Coccinelle

Tool support for automated CERT C Secure Coding Standard certification

Olesen, Mads Chr.; Hansen, Rene Rydhof; Lawall, Julia L.; Palix, Nicolas Jean-Michel

Published in:
Science of Computer Programming

DOI (link to publication from Publisher):
10.1016/j.scico.2012.10.011

Publication date:
2014

Document Version
Early version, also known as pre-print

Coccinelle: Tool support for automated CERT C Secure Coding Standard certification[☆,☆☆]

Mads Chr. Olesen[a], René Rydhof Hansen[a,*], Julia L. Lawall[b], Nicolas Palix[b]

[a]*Department of Computer Science, Aalborg University, Selma Lagerlöfs Vej 300, DK-9220 Aalborg East, Denmark.*
[b]*Department of Computer Science, University of Copenhagen (DIKU), Universitetsparken 1, DK-2100 Copenhagen East.*

Abstract

Writing correct C programs is well-known to be hard, not least due to the many low-level language features intrinsic to C. Writing secure C programs is even harder and, at times, seemingly impossible. To improve this situation the US CERT has developed and published a set of coding standards, the "CERT C Secure Coding Standard," that (currently) enumerates 122 rules and 180 recommendations, with the aim of making C programs (more) secure. The large number of rules and recommendations makes automated tool support essential for certifying that a given system complies with the standard.

In this paper we report on ongoing work on adapting the Coccinelle bug-finder and program transformation tool, into a tool for analysing and certifying C programs according to, *e.g.*, the CERT C Secure Coding Standard or the MISRA (the Motor Industry Software Reliability Association) C standard. We argue that such a tool must be highly adaptable and customisable to each software project as well as to the certification rules required by a given standard.

Furthermore, we present current work on integrating Clang (the LLVM C front-end) as a program analysis component into Coccinelle. Program analysis information, *e.g.*, from data-flow or pointer analysis, is necessary both for more precise compliance checking, *i.e.*, with fewer false positives, and also for enabling more complete checking, *i.e.*, with fewer false negatives, *e.g.*, resulting from pointer aliasing.

Keywords: automated tool support, CERT C Secure Coding, certification

[☆]Supported by the ISIS project (FTP grant number 274-08-0214).

[☆☆]This is an extended version of the paper "Clang and Coccinelle: Synergising program analysis tools for CERT C Secure Software Certification" that was presented at the 4th International workshop on Foundations and Techniques for Open Source Software Certification.

[*]Corresponding author

Email addresses: mchro@cs.aau.dk (Mads Chr. Olesen), rrh@cs.aau.dk (René Rydhof Hansen), Julia.Lawall@inria.fr (Julia L. Lawall), npalix@diku.dk (Nicolas Palix)

1. Introduction

Writing correct C programs is well-known to be hard. This is, in large part, due to the many programming pitfalls inherent in the C language, such as low-level pointer semantics, a very forgiving type system and few, if any, run time checks. Writing a *secure* C program is even more difficult, as witnessed by the proliferation of published security vulnerabilities in C programs: even seemingly insignificant bugs may lead to a complete compromise of security.

In an effort to improve the quality of security-critical C programs, the US CERT[1] organisation is maintaining and developing a set of rules and recommendations, called the *CERT C Secure Coding Standard* (CCSCS), that programmers should observe and implement in C programs in order to ensure at least a minimal level of security. The version of CCSCS currently under development enumerates 122 rules and 180 recommendations covering topics ranging from proper use of C preprocessor directives and array handling to memory management, error handling and concurrency. The sheer number of rules and recommendations makes it almost impossible for a human programmer to manually guarantee, or even check, compliance with the full standard. Automated tool support for compliance checking is therefore essential.

In this paper we describe work in progress on a prototype tool for automated CCSCS compliance checking. The tool is based on the open source program analysis and program transformation tool *Coccinelle* that has been successfully used to find bugs in the Linux kernel, the OpenSSL cryptographic library, and other open source infrastructure software [1, 2, 3]. Coccinelle is scriptable using a combination of a domain-specific language, called the *Semantic Patch Language* (SmPL), as well as OCaml and Python. The scripts, called *semantic patches*, specify search patterns partly based on syntax and partly on the control flow of a program. This makes Coccinelle easily adaptable to new classes of errors and new code bases with distinct API usage and code-style requirements.

Coccinelle however, does not perform program analysis in the traditional sense, *e.g.*, data-flow analysis or range analysis. For the purposes of program certification and compliance checking such analyses are essential, both to ensure soundness of the certification and to improve precision of the tool. For this reason we are currently working on integrating the *Clang Static Analyzer* with Coccinelle in order to enable Coccinelle to use the analysis (and other) information found by Clang. The Clang Static Analyzer is part of the C front-end for the LLVM project.[2] In addition to classic compiler support, it also provides general support for program analysis, using a monotone framework [4], and provides a framework for checking source code for (security) bugs. The emphasis in the source code checkers of the Clang project is on minimising false positives (reporting "errors" that are not really errors) and thus it is likely to miss some real error cases. To further enhance the program analysis capabilities of

[1] Formerly known as the US Computer Emergency Response Team (`http://www.cert.org`)
[2] `http://clang.llvm.org`

Clang, in particular for inter-procedural program analyses, we have integrated a library, called WALi[3] for program analysis using weighted push-down systems (WPDS) [5] into Clang.

The rest of the paper is organised as follows. In Section 2 we give an overview of the CERT C Secure Coding Standard including a brief description of the rule categories. Section 3 introduces and describes semantic patches and the basic functionality of Coccinelle. Section 4 illustrates how Coccinelle can be used as a compliance checker, by implementing a rule from each category as a semantic patch. Section 5 describes how the Coccinelle rules can benefit from having access to program analysis information. Section 6 discusses current and future work, including experiments and the integration of Clang and Coccinelle. Related work is discussed in Section 7. Finally, Section 8 concludes.

2. The CERT C Secure Coding Standard

The CERT C Secure Coding Standard (CCSCS) is a collection of rules and recommendations for developing secure C programs. The first version of the CCSCS was published in 2008 [6]. However, in this paper we focus on the version currently being developed as of May 5, 2011. The development process is a collaborative effort managed through the CCSCS web site.[4] This version of the CCSCS consists of 122 rules and 180 recommendations. The rules and recommendations are divided into 16 categories covering the core aspects of the C programming language. Figure 1 shows an overview of these categories and a summary of the number of rules and recommendations in each category.

2.1. Overview of the CCSCS

Experience shows that when programming in C, certain programming practices and language features, *e.g.*, language features with unspecified or compiler-dependent behaviour, result in insecure programs or, at the very least, in programs that are hard to understand and check for vulnerabilities. This experience is at the heart of the CCSCS. Many of the observed problems arise when programmers rely on a specific compiler's interpretation of behaviour that is undefined in the ANSI standard for the C programming language (ANSI C99). Other problems are caused, or at least facilitated, by the flexibility of the C language and the almost complete lack of run-time checks.

Based on the observed problems, the US CERT has identified a number of key issues and developed a set of *rules* that specify both how to avoid problematic features and also constructively how to use potentially dangerous constructs in a secure way, *e.g.*, programming patterns for securely handling dynamic allocation and de-allocation of memory. The rules in the CCSCS are almost all unambiguous, universal and generally applicable in the sense that they do not

[3] `http://www.cs.wisc.edu/wpis/wpds/`

[4] `https://www.securecoding.cert.org/confluence/display/seccode/CERT+C+Secure+Coding+Standard`

Code	Long name	# of Rules/Recomm.	
01-PRE	Preprocessor	3	14
02-DCL	Declarations and Initialization	11	21
03-EXP	Expressions	12	22
04-INT	Integers	6	18
05-FLT	Floating Point	7	7
06-ARR	Arrays	7	3
07-STR	Characters and Strings	9	11
08-MEM	Memory Management	6	13
09-FIO	Input Output	16	19
10-ENV	Environment	3	5
11-SIG	Signals	6	3
12-ERR	Error Handling	4	8
13-API	Application Programming Interfaces	N/A	8
14-CON	Concurrency	9	2
49-MSC	Miscellaneous	11	22
50-POS	POSIX	12	4

Figure 1: Categories in the CERT C Secure Coding Standard

depend on the specific application being developed. Furthermore the rules are, for the most part, formulated at the level of individual source files or even parts of source files and thus require little or no knowledge of the surrounding application or the context in which it is used. This makes the rules ideally suited for automated checking.

