch

```
@@ expression E; @@
- f(E);
+ g(E);
```

Listing 1.4: SmPL patch using expression meta-variable

```
@@ expression E; @@
 while (...) {
  if (...) {
    ...
-   f(E);
+   g(E);
    ...
  } else {
    ...
-   h(E);
+   g(E);
  }
 }
```

Listing 1.5: Contextual SmPL patch

```
@@ expression E1, E2; @@
- E1
+ 2
 + E2
```

Listing 1.6: Replacing a single function argument using SmPL

contain information about the positions in the source code where they occur, so the concept of a positional meta-variable was created instead. These meta-variables can be attached to any SmPL token, but we will only need to attach them to expression meta-variables. An example of this is shown in Listing 1.7 (note that in C the function name is an expression) where we match a free to an expression `E` and attach position `p1` to it, and a subsequent use of `E` where we attach position `p2`.

Regardless of the semantic patch, Coccinelle is insensitive to any whitespace and comments interspersing the constructs being matched.

1.1.2 Patterns for matching code

The semantic patches we have seen so far have stayed fairly close to the patch origins of SmPL. SmPL does, however, contain a number of other ways to match code that may be useful when we are searching for bugs. We have already seen the '`...`' pattern for abstracting away control flow, but SmPL also contains patterns for searching for zero or more occurences of something (Listing 1.8), as well as one or more occurrences (Listing 1.9).

Using the '`...`' pattern requires that what comes before and after it must exist in the control flow graph in order to return a match. By using '`<...`α`...>`' instead, α is not required to exist in the control flow graph for there to be a match, but if α is in the control flow graph all such matches are returned. Finally, using '`<+...`α`...+>`' matches if there is at least one use of α.

Another type of pattern that SmPL supports is the selection pattern where different items can be matched. This is illustrated in Listing 1.10. This pattern matches the declaration of an identifier I that is assigned by `malloc` later in the function, and later again it has either been assigned a new value, or has been indexed with some value `E2`. This pattern may, for example, form the basis of a patch for finding buffer overflows.

Lastly, SmPL supports to constrain matches on the different forms of '`...`' patterns using the `when` construct as illustrated in Listing 1.11 where we indicate that there should be no match if `I` is assigned with an arbitrary expression between the `malloc` and use.

Coccinelle supports several other patterns for expressing abstractions over paths that we will not cover here as we do not need them for finding bugs in this thesis [Padioleau et al., 2006b, 2007].

1.1.3 Isomorphisms

Isomorphisms in Coccinelle are user-programmable rules that specify equivalences between different constructs in the C programming language that are automatically expanded when Coccinelle matches semantic patterns to source code. This ensures that a user does not need to enumerate all possible ways to express a pattern in every semantic patch he writes, as they can be placed in a file containing all the relevant isomorphisms.

By default, Coccinelle contains a number of useful isomorphisms located in the `standard.iso` file. One such isomorphism is shown in Listing 1.12. The conditionals

```
@@
expression E; position p1, p2;
@@
free@p1(E);
...
E@p2
```

Listing 1.7: Using positional meta-variables in a semantic patch

```
@@
type T; expression E1, E2; identifier I;
@@
T I[E1];
<... I[E2] ...>
```

Listing 1.8: SmPL construct for matching zero or more matches

```
@@
type T; expression E1, E2; identifier I;
@@
T I[E1];
<+... I[E2] ...+>
```

Listing 1.9: SmPL construct for matching one or more matches

```
@@
type T; expression E, E2; identifier I;
@@
 T* I;
 ...
 I = (T)malloc(E);
 ...
(
 I = E2
|
 I[E2]
)
```

Listing 1.10: SmPL construct for selecting different matches

```
@@
type T; expression E, E2, E3; identifier I;
T* I;
...
I = (T)malloc(E);
... when != I = E3
I[E2]
```

Listing 1.11: SmPL construct for constraining path abstraction matches

are as one would expect, so given an expression `X` comparing `X` to zero is equivalent whether it is on the right or left-hand side, and it is the same as testing the negation of `X`. The equivalence to `!X` is not biconditional since if `X` is bound to a pointer variable, `NULL` is not the same as `0` [ISO/IEC 9899:1990],[2] unlike C++ where `NULL` is defined as `const int NULL = 0;` [ISO/IEC 14882:1998].

In Listing 1.13 we define a special isomorphism rule that enumerates some possible ways to redefine a variable, regardless of whether the equivalences make sense semantically. An isomorphism rule that enumerates all possible ways to redefine a variable will be used in Chapter 3.

1.1.4 Tweaking the matching

In CTL-VW, formulas can be existentially quantified (the formula must be true on one path) and universally quantified (the formula must be true on all paths), however the translation from SmPL to CTL-VW currently only supports that a formula is existential or universal.[3] Consider for example the function in Listing 1.14. If we were to match universally for the pattern `f(); ... g();` then it would fail since `g()` is not called on all paths from where `f()` appears. Instead, Coccinelle tries to reason about these *error paths* and tries to quantify universally, *except* on the error paths, thus matching the two calls in the function, even though a path exists where `g()` is not called.

This works very well for the semantic patches for source code evolution, but for finding bugs it does not really matter whether the fault is in or outside the error path, we just care whether a path exists with a bug on it. For this situation, Coccinelle provides the option `exists` that can be placed as shown in Listing 1.15. Here the `rulename` is merely a name for the rule—if it is absent Coccinelle interprets `exists` as the rule's name and not an option. In other situations one might want to ensure that something holds on all paths, including error paths. This can be done using the `when` syntax, but rather than using the '`!=`' syntax from Listing 1.11, one may use '`when strict`' and the tokens that come before and after the dots must be there on all paths.

The last rule option that we will describe here is the `using` option that allows one to add isomorphism rules like the ones shown in §1.1.3. The `using` option takes a filename as an argument as shown in Listing 1.16.

[2] `NULL` is defined as '`#define NULL ((void*)0)`'.
[3] Work is under way to remove this limitation.

```
Expression
@ is_zero @
expression X;
@@
X == 0 <=> 0 == X => !X
```

Listing 1.12: SmPL example isomorphism rule

```
Expression
@ redef @
expression E1, E2;
@@
E1 = E2 <=> E1 += E2 <=> E1 -= E2 <=> E1 *= E2
```

Listing 1.13: Example isomorphism for matching variable redefinitions

```
int foo(void) {
  int x = f();

  if (!x) {
    printf(stderr, "Failed when invoking f()\n");
    return x;
  }

  g();

  return x;
}
```

Listing 1.14: C function with an error path

```
@ rulename exists @
type T; identifier I; expression E1, E2;
@@
T I[E1];
<+... I[E2] ...+>
```

Listing 1.15: Using existential quantification in a SmPL patch

```
@ rulename using "redef.iso" @
type T; identifier I; expression E1, E2;
@@
 T* I;
 ...
 I = malloc(E1);
 ...
(
 I = E2
|
 I[E2]
)
```

Listing 1.16: Adding isomorphism rules to a SmPL rule

1.1.5 Chaining rules

Up until this point we have only seen small, isolated rules, but there are often situations where it is useful to first match one thing and then dependent on the first match, match something else. This can, for instance, be useful if you are describing a collateral evolution where the naming of a function is up to the given driver, but the change should only be made in that one specific function and not in general. The chained rule in Listing 1.17 is taken from Padioleau et al. [2007, p. 5]. As can be seen, rules need to be named in order for later rules to use things from earlier ones. In `rule1` we search for something of type `struct SHT` whose field `proc_info` is assigned `proc_info_func`, a function pointer. In `rule2` the meta-variable `proc_info_func` is inherited from `rule1` using the notation `identifier rule1.proc_info_func`, this indicates that the bound value from `rule1` is to be used in `rule2`, and `rule2` then proceeds to alter the function parameter list to have a pointer passed to it instead of the `hostno`.

If nothing is bound to a meta-variable in an earlier rule that the current rule uses, the current rule is simply never run. We can use this functionality to discard uninteresting things when searching for bugs by only assigning positions to interesting tokens and then use the position variables in a later rule that reports whether the code matched might be a bug.

SmPL supports other options to indicate further dependencies between rules that we do not need for finding bugs. Some of these can be seen in the work by Padioleau et al. [2007].

1.2 Program analysis

Using programs to evaluate properties of code is employed in many places in Computer Science. When optimising a program in a compiler, we may restructure the flow of a program by moving computations outside a loop if their values are not dependent on the loop (code hoisting), or by letting the compiler evaluate all the constants in the

```
@ rule1 @
struct SHT ops;
identifier proc_info_func;
@@
 ops.proc_info = proc_info_func;

@ rule2 @
identifier rule1.proc_info_func;
identifier buffer, start, offset, inout, hostno;
identifier hostptr;
@@
 proc_info_func (
+ struct Scsi_Host *hostptr,
    char *buffer, char **start, off_t offset,
- int hostno,
    int inout) { ... }
```

Listing 1.17: Collateral evolution to proc_info_func

program as close to their use as possible (constant propagation). A partial evaluator will even generate different special-purpose functions depending on the values that are known at the time of specialisation [Jones et al., 1993]. Finally, analysis tools for finding software faults will need to reason about the values in the program to determine whether a computation might lead to an unwanted situation, e.g. a buffer overflow [Verbrugge et al., 1996, Xie et al., 2003].

Bug finders are usually divided into one of two categories: the static tools that parse and analyse the code without running it, and the dynamic tools that run the program and observe what happens as the program executes. Finally, a third group, the hybrid tools, has emerged that both employ static and dynamic solutions. Compilers typically only employ simple static analyses in order to keep the time of compilation short, however some compilers also support dynamic analyses in order to find parts of a program that requires more thorough optimisation—this feature is typically known as profiler guided optimisations. Bug finders, on the other hand, use all three categories, e.g. xg++ uses static analyses to find primarily structural bugs [Engler et al., 2000], Valgrind uses dynamic analyses to find memory issues [Seward and Nethercote, 2005], and CCured uses both static analyses to infer safe use of pointers and dynamic analyses to evaluate the safety of the pointers it could not deem safe statically [Necula et al., 2005].

We will briefly describe the concepts of static analysis in §1.2.1 and dynamic analysis in §1.2.2. The static analysis concepts will be needed as part of our extensions in Chapter 3 and the dynamic analysis concepts as part of our comparisons with other tools in Chapter 5. We will also briefly touch on hybrid analysis in §1.2.3.

1.2.1 Static analysis

There are several kinds of static analyses including Hoare invariants, type and effect systems, constraint based analysis and abstract interpretation [Cousot and Cousot, 1977, Nielson et al., 1999, Huth and Ryan, 2004]. For this thesis we have chosen to only employ abstract interpretation as it is simple and adequate for showing the bug finding capabilities of Coccinelle. We will therefore only discuss abstract interpretation in this section. Note that Coccinelle already employs model checking using CTL-VW, so the abstract interpretation will only be necessary for reasoning about the flow of data.

Abstract interpretation does almost what the name suggests, it executes an abstract model of the program, though with some caveats, since the evaluation of a program may e.g. lead to infinite loops and undefined behaviour. Thus, abstract interpretation is typically set to only run a finite number of loop iterations and to warn about undefined behaviour. In effect, an abstract model of (a part of) the program is run repeatedly until the values it infers are consistent between two runs. More stringently, the result of an abstract interpretation of a data flow function f is the least fixed point of f, defined on a lattice [Cousot and Cousot, 1977, Nielson et al., 1999].

The abstract interpretation can be divided into two categories: the interprocedural analysis that analyses the whole program at once, or the intraprocedural analysis that analyses each function separately. Since Coccinelle is often run only on fragments of the whole program (e.g. only a single subsystem of the Linux kernel may be analysed at one time rather than the entire Linux kernel with all possible modules), and a single Coccinelle rule is matched intraprocedurally, we will also adopt intraprocedural analyses for reasoning about the data flow of programs.[4]

Intraprocedural and interprocedural analyses can be subdivided into the different categories described below.

Flow sensitivity is used to determine whether the data flow information is evaluated based on the control flow. If we take Listing 1.18 on the next page as an example, computing the possible values of `i` would give $i \in [0;30)$ in a flow-insensitive algorithm, but the values $i \in [0;20)$ in the control flow graph node corresponding to line 5 and $i \in [0;30)$ in the control flow graph node corresponding to line 8 in a flow-sensitive algorithm. Thus, a flow-sensitive algorithm will provide more accurate information for some problems.

To illustrate the usefulness of added precision in a data flow analysis, we can try to use the flow-sensitive and flow-insensitive analyses just described to see if the program in Listing 1.18 contains a bug. Using the flow-insensitive information, `buffer1` would be indexed with $i \in [0;30)$, which is clearly a bug since `buffer1` only has allocated space for 20 elements. However, in practice there is no such error, as can also be seen with the improved information for the flow-sensitive algorithm.

[4]There are no technical limitations for doing whole program analysis with Coccinelle if it is run on the entire program, but we have deemed the added precision of an interprocedural analysis unnecessary to illustrate the usefulness of data flow analyses for using Coccinelle to find bugs.

```
void foo(int buffer1[20], int buffer2[30]) {
 int i;

 for (i = 0; i < 20; ++i)
  buffer1[i] = bar(i, 1);

 for (i = 0; i < 30; ++i)
  buffer2[i] = bar(i, 2);
}

int bar(int i, int run) {
 if (run == 1)
  return i;
 else
  return 100 - i;
}
```

Listing 1.18: Simple C program

When considering flow-sensitive algorithms, we furthermore discern between path sensitivity and path insensitivity. An analysis is path-sensitive if it uses information from branch nodes (e.g. `i < 20`) to constrain the possible values according to the branch node for the *true* branch and *false* branch respectively.

Context sensitivity is used in interprocedural analyses to disambiguate function call-sites. In the context of Listing 1.18, a context-insensitive algorithm for estimating program values would merge the possible values that `bar` could return so that the possible values of `i` would be $[0; 100]$. However, in a context-sensitive algorithm, the values of `i` would be $[0; 20)$ at the first call-site, and $(70; 100]$ at the second call-site. We will not implement interprocedural analyses in this thesis, but their future use might help find bugs that were not otherwise found, as well as help remove false positives (reported faults that are not actual faults).

As an example of static analysis using abstract interpretation, we will present constant propagation as an intraprocedural, flow-sensitive, path-insensitive analysis. Informally, the problem can be stated as: if at some control flow graph node there is a use of variable x and that it is only reached from nodes where x is constant then we can rewrite x to be this constant, e.g. removing the need for allocating a register to the variable. More formally, constant propagation of integers is defined on the lattice $(\mathcal{L}, \sqcup, \sqcap)$ shown in Figure 1.2, that is $\mathbb{Z}$ extended with $\top$ and $\bot$ elements ($\mathcal{L} = \mathbb{Z} \cup \{\top, \bot\}$), where $\bot$ signifies that we do not have any information about a variable (all variables are set to $\bot$ initially) and $\top$ signifies that a variable is not a constant. We define $\sqcup$ as follows.

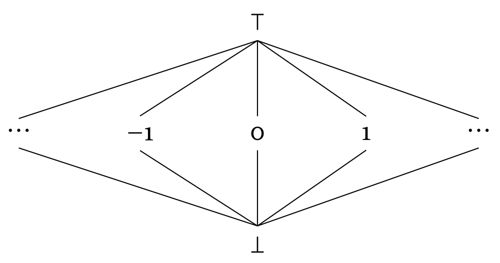

Figure 1.2: Constant propagation lattice

$$c_1 \sqcup c_2 = \begin{cases} \top & \text{if } c_1 = \top \vee c_2 = \top \vee c_1, c_2 \in \mathbb{Z}.c_1 \neq c_2 \\ \bot & \text{if } c_1 = \bot \wedge c_2 = \bot \\ c_1 & \text{if } c_2 = \bot \vee c_1, c_2 \in \mathbb{Z}.c_1 = c_2 \\ c_2 & \text{if } c_1 = \bot \vee c_1, c_2 \in \mathbb{Z}.c_1 = c_2 \end{cases}$$

We can now iterate a solution such that the input of each control flow graph node is the join ($\sqcup$) of each variable from all its predecessors, and the output is the input without the variables that are assigned.[5] After a finite number of iterations we will know what variables are constant at each control flow graph node, including the value of the constant. The number of iterations is finite since a loop will only have to be executed three times: once for the initial run. The second run will push the lattice value to $\top$ if a variable is not a constant, or maintain the constant c, the final run is just to verify that nothing has changed. We will generalise constant propagation in Chapter 3 in order to track the possible values a variable may assume during the execution of a function.

1.2.2 Dynamic analysis

As opposed to static analysis, dynamic analysis runs the program and tracks its state along its execution path in order to find bugs or evaluate invariants. The dynamic analysis programs we will consider track a program's state by instrumenting the binary program code with extra checks [Seward and Nethercote, 2005]. Unlike a static analysis, a dynamic analysis only deals with a single execution path, so the program being analysed needs a thorough test library that covers most parts of the program in order to ensure an accurate analysis. Furthermore, care should be taken when the dynamic analysis tool rewrites program code not to change timings drastically. Changes to timings may mask possible race conditions, such that they are never discovered by the analysis tool, but occur when the program is deployed.

Several dynamic analysis tools are widely used to check programs for bugs today, including the commercial IBM Rational Purify, and the open source tools, Valgrind

[5]This can be defined as a transfer function, see Nielson et al. [1999].

[Seward and Nethercote, 2005, Nethercote and Seward, 2007a] and ElectricFence.[6] Current information on the exact workings of IBM Rational Purify are almost non-existent. Valgrind instruments the binary code of a program and inserts checks to track every memory access and every value computed in order to be able to report on erroneous uses of memory. ElectricFence uses the virtual memory hardware to create an interrupt zone around any buffer such that if the outside of a buffer is touched, a debug interrupt is immediately triggered.

Another approach, which does not seem to have widespread use in the popular tools, is to insert `assert` checks several places in the code, e.g. to catch out of bounds memory accesses. Looking at the example program in Listing 1.19, we should, to be safe, check every use of `buffer` to ensure we do not write beyond the allocated space. We can use a transformation program like Coccinelle to insert the `assert` checks as can be seen in Listing 1.20 on the following page. This, of course, causes a problem of what to do with return values from a function, as extending the return type to also include the size, e.g. in a struct, would change the interface of the function and require changes at all uses of the function. If it is a library that we are instrumenting, all programs using this library will have to be transformed in the same way (conversely, all the libraries would have to be transformed as well if a program is instrumented and it passes a buffer to a library as a function argument), requiring a sizable time investment.

The primary problem of dynamic analysis is illustrated in Listing 1.21 where the function `foo` is never evaluated, thus never triggering the bug (provided `size` is supposed to be the length of `buffer`). Another problem with dynamic analyses is that they typically slow down the execution drastically, e.g. Valgrind executes the program between 10 and 50 times more slowly than running it natively. A dynamic analysis will typically never touch all possible execution paths and can thus easily miss bugs in the program. Finally, unlike static analysis, a dynamic analysis always requires the entire program in order to check it.

1.2.3 Hybrid analysis

The slow execution of dynamic analysis tools sparked an interest in removing as many of the dynamic checks as possible, by using static analyses to deem some of the memory accesses safe. This is the basis of the hybrid analysis tools, which try to draw on the best of both worlds.

One of the best known tools that performs hybrid analysis is CCured that uses static analysis to infer that pointer accesses are safe, and add runtime code to check pointer accesses that may be unsafe [Necula et al., 2005]. Other systems that use a hybrid analysis are some of the Ada compilers, since Ada requires that each array check is verified to be within the bounds of the array [ISO/IEC 8652:2007(E), §4.1.1]. The Ada compilers may then use static analysis to remove as many runtime checks as possible and maintain the dynamic checks for the remaining locations [Møller, 1994, Bernstein

[6] Sales-information on IBM Rational Purify can be obtained from `http://www.ibm.com/software/awdtools/purify/`, Valgrind is available from `http://www.valgrind.org`, and ElectricFence can be obtained from `http://perens.com/works/software/ElectricFence/`.

```c
int* create_buffer(int size, int init) {
  int i;

  int* buffer = malloc(size * sizeof(int));

  for (i = 0; i <= size; ++i)
    buffer[i] = init;

  return buffer;
}
```

Listing 1.19: Sample buffer allocation function

```c
int* create_buffer(int size, int init) {
  int i;

  int buffer_size = size * sizeof(int);
  int* buffer = malloc(buffer_size);

  for (i = 0; i <= size; ++i) {
    assert(i >= 0 && i < buffer_size);
    buffer[i] = init;
  }

  return buffer;
}
```

Listing 1.20: Sample buffer allocation function, checked

and Duff, 1999]. Likewise, Java also requires a bounds check for each array access to ensure safe execution of a program [Gosling et al., 2005] and proposals have been made to extend the HotSpot™ Java Virtual Machine (JVM) just-in-time (JIT) compiler with an analysis to remove some of these checks [Würthinger et al., 2007].

We can use Coccinelle as a hybrid analysis tool by using the matching engine to find bugs and the transformation engine to add code checks in the places that cannot be determined safe or faulty, as illustrated by Stuart et al. [2007]. However, to show that Coccinelle can be used as a bug finding tool, we will only use it as a static analysis tool in this thesis.

