International Journal of
INTELLIGENT SYSTEMS AND APPLICATIONS IN ENGINEERING
ISSN:2147-6799 www.ijisae.org Original Research Paper

Comprehensive Empirical Study of Static Code Analysis Tools for C Language

Vishruti V. Desai [*1], Dr. Vivaksha J. Jariwala[2]

Submitted: 10/09/2022 **Accepted:** 20/12/2022

Abstract: A developing trend in current science and technology is the emphasis on software codes, which places greater attention on the quality of software codes. In today's quality assurance procedure, static analysis plays a significant role. The important feature is that any fault or vulnerability in the code is discovered without the need to execute it. The key challenge is identifying complex code blocks and possible system faults. For unsafe programming languages like C and C++, various static code analyzers are used. Each of them has unique importance and constraints. To date, no technique has yet been able to guarantee that the software will not ever halt, crash, or behave bizarrely. However, more effective techniques may be chosen to reduce software coding defects. Our objective is to examine various static analysis tools to identify their uniqueness and specification. In this paper, we examine static analysis tools, their methods and determine their performance measures. Our focus is to compare various tools that assess C programs according to capabilities for detecting vulnerabilities and to identify the strengths and limitations of each tool. As an empirical study, we evaluate various performance parameters for the Juliet Test suit for C programming language.

Keywords: *C language, Common Weakness Enumeration (CWE) , Programming language, Security Vulnerability, Static Code Analysis*

1. Introduction

Software needs to be reliable and secure in today's software-driven environment. Security and software quality issues are becoming important in the rapidly expanding world of modern technology. The software application is defined by the quality of the code and the coding standards that it conforms. An effective way to examine the software system coding strategy is through Static Code Analysis (SCA). Tools for SCA assist programmers in creating robust software that is free of flaws and vulnerabilities [1][2]. Inadequate software quality is the primary cause when it comes to information security breaches. Low-quality code that has a few flaws might result in insecure software and because of it, knowledgeable adversaries can take advantage [3]. Manual inspection through the source code and seeking faults is a time-consuming and laborious process. In the worst-case scenario, it could not even be a practical strategy. Static analysis tools are useful in such situations [4]. Tools save time and effort for the inspector by alerting them to potential defects and code mistakes. But still, require someone to run them and manually review the indicated potential problems. If analysis results are used skilfully, they are a great resource for flaw finding [5].

Static analysis tools have significantly improved over the last 10 years, going beyond simple lexical analysis to incorporate far more complex algorithms. However, most static analysis challenges are undecidable, that is, it is difficult to devise an algorithm that always yields the right response in every situation. Therefore, not all vulnerabilities in source code are found by static code analysis tools and are likely to provide information that, upon closer inspection, is not a security vulnerability [6]. An SCA tool must uncover as many vulnerabilities as feasible, ideally all of them, with the least number of false positives—ideally none—to be useful. This work aims to better understand the advantages and disadvantages of static code analysis techniques by conducting a comprehensive empirical evaluation of their capacity to identify security flaws. The following are the primary contributions of this paper:

1) We represent comprehensive details of tools and prepare the comparison study between them.
2) We examine tool evaluation on Juliet Test Suit. It is a benchmark for assessment of static code analysers tools.
3) We consider performance measures such as accuracy, recall, time for execution, detection ratio for accountable Common Weakness Enumeration (CWE).

The remaining contents of this paper is organized as follows: The Section 2 contains the literature review; the Section 3 describes tool overview in detailed comprehensive way. Section 4 gives analysis of each tool mentioned in section 3. Section 5 discusses the result and performance measures affected by each tool followed by conclusion.

