Electronic Communications of the EASST
Volume 33 (2010)

Proceedings of the
Fourth International Workshop on
Foundations and Techniques for
Open Source Software Certification
(OpenCert 2010)

Clang and Coccinelle: Synergising program analysis tools for CERT C Secure Coding Standard certification

Mads Chr. Olesen, René Rydhof Hansen, Julia L. Lawall, Nicolas Palix

18 pages

Guest Editors: Luis S. Barbosa, Antonio Cerone, Siraj A. Shaikh
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Clang and Coccinelle: Synergising program analysis tools for CERT C Secure Coding Standard certification

Mads Chr. Olesen[1], **René Rydhof Hansen**[1], **Julia L. Lawall**[2], **Nicolas Palix**[2]

[1]rrh,mchro@cs.aau.dk, http://www.cs.aau.dk
[2]julia,npalix@diku.dk, http://www.diku.dk

Abstract: Writing correct C programs is well-known to be hard, not least due to the many language features intrinsic to C. Writing secure C programs is even harder and, at times, seemingly impossible. To improve on this situation the US CERT has developed and published a set of coding standards, the "CERT C Secure Coding Standard", that (in the current version) enumerates 118 rules and 182 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 is in compliance with the standard.

In this paper we report on ongoing work on integrating two state of the art analysis tools, Clang and Coccinelle, into a combined tool well suited for analysing and certifying C programs according to, e.g., the CERT C Secure Coding standard or the MISRA (the Motor Industry Software Reliability Assocation) C standard. We further 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.

Clang is the C frontend for the LLVM compiler/virtual machine project which includes a comprehensive set of static analyses and code checkers. Coccinelle is a program transformation tool and bug-finder developed originally for the Linux kernel, but has been successfully used to find bugs in other Open Source projects such as WINE and OpenSSL.

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

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 and compilers, 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 or "small" 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*

[1] Formerly known as the US Computer Emergency Response Team (www.cert.org)

Coding Standard (CCSCS), that programmers should observe and implement in C programs in order to ensure at least a minimal level of security. The current version of the CCSCS enumerates 118 rules and 182 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 [LBP$^+$09, LLH$^+$10, PLM10], and other open source infrastructure software. Coccinelle is scriptable using a combination of a domain specific language, called SmPL for *Semantic Patch Language*, as well as in O'Caml and Python. The scripts 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 codebases with distinct API usage and code style requirements. Coccinelle does not perform program analysis in the traditional sense, e.g., data flow analysis or range analysis. However, 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 integrate 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 frontend for the LLVM project[2]. In addition to classic compiler support, it also provides general support for program analysis, using the monotone framework, and 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 is likely to miss some real error cases. To further enhance the program analysis capabilities of 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)[RSJM05] into Clang. To enable rapid prototyping and development of new or specialised analyses, we have implemented Python bindings for the WALi library.

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 illustrates how a few of the coding rules can be automatically checked using the Coccinelle tool. Section 4 describes how the Coccinelle rules can benefit from having access to program analysis information. Section 5 discusses current work in progress, including experiments and the integration of Clang and Coccinelle. Finally Section 7 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. One version of the CCSCS was published in 2008 as [Sea08].

[2] Web: http://clang.llvm.org

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

Code	Short name	Long name	# of Rules	# of Recomm.
01	PRE	Preprocessor	4	12
02	DCL	Declarations and Initialization	9	22
03	EXP	Expressions	11	21
04	INT	Integers	7	18
05	FLT	Floating Point	8	6
06	ARR	Arrays	8	3
07	STR	Characters and Strings	9	12
08	MEM	Memory Management	6	13
09	FIO	Input Output	15	20
10	ENV	Environment	3	5
11	SIG	Signals	6	3
12	ERR	Error Handling	4	8
13	API	Application Programming Interfaces	N/A	10
14	CON	Concurrency	6	2
49	MSC	Miscellaneous	10	23
50	POS	POSIX	12	4

Figure 1: Categories in the CERT C Secure Coding Standard

However, in this paper we focus on the version currently being developed. The development process is collaborative through the CCSCS' web site[4]. The current version[5] of the CCSCS consists of 118 rules and 182 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 practises 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

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

[5] Last checked 6 July 2010

almost all unambiguous, universal and generally applicable in the sense that they do not 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 potentially ideally suited for automated checking, although see Section 3 for a more detailed discussion of this.