1.3 Outline of the thesis

In Chapter 2 we construct a taxonomy for the bugs that we search for; this taxonomy will include information on what patterns to search for, and how to remove false positives from the matches. In Chapter 3 we describe the theory and implementation of the static analyses required for filtering the false positives from Chapter 2. In Chapter 4

```
int foo(int* buffer, int size) {
  int i;
  for (i = 0; i <= size; ++i)
    buffer[i] = i;
}

int bar(int* buffer) { return buffer[0]; }

int main() {
  int i;
  int *buffer = malloc(10 * sizeof(int));

  for (i = 0; i < 10; ++i)
    buffer[i] = 10 - i;

  printf("%d\n", bar(buffer));

  return 0;
}
```

Listing 1.21: Illustration of the shortcomings of dynamic analysis

we use the developed bug finding patterns and analyses to try to find bugs in Open Source software code-bases and evalute its usefulness. Chapter 5 will look at our success rates of finding bugs and match it against other available bug finding tools. Finally, Chapter 6 will conclude on our efforts and remark on what future initiatives will help improve Coccinelle for finding bugs.

Chapter 2

Bug taxonomy

While there is a fairly ubiquitous understanding of 'a bug' in Computer Science and programmer circles alike, the understanding of the underlying flaw of a bug might differ slightly. If we furthermore try to ascertain whether using memory after it has been freed is a memory issue or a resource issue, then the answer will be highly dependent on the point of view of the individual programmer. Even if we have many different categories to place bugs into, we may have no guarantee that we cannot categorise the individual bugs radically differently. A systematic categorisation is also known as a taxonomy. In this chapter we will consider a taxonomy of software faults that will allow us to approach finding bugs in software by describing software fault patterns systematically.

In order for a taxonomy to categorise the same bug in the same way repeatedly it must have a number of properties. Lough [2001] and Hansman and Hunt [2005] provide some of the best summaries of what a good taxonomy should be, based on many of the existing works on taxonomies. We provide the main points of the summary by Hansman and Hunt [2005]:[1]

Accepted The taxonomy should be structed so that it can become generally approved.

Comprehensible It should be understood by people in the security field.

Completeness It should account for all possible flaws and provide categories accordingly.

Determinism Classification should be clearly defined.

Mutually exclusive Each attack should belong to at most one category.

Repeatable It must be possible to repeat the same classification more than once.

Useful It can be used both in the security industry and for research.

Despite the fact that the above list is by many considered, with minor variations, to be a list of good properties for a taxonomy, several of the taxonomies proposed in the literature do not adhere to all the points above. In particular, many do not categorise a bug uniquely in one category.

[1] It is our belief that some of their points overlap, e.g. 'terminology complying with established security terminology' could be comfortably grouped under their 'comprehensible' point. For such overlaps we will omit the point without further remarks.

In this chapter we will first look briefly at some of the existing taxonomies for software faults, then we will describe our rationale for extending an existing taxonomy rather than constructing our own, and finally describe some concrete taxonomy elements.

2.1 Previous work

The work on software fault taxonomies is largely divided into three different categories: the ones that are based on the type of attack [Lindqvist and Jonsson, 1997, Weber, 1998, Lippmann et al., 2000, Weaver et al., 2003, Hansman and Hunt, 2005, CAPEC], the ones that are based on how to defend against an attack [Killourhy et al., 2004], and the ones that are based on the underlying vulnerability [Bisbey and Hollingworth, 1978, Landwehr et al., 1994, Aslam, 1995, Bishop, 1995, Aslam et al., 1996, Krsul, 1998, Martin et al., 2006, Tsipenyuk et al., 2006, CWE]. Since our concern is to find bugs, we will primarily look at the vulnerability-based taxonomies as they focus on describing and classifying the actual bug and not like the attack taxonomies how to attack bugs.

Some of the earliest work on a vulnerability taxonomy was made by Bisbey and Hollingworth [1978]. They were trying to understand operating system vulnerabilities in an effort to propose automatic measures for identifying them, what is today known as Intrusion Detection Systems (IDS). Interesting to this work is not so much their taxonomy that several later papers have pointed out is inadequate [Aslam, 1995, Weber, 1998], but their approach on finding errors, which they tried to solve using patterns that expressed properties that were to occur in order for a vulnerability to be exploited in the system. An example of a generalised pattern that detects race conditions where an attack may modify a variable between its check and use is shown in Listing 2.1. They did not, however, succeed in applying their pattern matching approach widely as the computing power of the time was insufficient.

Landwehr et al. [1994] take a slightly different approach by categorising how a vulnerability entered the system (inadvertendly or maliciously), when it entered the system (in the design, development, maintenance or execution phase), and by location (hardware or software). The principal goal in their research is to be able to locate when the bugs enter the system in an effort to understand which part of the development process should receive further attention in order to eliminate the bugs. Several people including Aslam [1995] and Lindqvist and Jonsson [1997] indicate that this taxonomy is virtually impossible to use if you do not have access to the source code of the program that the vulnerability occurs in, as well as detailed knowledge of the software's progress through the development cycle.

In his thesis Aslam creates a taxonomy for faults in the UNIX operating system and uses it to categorise fault reports from the Computer Emergency Response Team [CERT] into a database for use in an IDS [Aslam, 1995, Aslam et al., 1996]. Krsul [1998] later argues that Aslam's work is merely a categorisation and not a taxonomy since it does not adequately generalise and discuss the predictive properties of the classification (Krsul addresses the shortcomings in his dissertation). It has been used in part as a

```
B:M(X) and for some operation L occurring before M,
  [for operation L which does not modify Value(X),
   Value(X) before L NOT = Value(X) before M], and
  Value(X) after L NOT = Value(X) before M.
```

Listing 2.1: Generalised pattern from Bisbey and Hollingworth [1978]

basis for the Common Weakness Enumeration [CWE] taxonomy as well as Krsul's own taxonomy.

Like many of the other articles we have looked at, Bishop [1995] also develops a taxonomy to be used for IDS. The work builds upon the original categories of Bisbey and Hollingworth [1978] and Landwehr et al. [1994]. It is source code-oriented and does thus not escape the issues raised by Aslam [1995] and Lindqvist and Jonsson [1997] that it is not easy to use for programs for which the source code is not available. Apart from being used for intrusion detection, a concern of Bishop [1995] largely mirrors the motivation of Landwehr et al. [1994] in giving developers advice on considering abstract interfaces to code modules in an effort to avoid known errors.

In his dissertation, Krsul [1998] builds on the work by Aslam [1995] in order to construct a taxonomy of software faults. He creates a taxonomy that adheres to all the properties listed in the beginning of this chapter. This is done using a number of decision trees to construct a unique and unambiguous way to classify software faults.

Finally, Tsipenyuk et al. [2006] created a fairly exhaustive taxonomy for software faults to be used with their commercial analysis tool that seeks to be able to encompass many different bugs. It is, as they state, divided into 'seven different categories (plus one for environment settings)', among others input validation issues and API abuse, which contains buffer overflows and weak string operations like `strcpy` respectively. This taxonomy has later been used, among others, as a basis for CWE.

In 2004–2006 several people started working more actively towards a unified taxonomy of software faults in an effort to provide a common vocabulary and reference [Polepeddi, 2004, Hansman and Hunt, 2005, Martin et al., 2006]. Polepeddi [2004] created a consolidated vulnerability database that collected faults from many different sources (e.g. BugTraq and Secunia). The success rate of including faults from each source is heavily dependent on the source's adoption of the Common Vulnerability and Exposures [CVE] identification number that Polepeddi uses as his database's primary key. This work showed that it was possible to get a sizable number of existing bug reports consolidated with his taxonomy. Around the same time, Martin et al. [2006] also propose using CVE as a basis for a Common Weakness Enumeration taxonomy [CWE]. However, unlike Polepeddi's endeavour, this effort is backed by a number of security researchers, a large part of the security industry as well as several US government agencies, providing it a greater leverage toward common adoption. CWE has been constructed using a large number of existing taxonomies, including those of Aslam [1995] and Tsipenyuk et al. [2006]. Since its first introduction, CWE has been greatly extended and seen a number of updates and is now actively being used by CVE for cross-referencing vulnerabilities [Martin and Barnum, 2008] and it is furthermore set to release in a first stable version in August 2008. This is likely to be the most promising work on a common vocabulary for software vulnerabilities to date.

2.2 Extending the Common Weakness Enumeration taxonomy

As part of the progress of CWE, Martin and Barnum [2008] have discovered that merely presenting source code examples is often not adequate to allow the people using the taxonomy to understand the exact vulnerability. To remedy this they have added information to CWE that indicate the lines that are involved in a specific vulnerability [Martin and Barnum, 2008]. However, like many before us, we believe that a better approach than to only give examples is to use a general pattern to describe the underlying fault [Bisbey and Hollingworth, 1978, Alexander et al., 2002, Hovemeyer and Pugh, 2004]. While it is most likely impossible to describe a fault using a general pattern, it should hopefully be possible to describe a fault using a general pattern for a specific programming language.

Using SmPL we will extend a few CWE elements with a general pattern to describe that fault. Each CWE element we consider will be structured as follows: the CWE ID and URL, a description of the issue in our own words, an example of a fault, a pattern description in SmPL matching the general structure of the code, and finally one or more refinements discussing false positives and false negatives. We will strive to use real-world faults in an effort to underline the necessity of bug finding tools, unlike CWE that just gives made-up examples.[2] To the extent that the bug relies on more than the structural properties of the program, the full SmPL pattern will be given in Chapter 3.

For ease of reference, we present a tree-view of where the bugs we look at fit into the CWE taxonomy in Figure 2.1, and for each of the leaf elements we refer to a section and page number in this chapter where it is extended.

2.2.1 Stack-based buffer overflow

CWE: 121 — `http://cwe.mitre.org/data/definitions/121.html`

Description: A stack-based buffer overflow occurs when a buffer on the stack has data written past its bounds. This may often lead to either crashes, or in targeted attacks, arbitrary code execution. An example of a simple buffer overflow can be seen in Listing 2.2.

General pattern: When constructing an array, it must have a constant size when placed on the program stack [ISO/IEC 9899:1990]. However, the constant size may be

[2]CWE does refer to real-world cases in the CVE, though, but CVE does usually not have associated source code fragments.

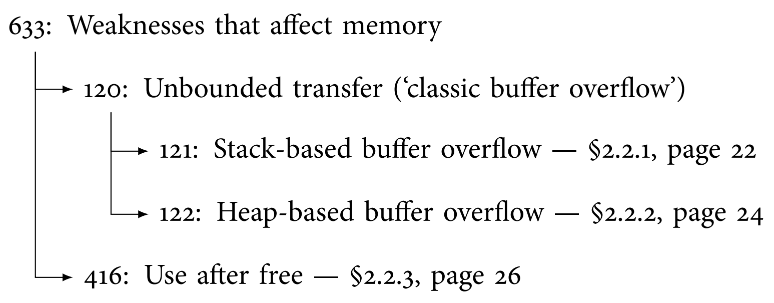
633: Weaknesses that affect memory
120: Unbounded transfer ('classic buffer overflow')
121: Stack-based buffer overflow — §2.2.1, page 22
122: Heap-based buffer overflow — §2.2.2, page 24
416: Use after free — §2.2.3, page 26

Figure 2.1: Taxonomy element structure

```
int buffer[size];
int i;

for (i = 0; i <= size; ++i)
 buffer[i] = i;
```

Listing 2.2: Example of stack-based buffer overflow

a computation based on other program constants as seen in Listing 2.3. Using the GCC or ISO/IEC 9899:1999 variable length array extension instead, the requirement that the array be constant sized is removed and variable length arrays can be placed on the stack.[3] Since the constant sized arrays are a special case of the variable length arrays, we merely consider variable length arrays in the following. Matching declarations and uses of variable lengths arrays can be expressed in Coccinelle as shown in Listing 2.4. It is furthermore possible to create multi-dimensional arrays in C, but for the sake of clarity, we will not make an effort to match them here.

Of the array index possibilities in Listing 2.4, perhaps the only one that is a bit esoteric is the last one, which is, in practice, a rather seldom used way to access arrays. This leaves the pattern `*I` which in ISO/IEC 9899:1990 would always be successful, but may fail using GCC's array of length zero extension or the flexible arrays in ISO/IEC 9899:1999.[4] SmPL currently does not support the last three array uses so we will have to omit matching them for the remainder of the thesis.

The SmPL patch matches all cases where we have a buffer definition followed by at least one use, so this will generate a large amount of false positives.

Refinements: There are no further structural refinements to be made to this pattern as the existence or absence of a bug hinges merely on whether the value of `E2` is greater than or equal to the corresponding value of `E1`. This pattern will be further refined in §3.5.1 on page 43.

[3]This GCC extension is described in detail here: `http://gcc.gnu.org/onlinedocs/gcc-4.3.0/gcc/Variable-Length.html`.

[4]`http://gcc.gnu.org/onlinedocs/gcc-4.3.0/gcc/Zero-Length.html`

```
const int x = 20;
int buffer[x + 2];
```

Listing 2.3: Example array construction in ISO/IEC 9899:1990

```
@@ type T; identifier I, fld; expression E1, E2; @@
 T I[E1];
 <+...
(
 I[E2]
|
 *(I + E2)
|
 (I + E2)->fld
|
 (I.fld)[E2]
|
 E2[I]
)
 ...+>
```

Listing 2.4: Stack-based buffer definition and usage match

2.2.2 Heap-based buffer overflow

CWE: 122 — `http://cwe.mitre.org/data/definitions/122.html`

Description: Heap-based allocation in the C Programming language typically occurs with the `malloc` function, but the possible buffer overflows that arise from using this function are symptomatic of all functions that return a buffer, like `calloc` in the C standard library, `kmalloc` in the Linux kernel, `g_malloc` in GLib from the Gnome project, and many other places. While a buffer returned from an allocation function might be placed on the stack as well, it is more common for it to be heap-based. We will match the use of memory returned by a function call together with the heap-based buffer overflow as the search pattern is the same. An example of a heap-based buffer overflow is shown in Listing 2.5.

General pattern: While `malloc` and `calloc` are essentially the same with regards to the characteristics of the buffer overflow, the matching of the allocation call is not, because we need to retrieve the buffer size as well, and for `malloc` this is dependent only on its single argument, however for `calloc` it is a multiple of its two arguments. This makes it hard to write a single SmPL pattern to match allocation-function based buffer allocations and uses. While not necessarily elegant, we can group all like-mannered functions into the same pattern as we can see in Listing 2.6—we will need equivalent SmPL patterns for other allocation functions with different arguments.

```c
int* buffer;
int i;

buffer = (int*)malloc(sizeof(int) * size);
if (!buffer)
  abort();

for (i = 0; i <= size; ++i)
  buffer[i] = i;
```

Listing 2.5: Example of allocation-function based buffer overflow

```
@ r exists @
identifier I;
expression E1, E2, E3, E4;
type T;
@@
(
  I = (T)malloc(E1)
|
  I = (T)kmalloc(E1)
)
  <+... when != I = E4
(
  I[E2]
|
  *(I + E2)
|
  *I
)
  ...+>
? I = E3
```

Listing 2.6: Allocation-function based buffer allocation and usage match

Apart from this, the pattern in §2.2.1 on page 22 (stack-based buffer overflows) is almost equivalent to the pattern presented in Listing 2.6, including the fact that we will match a lot more than necessary and thus have a lot of false positives in non-bug cases that we need to filter away.

Refinements: There are no further structural refinements to be made to this pattern as the existence or absence of a bug hinges merely on whether the value of `E2` is greater than or equal to the corresponding value of `E1`. This pattern will be further refined in §3.5.2 on page 45.

2.2.3 Use after free

CWE: 416 — `http://cwe.mitre.org/data/definitions/416.html`

Description: Using memory after free, e.g. freeing it twice, may lead to subtle bugs that do not manifest themselves until a much later point in the program execution. As an example, Listing 2.7 shows that `camera->sem` is accessed in line 21 after `camera` is freed in line 14, provided that `camera->buf` is `NULL` in line 12.

General pattern: The pattern for use after free can be generally expressed as seen in Listing 2.8. A use after free can happen with any function that deallocates memory, including `free` from the C standard library, `kfree` from the Linux kernel, etc. The SmPL patch must enumerate each function to be matched. The interesting matches are the ones where both `p1` and `p2` are bound. We will see in Chapter 3 how to use this information.

Refinements: Even with the guard against redefinitions, we will still generate numerous false positives in several code-bases, since it might only be a subexpression of `E` that is redefined between the free and the use. This can e.g. be seen in the source code for mplayer in Listing 2.9.[5]

With the redefinition check we risk getting false negatives instead, as illustrated by the admittedly contrived code fragment in Listing 2.10.

Given the huge number of possible ways to construct expressions that contain any number of subexpressions, it becomes prohibitively expensive to manually enumerate all of these. Instead we will consider an extension in Chapter 3 that can handle this for us.

[5]mplayer is an open source media player available at `http://www.mplayerhq.hu`. The source code is taken from svn revision 27095, `stream/tvi_dshow.c` lines 2991–2993.

```
static void camera_disconnect(struct usb_device *dev, void *ptr)
{
      struct camera_state *camera = (struct camera_state *) ptr;
      int   subminor = camera->subminor;

      down (&state_table_mutex);
      down (&camera->sem);

      /* If camera's not opened, we can clean up right away.
       * Else apps see a disconnect on next I/O; the release cleans.
       */
      if (!camera->buf) {
            minor_data [subminor] = NULL;
            kfree (camera);
      } else
            camera->dev = NULL;

      info ("USB Camera #%d disconnected", subminor);
      usb_dec_dev_use (dev);

      up (&camera->sem);
      up (&state_table_mutex);
}
```

Listing 2.7: Use-after-free bug in linux-2.4.1/drivers/usb/dc2xx.c

```
@ bug exists @ expression E, E2; position p1, p2; @@
(
 kfree@p1(E)
|
 free@p1(E)
)
 ...
(
 E = E2
|
 E@p2
)
```

Listing 2.8: Use after free match

```
for (i = 0; chain->arStreamCaps[i]; i++) {
   free(chain->arStreamCaps[i]);
}
```

Listing 2.9: False positive for use after free match

```
free(foo);
foo = foo;
free(foo);
```

Listing 2.10: False negative for double free match

Chapter 3

Extending Coccinelle

Coccinelle is at its core a source-to-source transformation tool that takes a semantic patch and one or more source code files as input and generates transformed files and a diff that describes the changes from the original to the processed files. This is very useful as long as you wish to transform code, but statically analysing programs in an effort to find bugs only requires half of this: the source code matching based on the semantic patch.

SmPL requires us to repeat code, and recompute a match if we just want to constrain our match to a part of what was written in the SmPL patch. Thus, we would like to create a facility for more easily processing and reporting found matches. Furthermore, Coccinelle makes no provisions for using data flow information, so we would like to implement a very simple data flow analysis as a proof of concept that can handle some of the false positives that we discussed in §2.2.1 and §2.2.2.

In order to provide a general-purpose processing and reporting facility that falls in line with Coccinelle's pursuit of being easy to understand for developers working with the C Programming language, and the Linux kernel in particular, a solution would be to integrate a scripting language. When you consider a scripting language that is familiar to Linux kernel developers, only two come to mind: Perl and Python, both of which are used already for various processing tools around the kernel. Bindings for integrating either language with OCaml exist, but we have opted for integrating Python with Coccinelle. This allows us to provide all the facilities easily: processing, reporting, and testing data flow analyses.

In this chapter we will first describe the Python extension for Coccinelle, then we will discuss the theory of generalised constant propagation and our implementation of it, which we will use to find buffer overflows, then we will look at how to mitigate the number of false positives for the *use-after-free* bug (see §2.2.3), and finally we will complete the taxonomy elements from Chapter 2 using our extensions.

3.1 Scripting Coccinelle

Using Coccinelle for finding bugs requires a way to report possible bug sites since without code transformation, Coccinelle does not generate any output. Furthermore, a way to prototype new features without having to make substantial changes to the parsing, interpretation and matching code in Coccinelle for the Semantic Patch Language would allow us to more easily experiment with data flow analyses and other ways of filtering matches. In particular filtering matches was considered in our preliminary work on Coccinelle [Stuart et al., 2007], but the other requirements can also be addressed by integrating a scripting language. This means a one time change of SmPL to allow scripting language rules on line with the existing SmPL rules (henceforth called Coccinelle rules to disambiguate from scripting rules).

We will first describe the integration of Python into Coccinelle and then describe how this can be used for reporting bugs, filtering matches and representing Coccinelle's meta-variables for use in the scripting rules.

To keep future possibilities open, we will allow the integration of any scripting language into Coccinelle using the same SmPL extension. It will be the integrator's responsibility to bridge features from Coccinelle into the scripting language. We will integrate Python with Coccinelle's OCaml code using the Open Source project *pycaml*.[1]

The overall structure of a scripting rule is illustrated in Listing 3.1. The *scripting-language-identifier* is the name of the scripting language, e.g. python. The *meta-variable-inheritance-list* is a list with zero or more bindings of meta-variables from previous rules. These bindings are on the form *local-name* << *rule-name.meta-variable-name;*, where *local-name* is the name of a valid identifier in the scripting language, and *rule-name.meta-variable-name* is an inherited meta-variable like in Coccinelle rules. Finally, *scripting-language-source-code* is a program in the scripting language that may use some of the functions provided by the scripting language integration (see §3.4). Listing 3.2 shows a short SmPL patch that prints all identifiers in a program. This could, for instance, be used as the basis of a design verification tool for checking naming conventions in a project.

Python `print` statements are sufficient to easily report matches, however, we will create a more elaborate mechanism for reporting errors that allows the user a greater deal of autonomy, including logging to a file, printing to the monitor, and presenting results in a graphical user interface. In an effort to provide users with these features and to provide a more Python-esque programming environment—wrapping the functional aspects of Coccinelle into an object-oriented interface—we will construct a small class library that can be used in the scripting rules called *coccilib*.

