Comparison of Static Code Analysis Tools

Matti Mantere and Ilkka Uusitalo
VTT Technical Research Centre of Finland
Kaitoväylä 1, Oulu, Finland
firstname.lastname@vtt.fi

Juha Röning
OUSPG Oulu University Secure Programming group
90014 University of Oulu, Finland
juha.roning@oulu.fi

Abstract

In this paper we compare three static code analysis tools. The tools represent three different approaches in the field of static analysis: Fortify SCA is a non-annotation based heuristic analyzer, Splint represents an annotation based heuristic analyzer, and Frama-C an annotation based correct analyzer. The tools are compared by analysing their performance when checking a demonstration code with intentionally implemented errors.

1. Introduction

Low software quality is the biggest culprit when it comes to compromized information security. Low code quality can leave the resulting program seemingly functional, while hiding a plethora of weaknesses for skilled antagonists to exploit. In today's software-driven world there is an increased need for software to be secure and dependable. Even though nearly all of the security problems are caused by software failure, comparatively little published security material has focused on this [8, p. 1-14]. An increasing number of safety-critical applications is in use, with possibly human lives depending on their uninterrupted functionality. The introduction of new tools in the field of static analysis has brought new power to the security team [5, p. 21-46]. The task of manually going through the source code while searching for flaws is often enormously time consuming and tedious. In the worst case, it might not be a feasible approach at all. This is the area where the static analysis tools come into play: they aid the auditor by pointing to possible bugs and failures in coding practice, thus saving time and effort. They still need someone to operate them and manually check the flagged possible errors, but are a valuable resource if applied with skill [5, p. 47-70].

In this paper we discuss the possible benefits of incorporating source code analysis tools into the software development process. We consider three different static analysis tools and demonstrate their use by analysing a demonstration code with intentionally implemented errors. These three tools, Fortify Source Code Analyzer (SCA), Frama-C, and Splint, will also be compared to each others. All this is done with emphasis on the software security perspective. [12, p. 1-28] [5, p. 1-20] [8, p. 1-35] [10, p. 105-137]

The rest of this paper is organised as follows. In section 2 we give the reasoning behind our selection of tools and shortly introduce the differences between them. Section 3 discusses the chosen tools in more detail. Section 4 is the core of this paper, discussing the comparisons and test done with the tools. Finally, the conlusions are given in section 5.

2 Selected Tools

Tools for static analysis have rapidly matured during the beginning of the 21st century; they have evolved from simple lexical analysis to using much more accurate and complex techniques. The current commercial tools are generally relatively easy to use and effective. The cutting edge open source tools are mostly found in research use and are not likely to be mature enough for commercial use even if very accurate when properly used. The more mature open source tools are currently more or less one step behind the cutting edge tools, but are free of charge, mature enough and provide lightweight software projects with added safeguards against bugs for minimal monetary investment.

The tools studied in this paper are introduced in the next section. In this section the reason behind the selection is

978-0-7695-3668-2/09 $25.00 © 2009 IEEE
DOI 10.1109/SECURWARE.2009.10

briefly explained, as well as the differences amongst the tools selected prior to their more detailed descriptions.

2.1 Selection

The reason for selecting this particular set of tools, Fortify SCA, Frama-C and Splint, is mainly to introduce different approaches to the same problem, as well as tools from different realms of development. SCA is commercial software and Frama-C and Splint are from the open source realm, Frama-C being fresh out of research laboratory. While Frama-C offers much wider and deeper analysis possibilities than Splint, Splint offers the core functions needed for annotations-based software, is mature, and works nicely for the demonstrative purposes of this paper. The tools represent three approaches to analyzers: Fortify SCA is a non-annotation based heuristic analyzer, Splint an annotation based heuristic analyzer, and the Frama-C an annotation based correct analyzer. All the tools were used both in Microsoft Windows XP and Ubuntu 8.04 environments without significant differences in usage or results.

2.2 Commercial or Open Source

The question of whether to choose tools provided for a cost by commercial producers or whether to adopt one of the freely distributed open source tools has galvanized some of the people in the software industry. While the acquisition of the open source tool itself might be free, personnel still needs to be educated in its use, and this will bring costs. While for some the choice between commercial or open source software remains ideological, attempts have been made not to include any bias towards either field in this paper. [10, p. 109-10][10, p. 122-137]

3. Tools

In this capter we introduce the chosen tools, SCA, Frama-C and Splint, in little more detail.