In addition to the above mentioned rules, the CCSCS also contains an even larger number of *recommendations*. The recommendations often represent the *best practise* 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 may also cover more global 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 that are considered safe or how to canonicalize file names. A programmer 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 3 we focus on a few specific rules and discuss them in more 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 at least 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 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 practises, 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 while 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 orders, type conversions, sizes of data types, general use of pointers, and so forth.

Below we show how rule EXP34-C (do not dereference null pointers) can be checked using the Coccinelle tool.

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 checking in the program itself to avoid the dangerous situations. In some cases it is possible to use Coccinelle to automatically insert the proper checks. However, inserting such checks automatically would seem to violate the point of a security certification.

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 can be difficult and require sophisticated 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 that strings are null terminated, 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, including not accessing freed memory, do not “double free” memory, only freeing dynamically allocated memory 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. Below we discuss rule MEM30-C (do not access freed memory) in more detail and show how it can be checked using Coccinelle.

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, i.e., environment variables, and calls to external command processors are covered by the rules and recommendations in the ENV category.

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`. Below we discuss the rule ERR33-C (detect and handle errors) in more detail and examine how Coccinelle can be used to check this rule. Note that this rule is different from most other rules in that it is actually *application dependent* since errors are detected and handled differently in different applications. Consequently, in order for an automated tool to support checking of this rule, it must be possible to customise and adapt the tool to a specific project's error handling strategy.

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 never reaches the end of the function (MSC37-C). Below we discuss rule MSC37-C (ensure that control never reaches the end of a non-void function) in more detail and show how this rule can be checked using Coccinelle.

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

3 Compliance Checking with Coccinelle

In this section we discuss how four rules, from the CCSCS categories presented in the previous section, can be checked using the Coccinelle tool. Before going into the details of the individual rules, we briefly introduce Coccinelle; for lack of space we cannot give a thorough introduction to Coccinelle and the languages used to script it, instead we refer to previous work [LBP$^+$09, BDH$^+$09, PLHM08].

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* [PLHM08]. 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 [BDH+09] enabling search not only for specific syntactic patterns but also for control flow patterns. Individual program searches (and transformations) are specified in a domain specific language, called SmPL (for Semantic Patch Language), designed to be similar to the unified patch format widely used by Linux kernel developers and other open source developers. 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. It has been successfully used to search for bugs in open source infrastructure software such as the Linux kernel and the OpenSSL cryptographic library [LBP+09, LLH+10]. The Coccinelle tool is released under the GNU GPLv2 open source license.

3.1 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 are 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.

Below a Coccinelle semantic patch is shown that simply searches for all variable declarations. This simple search forms the heart of the semantic patch used to search for potential violations of the DCL32-C rule:

```
@@
type T;
identifier id;
@@
  T id;
```

Observe that this is very similar to what a variable declaration looks like in a C program.

In Figure 2 the full semantic patch is shown. It simply collects all identifiers of length 63 or more and warns if there are (potential) violations of the 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 practise. 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. The semantic patch includes a simple O'Caml script (lines 11 to 21) that collects all the found identifiers (of length 63 or more) and adds them to a hash table. Before adding an identifer to the hash table, it is checked for collisions, and thus potential violations, and a warning is printed if there are (potential) collisions (line 18).

The basic semantic patch searching for declarations has been augmented with a *position meta-variable* denoted `@pos` (line 9). The position meta-variable is bound to the position (line and column number) of each match.

```
@ initialize:ocaml @
let idhash = Hashtbl.create 128

@ decl @
type T;
identifier id;
position pos;
@@
  T id@pos;

@ script:ocaml @
p << decl.pos;
x << decl.id;
@@
if (String.length(x)) >= 63 then
  let sid = String.sub x 0 63 in
  let _ = if (Hashtbl.mem idhash sid) then
            print_endline (warn p "DCL32-C" "Found␣long␣(%d)␣identifier␣'%s'"
              (String.length(x)) x)
          else () in
  Hashtbl.add idhash sid (x,p)
```

Figure 2: Coccinelle script to find "long" identifiers.