2. Literature Survey

Static code is a method in which a source code is examined for quality and safety. For identifying vulnerabilities in C/C++ software, we look for automated tools. In [7], the author reviewed various programme analyzers that are used by Google, Facebook, and Microsoft. Many static analyzers did not consider all potential runtime faults. Others pay particular attention to the ones that were likely to be useful to them. We search for memory related vulnerability detection using such tools. Authors in [8], compared 12 different tools with various parameters and defect types. For

[1] *Ph.D. Scholar, Gujarat Technological University, Gandhinagar, Gujarat, India.*
ORCID ID: 0000-0002-5922-9374
[2]*Supervisor, Gujarat Technological University, Gandhinagar, Gujarat, India.*
ORCID ID: 0000-0003-3332-2033
** Corresponding Author Email: Vishruti.phd@gmail.com*

each tool, they have written their own parser for analysis of reports and mentioned the tools priority for specific defect type. In various comparisons papers [9], [4], [6], [10], [2], [11], they have either said about the two or three tools with limited vulnerabilities. Even, the names of the tools are not mentioned by authors. They also considered the fact that commercial instruments are both expensive and not readily available. Additionally, although not in comparison, they outlined each tool's specific strengths and weaknesses in publications.

Primarily, static code analysis tools use two basic approaches. It is a compiler based or machine learning based approach. Here in our survey, we found more work in compiler-based approach and very less in machine learning based. Researchers employ SCA for a variety of applications, including embedded systems, IoT systems, etc., in addition to various techniques in general application software. In [12], it used as fault detection tool for limited resources and considered phase-wise analysis for software development. In [3], IoT based applications, the authors took 18 open-source system and found unsafe commands related with memory and string. This direction is helpful for secure coding and high-quality standards. Apart from empirical evaluation on limited set, authors in [1] have counted and listed theoretical concepts of tools along with CWE categories that are available in manuals. There are tools that used general purpose languages and domain specific languages. Many papers used Toyota ITC Test Suit and Juliet Test Suit for evaluation of tools and listed out vulnerability on CWE category for defect type or their own category for it [13][4]. There was a possibility that the same vulnerability is listed with different name and CWE number as per their interpretations. Instead of using ITC Test Suit or Juliet Test Suit, researchers also worked on preparing their own test suits for analyzing various parameters [14].

Other work carried out with static code analysis is related with report generated by tools. Authors analyzed and labelled the vulnerability in report and performed analysis on report. In addition, they tried to find false positives and true positives in the report. In [15], they presented a systematic review of the work carried out till 2018 for SCA tools. They had more focused on the language domain instead of detailing of the performance of tools.

3. Tools Overview

In this section, we introduce 14 tools in more detail with their general information along with their capabilities mentioned in manual of tools. Below table 1, shows the tool general information, its description. Basically, we focus on tools that work in the C language. We also compare them based on information such as the platform they support, whether tools are extensible or not and availability of tools. We consider both types of tools that are open source and commercial products used in industry. We also reflect the form of the output generated by tools, their version and first release year.

Table 1. Tool Parameter Description

Parameter Name	*Description*
Language (L)	Language supported by specific tool.
Platform (P)	It describes the platform on that tool run.
Extensibility (E)	It is extendable or not. If yes(Y) else no(N)
Availability (A)	Tool is available Commercial (C), Open Source(O), or Free(F).
Output	It gives the format of output generated after tool runs. Command Line (CMD), HTML, XML, CSV
Version	It provides the current version number available in market.
Release Year	It gives release year of the tool when it was first in use.

Table 2 describes this information for all considered tools. Astree [17] developes for a specific domain of embedded systems. It looks for variables' issues, memory usage, dangling references in C. ClangSCA [18] is fast, light, and scalable. It has library-based architecture. CodeSonar [19] works as a listening software that scans for applications that could use a C/C++ compiler. CppCheck [20] is an open-source tool, detects many rules. The most recent version of 2017 covers a vast list of checks, whereas the prior version could only verify a small list of criteria. FlawFinder [21] is simple but useful tool. The built-in database is examined by this tool to check if any so-called "flaws" or vulnerabilities exist. It also lists the severity of the problems. Frama-C [22] is a reliable framework for the analysis of C program that includes a few plugins for static analysis or verification. Infer [23] checks for null pointer dereferences, memory leaks, coding standards, and inaccessible APIs. Infer is integrated with Facebook's code review system.