3.1 SCA

SCA is currently part of the Fortify 360 suite. In this paper the separate SCA 4.0 is used; the full suite includes many other functionalities. It is a commercial tool that has been adopted by several high-profile operators. The engine of SCA comprises four different analyzers- semantic, data flow, control flow and configuration analyzers -and supports Java, C, C++, C#, JSP, XML and PL/SQL. We used only its standard C capability. The toolset is extensible and is supplied with Rules Builder for creating custom rules for the knowledge base [10]. The compiler used is a MinGW port of the GCC 3.4.5.

3.1.1 Strengths and Weaknesses

The first and most obvious weakness is the initial cost. While it depends on many factors, commercial tools for industry are typically expensive. Even though the price may be high, it should be noted that these kinds of investments will likely pay for themselves if acquired and applied with enough forethought. The second weakness is the closed source nature of the software; there is no access to the source code. It would be beneficial to be able to perceive how the tool functions and how it is written.

3.1.2 Using the Software

The use of the software in very small projects is straightforward and simple. When run, the tool only needs to be assigned the compiler one wishes to use, and target source files for analysis. The tool brings the static analysis to the developers desktop, and does this with remarkable ease of use with a tight selection of command line options.

3.2 Splint

Splint is an annotation-based tool aimed at the security needs of software programmers. It is a tool for statical analysis of C programs for security vulnerabilities and programming mistakes [6]. If used without the annotations, it performs like the traditional Lint tool. Due to its similarity to Frama-C in that both use an annotation-based technique to implement the rules for checking inside the code, it is a very appropriate choice for this paper. It is also quite mature software, being much more polished and easier to use than the experimental research tool Frama-C. Splint does not include a GUI, so it is used via a command line interface. The ruleset of Splint mainly resides inside the source code itself, in the form of the annotations embedded in the code by the developer himself. The tool itself includes some rules for analyzing the code, but without annotations only Lint-level performance is achieved.

With proper use and properly annotated code, Splint can detect a wide range of possible problems[6]. The proper use of annotations is of paramount importance, and will ensure that the output is what is desired.

The version of Splint used in this work is 3.1.2 released on 12.7.2008. The latest version of Splint was published during July 2007.

3.2.1 Strengths and Weaknesses

Splint is quite lightweight, and using it is simple and straightforward. The strengths with the annotation system are mostly the same as in the Frama-C suite, with the Splint annotations being slightly more limited in scope.

Splint is a lightweight program with some limitations; the annotation language cannot be used to model every aspect of the code. Some errors pass through the filter, even with annotations crafted with the utmost care. The software is also slightly dated, and its development stagnated.

3.2.2 Using the Software

Using Splint is simple for small programs that require very little annotations to be made. The annotations are also quite simple. The annotations used direct the tool, providing it in effect with customized rules.

3.2.3 Splint Annotations

The annotations used with Splint are written inside stylized comments, beginning with /*@ and ending with @*/. The annotations provide functionality to detect a range of errors described in the Splint manual; they are fewer in number and narrower in scope when compared to the ACSL implemented in Frama-C [6, 2]. Using the annotations correctly and achieving the desired goal can be quite complex and demanding. Annotating ready code is generally a bad idea, at least if the codebase is large. [5, p. 96-105]

3.3 Frama-C

Frama-C is a framework for formal analysis of C programs. It is developed by INRIA and the CEA Software Reliability Laboratory, both from France. The framework tool is written in INRIA-developed OCaml programming language, and contains a type checker, parser and a source level linker. Frama-C is in the early phases of development, so it is not quite as usable as the older, more mature open source analyzers. The ANSI-C Specification Language (ACSL) annotation system is one of the most important and integral parts of the Frama-C suite of tools and will be covered in more detail later [4, 11]. Currently, Frama-C is in the beta phase of development, with many experimental or unimplemented ACSL functionalities [4]. Since the framework itself is under construction, many plugins are missing, for example the framework still lacks plugins for handling C-strings, which are known to cause a very large portion of buffer overflow errors in C [5, p. 175-234].