3.2 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 at leveraging null pointer dereferences into 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 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.

Using Coccinelle to find such code patterns is straightforward. In Figure 3 the corresponding semantic patch is shown: we first look for calls to the relevant allocation functions (lines 8 to 14). The possible allocation functions are specified using the *disjunction* pattern (denoted by '(', '|', and ')') that succeeds if either of the alternatives (separated by '|') match. Following that, the script looks for a *control flow* path, represented by '`...`', where the identifier (`x`) is *not* assigned to, i.e., a path where it is not modified (line 15), and where the identifier is not tested for "nullness" (line 16). The latter is in order to cut down on the number false positives. Here the '`... WHEN != x = E`' and the '`WHEN != if(E == NULL) S1 else S2`' means 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 the disjunction pattern to specify three common ways to dereference a pointer: as a pointer (line 18), as an array (line 20), or for field member access (line 22).

```
1  @@
2  identifier x;
3  expression E,E1;
4  type T1;
5  identifier fld;
6  statement S1, S2;
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
19 |
20   x[E1]
21 |
22   x->fld
23 )
```

Figure 3: Coccinelle script to find dereferencing of null pointers.

Note that, even though the semantic patch specifies that there can be no conditionals with a condition on 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, represent a large reduction in 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 3 will catch many common violations of rule EXP34-C, it cannot catch all possible violations. First of all, 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 currently overlooks violations occuring *after* a null test. It is possible to manually refine the semantic patch to take more tests into account in a proper way. In [LBP+09] a more comprehensive Coccinelle approach to dereferencing of null pointers is described. This approach covers not only standard allocations functions, but basically any function returning null. In addition, care is taken to consider null tests and handle them properly.

Another alternative would be if the semantic patch could make use of information from a dataflow analysis. That way it would not be necessary to explicitly cover all syntactic possibilities for null testing or dereferencing. In Section 4 we describe our current work on integrating analysis

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

Figure 4: Coccinelle script to find potential access to deallocated memory.

information into Coccinelle scripts.

3.3 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 practise, reading from deallocated memory may result in crashes, leaks of information, and exploitable security vulnerabilities. Rule MEM30-C ensures 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.

In Figure 4 a Coccinelle script covering some of the simple(r) cases of this rule is shown. The script first looks for any identifier (declared in line 2) that occurs as an argument to the `free()` function (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 the *disjunction* search pattern (denoted by ‘(’, ‘|’, and ‘)’) that succeeds if either of the alternatives (separated by ‘|’) match, the script looks for a *use* of the identifier that results in the actual violation. Here 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).

3.4 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 highly likely to lead to unexpected and/or undefined behaviour, it is therefore essential that the return value is always checked for all calls to a function

```
1  @ voidfunc @
2  function FN;
3  position voidpos;
4  @@
5    void  FN@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   }
```

Figure 5: Coccinelle script to find non-void functions without a `return` statement.

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 code 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 [LLH+10] 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 of the Linux kernel [LBP+09].

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

Non-void functions are required to return a value, using the `return` statement. It results in undefined behaviour to use 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. For this reason the CCSCS requires that all control flows in a non-void function *must* end in a (non-empty) return statement.

Figure 5 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 the functions we have earlier identified as `void` functions (line 13). Once such a function is found, we start looking for 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 3.2) in order to avoid unwanted, potential interference from the isomorphism system. The header also specifies 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, since we have a potential violation if there is even a single control flow path without a `return` statement.

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

4 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 beneift greatly from having access to proper 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 need to be covered. In the following we will illustrate both uses as well as how we intend to make program analysis information available for use in semantic patches. In Section 4.2 we show how such information can be obtained through the Clang tool and we discuss the current status of our integration of Clang into Coccinelle.

4.1 Pointer Analysis: Tracking NULL Pointers and Aliases

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

Below we show how such analysis information could be incorporated into a semantic patch. The following semantic patch is intended to illustrate one possible way to make analyis information available to semantic patches:

```
@@
identifier x, fld;
expression E1;
analysis[null] NINF;
@@
( *x@NINF
| x@NINF[E1]
| x@NINF->fld
)
```

The main thing to note in the above semantic patch is the ‘`analysis`’ declaration (line 4) that declares a meta-variable, called `NINF`. This meta-variable is then 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.