ITS4[24] effectively notices the huge number of code lines. It also works on pattern matching. MOPS [18] basically checks for security properties and identify whether it is observed or not. Parasoft [26] employs as a testing tool, but it also analyzes source code files. RATS (Rough Auditing Tool for Security) [27] notices destructive function calls. Its goal is to provide a reasonable starting point for performing manual security audits. Sparse [28] uses for kernel security and it is one of the command line semantic scanners for files written in C language. Splint is also an open-source static code analysis tool. It is only for C. VisualCodeGrepper (VCG) [10],[12],[15],[16] is open source and mostly used widely in industry.

Static code analysis tools work either on program files or binary files. They internally work on different rules from simple pattern matching to complex symbolic execution. They consider different flows within a code.

Table 2. Tool General Information

Citation	*Tool Name*	*L*	*P*	*E*	*A*	*Output*	*Version*	*Release Year*
[17]	**Astree**	C/C++	W	N	C	CMD	22.04	2001
[12]	**Clang SCA**	C/C++	L	Y	O	CMD, HTML	3.8	2009
[6]	**CodeSonar**	C/C++	WL	Y	C	Text, HTML, XML, CSV	3.3	2007
[7]	**CppCheck**	C/C++	WL	Y	O	Text, HTML, XML, CSV	1.86	2007
[10]	**FlawFinder**	C	WL	Y	O	Text, HTML, XML, CSV	2.0.6	2001
[3]	**Frama-C**	C/CC++	L	Y	O	Text	21	2008

[14]	**Infer**	C, Java, PHP	L/M	Y	O	Text, HTML, XML, CSV	0.15	2008
[12]	**ITS4**	C/C++	L	N	C	Text	-	2000
[18]	**MOPS**	C	L	Y	O	Text, HTML, XML, CSV	0.9.2	2004
[19]	**Parasoft**	C/C++, Java, Python	W	Y	C	Pie Chart / Text	-	1987
[26]	**RATS**	C/C++, Perl, PHP, python	WL	Y	O	HTML, XML	4.18	2001
[27]	**Sparse**	C	L	Y	F	Linux OS output	-	2003
[28]	**Splint**	C	L	Y	O	Text, HTML, XML, CSV	3.1.2	2007
[30]	**Visual Code Greeper VCG**	C++, C#, VB, PHP, Java, PL/SQL	W	Y	O	Graph, HTML, XML, CSV	2.2.0	2014

Table 3: Tools working Details

Tool Name	***Specific Purpose***	***Rule***	***Usage***	***Command***
Astree	Embedded Software Security	Abstract Interpretation	GUI, Batch mode with annotations	-
Clang SCA	Security	Symbolic Execution, inter procedural data-flow analysis	Command Line	scan-build clang filename.c
CodeSonar	Security	Data Flow, Symbolic execution	GUI	codesonar hook-html <project-name> <command>
CppCheck	General Purpose	Pattern Matching, AST, intra-procedure	Command line, GUI	cppcheck filename.c
FlawFinder	General purpose	Pattern Matching from built-in Database	Command Line	flawfinder filename.c
Frama-C	General / Security	abstract interpretation	Command Line/GUI	frama-c file.c -<plugin> / frama-c-gui file.c
Infer	Security, Runtime errors	biabduction, Inter procedural	Online, Command prompt, GUI	Infer run – gcc –c filename.c
ITS4	Bad function	Abstract Syntax Tree, Pattern matching	Command line	-
MOPS	Security	FSM, Control Flow Graph	-	gcc –B filename.c > filename.cfg
Parasoft	Security	pattern-based analysis, dataflow analysis, abstract interpretation	GUI Based	-
RATS	Security	Pattern Matching	-	rats filename.c
Sparse	Security	Semantic Checker for Kernel Code	-	sparse filename.c
Splint	security	Intra-procedural data flow analysis, Annotation	Command Line	splint filename.c
Visual Code Greeper VCG	Security	Pattern Matching	GUI and Command Prompt	visualcodegrepper.exe -c -v -l -t --results

They are data flow or control flow analysis. With reference to code information about data, they are either flow-sensitive or flow-insensitive, content-sensitive, or insensitive, field sensitive or insensitive. For the tools that have been tested, we haven't concentrated on the code information in our work. So, we have not included that part of the tool's details. Table 3 shows the rules, specific purpose, usage, and command used to run the tool.