This paper discusses one of the first attempts of using Frama-C without direct support from the developers. lack of support likely to affects the outcome of its use. This diverts some of the value of this experimental use from the findings by the use of the tool to the actual process of how the use of the tool is understood and adopted. One of the critical differences between Frama-C and the rest of the analyzers used, is the fact that Frama-C is a framework of correct analyzers, producing much information but no false negatives. Much of the limitations mentioned with Frama-C actually exists within the other tools as well, but since they do not claim to be correct analyzers, these limitations are not mentioned.

3.3.1 Using the Software

The use of Frama-C requires a high level of experience and understanding from its user when compared to Fortify SCA. Annotating the code correctly using ACSL syntax requires deep insight of program behaviour. Numerous plugins exist, and the user must select which ones would be of use for him. Frama-C is quite verbose in its output, printing a lot of false positives and vast amounts of information. The output has to be inspected very carefully and can be time consuming. Currently, the suite only supports sequential, embedded-like C code. The version used in this work is the latest Frama-C with GUI release Helium-20080701 [2]. The suite includes both GUI and command line interfaces and both are needed to fully exploit the tool.

3.3.2 Strengths and Weaknesses

The modular structure of Frama-C is built to accept additional plugins. This allows for customization by advanced users to fit the system into their projects. The locally customized rules can be used to obtain accurate results, even with highly complex and delicate code structures, given they are written properly. Another strength is the annotation system, in which the rules of the analysis tools are implemented into the source code, which results in rules appearing directly in their context. Frama-C is a correct analyzer, and hence produces massive amounts of information, including false positives. The positive side is that no false negatives are produced. The downside is the massive amount of false positives that can be very cumbersome and tedious to sort out. The annotation system is also a weakness of the software suite; creating annotations can be tedious, and requires considerable expertise in the ACSL language, and deep insight into the code being annotated. If not properly written, annotations can easily yield a false sense of security. Annotations with no syntax errors, but with serious design flaws, can easily create a situation where the code complies to the specifications, but the specifications are invalid. [5, p. 96-105]

3.3.3 Important Plugins

The modular structure of Frama-C is evident in that the functionality of the tool is achieved via the use of numerous plugins. Some of the plugins are currently much more experimental than others, and some are altogether yet unimplemented. The plugins currently distributed with the Frama-C software suite include the tools listed below. The plugins generally function to some level, even without

ACSL notations in the code. Mainly the value analysis component is used for this paper, although other components are also discussed briefly. Other plugins than what introduced here are also available for Frama-C.

Value analysis includes an interactive GUI called ValViewer, which allows the user to peruse to code and observe the corresponding output of the value analysis plugin. In the value analysis the possible value ranges of the functions and variables are computed, and printed for the user. This plugin functions without ACSL notations but is very limited in accuracy and functionality. Currently, only sequential code can be analyzed. [1, 11, 2]

Jessie plugin allows deductive verification of C programs if they are annotated with ACSL. The need for ACSL annotation efectively rules out the use of this plugin for some projects coded prior to the adaptation of the tool since annotating the ready code can be very time-consuming and tedious. This plugin also contains a usable GUI [2].

Slicing can be used to remove spare or unnecessary parts of the source code in relation to the part being currently analyzed as dictated by the user in the slicing criteria [2]. The slicing information can be presented to the plugin from the command line. Notations similar to ACSL notations can also be used in the source code itself to tell the plugin the exact criteria by which to slice the code.

Impact analysis is, as the name implies, intended for the automatic computation of selected parts of the C program in question. As it is said on the web page: "The impact analysis plug-in allows the automatic computation of the set of statements impacted by the side effects of a statement of a C program. Statements not appearing in this set are guaranteed not to be impacted by the selected statement. " [2].

3.3.4 ACSL

ACSL is a language for specifying the properties of C source code. ACSL enables the user to annotate the source code to prepare it for formal deductive analysis. The language is inspired by the *CADUCEUS* [7] tool, which was fundamentally inspired by *Java Modelling Language* [4, 9]. The use of ACSL is important if Frama-C is to be utilized to any real extent. The annotations and specifications made into the source code using ACSL make it possible to analyze structures that would otherwise be unintelligible to the Frama-C tools. Some instances of code can be analyzed without the use of the ACSL annotations, but skill in the use of ACSL is needed if any real code is to be analyzed.