For reporting and filtering we create the base class `Output` shown in Listing 3.3. Here `include_match` provides filtering capabilities as the method is overridden by Coccinelle to indicate whether a given match should be saved for further processing. By placing this functionality in a function, the choice of whether to keep a match can be entirely up to the logic in a scripting rule, providing a very solid filtering functionality. The `register_match` is a uniform way to report aspects about the matched code. User-supplied Python code for writing matches must override `register_match` with their logic (e.g. storing matches in a local database). The `combine` function is a convenience function that can attach inherited position meta-variables to inherited non-position meta-variables to give a more unified way to print messages about meta-variables. Finally, the `finalise` function allows the output code to execute some code prior to Coccinelle finishing. This is very useful when implementing a graphical user interface that should not close before the user exits the program.

[1]Pycaml is available from `http://pycaml.sourceforge.net`.

```
@ script:scripting-language-identifier @
meta-variable-inheritance-list
@@
scripting-language-source-code
```

Listing 3.1: SmPL scripting rule structure

```
@ idfind @ identifier I; @@
I
@ script:python @ x << idfind.I; @@
print 'IDENTIFIER:', x
```

Listing 3.2: SmPL scripting rule example for reporting a program's identifiers

```
class Output:
      def include_match(self, b):
            pass

      def register_match(self, include, messages):
            self.include_match(include)

      def combine(self, meta_variable, locations):
            nmv = deepcopy(meta_variable)
            nloc = [deepcopy(loc) for loc in locations]
            nmv.location = nloc[0]
            nmv.locations = nloc

            return nmv

      def finalise(self):
            pass
```

Listing 3.3: Output class definition

We can now create a SmPL patch that filters some things away. An instance of the Output class is available as `cocci` in the scripting rules. As we see in Listing 3.4, we have the full expressive power of Python and we can even define functions inside the scripting rule for automating tasks. Furthermore, we see the use of the Python code to filter away matches we are certain of are bugs. By adding another rule that transform the remaining matches to add bounds checking code, we can also use Coccinelle as a hybrid analysis tool.

Selecting the output class can be done using the `-pyoutput` option for Coccinelle. The actual output class can be anything that inherits from the `Output` class above, either some of the built-in classes from coccilib or classes from user-defined Python code that inherits from coccilib's `Output` class.

```
@ bug exists @
type T; identifier I; expression E1, E2, E3; position p1, p2;
@@
T I[E1@p1];
<+... I[E2@p2] = E3; ...+>

@ script:python @
array_size << bug.E1; array_index << bug.E2;
p1 << bug.p1; p2 << bug.p2;
@@

def is_int(s):
  try:
    int(str(s))
    return True
  except:
    return False

if is_int(array_size) and is_int(array_index) and
  int(str(array_index)) >= int(str(array_size)):
  cocci.include_match(False)

  cocci.register_match(False,
    [(p1[0], 'Definition of array'),
    (p2[0], 'Buffer overflow')])
else:
  cocci.include_match(True)
```

Listing 3.4: Example SmPL filtering code using Python

The work described in this section has been used by Lawall et al. [2008] to find numerous bugs in the Linux kernel.

3.1.1 Representing Coccinelle meta-variables

Given the taxonomy elements that we described in Chapter 2 that we wish to match using Coccinelle, we only really need to represent expressions and their positions in the scripting rules. Rather than reconstruct an entire abstract syntax tree in Python, we merely represent the expression meta-variables using the string representations of their values and 'attach' the meta-variable to the Python object so it can pass this back to OCaml code later on for further processing—we will use this to retrieve all subexpressions of an expression later in this chapter. The representation of expression meta-variables is shown in Listing 3.5.

The values of the position meta-variables are rather simple, namely just a filename and a list of start and end lines and columns, so we do not need to carry around the OCaml object for these as we can represent them entirely in Python. For ease of

```python
class Expression:
      def __init__(self, expr, repr):
            self.expr = expr
            self.repr = repr

      def __str__(self):
            return self.expr
```

Listing 3.5: Python class for representing expression meta-variables

using the positions in the Python code, we include the filename with each location unlike the OCaml code that only has a single filename and a number of positions. The representation for position meta-variables is shown in Listing 3.6.

3.2 Data flow analysis

While Coccinelle contains thorough features for analysing the control flow of a program, it entirely lacks a mechanism for analysing the propagation of values in the control flow graph. Being able to reason about data in a program is a necessity if we wish to be able to find buffer overflows, as we have to both track the possible size of an array and the possible values that the array is indexed with.

Before we opted to implement our data flow analysis directly in Coccinelle, we considered a number of existing tools to provide data flow analysis information, among others the GNU Compiler Collection (gcc), CIL, and clang.[2] However, by using these tools we would have had to figure out compilation flags for the source code (which are not needed by Coccinelle), the source would need to be compiled with the respective tool, and its output decoded into a format useful for Coccinelle. All this only for a relatively small gain: not having to write a few choice data flow analyses (for clang we would have had to develop as many analyses as only the general control flow graph traversal was in place when we started our work). In general it would have been nice had there been a common library for data flow analyses, but we have found nothing mature that was publically available.

Due to the scope of this thesis, we will only integrate a single data flow analysis into Coccinelle that allows us to reason about variable values. There are a number of other data flow analyses that could be interesting to implement, in particular in an effort to increase precision in the analysis we describe below, such as points-to analysis to resolve what variables pointers point to [Ghiya and Hendren, 1998, Ghiya, 1998].

[2]The GNU Compiler Collection is available from `http://gcc.gnu.org`, CIL from `http://manju.cs.berkeley.edu/cil`, and clang from `http://clang.llvm.org`.

```
class Location:
    def __init__(self, file, line, column, line_end, column_end):
        self.file = file
        self.line = line
        self.column = column
        self.line_end = line_end
        self.column_end = column_end
```

Listing 3.6: Python class for representing position meta-variables

3.2.1 Generalised constant propagation

Estimating program variable values has primarily been used in the context of optimising compilers where e.g. constant propagation [Kildall, 1973] is often used to simplify a program so it may more easily fit into the available machine registers, and to prune dead code by removing branches in the code that will never be taken. However, as computers have gotten more powerful and an increasing need for (automatic) parallelisation and better branch prediction has arisen, people have investigated and extended constant propagation to compute ranges of possible values for variables rather than just constants. These analyses are called anything from generalised constant propagation to value range propagation or symbolic range propagation [Harrison, 1977, Blume and Eigenmann, 1996, Patterson, 1995, Verbrugge et al., 1996, Bae and Eigenmann, 2006]. Several people have discovered, though, that generalised constant propagation is useful for more than merely parallelising programs: it can also be used to locate bugs [Cousot and Cousot, 1977, Verbrugge et al., 1996, Xie et al., 2003].

The fundamental idea in generalised constant propagation is to assign a range $[a; b]$ to each program variable. This information can then be used to determine whether a branch will always be taken, or whether an array is indexed beyond its range. Some of the more advanced algorithms (the symbolic propagation algorithms) [Blume and Eigenmann, 1996, Xie et al., 2003, Bae and Eigenmann, 2006] also track interdependent ranges such as $x : [0; 20] \wedge y \geq x$ and propagate these through the program. While this may increase precision in some places, we do not consider it necessary to illustrate the usefulness of Coccinelle as a means to find bugs.

The work that is most relevant for integrating into Coccinelle is the work of Verbrugge et al. [1996], which implements generalised constant propagation on a simplified abstract syntax tree where `goto` statements have been eliminated [Erosa and Hendren, 1994] and preprocessor macros have been expanded. Thus, the only places they need to iterate a fixpoint solution are in C's loop constructs as opposed to programs with `goto` statements intact that have to be iterated for all nodes in the control flow graph. It is also worth noting that their iterative solution is merely an adaptation of *reaching definitions* [Appel and Ginsburg, 1998, Chapter 17].

Verbrugge et al. [1996] describe three algorithms for generalised constant propagation, each with increased precision: intraprocedural analysis that makes worst-case assumptions about function calls by setting all variables that have had their address taken in a function and all global variables to $[-\infty;\infty]$,[3] intraprocedural analysis with read/write sets that describe what global variables are assigned so only these need to be set to $[-\infty;\infty]$, and finally interprocedural analysis with read/write sets that is context-sensitive in an effort to discard as little information as possible about ranges in a function.

Now that we have briefly covered some of the background for generalised constant propagation, we will look at our implementation of it for Coccinelle. We will create a flow- and path-sensitive data flow analysis based on the intraprocedural algorithm without read/write sets described by Verbrugge et al. [1996]. We will, however, implement it on the full control flow graph representing a C program. While using read/write sets or an interprocedural analysis would give us more accurate results, we will settle with implementing the intraprocedural analysis as a proof of concept for using Coccinelle to find data flow bugs.

Since we still have `goto` statements in our control flow graph, we must use the adapted reaching definitions algorithm on our entire graph—this may cause the recomputation of elements that are not strictly necessary, but optimising this for performance is beyond the scope of this thesis.[4]

Solving a flow equation on a control flow graph iteratively may mean that the result never converges to a fixpoint due to loops, so to avoid looping infinitely, a step-up solution is employed (also called widening/narrowing by Cousot and Cousot [1977]) that takes non-converging elements and steps them up to $\pm\infty$, thus forcing convergence at the loss of precision. Verbrugge et al. [1996] employs several step-ups if the number of iterations exceeds some value n. We will settle with using two step-ups, one moving the non-converging part(s) of the range to $\pm\infty$, e.g. if `i` is bound to $[0;10]$ in node n on iteration k and to $[0;11]$ in node n on iteration $k+1$ then we step it up to $[0;\infty]$, and the second to step-up the range to $[-\infty;\infty]$.

There are two places where information is generated for estimating variable value ranges: in assignments, and in conditionals. Assignments naturally generate information in that after the assignment '`i = 0`' `i` will be bound to $[0;0]$. To see that conditionals generate information consider the conditional $i \geq 5$ where $i:[0;10]$. In the *true*-branch i will be bound as $i:[5;10]$ and in the *false*-branch as $i:[0;4]$.

We let $\top = [-\infty;\infty]$ and $\bot$ indicate that we do not have any information about a variable's range yet. We define $x \sqcup y$ as follows.

$$x \sqcup y = \begin{cases} \top & \text{if } x = \top \vee y = \top \\ x & \text{if } y = \bot \\ y & \text{if } x = \bot \\ [\min(a,c); \max(b,d)] & \text{where } x = [a;b] \text{ and } y = [c;d] \end{cases}$$

[3]Strictly speaking then the $[-\infty;\infty]$ notation may be at odds with mathematical notation where ∞ cannot be inclusive in the range, however we will use ∞ as the maximum value for the underlying type, since the C Programming language's simple integer and floating point types are all finite.

[4]If we did not want to retain the source code as close to the original when we perform static analyses we could employ the same `goto` elimination as described by Erosa and Hendren [1994], or the simplifications employed by CIL [Necula et al., 2002].

Representing program variables using intervals requires that we define mappings from the language's operators to intervals. We formulate our data flow analysis using the flow equation (3.1). This flow equation uses the function *constrain* that is inductively defined over the possible C programming language constructs using a number of likewise inductively defined auxiliary functions (3.2–3.4). The definition of *constrain* is shown in (3.5). Also, *gen* and *kill* are used according to their definitions by Appel and Ginsburg [1998]. For the sake of brevity, we have only shown a couple of the inductive cases—the remaining cases are constructed similarly. Do note that some operations will split the ranges in several, distinct ranges, e.g. `n < 0 || n > 10`, but we only ever use one range for any variable, so we will lose precision here as we have to include the values between 0 and 10 as well to represent the range of `n`.

$$\begin{aligned} in[n] &= \bigsqcup_{p \in pred[n]} \mathrm{constrain}(n, p, out[p]) \\ out[n] &= gen[n] \sqcup (in[n] - kill[n]) \end{aligned} \tag{3.1}$$

$$[a;b] \oplus [c;d] = \begin{cases} [a+b; c+d] & \mathit{if} \oplus \equiv + \\ [a; \min(b,d) - 1] & \mathit{if} \oplus \equiv < \wedge \min(b,d) = d \\ [a; \min(b,d)] & \mathit{if} \oplus \equiv < \wedge \min(b,d) \neq d \\ [\max(a,c); b] & \mathit{if} \oplus \equiv \geq \\ \ldots \end{cases} \tag{3.2}$$

$$\begin{array}{lll} \multicolumn{3}{l}{\mathrm{range}(e, out[p]) =} \\ e \equiv \mathtt{x} & \mapsto [a;b] & \mathit{if}\,(\mathtt{x}, [a;b]) \in out[p] \\ & \mapsto [-\infty; \infty] & \mathit{if}\,(\mathtt{x}, [a;b]) \notin out[p] \\ e \equiv \mathtt{e}_1 \mathtt{>=} \mathtt{e}_2 & \mapsto [a;b] \geq [c;d] & \mathit{if}\,[a;b] = \mathrm{range}(\mathtt{e}_1, out[p]) \wedge \\ & & \quad [c;d] = \mathrm{range}(\mathtt{e}_2, out[p]) \\ \ldots & \mapsto \ldots & \end{array} \tag{3.3}$$

$$\begin{array}{lll} \multicolumn{3}{l}{\mathrm{constrain}'(e, n, p, out[p]) =} \\ e \equiv \mathtt{x} & \mapsto [a;b] \neq [0;0] & \mathit{if}\, p \to n = \mathit{true} \wedge \\ & & \quad [a;b] = \mathrm{range}(\mathtt{x}, out[p]) \\ & \mapsto [a;b] = [0;0] & \mathit{if}\, p \to n = \mathit{false} \wedge \\ & & \quad [a;b] = \mathrm{range}(\mathtt{x}, out[p]) \\ e \equiv \mathtt{e}_1 \mathtt{<} \mathtt{e}_2 & \mapsto [a;b] < [c;d] & \mathit{if}\, p \to n = \mathit{true} \wedge \\ & & \quad [a;b] = \mathrm{range}(\mathtt{e}_1, out[p]) \wedge \\ & & \quad [c;d] = \mathrm{range}(\mathtt{e}_2, out[p]) \\ & \mapsto [a;b] \geq [c;d] & \mathit{if}\, p \to n = \mathit{false} \wedge \\ & & \quad [a;b] = \mathrm{range}(\mathtt{e}_1, out[p]) \wedge \\ & & \quad [c;d] = \mathrm{range}(\mathtt{e}_2, out[p]) \\ \ldots & \mapsto \ldots & \end{array} \tag{3.4}$$

$$
\begin{array}{lll}
\text{constrain}(n, p, out[p]) = & & \\
p : \texttt{e} \quad \mapsto \text{constrain}'(\texttt{e}, n, p, out[p]) & \textit{if } p \to n \in \{\textit{true}, \textit{false}\} & \\
p : \texttt{...} \quad \mapsto out[p] & \textit{in all other cases} &
\end{array}
\tag{3.5}
$$

As an example of using the equations, consider the very simple program fragment '`if (i < 20) a[i] = 20;`' where '`i < 20`' is node 1 in a control flow graph and '`a[i] = 20;`' is node 2 and take $out[1]$ to be $i : [0; \infty]$. Since $1 \to 2 = \textit{true}$ we use constrain to find the value of $out[2]$. As node 2 is the child of a conditional we use constrain′ to find the value of `i`. As we are in the true branch our result will be $[0; \infty] < [20; 20]$ as the range of `i` is $[0; \infty]$ and the range of a constant is the single element, namely $[20; 20]$. Using the equations from (3.2) we resolve this to be $[0; 19]$, arriving at the correct bounds of `i`.

Since our analysis is intraprocedural, we will have to make some pessimistic assumptions about the function arguments to the function being analysed and global variables, namely that they must be $\top$ (this is the least accurate value we have). Function calls will set all variables that have had their address taken to $\top$ as well.[5] This loses a lot of information that could be refined by using read/write sets or an interprocedural analysis.

By only stepping up non-converging ranges twice, we can employ our implementation to locate the loop bounds for us rather than doing it explicitly like Verbrugge et al. [1996]. However, this means that we lose the monotonicity property of the flow equation, as a range may be subsequently constrained by a conditional (e.g. $[0; \infty]$ may be constrained to $[0; 19]$ by the conditional '`i < 20`'). Any such constraint will only occur once for a binding in a node and only after the first widening; all other operations will be monotonically increasing and the algorithm will thus terminate, but run more inefficiently.[6]

As an example, consider the function in Listing 3.7. Its control flow graph is shown in Figure 3.1 on page 39 and the result of the generalised constant propagation is shown in Table 3.1 on page 40. We let the step-up max iteration count be 2 here for illustrative purposes.[7] This means that when `i` is not converging in nodes 4–6 after 2 iterations, we step up the non-converging part of the range to ∞. In the next iteration, `i` is constrained to the size of the condition and in iteration 5 we have found the least fixpoint, which is verified in iteration 6. Had iteration 6 not verified the fixpoint property, we would have stepped up its non-converging ranges again and terminated the algorithm returning these bounds. Also note that we do not get any information in node 7 until iteration 4 since we can statically ascertain that the branch will never be taken given the value range of `i` we have inferred prior to this. Once the analysis finishes, it tells us that

[5] Strictly speaking a macro can change a variable even if the address of the variable has not been taken. We make no provision for handling this case.

[6] In retrospect it would have been better to employ the multiple step-ups as suggested by Verbrugge et al. [1996] and Cousot and Cousot [1977, §9.2] and avoid this issue entirely.

[7] In practice we will set it to a much larger value. Verbrugge et al. [1996] uses a max iteration count of 40.

`buffer` will be accessed with values in the range [0;19] that are all legal indices to `buffer` (Node 5), and once we return, `i` will be 20 (Node 7).

We have implemented the generalised constant propagation algorithm in OCaml and provide a function for the Python scripting rules that returns a range for a given meta-variable, `cocci.gcp(meta_var, position)`. This means that we can now construct a simple example of finding buffer overflows using the generalised constant propagation results. This is shown in Listing 3.8. For the full-fledged semantic patch, there will of course be a requirement of better error reporting, but this should suffice to illustrate how to use it.

3.3 Avoiding false positives in *use-after-free*

With the current features of Coccinelle and the extensions described in this chapter, we have no way to detect whether an expression changes value between two occurrences. In order to find *use-after-free* bugs (see §2.2.3), we can filter away matches where an expression or any of its subexpressions are redefined between the two occurrences as an approximation.

As an example, we can consider the false positive from *mplayer* that we identified in Listing 2.9 on page 27. Using a Coccinelle isomorphism file, *redef.iso*, that makes all types of redefinitions equivalent, the semantic patch in Listing 3.9 should stop the false positive from being matched when both `p1` and `p2` are bound. However, running the semantic patch still generates the false positive. This is caused by the fact that Coccinelle represents the control flow graph at the statement level to be able to perform structural transformations, so the increment code in the for loop is seen as belonging to the for header and is thus not detected on the path from the free to the subsequent use in the for check in the for header. This is illustrated in Figure 3.2.

In order to find bugs, we do not need to preserve the complete statements in the control flow graph, but we can expand them to their expression components, allowing us to match the redefinition on the path from the free to the possible use. We accomplish this using the Python extensions developed earlier in the chapter to create a hook into Coccinelle that can replace the control flow graph by invoking the function `cocci.set_expr_cfg()` from a scripting rule. Using the expression-based control flow graph, the for loop is represented as seen in Figure 3.3 where we explicitly have a node for the increment code (`incr`) that can be matched by Coccinelle.

The second problem we face with matching *use-after-free* bugs is that we need to ensure that not only the matched expression is not redefined from the free to the use, but also that any subexpression of the expression is not redefined. While we can, with some tricks, collect all subexpressions using Coccinelle rules, we will opt to provide a Python function for decomposing an expression into its subexpressions, `cocci.get_subexpressions(expr)`, for clarity.