In addition to the above mentioned rules, the CCSCS also contains an even larger number of *recommendations*. The recommendations often represent the *best practice* for programming secure systems. In contrast to the rules, the recommendations are not limited to constructs that are local to a single file or function, but often also cover more global issues and design issues, such as how to handle sensitive information, how to use and implement APIs, how to declare and access arrays, and so forth. While most of the recommendations are still amenable to automated analysis, it may take more work and, in particular, it will require configuring and specialising the automated tool to the specific project being checked, *e.g.*, by specifying which data in the program may contain sensitive information or which macros are considered safe or how to canonicalize file names. An application is not required to follow the recommendations in order to be compliant with the CCSCS.

The CCSCS is much too large to cover in detail here. Instead, we give a brief overview of the different categories and the kind of (potential) errors they are designed to catch. In Section 4 we focus on how to check the rules using Coccinelle and discuss one rule for each category in detail.

2.2. Categories of the CCSCS

Preprocessor (01-PRE). The rules and recommendations in this category are concerned with proper use of the C preprocessor. Most (large) C projects use

preprocessor directives, especially macro definitions, extensively. Since these can dramatically change the "look" of a program, it is very important to avoid the many common pitfalls enumerated in this category.

Many static analysis tools are not very good at checking these rules since they typically work on the expanded code and thus do not even see the macros. This is unfortunate since a lot of semantic information can be gleaned from well-designed macros and their use.

Declarations and Initialization (02-DCL). The rules and recommendations in this category mostly cover tricky semantics of the type system and variable declarations, such as implicit types, scopes, and conflicting linkage classifications.

The recommendations in this category codify good programming practices, *e.g.*, using visually distinct identifiers (DCL02-C) and using typedefs to improve code readability (DCL05-C). While many of the recommendations can be automatically verified, others (like DCL05-C) require human interaction.

Expressions (03-EXP). The rules and recommendations in this category are concerned with issues related to expressions, including (unspecified) evaluation order, type conversions, sizes of data types, general use of pointers, and so forth.

Integers (04-INT). The rules and recommendations in this category are concerned with issues related to proper handling of integers. The main emphasis for the rules is on avoiding overflows and wrap-around for very large or very small integer values. Automated checking for these rules can be difficult since that may require sophisticated data flow or interval analysis. Alternatively, a tool can instead check that a program includes sufficient validation in the program itself to avoid the dangerous situations.

The recommendations are similarly concerned with conversions, limits and sizes of the integer types. Like the rules in this category, automated checking of the recommendations may require sophisticated program analysis.

Floating Point (05-FLP). The rules and recommendations in this category are concerned with issues relating to proper handling of floating point types: loss of precision, proper use of mathematical functions, and type conversion. Automated checking is at least as difficult as for the integer case.

Arrays (06-ARR). The rules and recommendations in this category focus on avoiding out of bounds array indexing and pointer access to arrays. Automated checking is likely to require pointer analysis in order to ensure correctness and to minimise false positives.

Characters and Strings (07-STR). The rules and recommendations in this category are concerned with: ensuring null termination of strings, proper size calculation of strings, and bounds checking for strings.

Memory Management (08-MEM). The rules and recommendations in this category cover some of the many pitfalls surrounding dynamic memory allocation, such as accessing freed memory, "double freeing" memory, freeing memory that was not dynamically allocated and so forth. Implementing memory management correctly is notoriously difficult and even small bugs in this category are likely to result in a security vulnerability, *e.g.*, a buffer overflow or a null pointer dereference.

Input Output (09-FIO). The rules and recommendations in this category are mainly concerned with the proper use of library functions for (file) input and output, including proper opening and closing of files, creation of temporary files, as well as secure creation of format strings.

Environment (10-ENV). The rules and recommendations in this category are concerned with proper handling of the execution environment, *e.g.*, environment variables, and calls to external command processors.

Signals (11-SIG). The rules and recommendations in this category are concerned with raising and handling signals in a secure manner, including ensuring that signal handlers do not call `longjmp()` and do not modify or access shared objects.

Error Handling (12-ERR). The rules and recommendations in this category are concerned with detecting and handling errors and proper handling of the `errno` variable. Examples include not modifying the `errno` variable and not relying on indeterminate values of `errno`.

Application Programming Interface (13-API). In the version of CCSCS currently under development, this category has no rules, only recommendations, since proper API design is highly application specific. Similar to the error handling (ERR) category above, automated tool support requires a very adaptable tool.

Concurrency (14-CON). The rules and recommendations in this category are general observations concerning concurrent programming such as avoiding race conditions and deadlocks by locking in a predefined order.

Miscellaneous (49-MSC). The rules and recommendations in this category are those that do not fit into any other category, *e.g.*, it is recommended to compile cleanly at high warning levels (MSC00-C) and it is a rule that a non-void function's flow of control should never reach the end of the function (MSC37-C).

POSIX (50-POS). The rules and recommendations in this category cover compliance with and proper use of POSIX. Examples include particular things to avoid doing with POSIX, such as calling `vfork()` or not using signals to terminate threads.

3. Coccinelle

In the following we introduce the Coccinelle tool, including its basic use and functionality. Further features and advanced functionalities will be introduced as needed in Section 4.

The Coccinelle tool was originally developed to provide support for documenting and automating updates to Linux device drivers necessitated by a change in the underlying API, the so-called *collateral evolutions* [7]. Finding the right place to perform collateral evolutions in a large code base requires a highly configurable and efficient engine for code searching. In Coccinelle this engine is based on model checking of a specialised modal logic, called CTL-VW, over program models [8] enabling search not only for specific syntactic patterns but also for control flow patterns. Individual program searches (and transformations) are specified in the SmPL domain-specific language [9, 10, 11, 12] designed to be similar to the unified patch format widely used by Linux kernel developers and other open source developers [13]. Such program searches are called *semantic patches* or even Coccinelle scripts. The combination of easy configurability and efficient search capabilities makes Coccinelle an excellent tool for searching for code patterns that may lead to potential bugs or violations of coding standards.

Coccinelle is released[5] as open source under the GNU GPLv2 license, and has been successfully used to search for bugs in open source infrastructure software such as the Linux kernel and the OpenSSL cryptographic library [1, 2]. As an indication of the efficiency and effectiveness of the tool, Coccinelle has currently been used to create more than 800 patches that have been accepted into the Linux kernel[6] which, as of version 3.2, comprises 37.000+ files and more than 15 million lines of code.[7] The successful application of Coccinelle to such a large code base, demonstrates the scalability of the tool architecture and the underlying parsing and model checking technologies. One key aspect of this is that the tool generally works on one source code function at a time, which limits the state-space explosion.

Figure 2 shows a semantic patch, that we will use to illustrate the basic features of SmPL, the domain-specific language for specifying semantic patches. Essentially, this semantic patch searches for all variable declarations in a program, in such a way that matches for integer variables can be separated from matches for non-integer variables (see below for more details), which results in a semantic patch that can be used to find all non-integer variable declarations and print the corresponding variable names and their types.

The semantic patch in Figure 2 comprises two *rules*: the first (lines 1–10) is written in SmPL and the second (lines 12–20) is written in OCaml using Coccinelle's scripting interface to OCaml. The former rule specifies the search

[5] `http://coccinelle.lip6.fr/` and `https://github.com/coccinelle/coccinelle`

[6] `http://git.kernel.org/?p=linux/kernel/git/next/linux-next.git`; search for any of the last two authors.

[7] `http://go.linuxfoundation.org/who-writes-linux-2012`

```
1  @ find_Tvars @
2  type T;
3  identifier id;
4  position pos;
5  @@
6  (
7    int id;
8  |
9    T id@pos;
10 )
```

```
11 // print message
12 @ script:ocaml @
13 t << find_Tvars.T;
14 x << find_Tvars.id;
15 p << find_Tvars.pos;
16 @@
17 print_endline
18   ("Found identifier"^ x ^
19    " of type "^ t ^
20    "at line "^ p[0].line)
```

Figure 2: Example semantic patch that searches for non-integer variable declarations.

while the latter rule prints the results of the search. In the rest of this section, we will present this example semantic patch in detail.

Line 1 defines the name '`find_Tvars`' of the semantic patch between a pair of `@`'s. The name is optional but it is useful for sharing information between semantic patch rules, *e.g.*, through *inheritance* as discussed below. In lines 2–4 three *meta-variables* are declared, *i.e.*, SmPL variables that will be bound by matching the underlying C code. The first meta-variable, '`T`', matches types; the second, '`id`', matches identifiers; and the third, '`pos`', is a special meta-variable that matches anything but will be bound to the corresponding *position* in the underlying code, *i.e.*, line and column number. Position meta-variables are useful both for reporting results, and as "anchors" that can be used in further searches. The '`@@`' in line 5 indicates the end of the meta-variable declarations.