Syntax The syntax by which ACSL is used closely follows that of ANSI C, though there are some notable differences that should be kept in mind when using the notation system. The logic expressions follow pure C notations, but also introduces a new structure. The basic use and syntax of ACSL is demonstrated in the alpha version of the quick tutorial [11]. Full specification is also available, but it must be remembered that it is still subject to change as the syntax and semantics have not yet been fixed. The specifics of the ACSL are not listed here as they are readily available in [11]

Problems in ACSL Problems with the use of ACSL and similar annotation techniques lie in the fact that if the annotations have not been adopted in a sufficiently early phase of the project development, they can be problematic to add later. To gain full benefit, the ACSL notations should be incorporated into the code during the entire length of the development phase, as the developers usually know best what they are doing. Therefore, if the use of Frama-C or similar software requiring annotations is desired, it must be clear before the project moves to the actual coding phase. The ACSL annotation system strives to be very broad and to cover the entire scope of programming problems, hence it is also necessarily quite complicated and requires a lot of work for complete coverage. Annotating code accurately enough to entirely model the behaviour of the program requires the coder to actually create the program twice, once for C and once for ACSL.

4. Comparisons

The use of the tools is demonstrated in this section. The demonstration codebase is analysed with each tool focusing on each particular error class separately. The error classes are discussed in section 4.2. After each run the code is corrected based on the results from the tools, so that in effect every tool gets to analyse the code again with corrections based on its own previous output. The review will follow the cycle explained in [5, p. 47-70].

4.1 The Demonstration Code

The demonstration code implements a small program that stores the names and birthdays of people and informs the user of the closest one when executed. The entire length of the annotated test code was just under thousand lines, so only few code snippets are presented in this paper.The software is tiny but it presents ample opportunities for implementation of many types of security-related programming errors. The code was produced to be free of any intentional errors to begin with, and, after that, a number of controlled

errors were introduced, with necessary annotations to simulate a situation where the code was accurately created and annotated but would hide an error. The demonstration code resides in one .c file. The demonstration code was annotated with two different annotation languages for Splint and Frama-C.

ASCL Annotations are written in function contracts and in other parts of the code. The function contracts can be written to fully describe a function, but here only a partial definition is done. The annotations follow the ACSL definition document [4]. Some of the functions are not yet implemented, but the annotations are written as if the framework were to follow the definition document version 1.3 to the letter, as it is not fully known what functionalities are implemented and to what extent, and this will become clear in some cases during this demonstration. The functions are not fully defined with ACSL in this demonstration. It would be possible, but it would greatly increase the workload as in effect the whole functionality of the function would have to be re-implemented with the ACSL language.

Splint Annotations are located in the code in function contracts and in the main body of the code. The contracts are written at the beginning of the function prototypes. They define some important parts of the behaviour of the function. They do not, however, define the function completely, nor are they intended for such use [6]. Other types of annotations appear directly in the code and are explained more thoroughly in the next section where the implemented errors are listed.

4.2 Implemented Errors

Below we introduce the implemented errors. This demonstration code was written to demonstrate errors. It is irrelevant whether or not there are other errors amid the codelines. The errors are presented in the code with the annotations that are generally used to expose them. The implemented errors are not designed to be particulary exploitable and merely represent test cases for how such errors could be found in the actual code. This is also a good place to remember that it should not be necessary to demonstrate that a bug could be exploited before action is taken to remedy it [5, 12]: any found errors should be corrected. Still, it should also be kept in mind, that false positives are not errors in this fashion. Errors are implemented in a way that they will not produce warnings when compiled using the GCC compiler with `gcc -Wall` syntax. Errors from five different classes are implemented for a total number of 14 errors. The errors represent several classes of different programming mistakes that could lead to a security compromize.