4. Tool Analysis

The following is the primary justification for empirically examining the tools employing Juliet Test Suit test cases:

1) Use the tools and evaluate their results to identify various vulnerabilities.
2) Find false positive, true positive addition to that find true negative (i.e., tools not able to detect vulnerabilities). We were able to assess recall and accuracy using these performance measures.
3) We also examined a significant number of test cases to determine the tools' detection capability.

For each vulnerability, we have assigned a CWE number and divided them into two broad group such as input data vulnerability and operating system (OS) vulnerability. We part them as you focus on memory related vulnerabilities. The vulnerabilities that we find using tools are described in input data vulnerability (table

4) and operating system (OS) vulnerability (table 5).
We used the identical machine configuration to test each tool for evaluation. With 2GB RAM, we utilized Windows 11, Ubuntu 18.04. Our goal was to provide them with a comparable environment. It was to check security flaws in C code related to input as well as OS weaknesses.

Table 4. Input Data validation

CWE-No	***20***	***457***
Name	***Improper Input Validation***	***Uninitialized variable***
Astree	Y	Y
Clang SCA	N	N
CodeSonar	Y	Y
CppCheck	N	Y
FlawFinder	N	N
Frama-C	N	Y
Infer	Y	N
ITS4	N	Y
MOPS	Y	Y
Parasoft	Y	N
RATS	Y	Y
Sparse	N	N
Splint	Y	Y
VCG	Y	N

Table 5. OS vulnerability

CWE-No	***121***	***122***	***367***	***362***
Name	***Stack Overflow***	***Heap Overflow***	***TOC-TOU***	***Race Condition***
Astree	N	Y	Y	N
Clang SCA	N	Y	N	N
CodeSonar	Y	N	Y	Y
CppCheck	N	Y	Y	N
FlawFinder	Y	N	N	N
Frama-C	N	N	Y	Y
Infer	Y	Y	N	Y
ITS4	N	N	Y	Y
MOPS	Y	N	Y	Y
Parasoft	Y	Y	N	Y
RATS	Y	Y	N	Y
Sparse	N	Y	N	Y
Splint	N	Y	Y	N
VCG	N	Y	Y	N

In addition to further information, table 6 lists the security vulnerabilities we search for, and we express them using our notation and CWE number.

5. Result and Discussion

In our experiments, we investigated the detecting abilities of total fourteen tools. It quickly became evident that the tools built on annotations had promising results but also imposed more demands on their users. Other side, the highest technical competence was needed for Frama-C since it was difficult and time consuming to analyze its output. Even Splint verified software effectively with enough annotations, but this required about the same amount of programming effort in the annotation language, which may be unfamiliar to many developers. When employed with the necessary skill level, sufficiently integrated into the project, and used on the desired goals, all the tools that were chosen would be advantageous for software projects. Small enterprises found it exceedingly expensive to employ commercial tools. Modern tools were more stable and effective for detection. They also produce the output in user friendly way. CppCheck was extremely helpful, and it was also quite simple to comprehend its results. We noted that the CWE numbers provided by tools in reports varied from one another. They did not use the same CWE number. Therefore, it was challenging to distinguish between vulnerabilities based just on their CWE numbers. Due to space restrictions, not all screenshots from all categories are included here. Table 7 describes detection capabilities for finding security vulnerabilities for each tool.