With these things in place we have enough features to detect *use-after-free* bugs with most false positives filtered away. The full pattern will be explained in §3.5.3, but an example can be seen in Listing 3.14 on page 50.

```
void foo(int init) {
 int buffer[20];

 for (int i = 0; i < 20; ++i) {
  buffer[i] = init;
 }
}
```

Listing 3.7: Simple loop

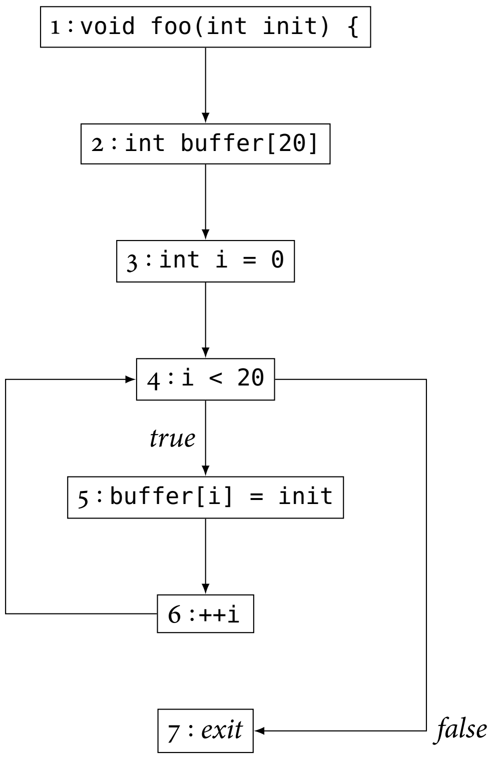

Figure 3.1: Control flow graph for Listing 3.7

	iteration 1		iteration 2		iteration 3	
n	*in*[*n*]	*out*[*n*]	*in*[*n*]	*out*[*n*]	*in*[*n*]	*out*[*n*]
1		init:⊤		init:⊤		init:⊤
2	init:⊤	init:⊤ buffer:⊥	init:⊤	init:⊤ buffer:⊥	init:⊤	init:⊤ buffer:⊥
3	init:⊤ buffer:⊥	init:⊤ buffer:⊥ i:[0;0]	init:⊤ buffer:⊥	init:⊤ buffer:⊥ i:[0;0]	init:⊤ buffer:⊥	init:⊤ buffer:⊥ i:[0;0]
4	init:⊤ buffer:⊥ i:[0;0]	init:⊤ buffer:⊥ i:[0;0]	init:⊤ buffer:⊤ i:[0;1]	init:⊤ buffer:⊤ i:[0;1]	init:⊤ buffer:⊤ i:[0;2]	init:⊤ buffer:⊤ i:[0;∞]
5	init:⊤ buffer:⊥ i:[0;0]	init:⊤ buffer:⊤ i:[0;0]	init:⊤ buffer:⊤ i:[0;1]	init:⊤ buffer:⊤ i:[0;1]	init:⊤ buffer:⊤ i:[0;2]	init:⊤ buffer:⊤ i:[0;∞]
6	init:⊤ buffer:⊤ i:[0;0]	init:⊤ buffer:⊤ i:[1;1]	init:⊤ buffer:⊤ i:[0;1]	init:⊤ buffer:⊤ i:[1;2]	init:⊤ buffer:⊤ i:[0;2]	init:⊤ buffer:⊤ i:[1;∞]
7						

	iteration 4		iteration 5		iteration 6	
n	*in*[*n*]	*out*[*n*]	*in*[*n*]	*out*[*n*]	*in*[*n*]	*out*[*n*]
1		init:⊤		init:⊤		init:⊤
2	init:⊤	init:⊤ buffer:⊥	init:⊤	init:⊤ buffer:⊥	init:⊤	init:⊤ buffer:⊥
3	init:⊤ buffer:⊥	init:⊤ buffer:⊥ i:[0;0]	init:⊤ buffer:⊥	init:⊤ buffer:⊥ i:[0;0]	init:⊤ buffer:⊥	init:⊤ buffer:⊥ i:[0;0]
4	init:⊤ buffer:⊤ i:[0;∞]	init:⊤ buffer:⊤ i:[0;∞]	init:⊤ buffer:⊤ i:[0;20]	init:⊤ buffer:⊤ i:[0;20]	init:⊤ buffer:⊤ i:[0;20]	init:⊤ buffer:⊤ i:[0;20]
5	init:⊤ buffer:⊤ i:[0;19]	init:⊤ buffer:⊤ i:[0;19]	init:⊤ buffer:⊤ i:[0;19]	init:⊤ buffer:⊤ i:[0;19]	init:⊤ buffer:⊤ i:[0;19]	init:⊤ buffer:⊤ i:[0;19]
6	init:⊤ buffer:⊤ i:[0;19]	init:⊤ buffer:⊤ i:[1;20]	init:⊤ buffer:⊤ i:[0;19]	init:⊤ buffer:⊤ i:[1;20]	init:⊤ buffer:⊤ i:[0;19]	init:⊤ buffer:⊤ i:[1;20]
7	init:⊤ buffer:⊤ i:[20;∞]	init:⊤ buffer:⊤ i:[20;∞]	init:⊤ buffer:⊤ i:[20;20]	init:⊤ buffer:⊤ i:[20;20]	init:⊤ buffer:⊤ i:[20;20]	init:⊤ buffer:⊤ i:[20;20]

Table 3.1: Example generalised constant propagation flow for Figure 3.1 with $m = 2$

```
@ bug exists @
type T; identifier I; expression E1, E2;
position p1, p2;
@@

T I[E1@p1];
<+... I[E2@p2] ...+>

@ script:python @
size_pos << bug.p1; size_var << bug.E1;
indx_pos << bug.p2; indx_var << bug.E2;
@@

size = cocci.combine(size_var, size_pos)
indx = cocci.combine(indx_var, indx_pos)

size_r = cocci.gcp(size, size_pos)
indx_r = cocci.gcp(indx, indx_pos)

if size_r.is_bottom() or indx_r.is_bottom():
      print 'Undefined variable in use'
elif size_r.is_top() or indx_r.is_top():
      print 'Possible buffer overflow. Check.'
elif indx_r.max() >= size_r.min():
      print 'Buffer overflow.'
```

Listing 3.8: SmPL patch using generalised constant propagation information

```
@ bug using "../../redef.iso" exists @
expression E, E2; position p1, p2;
@@

 free@p1(chain->arStreamCaps[i]);
 ...
(
 chain->arStreamCaps[i] = E
|
 chain->arStreamCaps = E
|
 chain = E
|
 i = E
|
 chain->arStreamCaps[i]@p2
)
```

Listing 3.9: Trying to avoid matching the mplayer false positive

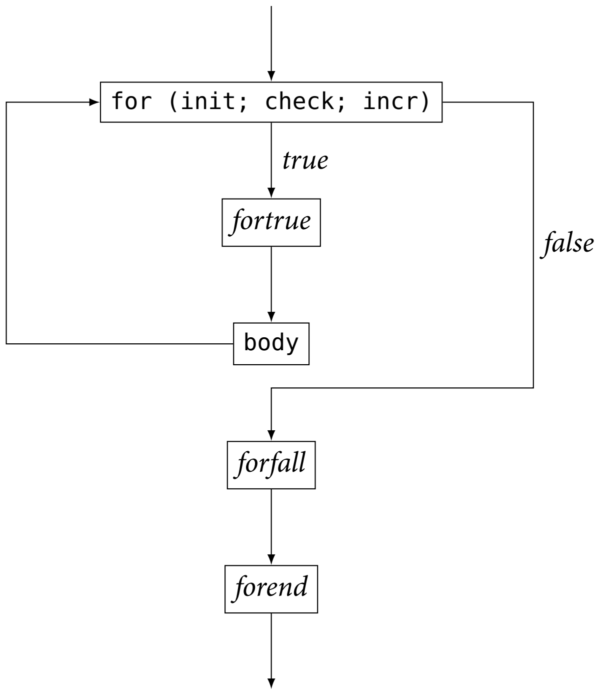

Figure 3.2: Coccinelle's control flow graph for a for loop

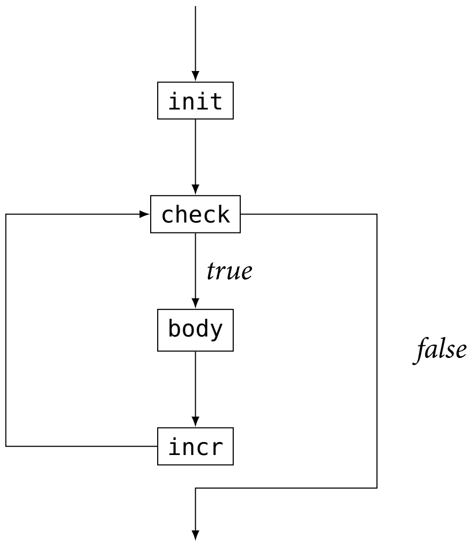

Figure 3.3: Expanded control flow graph for a for loop

3.4 Functions provided for Python by Coccinelle

For completeness' sake, this section contains a listing of all the functions that we have provided for the scripting rules to use to communicate with Coccinelle. All these functions are members of the default `cocci` instance.

`include_match(t)`: The value of `t` indicates whether the currently matched environment is kept for further processing in later rules. The function may be called any number of times during the course of a scripting rule; only the argument of the last call decides whether the environment is kept.

`set_expr_cfg()`: This function changes from the statement-based control flow graph to the expression-based control flow graph described in §3.3. There is currently no converse function, but providing one in the future should be trivial.

`print_cfg(prefix)`: Writes a GraphViz file to the file prefixN.dot where N is an increasing number that is unique for a single run of Coccinelle. The prefixN.dot file is furthermore compiled to prefixN.dot.ps. This function requires that GraphViz be installed on the system.

`get_subexpressions(expr_repr)`: This function retrieves a list of all subexpressions, represented as strings, to a given Coccinelle expression meta-variable (the Coccinelle expression meta-variable is attached to a Python expression as the `.repr` member variable).

`gcp(expr_repr, pos_repr)`: This function computes the range that `expr_pos` may be at the `pos_repr` location in the program. This is described in detail in §3.2.

3.5 Completing the taxonomy elements

In Figure 3.4, we present the overview of taxonomy elements from Chapter 2 with references to sections and pages in this chapter. For each of the taxonomy elements in this chapter we will merely give the details for matching the bugs, i.e. the full semantic patch and other details such as the need for multiple executions of Coccinelle.

3.5.1 Stack-based buffer overflow

Using the generalised constant propagation (see §3.2) on the expression-based control flow graph (see §3.3) we can locate possible stack-based buffer overflows using the SmPL patch in Listing 3.10.

There are some cases where we get false positives from this even when the generalised constant propagation successfully determines a bound, namely the cases where an array can have two or more sizes in one function, and two uses have different upper bounds where one is larger than the minimum of the two possible sizes of the array.[8] We make no provision for catching these.

[8]We have not observed any cases where this is a problem in all the code we have analysed, though.

```
@ bug exists @
type T; identifier I, fld; expression E1, E2;
position p1, p2;
@@

 T I[E1@p1];
 <+...
(
 I[E2@p2]
|
 *(I + E2@p2)
)
 ...+>

@ script:python @
@@
cocci.set_expr_cfg()
cocci.print_cfg()
cocci.include_match(True)

@ script:python @
e1 << bug.E1; e2 << bug.E2;
p1 << bug.p1; p2 << bug.p2;
@@

cocci.print_cfg()

print p1[0].file, p1[0].line, p1[0].column
array_size = cocci.gcp(e1.repr, p1[0].repr)
print ' array_size:', array_size

print p2[0].file, p2[0].line, p2[0].column
array_index = cocci.gcp(e2.repr, p2[0].repr)
print ' array_index:', array_index

if array_size.is_bottom() or array_index.is_bottom():
      cocci.register_match(True, [(p1[0], 'May be used
          uninitialised'), (p2[0], 'May be used uninitialised')
          ])
elif array_size.is_top() or array_index.is_top():
      cocci.register_match(True, [(p1[0], 'Array declaration.
          Size: %s' % array_size), (p2[0], 'Buffer use. May be
          used outside bounds: %s' % array_index)])
elif array_index.max() >= array_size.min():
      cocci.register_match(True, [(p1[0], 'Array declaration,
          size: %s' % array_size), (p2[0], 'Array use. May be
          outside bounds: %s' % array_index)])
```

Listing 3.10: SmPL patch for matching and reporting stack-based buffer overflows

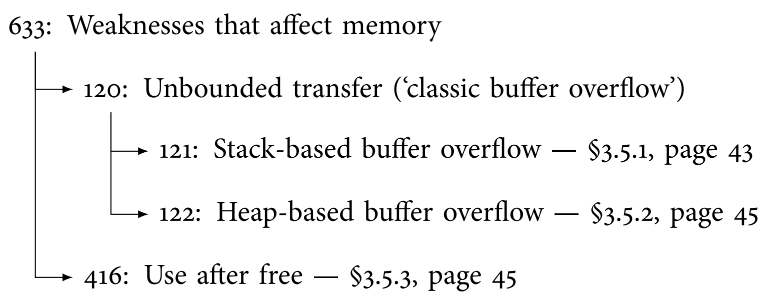
633: Weaknesses that affect memory
120: Unbounded transfer ('classic buffer overflow')
121: Stack-based buffer overflow — §3.5.1, page 43
122: Heap-based buffer overflow — §3.5.2, page 45
416: Use after free — §3.5.3, page 45

Figure 3.4: Taxonomy element structure

3.5.2 Heap-based buffer overflow

The semantic patch for matching heap-based buffer overflows, shown in Listing 3.11, is almost identical to the stack-based one. The only difference is the use of allocation functions rather than a statically defined array.

In order to compute the size of the array we must know the argument to the function that signifies the size of the returned buffer, `E1` in the case of `malloc` and `kmalloc`. The script would need to be adapted to e.g. `calloc` that uses two arguments to compute the size of the returned buffer. We only consider allocation functions where the first and only argument provides the size of the buffer.

3.5.3 Use after free

Matching *use-after-free* bugs will be done in two steps: first we find all places where there is a use after free, regardless of whether there is a redefinition of the freed expression or its subexpressions, and subsequently we test each of these places for whether there is a redefinition.

The first step, shown in Listing 3.12, can be done using the regular statement-based control flow graph, which may be faster since the control flow graph contains fewer nodes as each expression does not occupy a node in the graph.[9] Once a possible use after free location is matched, a new semantic patch is generated that will be used in the second step. This new semantic patch is generated from the template in Listing 3.13 where the different subexpressions are expanded into the '`[REDEF]`' placeholder and the code and location are expanded into the remaining '`[...]`' placeholders to ensure that the semantic patch only matches at the specific location that has been found (otherwise we could generate false positives by matching unrelated potential uses after free). For each match in step one, a line with the relevant file-name and generated semantic patch file-name is written to `bugs/useafterfree.bug` and by running each of these patches, we will find a closer count of the number of *use-after-free* bugs. Running all of these matches can easily be automated with a simple script.

[9] In practice we did not observe any significant difference between matching using the statement-based control flow graph and the expression-based control flow graph.

```
@ bug exists @
type T; identifier I; expression E1, E2, E3;
position p1, p2;
@@
 T* I;
 ...
(
 I = malloc(E1@p1);
|
 I = kmalloc(E1@p2);
)
 <+... when != I = E3
(
 I[E2@p2]
|
 *(I + E2@p2)
)
 ...+>

@ script:python @
e1 << bug.E1; e2 << bug.E2;
p1 << bug.p1; p2 << bug.p2;
@@

cocci.set_expr_cfg()

array_size = cocci.gcp(e1, p1)
array_index = cocci.gcp(e2, p2)

if array_size.is_bottom() or array_index.is_bottom():
      cocci.register_match(True, [(p1[0], 'May be used
          uninitialised'), (p2[0], 'May be used uninitialised')
          ])
elif array_size.is_top() or array_index.is_top():
      cocci.register_match(True, [(p1[0], 'Array declaration.
          Size may be unknown.'), (p2[0], 'Buffer use. May be
          used outside bounds, unable to verify.')])
elif array_index.max() >= array_size.min():
      cocci.register_match(True, [(p1[0], 'Array declaration'),
          (p2[0], 'Array use. May be outside bounds.')])
```

Listing 3.11: SmPL patch for matching and reporting heap-based buffer overflows

```
@ bug exists @
expression E; position p1, p2;
@@

(
 free@p1(E);
|
 kfree@p1(E);
)
...
E@p2

@ script:python @
e << bug.E; p1 << bug.p1; p2 << bug.p2;
@@

from tempfile import mkstemp
from os import write, close

template = open('bugs/useafterfree.templ', 'r')
content = ''.join(template.readlines())
template.close()

subexpr = [e] + cocci.get_subexpressions(e.repr)
redef = [str(x) + " = E" for x in subexpr]

p1 = p1[0] # only use principal location
p2 = p2[0] # ditto

content = content.replace('[EXPR]', str(e))
content = content.replace('[REDEF]', '\n|\n '.join(redef))
content = content.replace('[P1:FILE]', p1.file)
content = content.replace('[P1:LINE]', p1.line)
content = content.replace('[P1:COLUMN]', p1.column)
content = content.replace('[P2:FILE]', p2.file)
content = content.replace('[P2:LINE]', p2.line)
content = content.replace('[P2:COLUMN]', p2.column)

f, p = mkstemp('.cocci', 'uaf', 'bugs/tmp')
write(f, content)
close(f)

scr = open('bugs/useafterfree.bug', 'a')
scr.write('-cocci_file %s %s\n' % (p, p1.file)) # batch file
scr.close()
```

Listing 3.12: Finding all *use-after-free* locations

When expanded with the information from the first step (an example of this is shown in Listing 3.14), each semantic patch using the template shown in Listing 3.13 is structured to use the expression-based control flow graph and to discard the cases where the matched expression or any of its subexpressions are redefined between the free and the use.

This brings the number of false positives down, but there are still several kinds of false positives that remain, e.g. when the expression freed subsequently is the argument of an allocation-function that takes the address of the variable as an argument—this can be seen in `/arch/ia64/sn/kernel/xpc_channel.c` in the Linux-2.6 kernel where the freed expression, `ch->local_msgqueue_base` is later passed to `xpc_kzalloc_cacheline_aligned` as `&ch->local_msgqueue_base`, and its value is set inside the called function. Since we use Coccinelle as an intraprocedural analysis tool, we have no way to detect these cases automatically.

```
@ script:python @ @@
cocci.set_expr_cfg()
cocci.include_match(True)


@ bug using "../../redef.iso" exists @
expression E1, E;
position p1, p2;
@@

(
 free@p1(E1);
|
 kfree@p1(E1);
)
...
(
 [REDEF]
|
 E1@p2
)

@ script:python @
e << bug.E1; p1 << bug.p1; p2 << bug.p2;
@@

p1 = p1[0]
p2 = p2[0]

if str(e) == '[EXPR]' and p1.file == '[P1:FILE]' and
      p1.line == '[P1:LINE]' and p1.column == '[P1:COLUMN]' and
      p2.file == '[P2:FILE]' and p2.line == '[P2:LINE]'
      and p2.column == '[P2:COLUMN]':
      cocci.register_match(True, [(p1, 'Free'), (p2, 'Use')])
```

Listing 3.13: Template for finding faulty *use-after-free* locations

```
@ script:python @ @@
cocci.set_expr_cfg()
cocci.include_match(True)


@ bug using "../../redef.iso" exists @
expression E1, E;
position p1, p2;
@@

(
 free@p1(E1);
|
 kfree@p1(E1);
)
...
(
 pInfo->rx_buf = E
|
 pInfo = E
|
 E1@p2
)

@ script:python @
e << bug.E1; p1 << bug.p1; p2 << bug.p2;
@@

p1 = p1[0]
p2 = p2[0]

if str(e) == 'pInfo->rx_buf' and
      p1.file == 'linux-2.6/drivers/char/n_r3964.c' and
      p1.line == '1059' and p1.column == '1' and
      p2.file == 'linux-2.6/drivers/char/n_r3964.c' and
      p2.line == '1060' and p2.column == '42':
      cocci.register_match(True, [(p1, 'Free'), (p2, 'Use')])
```
Listing 3.14: Expanded example template for matching *use-after-free* bugs

Chapter 4

Results

As mentioned in Chapter 1, Coccinelle has been designed and tested primarily with the Linux kernel, and this is also where we will keep our focus in testing our extensions. However, in an effort to investigate Coccinelle's usefulness on other code-bases as well, we will apply it to two Internet application servers, tbaMUD and Icecast. These programs will be described in §4.3.

In this chapter, we will first investigate a number of constructed program fragments in order to illustrate the strengths and weaknesses of our approach. Subsequently, we will apply the semantic patches developed in Chapter 3 to the Linux 2.6 kernel, tbaMUD and Icecast, and for each project describe the bugs found and explain the false positives and what steps, if any, we can take to remedy them in future work.

4.1 Investigating the results of our extensions

Before we investigate the effectiveness of our results on real-world code, it seems prudent to subject them to some scrutiny to see what we can expect to work and what we cannot expect to work. This will also make it easier to understand why we may fail at finding issues in real-world code.

4.1.1 Buffer overflows

Buffer overflows on the stack are usually the worst as they potentially allow an attacker to overwrite the return address pointer, making it possible for the attacker to redirect the program's control flow to his own code. There are several ways that buffers are used that we cannot match using Coccinelle in the way that we have structured our SmPL patches.

At the very simplest, buffer overflows are typically caused by a programmer making an off-by-one error. An example of this is shown in Listing 4.1, where the comparison in line 5 should be a strict less-than comparison rather than less-than-or-equal, since we will be indexing one past the bounds of the array in line 6 in the last iteration otherwise. The result of running Coccinelle with our extensions is shown in Figure 4.1.

Many programs move the size of the buffers into a global constant to maintain consistency across the codebase, as shown in Listing 4.2. For this purpose we also scan and collect all global constants as part of the generalised constant propagation, as this allows us to more accurately state whether there is a bug or not.

However, this is where the ease of scanning for buffer overflows ends as there are many different places where arrays can be declared in program code, which influences

```c
void f() {
 int buffer[20];
 int i;

 for (i = 0; i <= 20; ++i)
  buffer[i] = i;
}
```

Listing 4.1: Simple stack-based buffer overflow

```
> ./runspatch.opt -cocci_file stackbuffer.cocci results/sbo1.c
results/sbo1.c:2:13: Array declaration, size: [20;20]
 results/sbo1.c:6:11: Array use: [0;20]. May be outside bounds.
```

Figure 4.1: Stack-based buffer overflow for Listing 4.1

```c
#define MAX_SIZE 25

void f() {
 int buffer[MAX_SIZE];
 int i;

 for (i = 0; i <= MAX_SIZE; ++i)
  buffer[i] = i;
}
```

Listing 4.2: Simple stack-based buffer overflow with global constant size

how a semantic patch might match it. Just moving the array outside the function as illustrated in Listing 4.3 causes our semantic patch not to match anything anymore. To find these we could create a two-part semantic match as shown in Listing 4.4, where we first search for all buffer definitions and then for all the uses of this buffer, and finally run our generalised constant propagation algorithm on the matched locations.