The SmPL code from line 6 to line 10 is called a *disjunction*, and starts with a '`(`' in the first column (line 6) and ends with a '`)`' in the first column (line 10). *Disjuncts* are separated by '`|`', likewise in the first column (line 8). The symbols must be placed in the first column in order not to confuse the disjunction with C code that should be searched for.[8] Here the two disjuncts define searches for integer variable declarations (line 7) and *any* variable declaration (line 9) respectively. The latter also records the position in the source code where the variable declaration was found, as indicated by the '`@pos`' notation.

Disjunction works by looking for the *first* disjunct that matches and then skipping the rest of the disjuncts. This "short circuit" evaluation of disjuncts makes it possible to find, and possibly filter away, special cases of a more general pattern, as in this example. In our tour of the CCSCS rules, this pattern will be used in several cases, *e.g.*, in Section 4.6.

Line 12 indicates the start of an OCaml rule. The meta-variable declarations for this rule (lines 13–15) demonstrate how meta-variables can be *inherited* from a prior rule simply by qualifying the meta-variable name with the rule name from which to inherit, *e.g.*, '`find_Tvars.T`' corresponds to the '`T`' in the

[8] For cases where it is inconvenient to use the first column for disjunctions the following alternative syntax is provided: disjunctions can be specified (in any column) using '`(`', '`|`', and '`)`'. An example of this can be seen in Section 4.1.

```
@@
expression E;
identifier x;
identifier mname ~= "UNSAFE";
position pos;
@@
  mname@pos(...,
      <+... \( E++ \| E-- \| ++E \| --E \| x = E \|
               x += E \| x -= E \| x *= E \|
               x /= E \| x |= E \| x &= E \|
               x(...) \) ...+>,...)

// print warning for every binding of 'pos'
```

Figure 3: Semantic patch to find potential side-effects in arguments to unsafe macros (PRE31-C).

'`find_Tvars`' rule. The '`<<`' operator is used in meta-variable declarations to transfer the content of SmPL meta-variables into the scripting interface. In this case, inheritance is used to give the OCaml rule access to the search results of the SmPL rule in order to print the name, type, and position of the found (non-integer) variable declarations (lines 17– 20). Many of the semantic patches discussed in the next section follow a pattern similar to the one we have discussed here.

4. Compliance Checking with Coccinelle

In this section we exemplify how Coccinelle can be used as an automated checker for CCSCS rules by implementing a set of Coccinelle semantic patches, each checking one rule, for the 16 rule categories in the CCSCS.

4.1. PRE31-C: Avoid side-effects in arguments to unsafe macros

In contrast to most other program analysis tools for C code, Coccinelle generally works on the *unexpanded* source code, making it possible to search for code patterns that involve preprocessor directives. In particular, Coccinelle interprets macro calls when possible as function calls and `#ifdef`'s as conditionals.

The CCSCS defines an *unsafe macro* to be a macro whose expansion may evaluate its arguments more than once. A programmer may be unaware that a given macro is unsafe or even that it is a macro, since it may look and otherwise behave like a function. In this case, the programmer may use an expression with a side-effect as an argument to the (unsafe) macro and thereby potentially crash the program or leave it vulnerable to attack.

In general, it is recommended that unsafe macros are clearly labelled as such, *e.g.*, by prefixing or suffixing the macro name with "UNSAFE." For this simple case, where unsafe macros contain "UNSAFE" in their name, we can use the semantic patch in Figure 3 to check that rule PRE31-C is followed. This

seemingly simple semantic patch introduces a number of new features of SmPL and we will discuss these along with the general discussion of the semantic patch.

The semantic patch works by first looking for function calls where identifiers that contain the string "UNSAFE" are used as the name of a function (line 7) in a function call. This is achieved by declaring a meta-variable of type '`identifier`' (line 4) and qualifying it with a regular expression, in this case the constant string "UNSAFE", using the '`~=`' operator. This means that the '`mname`' meta-variable will only match identifiers that also match the given regular expression.

Next, the semantic patch checks if any of the arguments in the function call (potentially) has a side-effect (lines 8 to 11). Here an inlined disjunction, as discussed in Section 3, is used to explicitly enumerate all the possible forms such an argument can take.

Observe that the side-effecting expression may actually be a sub-expression within a larger expression, *e.g.*, in the expression '`42 + (x++)`' only the sub-expression the '`x++`' is side-effecting. In order to take this into account, the inlined disjunction is placed inside a *nesting* operator, starting with a '`<+...`' (line 8) and ending with a '`...+>`' (line 11). The nesting operator extends the search to also cover sub-expressions and searches for *at least* one occurrence of the pattern searched for, in this case all the possible side-effecting expressions. In addition to the nesting operator, a further SmPL idiom is needed to search through all the arguments of the function call: the SmPL '`mname(...,<+...x ...+>,...)`' will look for '`x`' inside any of the arguments to '`mname`'. Instead of looking for the occurrence of a single identifier in an argument, the semantic patch in Figure 3 looks for any side-effecting sub-expressions in any argument. Thus, the combined effect is to search for at least one occurrence of a side-effecting expression, possibly occurring as a sub-expression in some argument of the macro.

An alternative to looking for macros with special names is to maintain a database of names of unsafe macros and then perform a lookup in this database whenever a macro is used in a potentially unsafe way. The semantic patch developed below in Section 4.2 in part illustrates such an approach, where OCaml scripting is used to maintain and use a hash-table of identifiers for checking rule DCL32-C. Ideally, such a database of unsafe macros could be populated automatically by first searching for potentially dangerous macros. However, since there is no requirement that macro-bodies form legal statements or expressions, this is in general only feasible for a subset of macros.

4.2. DCL32-C: Guarantee that mutually visible identifiers are unique

The ANSI C99 standard for the C programming language specifies that *at least* 63 initial characters are significant in an identifier. Thus, identifiers that share a 63 character long prefix may be considered identical by the compiler. The DCL32-C rule requires that all (in scope) identifiers are unique, *i.e.*, must differ within the first 63 characters.

In Figure 4 the semantic patch is shown. It collects all identifiers of length 63 or more into a hash table and warns if there are (potential) violations of the

```
1  @ initialize:ocaml @
2  let idhash = Hashtbl.create 128
3
4  @ decl @
5  type T;
6  identifier id;
7  position pos;
8  @@
9    T id@pos;
10
11 @ script:ocaml @
12 p << decl.pos;
13 x << decl.id;
14 @@
15 if (String.length(x)) >= 63 then
16   let sid = String.sub x 0 63 in
17   if (Hashtbl.mem idhash sid) then
18     (* print warning *)
19   else
20     Hashtbl.add idhash sid (x,p)
```

Figure 4: Coccinelle script to find "long" identifiers (DCL32-C).

rule. The rule does not take the scope of the declared identifiers into account and thus may give rise to unnecessary warnings (false positives). However, since identifiers of length 63 or more are rarely used this is unlikely to be a problem in practice. If, for a specific project, it turns out to be a problem, the semantic patch can be extended to take more scope information into account. Note that the semantic patch in Figure 4 is *sound*, in the sense that it will catch all violations of rule DCL32-C. The semantic patch includes a simple OCaml script (lines 11 to 20) that collects all the found identifiers, of length 63 or more, and adds them to a hash table. Before adding an identifier to the hash table, it is checked for collisions with previously added identifiers, and thus potential violations. A warning is printed if there are (potential) collisions (line 18), otherwise nothing is printed.

The SmPL rule searching for declarations uses a position meta-variable (line 9) as discussed in Section 3. Here, the position meta-variable is used for reporting purposes: the position found in the code search is inherited by the OCaml script (line 12), meaning that in the OCaml script the variable '`p`' contains an OCaml representation of the position where a variable declaration with a long identifier was found.

4.3. EXP34-C: Do not dereference null pointers

In the CCSCS, the rationale for this rule is that attempts to dereference null pointers result in undefined behaviour. In recent years, attackers and vulnerability researchers have had great success in leveraging null pointer dereferences into

```
1  @@
2  identifier x, fld;
3  expression E,E1;
4  type T1;
5  statement S1, S2;
6  position pos;
7  @@
8  (
9    x = (T1) malloc(...)
10 |
11   x = (T1) calloc(...)
12 |
13   x = (T1) realloc(...)
14 )
15   ... WHEN != x = E
16       WHEN != if(E == NULL) S1 else S2
17 (
18   *x@pos
19 |
20   x@pos[E1]
21 |
22   x@pos->fld
23 )
24
25 // print warning for every binding of 'pos'
```

Figure 5: Coccinelle script to find dereferencing of null pointers due to insufficient checks on memory allocation (EXP34-C).

full blown security vulnerabilities, making this rule very important for application security. The current version of the CCSCS contains an example involving the Linux kernel and the `tun` virtual network driver.