Table 6. Security Vulnerability notations

NO	**CWE**	**Name**
V1	**134**	Format string vulnerability
V2	**170**	Improper Null Termination
V3	**244**	Heap Inspection
V4	**251**	Often Misused: String Management
V5	**787**	Array Index Out of Bounds- Write
V6	**415**	Double Free
V7	**416**	Use After Free
V8	**468**	Unintentional pointer scaling
V9	**478**	Null Dereference
V10	**489**	Leftover Debug Code
V11	**125**	Out of Bound Array Indexing
V12	**190**	Integer Overflow or Wraparound
V13	**369**	Divide by Zero
V14	**785**	Use of Path Manipulation Function without Maximum Sized Buffer
V15	**401**	Memory Leak
V16	**120**	Buffer Overflow

We employed four metrics—accuracy, recall, false alarm probability (PF), and detection ratio—as performance measures. They stand for various tool performance qualities when it comes to identifying security vulnerabilities. The confusion matrix, which represents the total number of true positives (TP), false negatives (FN), false positives (FP), and true negatives (TN), had to be calculated before we calculated these performance measures. We then computed these measures for our experimental study using these confusion matrices. Below equations (1), (2) and (3) are the way these four measures are defined.
The essential criteria to compare these tools is another metric called detection ratio, which is used to categories vulnerabilities and determine whether a certain tool identifies a given type of vulnerability or not. Table 8 shows four performance measures for chosen SCA tools.

We found that just a few tools can perform the same task with greater accuracy than with lower recall. So, vulnerability categorization has been done and marked for each tool and then detection ratio has been calculated.

$$Accuracy = \frac{TN + FP}{TN + FP + FN + TP} \dots (1)$$

Table 7. Security Vulnerability detection

No	V1	V2	V3	V4	V5	V6	V7	V8	V9	V10	V11	V12	V13	V14	V15	V16
Astree	Y	N	N	Y	Y	Y	Y	N	Y	Y	Y	Y	N	N	N	N
Clang SCA	Y	Y	Y	N	N	Y	N	Y	N	N	N	N	N	N	Y	Y
CodeSonar	N	N	Y	Y	Y	Y	Y	N	N	Y	Y	N	Y	Y	N	N
CppCheck	Y	N	N	N	N	N	N	N	Y	N	N	N	N	N	N	N
FlawFinder	N	Y	Y	N	N	N	N	Y	Y	N	N	Y	N	Y	Y	Y
Frama-C	N	Y	N	Y	Y	N	Y	Y	Y	Y	Y	Y	Y	N	Y	Y
Infer	Y	Y	Y	Y	N	N	Y	Y	N	Y	Y	N	Y	N	Y	Y
ITS4	Y	N	N	Y	Y	Y	Y	N	Y	Y	Y	N	Y	Y	N	N
MOPS	Y	N	N	N	N	N	N	N	N	N	N	Y	N	Y	N	N
Parasoft	N	Y	Y	N	N	N	N	Y	N	N	N	Y	N	Y	N	N
RATS	N	Y	Y	Y	Y	Y	Y	N	Y	N	N	Y	N	N	N	Y
Sparse	Y	N	Y	N	Y	Y	Y	N	Y	Y	Y	N	N	N	Y	N
Splint	Y	Y	N	N	Y	Y	Y	Y	Y	N	N	N	N	N	Y	Y
VCG	N	N	N	Y	N	N	N	Y	N	N	N	N	Y	Y	Y	N

$$Recall = \frac{TP}{TP + FN} \dots (2)$$

$$PF = \frac{FP}{TN + FP} \dots (3)$$

Important empirical findings include:

1) None of the selected tools was able to detect all vulnerabilities. Specifically, out of the 22 C/C++ CWEs, none of the tools was able to detect at most 15 CWEs (i.e., 68%) were detected by a single tool and only 10 CWEs (approx. 31%) were detected by all tools.
2) The ability of the chosen SCA tools to identify security vulnerabilities for the C programming language did not demonstrate statistically significant differences.
3) One of the tools performed better than the others for C vulnerabilities in terms of probability of false alarm and recall.