Buffer Overflows are written in four slightly different instances, and represent some of the different types of errors concerning buffer addressing. The buffer overflow errors are very common and introduce serious vulnerabilities to the affected system. In C and C++ the programmer can introduce buffer overflows with remarkable ease, due to the nature memory is handled. One of the typical ways to introduce buffer overflows, is to be using the strings library of C standard, with potentially unsafe functions such as strcpy. Buffer overflows are also one of the most exploited vulnerabilities due to their abundance in existing software. All buffer overflows found in programs should be remedied, regardless of whether they are deemed to be immediately exploitable or not. [5, p. 175-234]

Below is a an example of buffer overflow error, as implemented on line 717 in the test code:

```
alphabets=malloc(20*sizeof(char));
if(alphabets==NULL)
   return ERROR MEMORY;
/**Buffer write overflow**/
strncpy(alphabets,
    " abcdefghijklmnopqrstuvwxyz", 30);
limiter = 29;
```

Memory Handling Two types of memory handling errors are implemented. Memory handling errors arise from dynamically allocating memory, and either using the memory after it is freed or by freeing it twice. Forgetting to free the memory altogether also depletes resources and can cause severe vulnerabilities, depending on the application in which it is manifest [3]. However, more typically, the use-after-free and double-free can expose the system to buffer overflow attacks. [5, p. 179-188]

Below is an example of memory handling error, as implemented on line 392 in the test code:

```
free (inbuff);
    //@ ghost isinbuffFreed=1;
/**Memory used after free**/
    //@ assert isinbuffFreed==0;
inbuff[0]=\0;
```

Dereferencing a Null Pointer Two null dereference errors are implemented. Using a NULL pointer as if it pointed to a valid memory area will invariably cause the program to crash. Even while the null dereference errors are common, they are usually easy to discover and fix. All pointers should be checked for null dereference before used. [3]

Control Flow Errors are implemented in four places in the code. Two kinds of control flow errors are implemented. Firstly the three instances of failing to check for return values of functions, and lastly not releasing a resource. These errors can possibly cause a risk, and should be remedied when discovered, even if no possible exploit could be immediately discerned. Failing to check the return value could lead to dere ferencing a null pointer or other problems. Not releasing a resource could lead to denial of service attacks, via depletion of resources. [5, p. 266-268] [3]

4.2.1 Input Validation

Input is received and not checked properly in two places in the code , leaving a possibility for tainted input. Input validation is a critical part of a secure software, and doing it poorly will leave the system open to a multitude of attacks such as metacharacter vulnerabilities and command injections. All input from all sources should be validated, preferrably using strong validation like whitelisting the allowed inputs. The tools used are not expected to discover lack of validation of input with the selected level of annotations and customized rules. [5, p. 117-173] [3]

4.3 Discovering the Implemented Errors

In this section an attempt is made to discover as many of the implemented errors as possible. The analysis tools will be run three times. After the first run, all the found errors are gone through and fixed based on this tool output. After this, the tools are run once more, and any remaining errors will be corrected. After this, there is a third and final run, after which it is be checked to see that no errors remain to be found with the used annotations and tools. The goal is to discover all the implemented errors.

4.3.1 Applying the Tools, First Phase

All three tools were run on the code.The finding of errors mostly depends on two factors: the quality of the automated tool, and the skill of the auditor using the tool. In the case of the annotation-based tools Frama-C and Splint, some errors are possibly overlooked due to the auditor's mistakes in annotating the code, or, in the case of Frama-C, the length of the generated output. Fortify's tool requires the least expertise to operate succesfully, leaving only the checking of the reported errors to the responsibility of the user. Table 1 lists the implemented errors, and how they were found with the tools. Symbol "F" is for found, and "-" means that the error remains yet undiscovered. "C" is used to indicate that the error was not directly reported but was also remedied when reported errors were corrected.

Fortify SCA directly discovered eight errors, and correcting these led to remediation of the two errors concerning null pointer dereferences as well. All reported errors were actual errors, no false negatives were discovered. One overflow error and the only underrun error were not detected by the tool directly, but the overflow error was remedied by correcting other bugs discovered. As expected, the input validation errors were not reported either, and discovering them would have required crafting a customized checking rule for the tool. With the audit workbench we could view all the locations where the tool accepts input, and from this information the validation status of each input could be analyzed.

Below is the code area where buffer underrun error that was left undiscovered by the tool was implemented:

```
for (i = 20; ((i <= (MAXLEN + 19)) &&
(inbuff != NULL)); i++)
{
if (inbuff[i] != '#' && inbuff[i] !=
'\0' && (i - 20) < 8)
{
/**Buffer read underrun error**/
temp.birthday[i - 25] = inbuff[i];
}
```

The instance of the buffer overflow that was left unnoticed was also similar in structure, appearing in a loop and reading, in this case, too many characters to an array.