One potential source of null pointers, as noted in the CCSCS examples, is when memory allocation functions, *e.g.*, `malloc()`, `calloc()`, and `realloc()`, fail and return null. If the return value from an allocation functions is not properly checked for failure, and handled accordingly, there is a high risk that a program will eventually, or can be made to, dereference a null pointer.

Figure 5 shows a semantic patch to find such code patterns. The semantic patch first looks for calls to the relevant allocation functions (lines 8 to 14). Here the '`...`' operator, used in the arguments for the allocation functions, means "with any number and type of arguments". It works like a specialised version of the function argument idiom discussion in Section 4.1 that does not search for anything specific in the arguments. The possible allocation functions are specified using a disjunction pattern as discussed in Section 3.

Following that, the script looks for a *control flow* path, represented by the '`...`' operator, where the identifier (`x`) is *not* reassigned (line 15) and where the identifier is not tested for "null-ness" (line 16). Note the different semantics for the '`...`' operator depending on the syntactic context where it occurs. In

order to cut down on the number of false positives, the semantic patch does not search along code paths that may change the "null-ness" of a variable. In SmPL this is formulated using the '`...`' operator qualified with '`WHEN !=x = E`' and '`WHEN !=if(E == NULL) S1 else S2`' meaning: along *any* control flow path where assignment to `x` does not occur, *i.e.*, any control flow path where `x` is not modified and which contains no null test on `x`.

Finally, we look for a dereference of `x` (lines 17 to 23), again using a disjunction pattern to specify three common ways to dereference a pointer: as a pointer (line 18), as an array (line 20), or as a field member access (line 22).

Note that, even though line 16 only explicitly considers conditionals with a condition of the form '`E==NULL`' (in line 16), Coccinelle will automatically also match variations of this condition such as '`NULL==E`', and '`!E`'. This feature is called *isomorphisms* and is a general, customisable, and scriptable feature of Coccinelle designed to handle syntactic variations of the same semantic concept, in this case, comparing a variable to the `NULL` pointer. Isomorphisms, while not strictly necessary, significantly reduce the amount of work a programmer has to do when developing a semantic patch. Isomorphisms are also useful in developing patches that are more complete (cover more cases) since corner and special cases need only be handled once.

While the semantic patch in Figure 5 catches many common violations of rule EXP34-C, it does not catch all possible violations. First, null pointers may come from many other places than the memory allocation functions, *e.g.*, user-defined functions and library functions. In principle it is of course possible to manually extend the semantic patch with all the functions possibly returning a null pointer, however, this quickly becomes unwieldy. Another drawback of the semantic patch, as shown, is that it overlooks violations occurring *after* a null test. It is possible to manually refine the semantic patch to take more tests into account. In [1] a more comprehensive Coccinelle approach to dereferencing of null pointers is described. This approach covers not only standard allocation functions, but any function returning null. In addition, some consideration is given to null tests and handling them properly.

Another alternative would be to enable the semantic patch to make use of information from a data-flow analysis. That way it would not be necessary to explicitly cover all syntactic possibilities for null testing or dereferencing. Many of the rules in this category (03-EXP) would likewise benefit from having access to information from program analyses such as pointer analysis, *e.g.*, rule EXP32-C, or data-flow analysis, *e.g.*, rule EXP41-C. In Section 5 we describe our current work on integrating analysis information into Coccinelle scripts.

4.4. INT30-C: Ensure that unsigned integer operations do not wrap

While it is not currently possible to perform complete checking of rules for secure handling of integers, Coccinelle can still be used to check that integer operations are suitably protected as suggested in the CCSCS, *e.g.*, by a pre- or post-condition on the integer operation in question ensuring that potential violations will at least be caught at run-time. The semantic patch in Figure 6 implements a simple, and very specific, search to find indexing into an array,

```
1  @ safe @
2  identifier arr;
3  unsigned int ui1, ui2;
4  position pos;
5  @@
6    if(UINT_MAX - ui1 < ui2) {
7      ...
8    } else {
9      ...
10     arr@pos[ui1 + ui2]
11     ...
12   }
13
14 @@
15 identifier arr;
16 unsigned int ui1, ui2;
17 position pos != safe.pos;
18 @@
19   arr@pos[ui1 + ui2]
```

Figure 6: Semantic patch to find unprotected integer additions in array indexing (INT30-C).

formed by the addition of integers that may potentially overflow. This is done by first finding and recording the *safe* array indexing instances (lines 1–12), *i.e.*, array indexing contained in a proper check for overflow, in order to rule them out in the search for potential violations. Next we look for all array indexing instances (lines 14–19), *except* for those that occur at a position we have previously recorded as being safe (line 17).

It is straightforward to generalise the semantic patch to cover more operations and other potentially dangerous uses. By using the program transformation capabilities of Coccinelle, such pre- or post-conditions can be *inserted* automatically into the source code of an application. While this is useful for program development or re-engineering, it is less useful for checking that an application is compliant with the CCSCS and therefore we do not go into further details here.

Most of the rules in the 04-INT category require data-flow analysis to fully check them. In particular, such information could be used to statically guarantee that certain integer expressions could not possibly overflow and thus eliminate the need for an explicit check.

4.5. FLP30-C: Do not use floating point variables as loop counters

Since the precision limits of floating point values are implementation/platform dependent, it can be non-portable to use floating point variables as loop counters. The general problem of automatically deciding which variables in a loop represent loop counters requires extensive program analysis. Instead of trying to catch every case, the semantic patch in Figure 7 looks for "suspicious"

```
1  @@
2  {float,double} x;
3  statement S;
4  position pos;
5  @@
6  (
7    for@pos(<+... x ...+>; <+... x ...+>; <+... x ...+>) S
8  |
9    while@pos(<+... x ...+>) {
10     <+... x ...+>
11   }
12 )
13
14 // print warning for every binding of 'pos'
```

Figure 7: Coccinelle script to find floating point loop counters (FLP30-C).

loops, *i.e.*, loops where a floating point variable is referenced in the loop condition and also in the loop body or the loop update expression for `while`-loops and `for`-loops respectively. In line 2 we use the notation '`{float,double}`' to declare that the '`x`' meta-variable matches any source code variable that is declared either as a '`float`' or as a '`double`'. Unfortunately, Coccinelle does not currently support `do`/`while` loops in semantic patches. Support for such loops has not been a priority for the Coccinelle project, mainly due to their relative scarcity in the Linux kernel and associated device drivers, the primary target for the Coccinelle project. However, support for them is expected in a future release of Coccinelle.

Similar to the rules for handling integers (Section 4.4), access to data-flow analysis information would enable more precise checking of the rules of this category (05-FLP) as well as checks for a wider range of potential violations.

4.6. ARR37-C: Do not add or subtract an integer to a pointer to a non-array object

Performing pointer arithmetic on pointers to non-array objects may result in stray or dangling pointers. In Figure 3 a semantic patch is shown that looks for such potentially dangerous pointer arithmetic. This semantic patch also relies on the short circuit semantics of the disjunction discussed in Section 3, *i.e.*, a disjunction matches if any of the disjuncts match and disjuncts are tried from left to right and top to bottom. Thus, pointer arithmetic performed on an array pointer will be matched by an early disjunct (and not used for anything else) in line 8, and any expression with pointer arithmetic matched by a later disjunct (lines 9–10) will involve a non-array pointer since those have already been "filtered out" by earlier disjuncts. The location (in code) of potential violations are recorded using the position meta-variable '`pos`' and later printed, *e.g.*, from an OCaml script.

Again for this category, 06-ARR, access to program analysis information would be very useful: data-flow analysis could be used to guarantee that array

```
@@
type T;
T[]  arr;
T  *p;
expression E;
position pos;
@@
  \( arr + E   \| arr - E \| arr += E \| arr -= E \|
    p@pos + E  \| p@pos - E \|
    p@pos += E \| p@pos -= E
  \)

// print a warning for every binding of 'pos'
```

Figure 8: Semantic patch to find pointer arithmetic on non-array pointers (ARR37-C).

```
@@
identifier str;
expression E1, E2;
position pos;
@@
  char str@pos[E1] = E2;

// print a warning for every binding of 'pos'
```

Figure 9: Semantic patch to find bounded and initialised character arrays (STR36-C).

indices are within bounds, *e.g.*, rules ARR30-C and ARR32-C, and pointer analysis could be used to handle aliased pointers, *e.g.*, rule ARR36-C and ARR37-C.