ISO/IEC 9899:1999 does, however, also allow one to create arrays of incomplete type (and thus with unknown size), e.g. '`int buf[]`' that can be initialised with an initialiser list and that obtain the size of the largest index value used in the initialiser list [ISO/IEC 9899:1999, §6.7.8]. This is shown in Listing 4.5 where `buf` is defined to be of size 6 and `buf2` is defined to be of size 11 with only three of its indices having been defined. None of our previously presented semantic patches support matching these buffer declarations. SmPL does not support matching values inside the initialiser lists and while we could match incompletely typed arrays, a meta-variable would never be bound to its size and we would have no expression to hand to the generalised constant propagation to evaluate the size of the array. We can, of course, extend our algorithm to be able to compute the size of these arrays, but we will leave that as a future extension.

```
#define MAX_SLOTS 20
static int foo[MAX_SLOTS];

int main() {
        foo[20] = 20;
}
```

Listing 4.3: Buffer overflow in global buffer

```
@ str @ type T; identifier I; expression E1; position p1; @@
T I[E1@p1];

@ bug exists @
identifier str.I; expression E2; position p2;
@@

I[E2@p2]

@ script:python @ @@
cocci.set_expr_cfg()
cocci.include_match(True)

@ script:python @
e1 << str.E1; e2 << bug.E2;
p1 << str.p1; p2 << bug.p2;
@@

import coccilib

array_size = cocci.gcp(e1.repr, p1[0].repr)
array_index = cocci.gcp(e2.repr, p2[0].repr)

if array_size.is_bottom() or array_index.is_bottom():
      cocci.register_match(True, [(p1[0], 'May be used
          uninitialised'), (p2[0], 'May be used uninitialised')
          ])
elif array_size.is_top() or array_index.is_top():
      cocci.register_match(True, [(p1[0], 'Array declaration.
          Size: %s' % array_size), (p2[0], 'Array use: %s. May
          be used outside bounds.' % array_index)])
elif array_index.max() >= array_size.min() or
      array_index.min() < coccilib.range.zero:
      cocci.register_match(True, [(p1[0], 'Array declaration.
          Size: %s' % array_size), (p2[0], 'Array use: %s. May
          be outside bounds.' % array_index)])
```

Listing 4.4: Global buffer semantic match

```
int buf[] = { 0, 1, 2, 3, 4, 5 };

int buf2[] = {
      [0] = 0,
      [10] = 3,
      [5] = 4
};

int main() {
      buf[6] = 6;
      buf2[11] = 12;
}
```

Listing 4.5: Buffer overflow in global array with initialiser

The different places that arrays can be defined in C do, of course, not end here, as arrays can also be declared inside structs and inside unions, and nested in these to arbitrary depths. One of the simplest such definitions is shown in Listing 4.6. We can then create a semantic patch that can match array definitions inside structs and subsequent uses of them, as shown in Listing 4.7. Consider, though, the program in Listing 4.8, where we no longer have a definition of a variable with the struct type it is contained in. We could, of course, create a semantic patch that matches any array definition at any level and then any field use with this identifier, e.g. '`...->I[E2@p2]`', however, we can easily consider two different structs with the same field name and each of these will be matched wrongly, providing a lot more false positives than we would care for.

Even though we might be able to catch a few more actual bugs using the semantic patches in Listing 4.4 and 4.7, the inaccuracy of the generalised constant propagation is already giving us a *huge* number of false positives, so we will limit ourselves to using the semantic patch shown in Chapter 3.

Finally, there are also the heap-based buffer overflows that we have not dealt with up until this point. The way that e.g. `malloc` works is by allocating a contiguous block of bytes to the caller, and thus the caller needs to specify the exact byte count needed at the point of allocation. This means that for anything apart from `char` arrays, `malloc` will be invoked with the `sizeof` expression that determines the size of a given type or expression on the specific platform that is being compiled for. Since we have chosen to only assign a single range to a given variable using our generalised constant propagation algorithm, it becomes impossible to assign anything but $[-\infty;\infty]$ when a `sizeof` expression is part of the term as its size is implementation specific.

As a remedy for this, we have considered filtering out all `sizeof` expressions in allocation functions, but the number of possible permutations for where it may occur is nearly endless (although in practice it will probably be more limited) that making semantic patches for these cases seems unnecessarily complex. The proper solution would be to create a full symbolic propagation algorithm for Coccinelle, but due to

```
struct s {
  int data[20];
};

void bar() {
  struct s x;
  x.data[20] = 22;
}
```

Listing 4.6: Buffer overflow in array defined in a struct

time constraints we will not pursue this solution. We will instead, sadly, omit matching buffer overflows that use values that have been allocated by `malloc`, etc., in this thesis.[1]

4.1.2 Use-after-free

One of the simplest mistakes that triggers a use-after-free bug is to free a structure first and then its members subsequently; this is shown in Listing 4.9 where `x` is freed in line 7 and is subsequently used in line 8. This may deallocate memory that no longer points to the expected value, potentially opening up an avenue of attack for a malicious user. Running our use-after-free SmPL patch on the example produces the output in Figure 4.2.

This example is, however, already matched by Coccinelle without any of our extensions. The places where our extensions make a difference are in the for and while-loops of C programs, as illustrated in Listing 4.10. Using the semantic match with Coccinelle without our expression-based control flow graph results in a bug being reported, while enabling our extensions removes this false positive. This is shown in Figure 4.3.

There are, however, interprocedural cases of use-after-free that we cannot hope to match. This is shown in Listing 4.12. If we call `send_to_all` in an expectation to send our buffer data to everyone who is connected to the server, then if someone has lost connection `send` will fail in `send_to_client`, line 4, and the buffer will be released in line 5 and the function returns to `send_to_all`. However, `send_to_all` fails to check the return value and proceeds to use the buffer if more clients are connected.[2] This means that each interprocedural use-after-free will give us a false negative, i.e. it is a flaw, but we will not be able to report it.[3]

One of the more unfortunate false positives we have is due to people printing the address of something that has just been freed, or using it in a conditional test as seen in Listing 4.11. While the subsequent use of a variable might be risky, it is not a bug

[1]Alternatively we could always consider the result of `sizeof` to be 1, but this will not support the construct '`sizeof(array)/sizeof(array[0])`' that is often used in the kernel.

[2]This is, of course, a bad program design, since the data might only get to the *n* first clients until we meet one that has lost connection, but much worse has probably seen the light of day in production code.

[3]Functions that free one or more arguments on some paths occur frequently in the Linux kernel. A work-around to matching them would be to create specialised semantic patches that enumerate these functions, but we will make no effort to do so in this thesis.

```
@ str @ type T; identifier S, I; expression E1; position p1; @@
struct S {
  ...
  T I[E1@p1];
  ...
};

@ bug exists @
identifier str.S, str.I, v; expression E2; position p2;
@@

struct S v;
...
v.I[E2@p2]

@ script:python @ @@
cocci.set_expr_cfg()
cocci.include_match(True)

@ script:python @
e1 << str.E1; e2 << bug.E2;
p1 << str.p1; p2 << bug.p2;
@@

array_size = cocci.gcp(e1.repr, p1[0].repr)
array_index = cocci.gcp(e2.repr, p2[0].repr)

if array_size.is_bottom() or array_index.is_bottom():
      cocci.register_match(True, [(p1[0], 'May be used
          uninitialised'), (p2[0], 'May be used uninitialised')
          ])
elif array_size.is_top() or array_index.is_top():
      cocci.register_match(True, [(p1[0], 'Array declaration.
          Size: %s' % array_size), (p2[0], 'Buffer use. May be
          used outside bounds: %s' % array_index)])
elif array_index.max() >= array_size.min() or array_index.min()
    < range.zero:
      cocci.register_match(True, [(p1[0], 'Array declaration.
          Size: %s' % array_size), (p2[0], 'Array use. May be
          outside bounds: %s' % array_index)])
```

Listing 4.7: Struct-defined buffer semantic match

```
> ./runspatch.opt -cocci_file uaf.cocci results/uaf1.c
results/uaf1.c:7:4: Free
  results/uaf1.c:8:9: Use
```

Figure 4.2: Use-after-free results for Listing 4.9 and 4.11

```
struct q {
  struct {
    int data[20];
  } x;
};

int main() {
  struct q x;
  x.x.data[20] = 22;
}
```

Listing 4.8: Buffer overflow in array defined in a nested struct

```
struct s {
  int* data;
};

void do_free(struct s* x) {
  if (x) {
    free(x);
    free(x->data);
  }
}
```

Listing 4.9: Simple use-after-free error with structs

```
1  void do_free(struct s* x) {
2    int i;
3
4    for (i = 0; i < x->len; ++i) {
5      free(x->data[i]);
6    }
7  }
```

Listing 4.10: Use-after-free in a loop

```
> ./runspatch.opt -cocci_file uaf-orig.cocci results/uaf3.c
results/uaf3.c:5:4: Free
  results/uaf3.c:5:9: Use

> ./runspatch.opt -cocci_file uaf.cocci results/uaf3.c
```

Figure 4.3: Use-after-free results for Listing 4.10

unless the memory it points to is used. There is no way to avoid this false positive with Coccinelle other than to exhaustively enumerate all cases where a use of the variable is not a use of the memory.

Lastly, we do not remove infeasible paths when we perform a match. This means that e.g. the code in Listing 4.13 will generate a false positive as it will not track that if we free `b` then `res` will never be `0` in line 9, and thus we will never use it in line 10. As Coccinelle does not currently support infeasible path pruning, every case of program construction like this will generate a false positive. We can verify this by running our SmPL patch on the code. This is shown in Figure 4.4.

4.2 Linux 2.6

We will run our semantic patches on the Linux kernel 2.6 at commit ID `baadac8b-10c5ac15ce3d26b68fa266c8889b163f` from the 11th of March 2008 in an effort to find bugs in real-world code.[4] This is a development version for the Linux 2.6.25 kernel and also the same version that was used by Lawall et al. [2008] for their results.

The great benefit of analysing Linux is that it is big and sees a continuous flurry of development activity, which means that if one looks at enough places, one is bound to uncover a bug sooner or later. We will exclusively look at the results and not the time it takes to run the semantic patches.[5]

4.2.1 Buffer overflows

As §4.1.1 indicated, finding buffer overflows by analysing a non-simplified control flow graph is difficult, and while it is not entirely unexpected, some of our implementation choices result in less than stellar results when it comes to finding buffer overflows in the kernel. Our overall results are presented in Table 4.1, where the 8 undecided cases are code fragments that were extremely complicated so we gave up trying to categorise them. The success rate of 0.2% that we have achieved is unequivocally, rather bad.[6] Rather than dwell on this, we will first look at the actual bug we have found and subsequently look at some of the cases where we fail to discard the false positive in an effort to provide means for a future strengthening of the generalised constant propagation algorithm.

The bug we have found is located in `arch/alpha/boot/main.c` and is a classical off-by-one error. The relevant code is shown in Listing 4.14, where '...' is used to signify irrelevant code. Here `callback_getenv` fills up to and including `sizeof(envval)` bytes into `envval` and returns this count. This means that `nbytes` can potentially be `sizeof(envval)` when the null terminator is written, thus overflowing `envval`. The

[4]Available from `http://git.kernel.org/?p=linux/kernel/git/torvalds/linux-2.6.git;a=commit;h=baadac8b10c5ac15ce3d26b68fa266c8889b163f`.

[5]As a point of interest, analysing the entire Linux kernel with our use-after-free semantic patch takes about 6 hours on our old AMD Sempron 1.6 GHz processor with 1 GB RAM.

[6]Do note that the success rate is only concerned with false positives and does not take false negatives into account.

```
void do_free(struct s* x) {
  struct foo* y;

  if (x) {
    free(x);

    for (y = global_list; global_list; ++y)
      if (y->data == x)
        y->data = NULL;
  }
}
```

Listing 4.11: Simple use-after-free error when freeing list member

```
int send_to_client(struct client* c, struct buffer* b) {
  int sent = 0;

  if ((sent = send(c->socket, b->data, b->length, 0)) == -1) {
    free_buffer(b);
    return -1;
  }

  return sent;
}

void send_to_all(struct buffer* b) {
  struct client* c;

  for (c = clients; c; c = c->next)
    send_to_client(c, b);
}
```

Listing 4.12: Interprocedural use-after-free

```
> ./runspatch.opt -cocci_file uaf.cocci results/uaf5.c
results/uaf5.c:6:4: Free
  results/uaf5.c:11:22: Use
```

Figure 4.4: Use-after-free results for Listing 4.13

Bugs found	1
False positives found	496
Undecided	8
Success rate	0.2%

Table 4.1: Success rates for finding buffer overflows in Linux 2.6

```

int send_data(struct client* c, struct buffer* b) {
  int ret = 0;

  if (!c->is_connected) {
    free_buffer(b);
    ret = -1;
  }

  if (ret == 0) {
    send_to_client(c, b);
  }

  return ret;
}
```

Listing 4.13: Infeasible path use-after-free false positive

likeliness of an exploit here is extremely low as `envval` is filled with data that is given as options to the bootloader, e.g. LILO, and while `start_kernel` does not check the size correctly, `callback_getenv` presumably does, so there will only be the possibility of having an extra zero written one past the bound. The code for `callback_getenv` is not part of the kernel but provided by the alpha architecture, but given the other uses of it where the check is '`nbytes < 0 || nbytes >= sizeof(envval)`', we postulate that this use is faulty.

There are, though, a lot more false positives that primarily are the result of simplifications we made in our implementation of the generalised constant propagation algorithm. When copying data from user-space to kernel-space, a typical program fragment is shown in Listing 4.15 where the intervals to the right indicate the value range of `count` on that line. As a function argument, we will pessimistically assign it the range $[-\infty; \infty]$, which results in our first mistake since `size_t` is an unsigned type, so $[0; \infty]$ would have been a more appropriate range. Our second mistake then comes from not representing `count` as a local variable (function parameters are represented at the same level as global variables), causing it to be widened to $[-\infty; \infty]$ again due to the function call in line 12. Had we properly made `count` a local variable then this widening would not have taken place, but without an extension to consider the unsignedness of the variable, we would have had the interval $[-\infty; 39]$ in line 15, and thus would still have had to report a buffer underrun and not a buffer overflow.

Another simplification we made was not to handle enumeration constants that, unfortunately, seem to be used quite frequently in the kernel. An example of this is shown in Listing 4.16 where the size of `PORT_NUM_EVENTS` will not be set and as such it will be set to $[-\infty; \infty]$, and thus `k` will be assigned the interval $[0; \infty]$ in line 17. Since `PORT_NUM_EVENTS` is assigned $[-\infty; \infty]$ the code will be flagged for investigation. We should be able to introduce this computation in the same manner as other global constant values, potentially eliminating a long list of false positives.

```
void start_kernel(void) {
  ...
  int nbytes;
  char envval[256];

  ...

  nbytes = callback_getenv(ENV_BOOTED_OSFLAGS,
    envval, sizeof(envval));
  if (nbytes < 0) {
    nbytes = 0;
  }
  envval[nbytes] = '\0';

  ...
}
```

Listing 4.14: arch/alpha/boot/main.c buffer overflow bug in Linux 2.6

```
static int parse_number(
  const char __user *p,
  size_t count,                                              [−∞; ∞]
  unsigned long *val)
{
  char buf[40];
  char *end;

  if (count > 39)                                            [−∞; ∞]
    return -EINVAL;                                          [40; ∞]

  if (copy_from_user(buf, p, count))                         [−∞; 39]
    return -EFAULT;                                          [−∞; ∞]

  buf[count] = 0;                                            [−∞; ∞]

  ...
}
```

Listing 4.15: False positive when copying from user-space to kernel-space

```
enum port_event {
  ...
  PORT_NUM_EVENTS = 5,
};

...

int sas_register_phys(struct sas_ha_struct *sas_ha) {
  ...

  static const work_func_t sas_port_event_fns[PORT_NUM_EVENTS] =
      { ... };

  ...

  for (k = 0; k < PORT_NUM_EVENTS; k++) {
    INIT_WORK(&phy->port_events[k].work,
        sas_port_event_fns[k]);
    phy->port_events[k].phy = phy;
  }

  ...
}
```

Listing 4.16: False positive when using enumerations

Our decision to not support the bitwise operators in C for intervals (by widening them to $[-\infty; \infty]$ when encountered) is the last issue that causes a long list of false positives. The bitwise and in line 9 of Listing 4.17 will ensure that `bit` is in the interval $[0; 63]$ and when that is shifted right 4 places in line 10, `hash_table` will only be indexed with values in the range $[0; 3]$, which falls inside the size of the array.

While most of these omissions make our results seem rather more abysmal than strictly necessary, we do have the means of strengthening the results by addressing the above-mentioned shortcomings. It is also worth noticing that our lack of finding intraprocedural buffer overflows might as well be attributed to the fact that few buffer overflows that occur in the kernel are intraprocedural (see e.g. §5.1 on page 75).

4.2.2 Use-after-free

The use-after-free bugs are, however, handled better by our extensions. Our results are presented in Table 4.2, and the false positives have furthermore been broken down in Table 4.3. The interprocedural cases are code-sites where the address of a variable is passed to a function that allocates it inside that function. The path prune code-sites are cases where the branch of code that causes the bug will never be taken. The non-expanded macros are cases where a return or kernel panic is hidden inside a macro that is not expanded (we use a feature of Coccinelle to expand all macros that are unique,

```
static void set_rx_mode(struct net_device *dev) {
  ...
  u16 hash_table[4];
  ...
  for (...) {
    unsigned int bit;

    ...
    bit = (ether_crc_le(6, mclist->dmi_addr) >> 3) & 0x3f;
    hash_table[bit >> 4] |= (1 << bit);
  }
}
```

Listing 4.17: False positive when using bitwise operators

i.e. that do not have multiple definitions, removing a long list of false positives). And the address cases are where just the address of the memory is used and not the memory at the address (as illustrated in Listing 4.11 on page 59). Before we look more closely at some of these false positives, we will describe a couple of the more interesting bugs that we have found.

Bugs

Using members of a freed structure as the freed storage can be potentially disastrous as it may have had other values written to it by another part of the system that has had the memory allocated to it in the meantime (when interrupts are enabled). A member use in freed storage occurs among other places in `drivers/serial/sunsu.c` as shown in Listing 4.18 where `up` is freed in line 9 and is subsequently accessed in line 14 and 15. In particular, in the call to `of_iounmap` then `up->port.membase`'s virtual page file may also be passed to `kfree`—if the value of `up->port.membase` has changed between being freed in line 9 and the free inside `of_iounmap` this is a bug.[7]

The situation becomes even more precarious if the program writes to freed memory, as this may allow a malicious user with some care and effort to direct the logic of a program to execute his own code. A memory write to freed memory happens, among other places, in `drivers/video/igafb.c` as shown in Listing 4.19 where `par->mmap_map` is freed in line 7 and is subsequently assigned in lines 15–19 and 22–26, provided we are compiling for a SPARC machine.

False positives

As noted in Table 4.3 the false positives we have found in Linux 2.6 are grouped into four primary categories: just using the address and not the memory it points to,

[7] Strictly speaking, it is always an error to access freed memory as far as the C programming language standard is concerned, but an error will usually only be triggered in practice if the memory has changed between the free and use.

Bugs found	17
False positives found	26
Success rate	40%

Table 4.2: Success rates for finding use-after-free bugs in Linux 2.6

Address	15
Interprocedural	5
Path pruning	4
Non-expanded macros	2

Table 4.3: Reasons for false positives for use-after-free bugs in Linux 2.6

```
static int __devexit su_remove(struct of_device *op) {
  struct uart_sunsu_port *up = dev_get_drvdata(&op->dev);

  if (up->su_type == SU_PORT_MS ||
      up->su_type == SU_PORT_KBD) {
#ifdef CONFIG_SERIO
    serio_unregister_port(&up->serio);
#endif
    kfree(up);
  } else if (up->port.type != PORT_UNKNOWN) {
    uart_remove_one_port(&sunsu_reg, &up->port);
  }

  if (up->port.membase)
    of_iounmap(&op->resource[0], up->port.membase, up->reg_size);

  dev_set_drvdata(&op->dev, NULL);

  return 0;
}
```

Listing 4.18: Use-after-free bug due to member access after free

```
int __init igafb_init(void) {
  ...

  if (!iga_init(info, par)) {
    iounmap((void *)par->io_base);
    iounmap(info->screen_base);
    kfree(par->mmap_map);
    kfree(info);
  }

#ifdef CONFIG_SPARC
  ...

  /* First region is for video memory */
  par->mmap_map[0].voff = 0x0;
  par->mmap_map[0].poff = par->frame_buffer_phys & PAGE_MASK;
  par->mmap_map[0].size = info->fix.smem_len & PAGE_MASK;
  par->mmap_map[0].prot_mask = SRMMU_CACHE;
  par->mmap_map[0].prot_flag = SRMMU_WRITE;

  /* Second region is for I/O ports */
  par->mmap_map[1].voff = par->frame_buffer_phys & PAGE_MASK;
  par->mmap_map[1].poff = info->fix.smem_start & PAGE_MASK;
  par->mmap_map[1].size = PAGE_SIZE * 2; /* X wants 2 pages */
  par->mmap_map[1].prot_mask = SRMMU_CACHE;
  par->mmap_map[1].prot_flag = SRMMU_WRITE;
#endif /* CONFIG_SPARC */

  return 0;
}
```

Listing 4.19: Use-after-free bug due to writing to a variable after free

interprocedural cases that make the bug report a false positive, path pruning cases where the bug occurs on a program path that can never occur, and non-expanded macros where Coccinelle parses a macro use as a function call—if the macro contains a `return` statement, the control flow graph will not reflect this. We will present an example of the three latter cases and state how we could extend Coccinelle to be able to deal with these cases.

The interprocedural case is likely the hardest to make Coccinelle able to uncover given Coccinelle's strongly intraprocedural nature. A typical example of this is from `arch/ia64/sn/kernel/xpc_channel.c` as shown in Listing 4.20. Here the member `ch->local_msgqueue_base` is freed in line 12 and subsequently used in line 9 where its address is taken. However, as we see in line 25, `xpc_kzalloc_cacheline_aligned` does not use the freed value, but merely assigns a new buffer to it. As a precautionary measure, we could state that taking the address of a variable is not a use of it, as part of our semantic patch, but we can easily imagine cases where this may just cause false negatives instead. Thus, there are no good short-term solutions to avoiding these false positive matches.