Taking this a step further, we can also use analysis information to find all (sub-)expressions that are potential dereferences and then simply search for all expressions that are both tagged as potentially dereferencing and also as potentially resulting in a null pointer. Here a dereferencing expression is taken to mean an expression that may in any way do a pointer dereference:

```
@@
expression E;
analysis[deref] DEREF;
analysis[null] NINF;
@@
  E@DEREF@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 a *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. The following semantic patch (with alias analysis information available) would capture this situation (see below for an explanation):

```
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 )
```

The idea in the above semantic patch 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 and use of *any identifier* `y` that is an *alias* for `x` (represented by `y@xyalias` in lines 9, 11, 13, 15, and 17).

4.2 Integrating Clang and Coccinelle

In the following we describe how program analysis information, such as described in the above section, can be computed using the Clang tool and discuss the current status of our integration of

Clang and Coccinelle.

Clang was chosen as the main 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, it provides a robust and proven infrastructure for manipulating C programs, and so forth.

The current version of our implementation of a Coccinelle/Clang integration is a "proof of concept" where the main emphasis has been on making the two tools work together and less on adding language features to the semantic patch language. As a result, it is not possible to use the '`analysis`' declaration illustrated in the semantic patches in the last section. Instead we use positions, as implemented by the '`position`' meta-variables, to look up relevant analysis information. Below we show how this works using Python scripting in the semantic patch:

```
@ initialize:python @

  # read in analysis information generated by Clang into
  # Python dictionaries: DEREF and NINF indexed by positions

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

@ script:python @
p << expr.pos
@@

  # lookup DEREF and NINF status in Clang data
  if not (DEREF[p] and NINF[p]):
    # remove the match
  else:
    # accept the match and continue
```

Currently we first run Clang on the source files in order to compute program analysis information. This information is then stored in a file that may subsequently be read by a semantic patch. However, it would be possible to start Clang from within the semantic patch, again using either O'Caml or Python scripting.

4.3 Clang and WPDSs

While Clang provides a good framework supporting the implementation of checkers and program analyses in various forms, e.g., using the monotone framework, they must be programmed directly in C++ and require recompiling the entire Clang tool. In order to make analysis development more flexible and convenient we have added a library for program analysis using weighted push-down systems (WPDS). This allows for program analyses to be specified at the more abstract level of WPDSs. We have also implemented Python bindings for the WPDS library enabling rapid prototyping of analyses without recompilation of Clang.

We have extended Clang with the analysis framework of WPDSs, using the library WALi. This enables us to model the control-flow from Clang as a push-down system, and plug-in different weight domains. Weight domains for different analyses have been presented [RLK07], such as affine-relations analysis, generalised gen-kill analysis and may-aliasing pointer analysis. We have used the gen-kill weight domain to implement a reaching definitions analysis within Clang, and plan to implement a pointer analysis as well. The analysis result can then be pre-processed in Coccinelle scripts, as illustrated above, e.g., to get maybe-null analysis information. The benefit of using Clang is that the control-flow graph of the program is readily available, with some infeasible paths automatically pruned.

The analysis is written as a special analysis pass that constructs the WPDS, assigns weights, and perform a query for each function. The analysis results (annotated weighted finite automata) are output, and subsequently interpreted by the concrete Coccinelle script when analysis information is needed. Currently the Python scripting interface is used with some additional support code for calling Clang and interpreting the output.

4.4 Current Work

The information that we have integrated at this point is the reaching definitions analysis. The output from Clang is a textual representation of the solved WFA, 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)}>
```

All lines are split into their components:

From state of the WPDS, which will be the state *p* in most cases.

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

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

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

The weight again needs to be parsed, in this case into its gen and kill set. 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 named from: the function they are defined in, their identifier and the position they are defined at. All positions are made up of: a file name, line number and 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 uninitialised variables being used, where the basic semantic patch is:

```
@ 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
)
```

Before being able to use a location from Coccinelle we have to account for small differences in how locations are presented in the CFG of Clang and Coccinelle, e.g. precisely where a variable is defined:

```
int a;
    ˆ Clang define location
   ˆ Coccinelle define location
```

Another example is that the use found might be part of a larger expression, so we will have to find the location of the entire expression. Currently we simply map a Coccinelle location to the closest Clang location on the same line.