4.7. STR36-C: Do not specify the bound of a character array initialized with a string literal

Explicitly specifying the bound of a character array initialised with a string literal may result in a string (represented as a character array) that is not null-terminated, since it is common for a character array variable to be bounded by the number of characters in the initialiser string. Using such a string under the assumption that it is null-terminated can easily lead to security vulnerabilities. For this reason, rule STR36-C forbids the use of explicit bounds for character arrays that are initialised with a string literal.

A simple semantic patch, shown in Figure 9, can be used to find violations of rule STR36-C. The semantic patch looks for declarations of character arrays with an explicit bound (as represented by the Coccinelle meta-variable '`E1`' of type '`expression`') as well as an initialiser (represented by the meta-variable '`E2`' also of type '`expression`').

Many of the rules in this category (07-STR) require both data-flow and pointer analysis to be checked in a complete manner, *e.g.*, to determine that all accesses to a string are within bounds.

```
1  @@
2  identifier x;
3  expression E,E1;
4  function f;
5  identifier fld;
6  position pos;
7  @@
8    free(x);
9    ... WHEN != x = E
10 (
11   f(...,x@pos,...)
12 |
13   *x@pos
14 |
15   x@pos[E1]
16 |
17   x@pos->fld
18 )
19
20 // print warning for every binding of 'pos'
```

Figure 10: Semantic patch to find potential "use-after-free" violations MEM30-C.

4.8. MEM30-C: Do not access freed memory

In the C programming language, as in most programming languages, using the value of a pointer to memory that has been deallocated with the `free()` function results in undefined behaviour. In practice, reading from deallocated memory may result in crashes, leaks of information, and exploitable security vulnerabilities. Rule MEM30-C requires that deallocated memory will not be accessed. The problem underlying this rule is very similar to that described in rule EXP34-C (do not dereference null pointers): instead of focusing on null pointers, this rule covers all pointers that have been freed.

Figure 10 shows a Coccinelle script, covering some of the simple(r) cases of this rule. The script first looks for any identifier that occurs as an argument to `free()` (line 7). Following that, the script looks for a *control flow* path where the identifier (`x`) is *not* assigned to, *i.e.*, a path where it is not modified (line 8). Finally, using a disjunction search pattern, as explained earlier, the script looks for a *use* of the identifier that results in the actual violation. Four common uses are covered: used as an argument to a function (line 10), dereferenced as a pointer (line 12) or an array (line 14), and dereferenced for member field access (line 16).

Proper memory management is essential for security critical C programs and is enforced by the rules in this category (08-MEM). Using Coccinelle to find various types of bugs caused by flawed memory management has been studied extensively in the context of the Linux kernel [14, 15].

Pointer analysis could help catch rule violations that occur through the use of an aliased pointer.

```
@@
int i;
identifier c;
position pos;
@@
(
  i = \( fgetc(...) \| getc(...) \| getchar(...) \|
         fputc(...) \| putc(...) \| putchar(...) \|
         ungetc(...) \)
|
  c@pos = \( fgetc(...) \| getc(...) \| getchar(...) \|
             fputc(...) \| putc(...) \| putchar(...) \|
             ungetc(...) \)
)

// print warning for every binding of 'c@pos'
```

Figure 11: Semantic patch to capture assignment of return values of I/O functions to non-integer variables (FIO34-C).

4.9. FIO34-C: Use `int` to capture the return value of character IO functions

Exploiting a security vulnerability most often requires an attacker to present some special input to the program under attack, *e.g.*, specially crafted strings that overflow a buffer and overwrite important system information. The 15 rules of the input/output rule category (09-FIO) illustrate some of the difficulty of handling input/output in a secure manner.

Some of the character input/output functions in the C standard library return an integer representing the character read/written. As noted in the CCSCS, casting such an integer to a character (of type `char`) may result in a character being misinterpreted as the end-of-file (EOF) marker with potentially devastating results. Rule FIO34-C requires programmers to use integer variables to hold such return values. The semantic patch, shown in Figure 11, for checking rule FIO34-C is quite simple: first look for "non-violations" (assignments to a variable of type `int`) and ignore them (the first disjunct), next look for potential violations, *i.e.*, assignments to variables of type other than `int`. A standard isomorphism automatically expands the rule to allow assignments to types that are synonymous with '`int`', namely '`signed`' and '`signed int`', thus reducing the number of false positives. See Section 4.3 for more details about isomorphisms.

4.10. ENV32-C: All `atexit` handlers must return normally

In order to provide programmers with an opportunity to execute cleanup code upon program exit, the C programming language allows programmers to register exit handlers, *i.e.*, functions that are executed immediately before the program terminates. If an exit handler is not allowed to run to completion it may lead to undefined behaviour. In particular, exit handlers must not be terminated by a call to `exit()` or by invoking `longjmp`.

```
1  @ atexit_call @
2  identifier fn;
3  @@
4    atexit(fn)
5
6  @@
7  identifier atexit_call.fn;
8  position pos;
9  @@
10   void fn(void) {
11       ...
12 (
13       exit@pos(...);
14 |
15       longjmp@pos(...);
16 )
17       ...
18   }
19
20 // print warning for every binding of 'fn' and 'pos'
```

Figure 12: Semantic patch to capture `atexit` handlers with non-normal return (ENV32-C).

The semantic patch in Figure 12 first looks for functions that are registered as exit handlers using the `atexit()` function (lines 1–4). The *inheritance* mechanism of Coccinelle is used in line 7 to transfer the names of functions registered as exit handlers to the subsequent rule (lines 6–18) that checks the definitions of exit handlers for calls to `exit()` or `longjmp()`; the position of any such call is recorded in the position meta-variable named `pos` to be printed later (lines 13 and 15).

4.11. SIG30-C: Call only asynchronous-safe functions within signal handlers

Similar to the `atexit` handlers discussed in Section 4.10, signals can be used by an attacker to manipulate the control flow of a program. Proper use of signals and signal handlers is the topic of category 11-SIG.

The CCSCS defines an *asynchronous-safe* function as a function that can be safely called without side effects from within a signal handler context. Calling a function that is not asynchronous-safe from a signal handler may result in undefined behaviour and should be avoided.

The set of asynchronous-safe functions is application and platform specific. Therefore the semantic patch shown in Figure 13, starts by initialising a hash table from a file with the names of functions that are asynchronous-safe (line 2). In a manner similar to that used for rule ENV32-C, as shown in Section 4.10, the names of functions registered as signal handlers are first collected (lines 4–7). These function names are then used to find the definition of any signal handler (lines 9–18) and check that all functions called in the signal handler are in the

```
1  @ initialize:ocaml @
2  let idhash = (* read hash table from file *)
3
4  @ signal @
5  identifier fn;
6  @@
7    signal(...,fn)
8
9  @ handler exists @
10 identifier signal.fn;
11 identifier fnc;
12 position pos;
13 @@
14   void fn(void) {
15     ... WHEN any
16     <+... fnc@pos(...) ...+>
17     ... WHEN any
18   }
19
20 @ script:ocaml @
21 func << handler.fnc
22 @@
23 if not (Hashtbl.mem idhash func) then
24   (* print warning (using 'pos') *)
```

Figure 13: Semantic patch to capture calls within signal handlers to functions that are not asynchronous-safe (SIG30-C).

hash table of allowed asynchronous-safe functions. The latter is handled by an OCaml script that performs a simple lookup in the hash table (line 23).

In order to find and check all possible function calls inside the signal handlers, we use the '`exists`' keyword in the head of the '`handler`' rule (line 9) to specify that the current rule should look for the *existence* of a control flow path with the required property, rather than checking for the property along *all* control flow paths as done by default. In addition, we use the '`...`' operator before and after the nesting pattern looking for function calls, qualified with the '`any`' keyword (lines 15 and 17). This is done for the same reason as using the '`exists`' keyword in the header: to ensure that the search looks for the existence, along *any* path, of a function call.

4.12. ERR33-C: Detect and handle errors

The lack of proper exceptions in the C programming language means that error conditions have to be explicitly encoded and communicated to other parts of the program. Most often a run-time error in a given C function will be communicated by returning an *error value*, frequently -1 or `NULL`. Ignoring an error condition is likely to lead to unexpected or undefined behaviour, it is therefore essential that the return value is always checked for all calls to a

function that may return an error value and that any error condition is handled properly. Rule ERR33-C formalises this requirement.