Splint reported numerous possible errors to be present in the code. After working each reported error through, it was discovered that several of the reported errors were propagated from the same initial error source, and that a few errors were false positives, such as those concerning the argument vector of the main function.

Frama-C produced a lot of output. The findings are discussed here and listed in Tables 1 and 2. In the first phase, the tool stopped propagation of the value analysis when the assertation annotations got the status invalid, so it was still possible to find the other problems, after the problems that caused the assertations to fail were corrected. Validations that received an unknown status are to be processed in the next phase. Frama-C is a correct analyzer, so the results here represent the users skill in weeding out the true positives from the extensive output listing. In other words: all the errors prior to the failed validation were discovered by the tool, but not necessarily by the user.

4.3.2 Fixing the Found Errors, First Phase

The found errors were corrected, and are listed in Table 1. This phase produced three different code lineages, but

they are not listed as they only differ from the original very slightly and the changes are easily spotted. When correcting the code, the line numbers were preserved to keep the information and tool output readable with regard to the original code with implemented errors.

Table 1. Errors found after first phase.

Implemented Error	Frama-C	Splint	Fortify SCA
Buffer Read Overflow	F	F	C
Buffer Read Underrun	F	-	-
Buffer Write Overflow	C	F	F
Buffer Write Overflow	C	C	F
Memory Used After Freed	F	F	F
Memory Not Freed	F	-	F
Null Pointer Dereferenced	-	F	C
Null Pointer dereferenced	-	F	C
Missing Check for Null	-	C	F
Missing Check for Null	-	C	F
Unreleased Resource	-	-	F
Unreleased Resource	-	-	F
Unchecked Input	-	-	-
Unchecked Input	-	-	-

4.3.3 Applying the Tools, Second Phase

The tools were then run again on the code corrected by using their own output.

Fortify SCA produces no errors after the corrections were made and the tool was run, it was concluded that no more information was to be gained by running this tool again, apart from fixing the input validations, which could have been corrected by using the Audit Workbench and the input listing. These are not listed as found errors in Table 2, as they were discovered manually, only using the tool to pinpoint where the program accepted input, and wading through these as listed.

Splint produced the same false negatives as before, without giving any additional information on how to proceed correcting the code. It was concluded that further runs of this tool would require additional annotations to provide more useful information. No additional corrections or annotations were made.

Frama-C again produced a lot of information. After examining the information more thoroughly, it was discovered that two assertations that received status unknown resulted from possibilities for null pointer dereferences. The unreleased resources were not discovered by the user with this level of annotations. Expanding the annotations and including some checks for input validation would have revealed the two input points missing the validation checks.

4.3.4 Fixing the Found Errors, Second Phase

The additional errors found were corrected as depicted in Table 2. The input validation was corrected for the final corrected version of the code by using the Fortify Audit Workbench to list the inputs and then by manually checking that all of the inputs were received through proper validation channels. The input validation was not listed as found by any of the tools due to the large amount of manual work needed.

Table 2. Errors found after second phase

Implemented Error	Frama-C	Splint	Fortify SCA
Buffer Read Overflow	F	F	C
Buffer Read Underrun	F	-	-
Buffer Write Overflow	C	F	F
Buffer Write Overflow	C	C	F
Memory Used After Freed	F	F	F
Memory Not Freed	F	-	F
Null Pointer dereferenced	F	F	C
Null Pointer dereferenced	F	F	C
Missing Check for Null	C	C	F
Missing Check for Null	C	C	F
Unreleased Resource	-	-	F
Unreleased Resource	-	-	F
Unchecked Input	-	-	-
Unchecked Input	-	-	-

4.3.5 Applying the Tools, Third Phase

In the third phase the tools were run again, but no additional information was gained from the analyzers; Splint and SCA produced the exact same output as in phase two, and Frama-C's output did not produce any more insights in to the remaining errors with the current level of user skill. Using more time and effort on the Frama-C output should have uncovered the remaining errors as well due to the correct nature of the analyzer, but consumed disproportioned amount of time.