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.

5 Work in Progress

In this section we discuss the current status and work-in-progress for using Coccinelle to check for CCSCS compliance. In particular we we discuss compliance checking of the full standard for real world software projects.

5.1 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”. Especially 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 a great tool for reducing the number of false positives, since it enables a programmer to refine the semantic patches to take the project specific code styles into account that give rise to the most false positives.

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.

5.2 Implementing Checkers for the Full Standard

While we have only detailed the implementation of Coccinelle checkers for four of the 118 rules in the CCSCS, we have implemented checkers for approximately 25 rules and 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 3.4), it may be possible to provide an “abstract” semantic patch that can be instantiated with project specific details similar to the approach taken in [LLH+10].

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

6 Related Work

The past decade has seen the development and release of numerous compile time tools for program navigation, bug finding and code style checking for programs written in C, as well as many other languages. These tools include the MC tool [ECCH00] (later used as basis for the commercial tool Coverity Prevent). Similar to Coccinelle, the MC tool is a bug finder that can be adapted to specific projects, however the source code for MC has never been released.

Splint [LE01] and Flawfinder [Whe06] are two examples of Open Source bug finders. Both are able to check for a relatively small set of bugs. Both are somewhat adaptable but requires either (light-weight) annotation of the source code or Python programming.

While several commercial static analysis tools support compliance checking[6] for a wide spectrum of coding standard, including the CCSCS, we are not aware of any Open Source bug finder tools working towards this goal.

7 Conclusion

In this paper we have shown that the Coccinelle tool is very well suited for checking some of the rules comprising the CERT C Secure Coding Standard. We have further argued that integrating program analysis information would facilitate even more comprehensive, more expressible, and even more flexible semantic patches to be written.

Bibliography

[BDH$^+$09] 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 Shao and Pierce (eds.), *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009*. Pp. 114–126. ACM, Savannah, GA, USA, Jan. 2009.

[ECCH00] D. R. Engler, B. Chelf, A. Chou, S. Hallem. Checking System Rules Using System-Specific, Programmer-Written Compiler Extensions. In *Fourth USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Pp. 1–16. San Diego, CA, Oct. 2000.

[LBP$^+$09] 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*. Pp. 43–52. IEEE, Estoril, Lisbon, Portugal, June/July 2009.

[LE01] D. Larochelle, D. Evans. Statically Detecting Likely Buffer Overflow Vulnerabilities. In *Proc. of the 10th USENIX Security Symposium*. USENIX, Washington D.C., USA, Aug. 2001.
http://lclint.cs.virginia.edu/

[6] See the CCSCS web page for details on tool support.

[LLH$^+$10] 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*. Pp. 191–196. IEEE Computer Society, Valencia, Spain, Apr. 2010.

[PLHM08] Y. Padioleau, J. L. Lawall, R. R. Hansen, G. Muller. Documenting and automating collateral evolutions in linux device drivers. In Sventek and Hand (eds.), *Proceedings of the 2008 EuroSys Conference*. Pp. 247–260. ACM, Glasgow, Scotland, UK, Apr. 2008.

[PLM10] N. Palix, J. L. Lawall, G. Muller. Tracking code patterns over multiple software versions with Herodotos. In Jézéquel and Südholt (eds.), *Proceedings of the 9th International Conference on Aspect-Oriented Software Development, AOSD 2010*. Pp. 169–180. ACM, Rennes and Saint-Malo, France, Mar. 2010.

[RLK07] 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*. Pp. 23–51. Springer-Verlag, 2007.

[RSJM05] 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):206–263, 2005.

[Sea08] R. C. Seacord. *The CERT C Secure Coding Standard*. Addison-Wesley, 2008.

[Whe06] D. Wheeler. Flawfinder Home Page. Web page: http://www.dwheeler.com/flawfinder/, Oct. 2006.
http://www.dwheeler.com/flawfinder/