This rule differs from most of the other rules in the CCSCS in that it is almost entirely *application dependent*, since it is up to each application or software project to decide how, specifically, error conditions are signalled, what error values are used, what they mean, and how they must be checked and handled. It is therefore impossible to come up with a single, or even a few, rules that will cover the entire spectrum of possibilities. Thus, for a tool to be useful and effective it *must* be very customisable in order to adapt it to project-specific coding styles and policies. We believe that the specialised semantic patch language (SmPL) used in Coccinelle provides an excellent, and highly adaptable, platform for developing project-specific rule checkers.

As an example of how Coccinelle can be customised for project-specific error-handling standards, we show in [2] how Coccinelle was used to find several bugs in some error handling code in the OpenSSL cryptographic library. Coccinelle has also been used to find flaws in the error handling code of the Linux kernel [1].

4.13. Application Programming Interfaces (13-API)

At the time of writing, the latest version of CCSCS does not define any rules for the *Application Programming Interface* category.

4.14. CON33-C: Avoid race conditions when using library functions

Several of the functions defined in the C standard library, including `getenv()` and `rand()`, are not guaranteed to be thread-safe and their use in multi-threaded programs may lead to race conditions that can potentially be leveraged to break security.

A semantic patch almost identical to that described in Section 4.11 can be used to find and flag uses of such library functions. For this rule the hash table would contain the names of the potentially *unsafe* library functions instead of the names of asynchronous-safe functions.

4.15. MSC37-C: Ensure that control never reaches the end of a non-void function

Non-void functions are required to return a value, using a `return` statement. Using the return value of a non-void function where control flow reaches the end of the function, *i.e.*, without having explicitly returned a value, results in undefined behaviour. For this reason the CCSCS specifies that all control flows in a non-void function *must* end in a (non-empty) return statement.

Figure 14 shows a semantic patch, that finds non-void functions with a control flow path not ending in a non-empty return statement. The overall strategy for this search is to first find all `void` functions (line 1 to 7), *i.e.*, functions that are not supposed to return a value, in order to rule them out in our search. Next, we find all function declarations *except* for those declarations we have earlier identified as declaring `void` functions (line 13). Once such a

```
1  @ voidfunc @
2  function voidfn;
3  position voidpos;
4  @@
5    void  voidfn@voidpos(...) {
6      ...
7    }
8
9  @ func disable ret exists @
10 type T;
11 expression E;
12 function fn;
13 position pos != voidfunc.voidpos;
14 @@
15   T  fn@pos(...) {
16     ... WHEN != return E;
17   }
18
19 // print warning for every binding of fn and pos
```

Figure 14: Semantic patch to find non-void functions without a `return` statement (MSC37-C).

function declaration is found, we start looking for the existence of a control flow path that does *not* contain a `return` statement (line 16).

Observe that the head of the latter search pattern (line 9) not only contains the name of the search pattern (`func`) but also a directive to Coccinelle that it should disable the use of the '`ret`'-isomorphism (cf. the discussion of isomorphisms in Section 4.3) in order to avoid unwanted, potential interference from the isomorphism system. Similar to rule SIG30-C (see Section 4.11), the header also specifies that the current rule should look for the existence of a control flow path with the required property, instead of requiring it to hold over all control flow paths, as done by default, since we have a potential violation if there is even a single control flow path without a `return` statement.

The problem caught by the semantic patch in Figure 14 is inherently syntactic and control-flow based, and thus is very well suited for Coccinelle searches. Furthermore, checking for violations can be done in a universal and application-independent way.

4.16. POS33-C: Do not use `vfork()`

The POSIX category (50-POS) is concerned with guidelines for proper use of POSIX functions and it is *not* part of the core CCSCS. It is included to show how other coding standards can be integrated with the CCSCS.

According to the CCSCS, the POSIX '`vfork()`' function to create a child process must not be used, due to numerous potential problems, including race conditions, security issues and undefined behaviour. A trivial semantic patch simply searches for uses of the '`vfork()`' function and prints a warning for every use:

```
@@
position pos;
@@
  vfork@pos(...)

// print warning for every binding of 'pos'
```

The CCSCS explicitly recommends always using '`fork()`' function rather than the '`vfork()`' function. As already mentioned in Section 4.4, Coccinelle can be used to automatically perform program transformations. In this case it is possible to replace all calls to '`vfork()`' with calls to '`fork()`':

```
@@
param list params;
@@
-  vfork(params)
+  fork(params)
```

In general, automatic program transformation is not useful for verifying that a program is compliant with the CCSCS, but in this particular case where the problem identification and the solution are unambiguous and context independent it can be a very useful and effective tool.

5. Adding Program Analysis Information

From the discussion in the previous section of the categories and how specific rules can be checked using Coccinelle, it should be clear that while Coccinelle is useful for compliance checking it would benefit greatly from having access to more program analysis information, *e.g.*, for more precise and comprehensive tracking of potential null pointers. Such information could also be used to make checkers more succinct and efficient because fewer syntactic cases would need to be covered.

In this section we first discuss how SmPL, the domain-specific language used for writing semantic patches, could be extended in the future to make program analysis information available in an intuitive and easy to use manner. We have not (yet) implemented or evaluated these changes to SmPL, and thus we also show how program analysis information can be made available in semantic patches through the scripting interface of Coccinelle. This enables access to analysis information without having to change SmPL, allowing for rapid integration and evaluation of new analyses. Finally, we describe an experimental integration of an external program analysis engine into Coccinelle, namely the Clang Static Analyzer.

5.1. Extending SmPL with Analysis Information

Consider the rule EXP34-C (do not dereference null pointers) in Figure 5. Here the problem is to find all expressions that may potentially dereference

```
1 @@
2 identifier x, fld;
3 expression E1;
4 analysis[null] NINF;
5 @@
6 ( *x@NINF
7 | x@NINF[E1]
8 | x@NINF->fld
9 )
```

Figure 15: Semantic patch using analysis information to find `null`-pointers.

a null pointer. With access to pointer analysis information, every expression that may result in a null pointer can be found and tagged. Note that this is independent of how an expression may result in a null pointer, *i.e.*, it is no longer necessary to explicitly track information only from allocation functions in the semantic patch, since this is handled by the analysis.

In Figure 15 we show how such analysis information could be incorporated into a semantic patch, essentially by adding a new type of SmPL meta-variables, declared with the '`analysis`' keyword (line 4 in Figure 15). The '`analysis`' declaration (line 4) declares a meta-variable, called `NINF`. This meta-variable is used in much the same way as position meta-variables: by "tagging" an expression with the '`NINF`' meta-variable, *e.g.*, like '`x`' in line 7, only expressions that match the syntax (in this case an array) and that may also result in a null pointer are matched by the semantic patch.

The semantic patch is intended to illustrate one possible way to make analysis information available to semantic patches. In Section 5.2 we discuss a different approach that uses the Python interface of Coccinelle and therefore does not require changes or additions to SmPL.

By relying on the isomorphisms of Coccinelle (see the discussion in Section 4.3) to expand a simple pointer dereference (see line 5 in the semantic patch below) to cover all possible forms of pointer dereferences, we can use the following semantic patch to find all (sub-)expressions that are both potentially dereferencing and also tagged as potentially resulting in a null pointer:

```
1 @@
2 expression E;
3 analysis[null] NINF;
4 @@
5   *E@NINF
```

Since pointers in C may be *aliases* for the same location in memory, it is important that the pointer analysis not only tracks potential null-pointers but also tracks all potentially aliasing pointers. This is often called an *alias analysis* or a *points-to analysis*. Such analysis information would be useful in many other situations, *e.g.*, in the rule MEM30-C (do not access freed memory) where access may occur through an alias.

```
1  @@
2  identifier x, y;
3  expression E,E1;
4  function f;
5  identifier fld;
6  analysis[alias] xyalias;
7  @@
8    free(x@xyalias);
9    ... WHEN != y@xyalias = E
10 (
11   f(...,y@xyalias,...)
12 |
13   *y@xyalias
14 |
15   y@xyalias[E1]
16 |
17   y@xyalias->fld
18 )
```

Figure 16: Semantic patch using alias analysis information to search for "use-after-free" bugs.

Figure 16 shows a semantic patch which integrates alias analysis information and searches for pointer dereferences that may access freed memory. The idea is that we first declare an analysis meta-variable in line 6 (called '`xyalias`'). Then, in line 8, we match a call to '`free()`' on an identifier '`x`' and bind the '`xyalias`' meta-variable to any available alias analysis information for '`x`'. Following that we match any assignments to any use of *any identifier* '`y`' that is an *alias* for '`x`' (represented by '`y@xyalias`' in lines 9, 11, 13, 15, and 17).

5.2. Integrating Clang and Coccinelle

Clang was chosen as the external program analysis engine for Coccinelle for several reasons: it is open source, it is being (very) actively developed, it has good support for writing new analyses, and it provides a robust and proven infrastructure for manipulating C programs.