The lack of path pruning in Coccinelle has a potential to cause errant behaviour in any semantic patch. We present one such case in Listing 4.21 from `arch/x86/pci/acpi.c` where `sd` is freed in line 8 provided that `bus` is `NULL`, and it is subsequently used in line 14, provided that `bus` is different from `NULL`, thus the two branches in the example will never both be taken and there is thus no bug here either. Like with the interprocedural case there is no easy way that we can avoid these false positives given the current workings of Coccinelle, but a path pruning algorithm would be very high on our wish list for future extensions.

Lastly there are the non-expanded macros that masquerade as function calls rather than macro uses. This is shown in Listing 4.22 that is taken from `drivers/ieee1394/pcilynx.c` where we have simplified the structure of the 341 line long function. We see here that `i2c_ad` is freed in line 13 and subsequently used in line 20, however we never make it there as there is a `return` statement as part of the `FAIL` macro in line 15. We can solve these false positives manually one at a time by exploiting Coccinelle's feature to automatically expand all macros that are listed in the file specified by `-macro_file` when invoking it, but this becomes cumbersome quickly. A better alternative would be to let Coccinelle determine whether there are multiple definitions of a macro, and if there is not, then Coccinelle could expand the macro automatically when we are searching for bugs.[8] It would always be safe to expand macros with only a single definition for bug hunting.[9]

[8]Automatic expansion would not necessarily be interesting for transforming code as one may wish to transform the use of one macro with the use of another.

[9]Unless we are trying to find bugs in the use of macros, of course.

```
static enum xpc_retval
xpc_allocate_local_msgqueue(struct xpc_channel *ch)
{
  ...
  for (nentries = ch->local_nentries; nentries > 0; nentries--) {
    nbytes = nentries * ch->msg_size;
    ch->local_msgqueue = xpc_kzalloc_cacheline_aligned(nbytes,
          GFP_KERNEL,
          &ch->local_msgqueue_base);
    ...
    if (ch->notify_queue == NULL) {
      kfree(ch->local_msgqueue_base);
      ch->local_msgqueue = NULL;
      continue;
    }
    ...
  }
  ...
}

static void *
xpc_kzalloc_cacheline_aligned(size_t size, gfp_t flags, void **base)
{
  ...
  *base = kzalloc(size, flags);
  ...
}
```

Listing 4.20: Use-after-free false positive due to interprocedural flow

```
struct pci_bus * __devinit pci_acpi_scan_root(
 struct acpi_device *device, int domain, int busnum
)
{
 ...
 bus = pci_scan_bus_parented(NULL, busnum, &pci_root_ops, sd);
 if (!bus)
  kfree(sd);

#ifdef CONFIG_ACPI_NUMA
 if (bus != NULL) {
  if (pxm >= 0) {
   printk("bus %d -> pxm %d -> node %d\n",
    busnum, pxm, sd->node);
  }
 }
#endif
 ...
}
```

Listing 4.21: Use-after-free false positive due to lack of path pruning

```
static int __devinit add_card(struct pci_dev *dev,
                              const struct pci_device_id *
                                  devid_is_unused)
{
#define FAIL(fmt, args...) do { \
    PRINT_G(KERN_ERR, fmt , ## args); \
    remove_card(dev); \
    return error; \
    } while (0)

 ...
 else {
  kfree(i2c_ad);
  error = -ENXIO;
  FAIL("read something from serial eeprom, but it does not seem
       to be a valid bus info block");
 }
 ...
 i2c_del_adapter(i2c_ad);
 kfree(i2c_ad);
 ...
}
```

Listing 4.22: Use-after-free false positive due to non-expanded macro

4.3 Other code-bases

While Coccinelle has almost exclusively been applied to the Linux kernel in previous literature, there should be nothing that causes it to be tied to this source code base. To verify this we will look at two code-bases that are meant to be run continuously and be exposed to the Internet.

We will present each of the code-bases below along with the bugs we have found in them. For the sake of being able to match *something*, we will run our semantic patches on prior versions of the code-bases that are known to contain bugs. There is nothing that prohibits running our semantic patches on code-bases without known flaws, but one would most likely need to be prepared to investigate several code-bases before finding something that actually contains a bug that we can find, given the low number of flaws we have developed patches for.

4.3.1 tbaMUD

tbaMUD is a text-based multiplayer online roleplaying game that is meant to run continuously around the clock and provide a virtual world where players can log in and engage each other and computer controlled entities in combat and puzzle-solving. The game server is available from `http://www.tbamud.com`. We will run our semantic patches against revision 103 from their subversion repository. Revision 103 contains two known buffer overflow bugs in `src/genqst.c`.

Our somewhat sparse results for buffer overflows are shown in Table 4.4. We found no use-after-free bugs in the source code. The two buffer overflow bugs are actually the same bug, but in two different accesses to the same array that may both overflow. The code is shown in Listing 4.23 and both the possible overflows that our semantic patch detects are in line 8. The actual buffer overflow, though, may happen in line 7 provided that the string representation of the float is more than 19 characters (plus one for the null terminator). This may easily happen as even on our 32-bit machine, the maximum float value in string representation is '340282346638528859811704183484516925440.000000', which is clearly beyond 19 characters.

The two known buffer overflows are not detected as the declaration is global, and thus outside the function where it is used in error (see §4.1.1). The known bugs are shown in Listing 4.24 where `QST_MASTER(rnum)` in lines 3, 4, 5, and 6 might be `NOBODY`, which is defined as '`(unsigned short int)~0`', i.e. the maximum value for an unsigned short integer.

4.3.2 Icecast

Our second code-base is an open source server for streaming multimedia across the Internet that is used by several online radio stations. Software faults in this service might take out an entire radio station and potentially lose the station a lot of revenue from advertisements. Icecast is available from `http://www.icecast.org` and we will

Bugs found	2
False positives found	1
Success rate	67%

Table 4.4: Success rates for finding buffer overflows in tbaMUD

```c
void do_float(FILE * shop_f, FILE * newshop_f)
{
 float f;
 char str[20];

 fscanf(shop_f, "%f \n", &f);
 sprintf(str, "%f", f);
 while ((str[strlen(str) - 1] == '0') &&
   (str[strlen(str) - 2] != '.'))
   str[strlen(str) - 1] = 0;
 fprintf(newshop_f, "%s \n", str);
}
```

Listing 4.23: Buffer overflow in `util/shopconv.c`

be analysing revision 11411 from their subversion repository, which contains a single known use-after-free bug.

We only find the known use-after-free bug, and no other faults or false positives at all. The known use-after-free bug is illustrated in Listing 4.25 where `fullpath` is freed in line 4 and subsequently used in line 6, i.e. a classic use-after-free bug.

4.4 Summary

We have shown that our extensions can help find bugs in real-world applications, ranging from smaller Internet servers like tbaMUD and Icecast to large-scale operating system kernels such as Linux. Our buffer overflow extension in particular suffers from a large number of false positives, but we have outlined several steps that can be taken to further strengthen its usefulness.

```c
int add_quest(struct aq_data *nqst) {
  ...
  if (mob_index[QST_MASTER(rnum)].func &&
    mob_index[QST_MASTER(rnum)].func != questmaster)
    QST_FUNC(rnum) = mob_index[QST_MASTER(rnum)].func;
  mob_index[QST_MASTER(rnum)].func = questmaster;
  ...
}
```

Listing 4.24: Known buffer overflow in `genqst.c`

```c
int fserve_client_create (client_t *httpclient, const char *path)
     {
  ...
  file = fopen (fullpath, "rb");
  free (fullpath);
  if (file == NULL) {
    WARN1 ("Problem accessing file \"%s\"", fullpath);
    client_send_404 (httpclient, "File not readable");
    return -1;
  }
  ...
}
```

Listing 4.25: Known use-after-free bug in `fserve.c`

Chapter 5

Comparing Coccinelle to other bug finders

While we have already established that Coccinelle can be used as a bug finding tool in Chapter 4, and in the work by Stuart et al. [2007] and Lawall et al. [2008], it is also interesting to see how we compare to other bug finding tools, as this will both give us an idea of our current effectiveness and possible avenues for future work.

Coccinelle has previously been compared to the work by Engler et al. [2000] by Stuart et al. [2007]. In this chapter we will compare our work to Splint [Evans, 1996, Larochelle and Evans, 2001] and Valgrind [Seward and Nethercote, 2005, Nethercote and Seward, 2007a,b].[1]

There are a few more publically available tools that we will not try to compare with, among others CCured [Necula et al., 2005], which requires the user to change a potentially large amount of source code lines in order to 'cure' a program, and BOON [Wagner et al., 2000], which tracks buffer overflows in programs. However, we could not get BOON to compile properly. There are furthermore a number of simple lexical-based tools that are basically extensions of *grep* that understand C tokens such as ITS4 and Flawfinder.[2] Lastly, there are some more dynamic analysis tools such as Electric Fence. Valgrind is by far the most polished and functional dynamic analysis tool available, so we will focus on this.

In order to utilise the work we have already done, we will compare the accepted results of Coverity (by Engler et al.) on the Linux kernel with Coccinelle in §5.1, and we will compare our results on tbaMUD and Icecast with Splint and Valgrind in §5.2. Before we proceed with the comparisons, we will briefly introduce each tool and its functionality.

It is worth mentioning here that while each of the tools in this chapter support finding a large number of kinds of bugs, we will only use them to find use-after-free bugs and buffer overflows.

Coverity Prevent

What was originally an academic effort by Engler et al. [2000], has now spun off into a commercial product called Coverity Prevent [Coverity]. Coverity has in collaboration with the American Department of Homeland Security launched a great effort in providing 'free' scan results for a large number of open source software projects. At the time

[1]Splint is available from `http://www.splint.org` and Valgrind from `http://www.valgrind.org`.

[2]ITS4 is available from `http://www.cigital.com/its4/` and Flawfinder from `http://www.dwheeler.com/flawfinder/`.

of writing there are 270 projects that are checked frequently.[3] However, due to policy restrictions, anyone developing a competing product (like Coccinelle), is prohibited from using these scan results due to 'intellectual property rights':

> *Coverity's Intellectual Property may include elements that would assist competitors in creating or improving products competitive to Coverity's tools. You agree that by accepting access to the Service you commit not to distribute or share details of the service or its analysis with any entity without prior authorization from Coverity.*

As an alternative, Engler et al. [2000] used to have their full results from using the Stanford Checker on the Linux 2.4.2 kernel (and a few earlier versions) available on a website, however, access to this site has been closed for several months now, making a comparison with these results impossible as well.[4]

As a last resort, we have chosen to compare our results against the kernel patches that credit Coverity for finding a bug, *and* the bug is either a *use-after-free* bug or a buffer overflow. While this will afford us no knowledge of the comparative false positive rates, it can tell us whether we are as good at matching flaws as they are.

While Coverity today is applied to a large number of open source projects, a few samples from the revision control systems of other projects show no similar endeavour to credit Coverity with finding a bug. We will therefore only be able to compare our results on the Linux kernel with Coverity.

Splint

Splint is, like Coverity and Coccinelle, a static analysis tool, but unlike Coverity and Coccinelle it relies on user-defined comments in the source code to direct its static analysis. Splint contains features for detecting both buffer overflows [Larochelle and Evans, 2001] and use-after-free bugs [Evans, 1996]. Splint may process files with and without extra user comments directing its efforts—we will only use it on programs that are not adorned with such comments.[5]

Unlike Coccinelle, Splint works on preprocessed code, so it will only check a single configuration of a program. We will configure the programs using a standard Ubuntu 8.04 distribution on a Linux 2.6.24 kernel, and run Splint on this configuration.

Valgrind

Unlike the other tools, Valgrind is a dynamic analysis tool that is both a platform for writing tools that check a program for certain properties and a virtual processor on which the program is run that calls into a specific Valgrind tool. We will only use the

[3]The results are available at `http://scan.coverity.com`.

[4]We have contacted them about making the database available, but the latest response we have had is that the server is corrupted and they are looking for a backup if it exists.

[5]tbaMUD contains user comments that indicate whether some instances of using `strcpy` are safe, however they are not in a format can Splint will use them.

default tool for Valgrind, called memcheck, that tries to verify that the program being executed does not write to unallocated memory, that freed memory is not used, etc. Memcheck accomplishes this using Valgrind's shadow memory model [Nethercote and Seward, 2007b] that tries to ensure that all memory accesses are legal.

Given Valgrind's dynamic nature, it will, like Splint, only work on a single configuration of a program. Furthermore, it only detects flaws in the parts of the program that are executed.

5.1 Coverity and Linux 2.6

The comparison with Coverity is based on the Linux 2.6 kernel at commit ID `baad-ac8b10c5ac15ce3d26b68fa266c8889b163f`.[6] We have searched through the log for the git repository, dating from April 2005 to March 2008, and we have found mention of 38 buffer overflows and 37 use-after-free bugs that are attributed to Coverity.

For each of these bugs we have run the SmPL patches from Chapter 3 on them and recorded whether we also match the bug. Of the 38 buffer overflows, we match 2, and of the 37 use-after-free bugs we were able to match 24, where the ones we miss are either interprocedural or hit some corner case we have not implemented for the expression-based control flow graph.

Below, we cover the things that Coverity has found, what we have found, and more importantly, what we have missed.

5.1.1 Buffer overflow bugs in the Linux-2.6 kernel

Table 5.1 lists all the 38 commits where a buffer overflow that has been attributed to Coverity has been fixed, and for each of these commits we indicate whether we could match it, or if not, why. We will start with looking at the two cases that we do match, and then we will look at a single case where we cannot match the code due to it being interprocedural. One of the failures is caused by a short-coming in how we deal with empty ranges in the generalised constant propagation algorithm and the other is because the array is inline-initialised without specifying an explicit size to the array, as covered in §4.1.1.

The first case that we match, the code in the commit just before commit ID `8ea3-71fb6df5a6e8056265e0089fd578e87797fc`, is shown in Listing 5.1. It is a classic case of copy-paste coding, where the use in line 17 may be out of bounds as `j` is constrained to the bounds of `period` rather than `delay`. Since `ARRAY_SIZE` expands into a `sizeof` computation, we match this buffer overflow due to our cautionary measure of resolving any `sizeof` computation to the bounds $[-\infty; \infty]$. As such we also mark the access of `period[i]` in line 13 to be a potential buffer overflow, even though it is not.

Looking at the code just before commit ID `d6d21dfdd305bf94300df13ff47214-1d3411ea17` in Listing 5.2, we see that this is a classical off-by-one error that causes a

[6] Available from `http://git.kernel.org/?p=linux/kernel/git/torvalds/linux-2.6.git;a=commit;h=baadac8b10c5ac15ce3d26b68fa266c8889b163f`.

Commit ID with fix	**Coccinelle match**
`84ea77635b91a6ca1c0c592ee5ddc0c780856b97`	Interprocedural
`80c6e3c0b5eb855b69270658318f5ccf04d7b1ff`	Interprocedural
`5fd571cbc13db113bda26c20673e1ec54bfd26b4`	Interprocedural
`9f13fae2479ed2e2801d538d6a22309123c704f6`	Interprocedural
`a6a61c5494145c904bead0cceadd94080bd3a784`	Interprocedural
`d698f1c72629ff43d0cb6b9f1d17c491c057a0d9`	Interprocedural
`1a34456bbbdaa939ffa567d15a0797c269f901b7`	Interprocedural
`6dde432553551ae036aae12c2b940677d36c9a5b`	Interprocedural
`d93c2efc93f61c95808e303982f12fe6f5987270`	Interprocedural
`e60b6e2f747e94358fed9a23afd6abd738de4bf7`	Interprocedural
`65b07ec29354b345ff93914d064c2467aef4c862`	Interprocedural
`51af33e8e45b845d8ee85446f58e31bc4c118048`	Interprocedural
`805d92dfa627acad3d4a78966bc5e4f8183d48b3`	Fail
`221c80cf03d77490b8e45184a273834d0259b9e0`	Interprocedural
`6551198a201a70cb11e25712b1d0b2a369bb8a4c`	Interprocedural
`64e862a579015d229b8e40b6bc4ac3612e9656e1`	Interprocedural
`c899a575fa9cc802a4a77f6c5078b14fc1d12487`	Interprocedural
`69b311a4dabc9163288be1fe993cb7db47541e67`	Interprocedural
`e6a5fdf56e3a5fc179cd8c8c19081a9a11882b0c`	Interprocedural
`a0a74e45057cc3138c29173e7b0b3db8b30939ae`	Interprocedural
`88ae704c2aba150372e3d5c2f017c816773d09a7`	Interprocedural
`32a70a817acbb96fcfcc7543932222467c771207`	Interprocedural
`23c15c21d34a4b4b4d7b9a95ce498991c5339c77`	Fail
`bf703c3f199342da440a30798b6120f391741ffe`	Interprocedural
`d24030f0f71390b1a01796d664445352bd403269`	Interprocedural
`05052f7f130b1232faeee1674a5bc41f67746cff`	Interprocedural
`b196872cd65a06ad65853c4513e0d0f24452d32e`	Interprocedural
`0d0d871b3f3395820ec33a78fb2cc101b9bdcced`	Interprocedural
`8ea371fb6df5a6e8056265e0089fd578e87797fc`	✓
`0a3a6d69b7e9f1d7fa5add7db528e7b81cbd422e`	Interprocedural
`68a26aecb3829d013f612def3c8995efdbad3306`	Interprocedural
`052bb88e180d010f7da7186e6f21eea3be82a903`	Interprocedural
`9ec85c03d045d5ec24d6f15649a68646aefe88ba`	Interprocedural
`e3a5cd9edff9a7a20de3c88c9d479704da98fb85`	Interprocedural
`3b71797eff4352b4295919efc52de84f84d33d94`	Interprocedural
`d6d21dfdd305bf94300df13ff472141d3411ea17`	✓
`5bab2482083077d1e14644db2546c54534156849`	Interprocedural
`3a63e44420932852efd6a7d6d46cdad4d427f880`	Interprocedural

Table 5.1: Buffer overflow bugs from the Linux 2.6 kernel

buffer overflow, i.e. the loop runs one place too far, and thus the `'\0'` written in line 7 is one beyond the bounds of the buffer. Like with the other buffer overflow we detect, we only detect this by virtue of imprecision since we assign `i` the interval $[-\infty; \infty]$ since we have no better approximation of what `sizeof` is in our implementation. This means that we in reality also flag the array access in line 6 as a possible buffer overflow, even though it is not.

As an example of interprocedural use, we may look at the code just prior to commit ID `80c6e3c0b5eb855b69270658318f5ccf04d7b1ff` as shown in Listing 5.3 where the guard in line 7 should have been a '>=' to not cause a potential buffer overflow in line 9. The actual array, `scsi_device_types`, is defined globally in the same file as the function, and as such our semantic patch does not pick up its existence. This corresponds to the issue illustrated in Listing 4.3 on page 53.

```c
static int atkbd_set_repeat_rate(struct atkbd *atkbd)
{
  const short period[32] =
    { 33, 37, 42, 46, 50, 54, 58, 63, 67, 75, 83, 92, 100, 109,
        116, 125,
     133, 149, 167, 182, 200, 217, 232, 250, 270, 303, 333, 370,
         400, 435, 470, 500 };
  const short delay[4] =
    { 250, 500, 750, 1000 };

  struct input_dev *dev = atkbd->dev;
  unsigned char param;
  int i = 0, j = 0;

  while (i < ARRAY_SIZE(period) - 1 && period[i] < dev->rep[
      REP_PERIOD])
    i++;
  dev->rep[REP_PERIOD] = period[i];

  while (j < ARRAY_SIZE(period) - 1 && delay[j] < dev->rep[
      REP_DELAY])
    j++;
  dev->rep[REP_DELAY] = delay[j];

  param = i | (j << 5);
  return ps2_command(&atkbd->ps2dev, &param, ATKBD_CMD_SETREP);
}
```

Listing 5.1: Buffer overflow in the Linux-2.6 kernel (commit ID `8ea371fb6df5a6e805-6265e0089fd578e87797fc`)

```c
void __init efi_init(void) {
  ...
  char vendor[100] = "unknown";
  ...
  for (i = 0; i < sizeof(vendor) && *c16; ++i)
    vendor[i] = *c16++;
  vendor[i] = '\0';
  ...
}
```

Listing 5.2: Buffer overflow in the Linux-2.6 kernel (commit ID `d6d21dfdd305bf9430-0df13ff472141d3411ea17`)

```
const char * scsi_device_type(unsigned type)
{
  if (type == 0x1e)
    return "Well-known LUN ";
  if (type == 0x1f)
    return "No Device ";
  if (type > ARRAY_SIZE(scsi_device_types))
    return "Unknown ";
  return scsi_device_types[type];
}
```

Listing 5.3: Buffer overflow in the Linux-2.6 kernel (commit ID 80c6e3c0b5eb855b69-270658318f5ccf04d7b1ff)

Apart from the noted failures in our implementation, we are able to match all the intraprocedural buffer overflows that have also been found by Coverity. Some work remains in matching buffer overflows due to interprocedural properties.

5.1.2 Use-after-free bugs in the Linux-2.6 kernel

The commits containing use-after-free bug fixes attributed to Coverity are listed in Table 5.2. All of the matches we make are fairly straightforward use-after-free bugs, and the failures are due to some unresolved differences between the our expression-based control flow graph and Coccinelle's model checker, so we will not go into detail about any of these. Of more interest, however, is the reason for our inability to match the majority of the interprocedural cases.