4.4 Discussion

The tools in question are very different in use: being the only correct analyzer in the selected toolset Frama-C pro-

duces copious amounts of information and is very versatile. The toolset has a steep learning curve and demands a lot of initial study and testing from the user. While the tool discovers all the errors, they hide amid many false positives. Splint is more limited but easier to adopt. It also produces much less information concerning the errors, but naturally also missed more errors than Frama-C. Fortify SCA was by far the easiest to use; it also missed at least one error, but produced no false positives. The reason why the tools based on annotations did not do as well as SCA in this demonstration, was due to the level of annotations, and limited time to analyze Frama-C output. Using the tools based on annotations would be feasible in a software project where the developers are intimate with the annotation language and are able to accurately write them into the codebase as they work. Fortify SCA does not require that the programmers necessarily know anything about the inner functions of the tool, only that someone in the project is able to write customized rules as needed. SCA is also more suitable for projects using a large amount of earlier code, as discussed earlier. To automate input validation checking, more annotations would be required for Frama-C and Splint, and a custom rule would be necessary for Fortify SCA. Of the three tools studied, the most interesting one, though not the most usable at the time, was the Frama-C suite. Frama-C shows good potential and provides a possibility to develop customized plugins, as well as a ready package of plugins. Based on the specification of the ACSL, it is designed to be comprehensive in modelling the C language, thought not all of the ACSL language is yet implemented. A drawback with Frama-C is also the fact that it is currently solely a C language platform, with preferrence for embedded-like code, with C++ support upcoming in the future. Even though Frama-C was the most interesting, it is still in the beta phase and needs embedded-like code and very strong technical skill, so Splint would be recommended for a user wishing to adopt a static analysis tool for a light C language software project, at least until Frama-C matures. For a full-scale industrial project with a healthy budget, Fortify SCA currently provides a streamlined, easily adopted and mature tool for discovering and removing some of the security risks from the code.

5. Conclusions

Static code analysis tools can be very expensive, have steep learning curves, be too old for any serious work or need annotations to be written to libraries. They should be carefully studied before selecting which one to use in a software project.

Using up-to-date static analysis tools can be recommended for any serious software project as it will help to weed out some of the errors from the code. The output of the analyser must be analyzed further, and not understanding the output but doing the corrections anyway can lead to potentially disastrous consequences. One should always have an understanding of what it is one is correcting; correcting errors one does not understand can easily lead to disaster.

Three different static analyzers were studied in this work. It became evident that the tools based on annotations have good potential but demand more of their users. Frama-C required most technical skill, as analyzing its output presented a considerable challenge. The non-annotation-based cutting edge tools represented by Fortify SCA are easier to adopt and have more polished user interfaces. With enough annotations, the software can be checked adequately even with Splint, but this demands close to the original amount of programming work in the annotation language, which can be new to many developers. All the selected tools would benefit software projects when used with required skill level, integrated well enough to the project and used on intended targets.

6 Acknowledgements

This paper is part of the €-Confidential project, which is an EUREKA Σ!2023 / ITEA 05011 European Research Program project.

References

[1] Documentation of the static analysis tool valviewer. (accessed 10.5.2008).
[2] Frama-c official web site. (accessed 10.5.2008).
[3] Owasp web page. (accessed 13.9.2008).
[4] P. Baudin, J.-C. Fillitre, C. March, B. Monate, Y. Moy, and V. Prevosto. Acsl: Ansi/iso c specification language preliminary design (version 1.3, july 11, 2008). (accessed 22.7.2008).
[5] B. Chess and J. West. *Secure Programming With Static Analysis*. Addison-Wesley, Erewhon, NC, first edition, 2007.
[6] D. Evans and D. Larochelle. *Splint Manual*. (accessed 9.6.2008).
[7] J.-C. Fillitre, T. Hubert, and C. March. The caduceus verification tool for c programs. (accessed 20.5.2008).
[8] G. Hoglund and G. McGraw. *Exploiting Software: How to Break Code*. Addison-Wesley, Erewhon, NC, first edition, 2004.
[9] G. T. Leavens, A. L. Baker, and C. Ruby. Jml: A notation for detailed design. In H. Kilov, B. Rumpe, and I. Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 175–188. Kluwer Academic Publishers, 1999.
[10] G. McGraw. *Software Security: Building Security In*. Addison-Wesley, Erewhon, NC, first edition, 2006.
[11] V. Prevosto. Acsl mini-tutorial. (accessed 16.5.2008).
[12] J. Viega and G. McGraw. *Building Secure Software: How to Avoid Security Problems the Right Way*. Addison-Wesley, Erewhon, NC, first edition, 2002.