In this prototyping phase, the emphasis has been on tool integration rather than extending SmPL. Consequently, it is not yet possible to use the '`analysis`' declaration illustrated in the last section. Instead we use positions, as implemented by the '`position`' meta-variables, to look up relevant analysis information (see below for details).

For the prototype we run Clang as a pre-processor to Coccinelle, computing the relevant program analysis information and storing it in a file, which may subsequently be read by a semantic patch. This approach, using Python scripting, is illustrated by the semantic patch shown in Figure 17.

5.2.1. Information Exchange Format

In order to exchange analysis information between Clang and Coccinelle, it is necessary to define an exchange format that can adequately and flexibly

```
@ initialize:python @
  # read analysis information generated by Clang into
  # Python dictionaries: NINF indexed by positions

@ expr @
expression E;
position pos;
@@
  *E@pos

@ script:python @
p << expr.pos
@@
  # lookup NINF status in Clang data
  if not (NINF[p]):
    # no match: the expression at position 'p'
    # cannot result in a null pointer
  else:
    # print warning
```

Figure 17: Example semantic match showing prototype integration of Clang data into Coccinelle using Python scripting.

encode both the control flow information of the program as well as abstract analysis information from a wide range of program analyses. To ensure maximum flexibility and future extensibility, we have chosen to use the framework of weighted push-down systems [16] (WPDS) as the exchange format. WPDSs have previously been proposed as a generalisation of monotone frameworks for program analysis and have been used to specify and implement a number of program analyses [5, 17]. By changing the weight domain of a WPDS modelling a program, different analyses can be obtained, such as affine-relations analysis, generalised gen-kill analysis, and may-aliasing pointer analysis as defined in [17].

In order to follow this approach we have extended Clang with the analysis framework of WPDSs, using the WALi library.[9] This enables us to model the control-flow extracted from Clang as a WPDS, plug in a weight domain suitable for describing the analysis of interest, and use the WPDS model checking algorithms to compute the analysis result. Specifically, we have used the gen-kill weight domain to implement a reaching definitions analysis within Clang. The analysis result can then be pre-processed in Coccinelle scripts, as illustrated above, *e.g.*, to get maybe-null analysis information.

The WPDS extension for Clang is written as a special analysis pass that constructs the WPDS, assigns weights, and performs a query for each function. The analysis results (annotated weighted finite automata) are output, and subsequently interpreted by the Coccinelle script when analysis information is

[9] `http://research.cs.wisc.edu/wpis/wpds/index.php`.

needed. Currently the Python scripting interface is used with some additional support code for calling Clang and interpreting the output.

The information that we have so far integrated is the reaching definitions analysis. The output from Clang is a textual representation of the solution (represented by *weighted finite automaton*). An example of one line of this output is:

```
( p , ( uninit_use.c , ( 5 , 9 ) ) , accept )
  <\S.(S - {NULL}) U
   {(simple:a@uninit_use.c:3:9@uninit_use.c:4:9,1)}>
```

The output lines from Clang are composed in the following way:

"From" state of the WPDS.

Symbol in this case "*(uninit_use.c , (5 , 9))*" indicating the program point.

"To" state which will always be the accepting state *accept*.

The weight associated with this transition, which is the program analysis information associated with the program point.

The weight needs to be parsed by the Coccinelle interface to Clang, in this case into its component "gen" and "kill" sets. In the above example the kill set is empty, and the gen set adds a definition point of the variable *simple:a@uninit_use.c:3:9*, namely that it can be defined at *uninit_use.c:4:9*. Variables are referenced as the concatenation of the name of the function they are defined in, their identifier and the position they are declared at. All positions are made up of a file name, a line number, and a column number. Finally, a dictionary data structure is constructed such that the reaching definitions for a variable at a program point can be looked up.

One use is to look for uses of uninitialised variables, where the basic semantic patch is shown in Figure 18.

We can then discard false positive matches, based on whether the data can actually flow from the found definition to the found use, in a somewhat cleaner way than specifying all possible ways the variable could have been modified. The approach of course becomes much more powerful when including analysis information from a pointer analysis.

6. Discussion and Future Work

In this section we discuss the current status and work-in-progress for using Coccinelle to check for CCSCS compliance. In particular we discuss an improved approach to integrating Clang and Coccinelle, as well as compliance checking of the full standard for real world software projects.

```
@ uninituse @
type T; identifier I;
position defloc, useloc;
identifier FN;
@@
// look for declarations with no assignment
T@defloc I;
... WHEN any
//which are then used
(
    FN@useloc(...,I,...);
|
    I@useloc
)
```

Figure 18: Semantic patch enhanced with def/use analysis information to find potential uses of uninitialised variables.

6.1. Clang in Coccinelle

Our prototype gave us some experience and insights to guide further integration between Clang and Coccinelle:

- Not all potentially useful information fits nicely into our exchange format, *e.g.*, which cases an enum contains.
- We would like to take advantage of more of Clang's features: for example Clang has a path-based reachability engine that we could use to rule out false positives.
- The fact that we use Clang as a pre-processor means we need to find all possibly needed information, before Clang exits, and then read it into Coccinelle. We would not be able to do an incremental analysis, only asking for smaller pieces of information at a time.

Based on these insights we came up with a second approach: Instead of using Clang as a pre-processor, it should be run directly under the control of Coccinelle. This can be achieved by exploiting the existing Python or OCaml bindings provided by Clang for internal data structures, and using these directly in semantic patches. In this way, instead of having two separate processes, Clang would be running within the address space of the Coccinelle process. This could solve the problem of defining an exchange format: all the information available to Clang will also be available directly through the Python or OCaml scripting interface in the semantic patch.

In addition we would be able to make calls to different parts of the modular Clang code base, for exploiting the internal parts of Clang in specific cases when it makes sense. We would also be able to much more closely query for only the information we need, and allow for an incremental analysis.

One disadvantage of this approach would be that the semantic patches might become very dependent on the internal structure of Clang (exposed through the

Python/OCaml bindings), which naturally is not guaranteed to be stable by the Clang developers. This could be alleviated to some degree by defining abstract interfaces at the Python/OCaml level that enable access to the most commonly used analysis information from Clang in a uniform way. In this way only the interfaces would have to be maintained. However, since we would not have a data exchange format it will be harder to integrate with other tools, *e.g.* specialised analysis tools or checkers, since each tool will have to be handled separately.

We are currently in the process of creating the Python bindings and the library for a common interface. We are also investigating how best to exploit the different features present in the Clang code-base. The bindings are created using SWIG,[10] an "interface compiler" that can automatically generate wrapper code for accessing C/C++ code from scripting languages such as Python, with the goal of reducing the maintenance burden as the Clang code base evolves.

6.2. Compliance Checking Real World Software

Coccinelle has already been used successfully to find numerous bugs in the Linux kernel, the OpenSSL library, and other open source projects used in the "real world." In particular, the experience with bug finding in the Linux kernel shows that the approach scales well even to very large software projects.

One of the biggest problems when checking such large projects is the number of *false positives*, *i.e.*, warnings of potential violations that turn out not to be violations. Here the customisability of Coccinelle has turned out to be very beneficial for reducing the number of false positives, since it enables a programmer to refine the semantic patches to take project-specific coding style into account.

The integration of program analysis information, *e.g.*, obtained from Clang, will enable a code search to take (more) semantic information into account and will thus reduce the number of false positives further.

6.3. Implementing Checkers for the Full Standard

While we have only detailed the implementation of Coccinelle checkers for one rule from each category, we plan to implement Coccinelle checkers for all the CCSCS rules that are suitably application independent. For rules that are application dependent, such as rule ERR33-C (discussed in Section 4.12), it seems possible to provide an "abstract" semantic patch that can be instantiated with project specific details, similar to the approach taken in [2].

We intend to make the complete set of checkers available for download as open source.

7. Related Work

The past decade has seen the development of numerous compile-time tools that can perform a wide range of anlysis, validation, and verification tasks,

[10] `http://www.swig.org`

including program comprehension, fault localisation ("bug hunting"), and code style checking. This includes both open source tools as well as commercial offerings, with the tool suites available from IBM[11] (Rational Software), Coverity,[12] Klockwork,[13] and Fortify[14] as notable examples of the latter.

The commercial tools mentioned above all support checking CCSCS rules to a varying degree[15] and can all be extended and adapted to project-specific coding styles and requirements. However, we are not aware of any commercial tools that provide a domain specific language [9, 11, 12] in the style of SmPL with a strong focus on code search and program transformation.