A typical interprocedural case, taken from the code just prior to commit ID 8dc22d2b642f8a6f14ef8878777a05311e5d1d7e, is shown in Listing 5.4. If the call to rose_route_frame in line 4 succeeds then skbn is actually freed, and as such the use of skbn in line 11 is invalid. This is symptomatic of the majority of the cases marked 'Interprocedural' in Table 5.2. We could utilise the same protocol finding techniques as employed by Lawall et al. [2008] to find all functions that may free a passed variable as part of either success or failure, and then create a semantic patch that utilises this information in matching use-after-free cases. This should be able to match the majority of the interprocedural cases in the Linux kernel, but it will not be a general substitution against other code-bases that are less structured.

5.2 Splint, Valgrind and the other code-bases

While we could not provide a direct a comparison with Coverity, we can do that with Splint and Valgrind. In the sections below we will describe our approach to analysing tbaMUD and Icecast using Splint and Valgrind respectively, as well as explain the faults we find, the false positives, and in particular look at the differences to Coccinelle.

Commit ID with fix	Coccinelle match
2daa48729dfafd349c2a52520734de2edb9dc805	✓
8dc22d2b642f8a6f14ef8878777a05311e5d1d7e	Interprocedural
bafefc0cf8e4b34fbb159ea2e2aef2358ebff935	✓
7c908fbb0139fa1080412d0590189abfe2df87eb	(5 matches) ✓
cdee5751bf91d02616aaf30a5affef56105e3b79	Interprocedural
5185c7c20a4b88892f868ad8d92d1b640b1edba9	✓
a2e9c384ce76993cd68d6de57eaa81985b4618e3	✓
f84fba6f969065c6622669bbaa955c26fc1461ae	✓
ad008d42bcec99911b3270a8349f8ec8405a1c4e	(2 matches) ✓
651be3a2ba95bc30fcb737985741736e63231cdf	✓
2fa993423a345fd484f7295797ddb59b7738ad38	Interprocedural
d5cd97872dca9b79c31224ca014bcea7ca01f5f1	✓
1544fdbc857cbe8afca16a521d3254346befeb06	✓
fcf94c89af8acccb14ce37b1c9e8dd6bd32a999d	Interprocedural
bdc3e603cda3433c2ccc2069d28f7f3cd319cfc6	Interprocedural
1a3cac6c6d1f56dc26939eb41be29844f897c15a	✓
07ddf768d860bee7bd6581b7af3ce1009dbd05d0	✓
de47b69c7b7be46b0848b2c4f8e23c478cd68690	✓
c9b3febc5b9c55a76b838c977b078195ec8bb95e	Fail
09c7d8293a2d1317d16ef4ddb9f6dd2553d0694e	✓
190644e180794208bc638179f4d5940fe419bf9c	Interprocedural
98ac0e53facc851f8bc5110039ab05005c0c4736	✓
c4e90ec0134d7bedebbe3fe58ed5d431293886d4	Fail
d04d01b113be5b88418eb30087753c3de0a39fd8	✓
835d90c4218dffe6f9e7ac1ed79795197a4970c4	Fail
104326f8df9925317cca64b84249d3eac5de7c74	✓
699756199d65700e8deed59ae250439ca8684686	✓
8abceaf1cf44b9d95bcc366fa277b33e292141c4	✓
3de4414e798795ef5d719622dbf12bbe27a9e72e	Interprocedural
bcc54f9a563f146e723ead16c76f842bcaeb694e	Interprocedural
c27e8c591854ef349fdf5bec777355dae04bb48f	Interprocedural
a2df813beab42740fa8043b3fdc9e1d15784b9ec	Interprocedural

Table 5.2: Use-after-free bugs from the Linux 2.6 kernel

```c
static int rose_rebuild_header(struct sk_buff *skb)
{
  ...
  if (!rose_route_frame(skbn, NULL)) {
    kfree_skb(skbn);
    stats->tx_errors++;
    return 1;
  }

  stats->tx_packets++;
  stats->tx_bytes += skbn->len;
  ...
}
```

Listing 5.4: Use-after-free bug from the Linux-2.6 kernel (commit ID 8dc22d2b642f-8a6f14ef8878777a05311e5d1d7e)

5.2.1 Splint

To analyse the programs using Splint, we have configured each of the programs using their regular automake setup on a standard Ubuntu 8.04 system on a single 32-bit AMD processor as this will generate some needed header files. For the analysis we will be using Splint version 3.1.2. As stated we will only use Splint to look for buffer overflows and use-after-free bugs, and we will therefore run Splint with a number of switches that makes it disregard other issues. The exact switches for each of the two programs are given below. We will not explain what the switches do, but rather refer to Splint's documentation.

Each issue reported by Splint is on the same form as shown in Listing 5.5 where Splint has discovered a possible use-after-free bug in `dg_olc.c`.

tbaMUD

We have run Splint on each .c file in tbaMUD using the switches shown in Listing 5.6. This has produced a total number of 440 possible faults, where 165 are possible buffer overflows and the remaining 275 are possible use-after-free bugs. After having gone through this list of possible faults we have discovered 8 places where buffer overflows may occur and 2 places where a use-after-free bug may occur.[7] The results are listed with success rates in Table 5.3 and 5.4.

As noted in §4.3.1, the version of tbaMUD we are checking contains a known buffer overflow in `genqst.c`, and like Coccinelle, Splint does not find this bug. The reason that Splint is not able to detect this bug is because the array accessed, `mob_index`, is heap allocated and with a size that is dependent on the data files for the MUD.

Most of the actual buffer overflow bugs are due to uses of `strcpy` that we do not track explicitly in Coccinelle, and in fact it seems as if Splint does nothing more than alert on each case of `strcpy` merely by virtue of being an error-prone function. As an example of one of these bugs, we may consider Listing 5.7 where when the player can see the relevant object, then its `short_description` may be `MAX_STRING_LENGTH` long, which is a good deal longer than the 128 characters available in `buf`. Since string operations are a very error-prone aspect of the C programming language, it would make sense to create patterns for matching these cases in Coccinelle. The remaining buffer overflows that are found by Splint are almost identical to this one. As a point of interest it also finds the buffer overflows that Coccinelle does, however, Splint finds them due to the use of `strcpy` rather than the array access after the `strcpy`.

Both the use-after-free bugs that Splint finds are fairly straightforward uses of memory after it has been freed. The reason that Coccinelle finds neither of these is that the two files they belong to has hit upon a flaw in our implementation where the expression-based control flow graph is not in a form that Coccinelle's model checker expects, and thus nothing is found. We expect that if this flaw were corrected, then Coccinelle should find both use-after-free bugs without any further issues. One of the

[7] This corresponds to success rates of 4.6% for the buffer overflows and only 7‰ for the use-after-free bugs.

```
    dg_olc.c:459:13: Field proto->arglist used after being
        released
       dg_olc.c:419:10: Storage proto->arglist released
```

Listing 5.5: Splint error report

```
splint -I. -varuse -noret -initallelements -formatconst
  -fixedformalarray -firstcase -ifempty -castfcnptr
  -aliasunique -immediatetrans -noeffect -dependenttrans
  -observertrans -macrovarprefixexclude -nullassign
  -statictrans -shadow -exitarg -unreachable -globstate
  -unqualifiedtrans -compmempass -exportlocal -kepttrans
  +charindex -temptrans -shiftimplementation
  -unsignedcompare -compdestroy -onlytrans -casebreak
  -modobserver -formattype -nullret -unrecog -nullderef
  -branchstate -mustfreeonly -predboolothers -usedef
  -compdef -evalorder -nullstate -incondefs -predboolint
  -paramuse -mustfreefresh -shiftnegative -type -nullpass
  -retvalint -retvalother -boolops +posixlib -D__GNUC__
```

Listing 5.6: Splint switches for analysing tbaMUD

Bugs found	8
False positives found	157
Success rate	4.8%

Table 5.3: Success rates for finding buffer overflows in tbaMUD with Splint

Bugs found	2
False positives found	273
Success rate	0.7%

Table 5.4: Success rates for finding use-after-free bugs in tbaMUD with Splint

```
#define MAX_STRING_LENGTH 49152

#define OBJS(obj, vict) (CAN_SEE_OBJ((vict), (obj)) ? \
      (obj)->short_description : "something")

static int Crash_report_unrentables(struct char_data *ch, struct
     char_data *recep, struct obj_data *obj)
{
  char buf[128];
  ...
  sprintf(buf, "$n tells you, 'You cannot store %s.'", OBJS(obj,
      ch));
  ...
}
```

Listing 5.7: Example of a buffer overflow in tbaMUD discovered by Splint

use-after-free bugs that Splint has uncovered is illustrated in Listing 5.8. This is only a use-after-free bug on some paths, namely the ones where `trg->arglist` in line 4 is `NULL`, because `proto->arglist` in line 13 will refer to the already freed memory from line 9.

The false positives do, however, tell us a lot more about the way that Splint works. The buffer overflow reports in particular are all due to tbaMUD's extensive use of `sprintf` (rather than `snprintf`), and these are the only possible sites for buffer overflows that Splint considers on the tbaMUD code-base. We can verify that this is merely an extremely naïve implementation of reporting `sprintf`-uses given the code in Listing 5.9 where it is clear that the string length of the formatted string in line 6 will only be 3 characters (plus 1 for the null-terminator), which is clearly less than 49152 characters. The remaining false positives for buffer overflows follow the same pattern, but a lot of them are not as obviously false positives.

The false positives for use-after-free are a bit more varied as Splint has a more general idea of when objects are released rather than just with a call to `free`.

Since C does not contain copy constructors like C++, the tbaMUD developers have solved the problem by assigning one struct to another using the syntax '`*dest = *src`', followed by another call that properly copies all pointers (e.g. strings) in the struct. This, however, is seen by Splint as `src` releasing all its members, and if it is thus used subsequently to this copying, a long list of false positives will be generated. This can, for instance, be seen in `genmob.c`, also shown in Listing 5.10, where `*mob` is copied to the prototype list of monsters and subsequently all its strings are copied properly to the prototype. In reality, `*mob` is not freed at the use in line 5, despite Splint's report to the contrary.

One of the more curious use-after-free false positives are shown in Listing 5.11 where Splint reports a use-after-free bug in the loop increment, '`ch = ch->next_in_room`', in line 4, as it considers that `ch` has been released at the `return` in line 8. We can clearly

```c
void trig_data_copy(trig_data *this_data, const trig_data *trg)
{
 ...
 if (trg->arglist) this_data->arglist = strdup(trg->arglist);
}

void trigedit_save(struct descriptor_data *d) {
 ...
 free(proto->arglist);
 ...
 trig_data_copy(proto, trig);
 ...
 if (proto->arglist)
  live_trig->arglist = strdup(proto->arglist);
 ...
}
```

Listing 5.8: Use-after-free bug in tbaMUD discovered by Splint

```c
#define MAX_STRING_LENGTH 49152

int format_text(char **ptr_string, int mode, struct
    descriptor_data *d, unsigned int maxlen, int low, int high)
    {
 ...
 char buf[MAX_STRING_LENGTH];
 sprintf(buf, "%c␣␣", *flow);
 ...
}
```

Listing 5.9: Buffer overflow false positive as reported by Splint

```c
int add_mobile(struct char_data *mob, mob_vnum vnum) {
 ...
 mob_proto[i] = *mob;
 mob_proto[i].nr = i;
 copy_mobile_strings(mob_proto + i, mob);
 ...
}
```

Listing 5.10: Use-after-free false positive as reported by Splint

see that this is not the case, but it illustrates that the algorithm used by Splint to detect use-after-free bugs is flow insensitive, as the `return` would otherwise have escaped the flow from the loop.

The last type of false positive where Splint is overly zealous is illustrated in Listing 5.12. The false positive occurs in line 7 as it states that `ch->player.short_descr`, which is freed in line 4, reaches that point and is passed as a parameter out of the function. While this may, indeed, lead to a use-after-free were the `affect_remove` function to use it, there is no such use in `affect_remove` and it is therefore a false positive. There are a lot of these false positives generated for several of the tbaMUD functions, given the way that e.g. characters are freed, and since Splint generates a false positive for every previously freed member, it may contribute substantially to the total count of false positives for some of the more complex structures.

As we have seen, Splint can find bugs in tbaMUD, but it does so with extreme caution by reporting all possible issues, even if a fairly simple verification could have removed an issue as a false positive. Furthermore, not removing false positives causes Splint to only obtain a 7‰ success rate for finding use-after-free bugs in tbaMUD. On the other hand, by exerting this caution it also finds 7 buffer overflows in tbaMUD that Coccinelle does not. This is, however, primarily due to the fact that we do not track uses of `strcpy` and other unsafe string operations in our semantic patches.

Icecast

We have run Splint on each .c file in Icecast using the switches shown in Listing 5.13. This has produced a total number of 23 possible faults, where 4 are possible buffer overflows and the remaining 19 are possible use-after-free bugs. Of these, there are zero buffer overflows and a single use-after-free bug, namely the known bug. The results for use-after-free bugs are listed in Table 5.5.

As all the false positives for Icecast mimic the behaviour from running Splint on tbaMUD, we will not present any of the false positive cases here.

5.2.2 Valgrind

For our tests we will be using Valgrind version 3.3.0 from the Ubuntu 8.04 repository. Each of the programs are run in their standard configuration where we try to exert the parts of the program that will be utilised as part of normal use. We will describe our approach to testing each program in more detail below. For both programs, Valgrind reports a number of other issues like using uninitialised variables, but we will disregard all of these for the purposes of comparing found buffer overflows and use-after-free bugs to Coccinelle.

tbaMUD

We have run the tbaMUD server as checked out without any modifications to it and have subsequently connected two clients to it, one to test some of the administrative features, and one to play the game as a normal player. We have tried to make some effort

```
static struct char_data *get_victim(struct char_data *chAtChar)
{
  ...
  for (ch = world[IN_ROOM(chAtChar)].people; ch; ch = ch->
      next_in_room) {
    if (FIGHTING(ch) == NULL)
      continue;
    ...
    return (ch);
  }
  ...
}
```

Listing 5.11: Use-after-free false positive as reported by Splint

```
void free_char(struct char_data *ch) {
  ...
  if (ch->player.short_descr)
    free(ch->player.short_descr);
  ...
  while (ch->affected)
    affect_remove(ch, ch->affected);
  ...
}
```

Listing 5.12: Use-after-free false positive as reported by Splint

```
splint -I.. -I/usr/include/libxml2 -I. -varuse -noret
  -initallelements -formatconst -fixedformalarray
  -firstcase -ifempty -castfcnptr -aliasunique
  -immediatetrans -noeffect -dependenttrans
  -observertrans -macrovarprefixexclude -nullassign
  -statictrans -shadow -exitarg -unreachable -globstate
  -unqualifiedtrans -compmempass -exportlocal -kepttrans
  +charindex -temptrans -shiftimplementation
  -unsignedcompare -compdestroy -onlytrans -casebreak
  -modobserver -formattype -nullret -unrecog -nullderef
  -branchstate -mustfreeonly -predboolothers -usedef
  -compdef -evalorder -nullstate -incondefs -predboolint
  -paramuse -mustfreefresh -shiftnegative -type -nullpass
  -retvalint -retvalother -boolops +posixlib -D__GNUC__
```

Listing 5.13: Splint switches for analysing Icecast

Bugs found	1
False positives found	19
Success rate	5%

Table 5.5: Success rates for finding use-after-free bugs in Icecast with Splint

to use multiple parts of the code-base, but by no means all of it. We have furthermore made sure that we activate the known buffer overflow bug. However, despite our best efforts, Valgrind only finds a single use-after-free bug as part of the shutdown procedure of the MUD. It does not find the known buffer overflow either.

The use-after-free bug that Valgrind finds is, indeed, fairly involved. As part of the MUD database being freed each character is processed and anyone following that character is stopped from following him and then the character is freed. If a room contains three characters A, B, and C, and C is following B and the characters are freed in alphabetical order, then as part of the procedure that stops C from following B a message will be printed to everyone in the room, 'C stops following B'. However, as A is already freed, but not removed from the room's character list, the MUD will try to send this message to that person, resulting in a use-after-free. The relevant code is located in `db.c` lines 472–478, but since it touches on so many parts of the MUD code-base, we will not try to present all the code here.

It is, however, more interesting that Valgrind also fails to detect the known buffer overflow. To understand why this is the case, we have to delve into how Valgrind checks that memory accesses are safe, which is in fact fairly simple: A memory access is safe when the memory is defined [Nethercote and Seward, 2007b]. This means that when we have the array `mob_index` that is allocated with size 3612 (default number of mobs), but indexed with 65535, then this is clearly semantically invalid, but Valgrind sees it as no problem if the address '`mob_index + 65535`' is defined. So even if we had some way to test all parts of the code in all possible configurations, we might still not discover these bugs as part of the program execution.

Icecast

Icecast is only really an intermediary webserver that facilitates access to media served by another program to clients that connect to the Icecast webserver using their media player. To test Icecast in as much of a production environment that we can replicate, we use IceS2 to provide a playlist of an Ogg Vorbis encoded version of the music album '*Michael Bublé - Call Me Irresponsible*', streaming data to the Icecast webserver at `/playlist.ogg`, and connect to this playlist using the Exaile music client and listen to it for the entire duration of the album.[8] Since the known bug lies exclusively in the webserver part of Icecast, we also make a request on a local file that we have removed read-permission from for the Icecast user, as this is what triggers the bug.

[8] IceS2 is available from `http://www.icecast.org/ices.php` and Exaile is available from `http://www.exaile.org`.

After having performed the test as described above, the only result we got was the known use-after-free bug, which is shown from the Valgrind log in Listing 5.14. Here the read in `fserve.c:471` corresponds to the use after the `fullpath` variable was freed in `fserve.c:468`.

5.3 Summary

We have compared our extensions to Coccinelle with the successful matches of Coverity on the Linux kernel, since their full results are not freely available, and found that we match the intraprocedural cases as well as Coverity, but, as expected, we fail to match interprocedural occurrences of bugs.

We have furthermore compared our extensions to the bug finding capabilities of the publically available static analysis tool, Splint, and the publically available dynamic analysis tool, Valgrind, and found that for the code-bases we have tested on, Splint produces a larger amount of false positives, but due to its general complaining about unsafe string operations also find some more bugs than we do with Coccinelle. Valgrind fails to match one of the known bugs that we had expected that it would find. Apart from this, Valgrind tends to find some more involved bugs, but will miss any bugs that are not part of the execution path.

After having reviewed each of the tools in relation to Coccinelle, we cannot say that Valgrind, Splint or Coccinelle is better than the others, or that one could replace the others as each tool finds bugs the others do not. We can furthermore not conclude anything useful about Coverity as we do not have data on its false positive rates.

```
==18907== Invalid read of size 1
==18907==    at 0x4024532: mempcpy (mc_replace_strmem.c:676)
==18907==    by 0x42ED04A: _IO_default_xsputn (in /lib/tls/i686/cmov/libc-2.7.so)
==18907==    by 0x42C6AE2: vfprintf (in /lib/tls/i686/cmov/libc-2.7.so)
==18907==    by 0x42E7C03: vsnprintf (in /lib/tls/i686/cmov/libc-2.7.so)
==18907==    by 0x80676B9: log_write (log.c:439)
==18907==    by 0x80586A8: fserve_client_create (fserve.c:471)
==18907==    by 0x805EE1F: add_authenticated_client (auth.c:360)
==18907==    by 0x805F0F4: add_client (auth.c:434)
==18907==    by 0x805091C: _handle_connection (connection.c:875)
==18907==    by 0x8065F77: _start_routine (thread.c:655)
==18907==    by 0x42714FA: start_thread (in /lib/tls/i686/cmov/libpthread-2.7.so)
==18907==    by 0x435BE5D: clone (in /lib/tls/i686/cmov/libc-2.7.so)
==18907==  Address 0x469f968 is 48 bytes inside a block of size 52 free'd
==18907==    at 0x402265C: free (vg_replace_malloc.c:323)
==18907==    by 0x80582A2: fserve_client_create (fserve.c:468)
==18907==    by 0x805EE1F: add_authenticated_client (auth.c:360)
==18907==    by 0x805F0F4: add_client (auth.c:434)
==18907==    by 0x805091C: _handle_connection (connection.c:875)
==18907==    by 0x8065F77: _start_routine (thread.c:655)
==18907==    by 0x42714FA: start_thread (in /lib/tls/i686/cmov/libpthread-2.7.so)
==18907==    by 0x435BE5D: clone (in /lib/tls/i686/cmov/libc-2.7.so)
```

Listing 5.14: Valgrind detection of the known use-after-free bug in Icecast

Chapter 6

Conclusion

We believe that we have succeeded in showing that Coccinelle can be used to find bugs, however the false positive rates are fairly large and it will require some changes to make Coccinelle into a bug-finding tool that can compete on equal terms with e.g. Coverity.

We have developed an extension for Coccinelle's front-end domain specific language, SmPL, that allows Coccinelle to be used easily for reporting possible bug-sites using scripting rules with embedded Python code. Using the full integration of the Python interpreter into Coccinelle, we have furthermore exploited the prototyping capabilities this affords us to implement an alternative control flow graph representation that simplifies some semantic patches for finding use-after-free bugs. Also using these prototyping capabilities, we have implemented generalised constant propagation in an effort to estimate the possible interval of values that a program variable may have during program execution, which we use to find possible buffer overflow bugs in fully defined intraprocedural array definitions and uses.

Using these extensions we have begun work on adding SmPL patterns for bug descriptions in the Common Weakness Enumeration taxonomy to provide a more rigorous foundation for indicating when these bugs occur in C programs. We have, however, only taken a few, short steps into this territory, as covering all possible cases where a bug may occur using SmPL patterns would be very time-consuming. Take as an example the stack-based buffer overflow taxonomy element. There are countless ways that a buffer overflow can be achieved in C, including using simple array accesses like the ones we have described, as well as string operations, pointer dereferences, system calls, and many, many other variations, both intraprocedural and interprocedural. We have made no effort in providing exhaustive SmPL patterns for any of the bugs we have looked at, but it is our belief that SmPL, or a variation thereof, might provide better understanding of bugs in a taxonomy, both for security researchers, but probably more importantly, for normal programmers who are trying to understand the nebulous properties of e.g. a stack-based buffer overflow. Specifying a fault succintly in a pattern will, however, require a good deal further research, in our opinion.

We have furthermore successfully run Coccinelle with our extensions on a development branch of the Linux 2.6 kernel, tbaMUD, and Icecast, finding bugs in all of them. Finding buffer overflows using generalised constant propagation has proven to be somewhat more difficult, though, since `sizeof` seems to be used extensively in e.g. the Linux kernel and we lack support for accurately computing the size of an expression. This has caused us to be overly conservative, giving us a very low success rate.

Finally, we have compared the results of our extensions with other analysis tools for finding bugs, in particular Coverity on the Linux kernel, and Splint and Valgrind on tbaMUD and Icecast. There is no conclusive evidence that any of the tools are better than the others, but with continued work, Coccinelle should be able to compete well with the other tools. For the cases we do support, we match virtually all the same intraprocedural faults as Coverity, Splint, and Valgrind do.

As part of this thesis, as well as in the work Stuart et al. [2007] and Lawall et al. [2008], we have taken the first successful steps toward using Coccinelle as a full-fledged bug hunting tool by using the scripting rule extensions together with Coccinelle's existing model checker. It is our belief that with continued work, Coccinelle could become a serious contender as a static analysis platform that allows the end-user programmer a lot more autonomy in what is matched, and how it is matched and reported, than competing tools do.

6.1 Future work

While Coccinelle *can* be used to find bugs in several categories, there are several aspects of Coccinelle that could easily be improved to better find bugs.

The easiest addition would most likely be to complete the remaining corner cases of the expression-based control flow graph implementation. This would allow us to match most of the bugs that we missed in Chapter 4 and Chapter 5.

Generalised constant propagation has shown itself to be inadequate for reliably finding bugs, and in particular in discarding false positives, on the Linux kernel. Implementing a stronger analysis such as symbolic range propagation that could handle `sizeof` symbolically rather than by value, would allow us to discard many of the false positives we found for buffer overflows in the Linux kernel and more accurately label the problematic cases that contain bugs.

A great boon for the accuracy of Coccinelle would be to add algorithms for pruning infeasible paths. This would avoid several of the remaining false positives we encountered when analysing the Linux kernel.[1]

Lastly, improved support for matching interprocedurally using Coccinelle would help in many cases, e.g. by alleviating the user of the need to specify all kinds of permutations for where an array may be defined and its subsequent use. Our decision not to implement all these permutations as semantic patches has meant that we were unable to discover many of the buffer overflows that Coverity found. Adding proper interprocedural matching to Coccinelle is, however, most likely not easy, as it has been designed for matching intraprocedurally.

[1]Coverity uses infeasible path pruning today to recognise several of the cases that we have found to be false positives in Chapter 4 and Chapter 5. Their path pruning filters out numerous false positives as described in detail by Kremenek et al. [2006].

Bibliography

[Alexander et al., 2002]: Roger T. Alexander, Jeff Offutt, and James M. Bieman. Syntactic fault patterns in OO programs. In *ICECCS '02: Proceedings of the Eighth International Conference on Engineering of Complex Computer Systems*, pages 193–202, Washington, DC, USA, 2002. IEEE Computer Society. ISBN 0-7695-1757-9. doi: http://doi.ieeecomputersociety.org/10.1109/ICECCS.2002.1181512.

[Appel and Ginsburg, 1998]: Andrew W. Appel and Maia Ginsburg. *Modern Compiler Implementation in C*. Cambridge University Press, 1998. ISBN 0-521-58390-X.

[Aslam, 1995]: Taimur Aslam. A taxonomy of software faults in the UNIX operating system. Master's thesis, Purdue University, August 1995.

[Aslam et al., 1996]: Taimur Aslam, Ivan Krsul, and Eugene H. Spafford. Use of a taxonomy of security faults. In *19th NIST-NCSC National Information Systems Security Conference*, pages 551–560, 1996.

[Bae and Eigenmann, 2006]: Hansang Bae and Rudolf Eigenmann. *Interprocedural Symbolic Range Propagation for Optimizing Compilers*, volume 4339 of *Lecture Notes in Computer Science*, pages 413–424. Springer Berlin / Heidelberg, 2006. ISBN 978-3-540-69329-1. doi: 10.1007/978-3-540-69330-7_28.

[Bernstein and Duff, 1999]: Sheri J. Bernstein and Robert S. Duff. Optimizing Ada on the fly. In *SIGAda '99: Proceedings of the 1999 annual ACM SIGAda international conference on Ada*, pages 169–179, New York, NY, USA, 1999. ACM. ISBN 1-58113-127-5. doi: http://doi.acm.org/10.1145/319294.319321.

[Bisbey and Hollingworth, 1978]: Richard Bisbey and Dennis Hollingworth. Protection analysis: Final report. Technical Report ISI/SR-78-13, Information Sciences Institute, University of Southern California, May 1978.

[Bishop, 1995]: Matt Bishop. A taxonomy of UNIX system and network vulnerabilities. Technical Report CSE-95-10, University of California at Davis, Davis, California, USA, 1995.

[Blume and Eigenmann, 1996]: William Blume and Rudolf Eigenmann. Demand-driven, symbolic range propagation. In *LCPC '95: Proceedings of the 8th International Workshop on Languages and Compilers for Parallel Computing*, pages 141–160, London, UK, 1996. Springer-Verlag. ISBN 3-540-60765-X.

[Brunel et al., 2008]: Julien Brunel, Damien Doligez, René Rydhof Hansen, Julia L. Lawall, and Gilles Muller. A foundation for flow-based program matching using temporal logic and model checking. Technical Report 08/2/INFO, Ecole des Mines de Nantes, Nantes, France, 2008.

[CAPEC]: CAPEC. Common Attack Pattern Enumeration and Classification. URL `http://capec.mitre.org/data/index.html`. CAPEC is a collaborative effort and is continually updated. This work refers to version 1.1 of the database.

[CERT]: CERT. Computer emergency response team. URL `http://www.cert.org`.

[Cousot and Cousot, 1977]: Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL '77: Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252, New York, NY, USA, 1977. ACM. doi: http://doi.acm.org/10.1145/512950.512973.

[Coverity]: Coverity. Coverity prevent. URL `http://coverity.com/html/about.html`. [Online; Retrieved on the 6th of August, 2008].

[CVE]: CVE. Common Vulnerability and Exposures. URL `http://cve.mitre.org`.

[CWE]: CWE. Common Weakness Enumeration. URL `http://cwe.mitre.org/`. CWE is a collaborative effort and is continually updated. This work refers to draft 9 of the database.

[Engler et al., 2000]: Dawson Engler, Benjamin Chelf, Andy Chou, and Seth Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *OSDI'00: Proceedings of the 4th conference on Symposium on Operating System Design & Implementation*, pages 1–16, Berkeley, CA, USA, 2000. USENIX Association.

[Erosa and Hendren, 1994]: Ana M. Erosa and Laurie J. Hendren. Taming control flow: A structured approach to eliminating goto statements. In *Proceedings of the 1994 International Conference on Computer Languages*, pages 229–240. IEEE Computer Society, 1994. ISBN 0-8186-5640-X. doi: 10.1109/ICCL.1994.288377.

[Evans, 1996]: David Evans. Static detection of dynamic memory errors. In *PLDI '96: Proceedings of the ACM SIGPLAN 1996 conference on Programming language design and implementation*, pages 44–53, New York, NY, USA, 1996. ACM. ISBN 0-89791-795-2. doi: http://doi.acm.org/10.1145/231379.231389.

[Ghiya, 1998]: Rakesh Ghiya. *Putting Pointer Analysis to Work*. PhD thesis, School of Computing, McGill University, Montreal, May 1998.

[Ghiya and Hendren, 1998]: Rakesh Ghiya and Laurie J. Hendren. Putting pointer analysis to work. In *POPL '98: Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 121–133, New York, NY, USA, 1998. ACM. ISBN 0-89791-979-3. doi: http://doi.acm.org/10.1145/268946.268957.

[Gosling et al., 2005]: James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Addison-Wesley Professional, 3rd edition, 2005. ISBN 0321246780. URL `http://java.sun.com/docs/books/jls/`.

[Hansman and Hunt, 2005]: Simon Hansman and Ray Hunt. A taxonomy of network and computer attacks. *Computers & Security*, 24(1):31–43, February 2005.

[Harrison, 1977]: W. H. Harrison. Compiler analysis of the value ranges for variables. *IEEE Transactions on Software Engineering*, 3(3):243–250, 1977. ISSN 0098-5589. doi: http://dx.doi.org/10.1109/TSE.1977.231133.

[Hovemeyer and Pugh, 2004]: David Hovemeyer and William Pugh. Finding bugs is easy. In *OOPSLA '04: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 132–136, New York, NY, USA, 2004. ACM. ISBN 1-58113-833-4. doi: http://doi.acm.org/10.1145/1028664.1028717.

[Huth and Ryan, 2004]: Michael Huth and Mark Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems*. Cambridge University Press, New York, NY, USA, 2004. ISBN 052154310X.

[Jones and Hansen, 2007]: Neil D. Jones and René Rydhof Hansen. The semantics of "semantic patches" in Coccinelle: Program transformation for the working programmer. In Zhong Shao, editor, *APLAS '07: Proceedings of the 5th Asian Symposium on Programming Languages and Systems*, volume 4807 of *Lecture Notes in Computer Science*, pages 303–318. Springer, November 2007. ISBN 978-3-540-76636-0. doi: http://dx.doi.org/10.1007/978-3-540-76637-7_21.

[Jones et al., 1993]: Neil D. Jones, C. K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall International, June 1993. ISBN 0-13-020249-5. URL `http://www.dina.kvl.dk/~sestoft/pebook/pebook.html`.

[Kildall, 1973]: Gary A. Kildall. A unified approach to global program optimization. In *POPL '73: Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 194–206, New York, NY, USA, 1973. ACM. doi: http://doi.acm.org/10.1145/512927.512945.

[Killourhy et al., 2004]: Kevin S. Killourhy, Roy A. Maxion, and Kymie M. C. Tan. A defense-centric taxonomy based on attack manifestations. In *DSN '04: Proceedings of the 2004 International Conference on Dependable Systems and Networks*, pages 102–111, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2052-9.

[Kremenek et al., 2006]: Ted Kremenek, Paul Twohey, Godmar Back, Andrew Ng, and Dawson Engler. From uncertainty to belief: inferring the specification within.

In *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation*, pages 161–176, Berkeley, CA, USA, 2006. USENIX Association. ISBN 1-931971-47-1.

[Krsul, 1998]: Ivan Victor Krsul. *Software Vulnerability Analysis*. PhD thesis, Purdue University, May 1998.

[Landwehr et al., 1994]: Carl E. Landwehr, Alan R. Bull, John P. McDermott, and William S. Choi. A taxonomy of computer program security flaws. *ACM Comput. Surv.*, 26(3):211–254, 1994. ISSN 0360-0300. doi: http://doi.acm.org/10.1145/185403.185412.

[Larochelle and Evans, 2001]: David Larochelle and David Evans. Statically detecting likely buffer overflow vulnerabilities. In *SSYM'01: Proceedings of the 10th conference on USENIX Security Symposium*, pages 14–14, Berkeley, CA, USA, 2001. USENIX Association.

[Lawall et al., 2008]: Julia L. Lawall, Julien Brunel, René Rydhof Hansen, Henrik Stuart, and Gilles Muller. WYSIWIB: A declarative approach to finding protocols and bugs in Linux code. Technical Report 08/1/INFO, Ecole des Mines de Nantes, Nantes, France, 2008.

[ISO/IEC 14882:1998]: ISO/IEC 14882:1998. *Programming Languages — C++*. International Organization for Standardization, Geneva, Switzerland, 1998.

[ISO/IEC 8652:2007(E)]: ISO/IEC 8652:2007(E). *Ada Reference Manual: ISO/IEC 8652:2007(E) with Technical Corrigendum 1 and Amendment 1*. International Organization for Standardization, Geneva, Switzerland, 3rd edition, 2007.

[ISO/IEC 9899:1990]: ISO/IEC 9899:1990. *Programming languages — C*. International Organization for Standardization, Geneva, Switzerland, 1990.

[ISO/IEC 9899:1999]: ISO/IEC 9899:1999. *Programming languages — C*. International Organization for Standardization, Geneva, Switzerland, 1999.

[Lindqvist and Jonsson, 1997]: Ulf Lindqvist and Erland Jonsson. How to systematically classify computer security intrusions. In *SP '97: Proceedings of the 1997 IEEE Symposium on Security and Privacy*, pages 154–163, Washington, DC, USA, 1997. IEEE Computer Society.

[Lippmann et al., 2000]: Richard P. Lippmann, David J. Fried, Isaac Graf, Joshua W. Haines, Kristopher R. Kendall, David McClung, Dan Weber, Seth E. Webster, Dan Wyschogrod, Robert K. Cunningham, and Marc A. Zissman. Evaluating intrusion detection systems: the 1998 DARPA off-line intrusion detection evaluation. In *DISCEX '00: DARPA Information Survivability Conference and Exposition*, volume 2, pages 12–26. IEEE Computer Society, 2000. ISBN 0-7695-0490-6. doi: 10.1109/DISCEX.2000.821506.

[Lough, 2001]: Daniel Lowry Lough. *A taxonomy of computer attacks with applications to wireless networks*. PhD thesis, Virginia Polytechnic Institute and State University, 2001.

[Martin and Barnum, 2008]: Robert A. Martin and Sean Barnum. Common weakness enumeration (CWE) status update. *Ada Lett.*, XXVIII(1):88–91, 2008. ISSN 1094-3641. doi: http://doi.acm.org/10.1145/1387830.1387835.

[Martin et al., 2006]: Robert A. Martin, Steven M. Christey, and Joe Jarzombek. The case for Common Flaw Enumeration. In Elizabeth Fong, editor, *SSATTM '05: Proceedings of Workshop on Software Security Assurance Tools, Techniques, and Metrics*. U.S. National Institute of Standards and Technology (NIST), February 2006.

[Møller, 1994]: Peter Lützen Møller. Run-time check elimination for Ada 9x. In *TRI-Ada '94: Proceedings of the conference on TRI-Ada '94*, pages 122–128, New York, NY, USA, 1994. ACM. ISBN 0-89791-666-2. doi: http://doi.acm.org/10.1145/197694.197713.

[Necula et al., 2002]: George C. Necula, Scott McPeak, Shree Prakash Rahul, and Westley Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *CC '02: Proceedings of the 11th International Conference on Compiler Construction*, pages 213–228, London, UK, 2002. Springer-Verlag. ISBN 3-540-43369-4.

[Necula et al., 2005]: George C. Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. CCured: type-safe retrofitting of legacy software. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 27(3):477–526, 2005. ISSN 0164-0925. doi: http://doi.acm.org/10.1145/1065887.1065892.

[Nethercote and Seward, 2007a]: Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI 2007)*, New York, NY, USA, 2007a. ACM.

[Nethercote and Seward, 2007b]: Nicholas Nethercote and Julian Seward. How to shadow every byte of memory used by a program. In *VEE '07: Proceedings of the 3rd international conference on Virtual execution environments*, pages 65–74, New York, NY, USA, 2007b. ACM. ISBN 978-1-59593-630-1. doi: http://doi.acm.org/10.1145/1254810.1254820.

[Nielson et al., 1999]: Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999. ISBN 3540654100.

[Padioleau et al., 2006a]: Yoann Padioleau, René Rydhof Hansen, Julia L. Lawall, and Gilles Muller. Semantic patches for documenting and automating collateral evolutions in Linux device drivers. In *PLOS '06: Proceedings of the 3rd workshop on*

Programming languages and operating systems, New York, NY, USA, 2006a. ACM. ISBN 1-59593-577-0. doi: http://doi.acm.org/10.1145/1215995.1216005.

[Padioleau et al., 2006b]: Yoann Padioleau, Julia L. Lawall, and Gilles Muller. SmPL: A domain-specific language for specifying collateral evolutions in Linux device drivers. In *International ERCIM Workshop on Software Evolution (2006)*, Lille, France, April 2006b.

[Padioleau et al., 2006c]: Yoann Padioleau, Julia L. Lawall, and Gilles Muller. Understanding collateral evolution in Linux device drivers. In *The first ACM SIGOPS EuroSys conference (EuroSys 2006)*, pages 59–71, Leuven, Belgium, April 2006c. Also available as INRIA Research Report RR-5769.

[Padioleau et al., 2007]: Yoann Padioleau, Julia L. Lawall, and Gilles Muller. Semantic patches, documenting and automating collateral evolutions in Linux device drivers. In *Ottawa Linux Symposium (OLS 2007)*, Ottawa, Canada, June 2007.

[Patterson, 1995]: Jason R. C. Patterson. Accurate static branch prediction by value range propagation. In *PLDI '95: Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*, pages 67–78, New York, NY, USA, 1995. ACM. ISBN 0-89791-697-2. doi: http://doi.acm.org/10.1145/207110.207117.

[Polepeddi, 2004]: Sriram S. Polepeddi. Software vulnerability taxonomy consolidation. Master's thesis, Carnegie Mellon University, December 2004.

[Seward and Nethercote, 2005]: Julian Seward and Nicholas Nethercote. Using Valgrind to detect undefined value errors with bit-precision. In *Proceedings of the USENIX'05 Annual Technical Conference*, pages 17–30, Berkeley, CA, USA, April 2005. USENIX Association.

[Stuart et al., 2007]: Henrik Stuart, René Rydhof Hansen, Julia L. Lawall, Jesper Andersen, Yoann Padioleau, and Gilles Muller. Towards easing the diagnosis of bugs in OS code. In *PLOS '07: Proceedings of the 4th workshop on Programming languages and operating systems*, pages 1–5, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-922-7. doi: http://doi.acm.org/10.1145/1376789.1376792.

[Tsipenyuk et al., 2006]: Katrina Tsipenyuk, Brian Chess, and Gary McGraw. Seven pernicious kingdoms: A taxonomy of software security errors. In Elizabeth Fong, editor, *SSATTM '05: Proceedings of Workshop on Software Security Assurance Tools, Techniques, and Metrics*. U.S. National Institute of Standards and Technology (NIST), February 2006.

[Verbrugge et al., 1996]: Clark Verbrugge, Phong Co, and Laurie J. Hendren. Generalized constant propagation: A study in C. In *CC '96: Proceedings of the 6th International Conference on Compiler Construction*, pages 74–90, London, UK, 1996. Springer-Verlag. ISBN 3-540-61053-7. doi: 10.1007/3-540-61053-7_54.

[Wagner et al., 2000]: David Wagner, Jeffrey S. Foster, Eric A. Brewer, and Alexander Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Network and Distributed System Security Symposium*, pages 3–17, San Diego, CA, February 2000.

[Weaver et al., 2003]: Nicholas Weaver, Vern Paxson, Stuart Staniford, and Robert Cunningham. A taxonomy of computer worms. In *WORM '03: Proceedings of the 2003 ACM workshop on Rapid malcode*, pages 11–18, New York, NY, USA, 2003. ACM. ISBN 1-58113-785-0. doi: http://doi.acm.org/10.1145/948187.948190.

[Weber, 1998]: Daniel James Weber. A taxonomy of computer intrusions. Master's thesis, Massachusets Institute of Technology, June 1998.

[Würthinger et al., 2007]: Thomas Würthinger, Christian Wimmer, and Hanspeter Mössenböck. Array bounds check elimination for the Java HotSpot™client compiler. In *PPPJ '07: Proceedings of the 5th international symposium on Principles and practice of programming in Java*, pages 125–133, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-672-1. doi: http://doi.acm.org/10.1145/1294325.1294343.

[Xie et al., 2003]: Yichen Xie, Andy Chou, and Dawson Engler. Archer: using symbolic, path-sensitive analysis to detect memory access errors. *ACM SIGSOFT Software Engineering Notes*, 28(5):327–336, 2003. ISSN 0163-5948. doi: http://doi.acm.org/10.1145/949952.940115.

Acknowledgements

First and foremost I would like to thank the Coccinelle team for providing an interesting and stimulating work environment. In particular I would like to thank René Rydhof Hansen for getting me off to the best possible start on my thesis, and to my advisor Julia L. Lawall who has gone beyond what could be expected in promptly answering all my many questions at all hours of the day—if only all advisors were as responsive. The collaboration with the Coccinelle team in writing the article [Stuart et al., 2007] also provided me with a thorough refresher in academic writing and a fascinating look at the peer review process—being 'at the other side of the fence' for once was very educational. The Coccinelle team's interest in the solutions I have produced as part of my thesis have been overwhelming, and made my work seem much more relevant. I have been more than happy to integrate several of the extensions I have developed into the official version.

Lastly, I would like to thank my wonderful wife and our unborn child for being there every single day and with a smile ensuring me that I could manage one more day of writing this thesis. You have made my life and this thesis better than it would have been in your absence.

Typeset using pdftex with microtypographic and hyperref extensions.
Illustrations created in TikZ.

Front page design by Henrik Stuart.
Faculty of Natural Sciences logo by Pete Burke.

Text composed in 11/13 pt Adobe Minion Pro.
Titles composed in Adobe Myriad Pro.
Mathematics composed in MnSymbol.

Adobe Minion Pro was designed by Robert Slimbach in 1990. Adobe Myriad Pro was designed by Robert Slimbach & Carol Twombly with Fred Brady & Christopher Slye in 1992. MnSymbol was designed by Achim Blumensath.