In addition to the commercial tools, there are also a number of excellent open source tools available for checking various properties of source code. For checking CCSCS compliance, the ROSE compiler infrastructure[16] is of particular interest since CERT, the organisation developing and publishing the CCSCS, is actively developing freely available rule checkers for CCSCS using this framework.[17] Since ROSE is a compiler infrastructure, rather than a source checking tool, it mainly provides access to an advanced and easy to use intermediate representation that facilitates program analysis and program transformation. As an obvious consequence, checkers built on top of ROSE must be programmed in one of the general programming languages supported by the infrastructure, e.g., C or Haskell. While this is certainly feasible, we believe that a dedicated domain specific language like SmPL is more convenient and easer to use, especially for developers that are not compiler experts.

In [18] an extension of the GNU Compiler Collection (GCC) is described that enables a developer to use the GCC to check for compliance with a set of coding rules. The coding rules can be specified by a variant of Prolog. Using the GCC as a basis for a compliance checker has the advantage that several languages can be targeted at once, since the GCC has multiple front-ends that uses the same mid- and back-ends. The use of logic programming to specify the coding rules of interest allows for succint and declarative rule specifications. However, it obviously requires developers or reviewers that are familiar with this programming paradigm to develop project-specific rule sets.

Splint [19] and Flawfinder [20] are two examples of open-source "bug hunters". Both are able to check for a relatively small fixed set of bugs and both are somewhat adaptable but require either (light-weight) annotation of the source code or Python programming.

Other open source tools of relevance for compliance checking are tools for

[11] `http://www.ibm.com/software/rational/`
[12] `http://www.coverity.com`
[13] `http://www.klockwork.com`
[14] `http://www.fortify.com`
[15] See the CCSCS web page for up-to-date status on tool support.
[16] `http://http://www.rosecompiler.org/`
[17] `https://www.securecoding.cert.org/confluence/display/seccode/ROSE+Checker+Code`

general software quality management, e.g., SonarSource.[18] Such high-level tools are useful for tracking the results and status of (other) compliance checking tools and monitor progress in a project. It would be quite feasible and interesting to integrate Coccinelle with SonarSource or similar tools to enable the use of Coccinelle in continuous monitoring of software quality. SonarSource provides an API for access to an intermediate representation, essentially an abstract syntax tree, of the monitored C code and, similar to the ROSE compiler infrastructure, this can be used to program additional code checking rules.

Finally, one could consider the use of *proof-carrying code* [21] (PCC) in conjunction with standards certification. While PCC is generally aimed at formally proving properties of programs at a semantic level, and not so much at the level of coding standards, vendors could conceivably embed sufficient proof into their products that a certain standard is followed and thus simplify later verification of this fact.

8. Conclusion

In this paper we have shown that the Coccinelle tool and the underlying notion of semantic patches are very well suited for checking the structural and application independent rules of the CCSCS. We have demonstrated this by developing a semantic patch for one rule in each of the categories of the CCSCS. We have further argued that the use of SmPL, a domain specific language dedicated to specifying semantic patches for code searches and transformations, makes it very simple for developers to adapt, configure, and extend the tool to cover project-specific needs and to check for application-specific CCSCS rules, e.g., rule ERR33-C (see Section 4.12).

In addition to the *rules* of the CCSCS, that *must* be followed to obtain certification, the standard also defines a large number of *recommendations*, i.e., additional coding practices that should be followed but are not mandatory. While the current work focuses on checking the rules, we observe that many of the recommendations follow the same form. Thus, we belive that Coccinelle could be applied equally well to checking the CCSCS recommendations.

Finally, we have argued that making program analysis information directly available in the semantic patches would facilitate writing more comprehensive and more precise semantic patches. We have suggested how SmPL can be extended to allow SmPL programmer to make use of program analysis information in a natural way in order to easily exploit this information in semantic patches. We have also shown an alternative way to make this information available, through the scripting interface of Coccinelle, without extending SmPL.

The Coccinelle tool can be downloaded as open source code from the Coccinelle web-site: '`http://coccinelle.lip6.fr/`' and is currently available in a number of Linux distributions.

[18] `http://www.sonarsource.com`

References

[1] J. L. Lawall, J. Brunel, N. Palix, R. R. Hansen, H. Stuart, G. Muller, WYSIWIB: A declarative approach to finding API protocols and bugs in Linux code, in: Proceedings of the 2009 IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2009, IEEE, Estoril, Lisbon, Portugal, 2009, pp. 43–52.

[2] J. L. Lawall, B. Laurie, R. R. Hansen, N. Palix, G. Muller, Finding error handling bugs in OpenSSL using Coccinelle, in: Eighth European Dependable Computing Conference, EDCC-8, IEEE Computer Society, Valencia, Spain, 2010, pp. 191–196.

[3] N. Palix, J. L. Lawall, G. Muller, Tracking code patterns over multiple software versions with Herodotos, in: J.-M. Jézéquel, M. Südholt (Eds.), Proceedings of the 9th International Conference on Aspect-Oriented Software Development, AOSD 2010, ACM, Rennes and Saint-Malo, France, 2010, pp. 169–180.

[4] J. B. Kam, J. D. Ullman, Monotone data flow analysis frameworks, Acta Informatica 7 (1977) 305–317.

[5] T. Reps, S. Schwoon, S. Jha, D. Melski, Weighted pushdown systems and their application to interprocedural dataflow analysis, Science of Computer Programming 58 (1-2) (2005) 206–263.

[6] R. C. Seacord, The CERT C Secure Coding Standard, Addison-Wesley, 2008.

[7] Y. Padioleau, J. L. Lawall, R. R. Hansen, G. Muller, Documenting and automating collateral evolutions in Linux device drivers, in: J. S. Sventek, S. Hand (Eds.), Proceedings of the 2008 EuroSys Conference, ACM, Glasgow, Scotland, UK, 2008, pp. 247–260.

[8] J. Brunel, D. Doligez, R. R. Hansen, J. L. Lawall, G. Muller, A foundation for flow-based program matching: using temporal logic and model checking, in: Z. Shao, B. C. Pierce (Eds.), Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, ACM, Savannah, GA, USA, 2009, pp. 114–126.

[9] J. Sprinkle, M. Mernik, J. Tolvanen, D. Spinellis, What kinds of nails need a domain-specific hammer?, IEEE Software 26 (4) (2009) 15–18.

[10] M. J. V. Pereira, M. Mernik, D. da Cruz, P. R. Henriques, Program comprehension for domain-specific languages, Computer Science and Information Systems 5 (2) (2008) 1–17.

[11] M. Mernik, J. Heering, A. Sloane, When and how to develop domain-specific languages, ACM Computing Surverys 37 (4) (2005) 316–344.

[12] A. Deursen, P. Klint, J. Visser, Domain-specific languages: An annotated bibliography, ACM SIGPLAN Notices 35 (6) (2000) 26–36.

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

[14] H. Stuart, Hunting bugs with Coccinelle, Master's thesis, Department of Computer Science (DIKU), University of Copenhagen (2008).

[15] H. Stuart, R. R. Hansen, J. Lawall, J. Andersen, Y. Padioleau, G. Muller, Towards easing the diagnosis of bugs in OS code, in: 4th Workshop on Programming Languages and Operating Systems (PLOS 2007), Stevenson, WA, USA, 2007.

[16] S. Schwoon, S. Jha, T. Reps, S. Stubblebine, On generalized authorization problems, in: Proc. 16th IEEE Computer Security Foundations Workshop (CSFW 2003), 2003, pp. 202–218.

[17] T. Reps, A. Lal, N. Kidd, Program analysis using weighted pushdown systems, in: Proceedings of the 27th international conference on Foundations of software technology and theoretical computer science, FSTTCS'07, Springer-Verlag, 2007, pp. 23–51.

[18] G. Marpons, J. Mariño, M. Carro, Á. Herranz, L.-Å. Fredlund, J. J. Moreno-Navarro, Á. Polo, A coding rule conformance checker integrated into GCC, Electronic Notes in Theoretical Computer Science 248 (2009) 149–159, proceedings of the Eighth Spanish Conference on Programming and Computer Languages (PROLE 2008).

[19] D. Larochelle, D. Evans, Statically detecting likely buffer overflow vulnerabilities, in: Proc. of the 10th USENIX Security Symposium, USENIX, Washington D.C., USA, 2001, pp. 177–190.
URL `http://lclint.cs.virginia.edu/`

[20] D. Wheeler, Flawfinder web page, Published online (Oct. 2006).
URL `http://www.dwheeler.com/flawfinder/`

[21] G. C. Necula, Proof-Carrying Code, in: Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 1997, ACM, Paris, France, 1997, pp. 106–119.