Table 8. Performance Measures

Tool Name	Accuracy	Recall	PF	Detection Ratio
Astree	52.15	37.76	0.43	55.22
Clang SCA	69.39	57.68	0.91	67.95
CodeSonar	63.14	50.21	0.98	73.57
CppCheck	69.55	55.99	0.32	53.09
FlawFinder	51.1	73.51	0.5	69.59
Frama-C	50.8	76.54	0.98	51.75
Infer	78.94	61.13	0.64	69.6
ITS4	47.62	58.41	0.29	59.54
MOPS	68.56	51.24	0.65	46.57
Parasoft	56.12	77.34	0.23	48.54
RATS	74.91	60.58	0.15	53.13
Sparse	50.49	33.28	0.42	74.02
Splint	64.14	51.58	0.67	60.97
VCG	80.62	55.83	0.78	52.06

6. Conclusion

In this paper, our focus was on evaluating static code analyzers' capacity to identify security vulnerabilities. We employed an experimental strategy based on the Juliet benchmark test suite for this objective. It enabled us to automatically assess many test cases covering a wide range of C vulnerabilities to determine how efficiently the tools performed. In a contrast to research interests work, we have also provided thorough information regarding the tools and rules required to make the tools function. Despite recent advancements in this area and claims made by the tool's

developers, our experimental findings indicated that static code analyzers do not effectively identify security vulnerabilities in source code. To conform the finding that static code analysis tools have large false negative rates, we will need to integrate open-source code in future studies.

References

[1] Fatima, S. Bibi, and R. Hanif, "Comparative study on static code analysis tools for C/C++," *Proc. 2018 15th Int. Bhurban Conf. Appl. Sci. Technol. IBCAST 2018*, vol. 2018-Janua, pp. 465–469, 2018, doi: 10.1109/IBCAST.2018.8312265.

[2] H. Kaur and P. Jai, "Comparing Detection Ratio of Three Static Analysis Tools," Int. J. Comput. Appl., vol. 124, no. 13, pp. 35–40, 2015, doi: 10.5120/ijca2015905749

[3] S. M. Alnaeli, M. Sarnowski, M. S. Aman, A. Abdelgawad, and K. Yelamarthi, "Source code vulnerabilities in IoT software systems," Adv. Sci. Technol. Eng. Syst., vol. 2, no. 3, pp. 1502–1507, 2017, doi: 10.25046/aj0203188.

[4] A. Wagner and J. Sametinger, "Using the Juliet Test Suite to compare static security scanners," SECRYPT 2014 - Proc. 11th Int. Conf. Secur. Cryptogr. Part ICETE 2014 - 11th Int. Jt. Conf. E-bus. Telecommun., pp. 244–252, 2014, doi: 10.5220/0005032902440252.

[5] I. Gomes, P. Morgado, T. Gomes, and R. Moreira, "An overview on the Static Code Analysis approach in Software Development," Fac. Eng. da Univ. do Porto, Port., 2009.

[6] K. Goseva-Popstojanova and A. Perhinschi, "On the capability of static code analysis to detect security vulnerabilities," Inf. Softw. Technol., 2015, doi: 10.1016/j.infsof.2015.08.002.

[7] M. Christakis and C. Bird, "What developers want and need from program analysis: An empirical study," ASE 2016 - Proc. 31st IEEE/ACM Int. Conf. Autom. Softw. Eng., pp. 332–343, 2016, doi: 10.1145/2970276.297.

[8] A. Arusoaie, S. Ciobaca, V. Craciun, D. Gavrilut, and D. Lucanu, "A comparison of open-source static analysis tools for vulnerability detection in C/C++ Code," Proc. - 2017 19th Int. Symp. Symb. Numer. Algorithms Sci. Comput. SYNASC 2017, pp. 161–168, 2018, doi: 10.1109/SYNASC.2017.00035.

[9] D. ucanu Andrei Arusoaie, Stefan Ciobaca, Vlad Craciun, Dragos Gavrilut, "A Comparison of Static Analysis Tools for Vulnerability Detection in C / C ++ Code," vol. 190, pp. 161–168, 2017.

[10] M. Mantere, I. Uusitalo, and J. Röning, "Comparison of static code analysis tools," 2009, doi: 10.1109/SECURWARE.2009.10.

[11] A. Kaur and R. Nayyar, "A Comparative Study of Static Code Analysis tools for Vulnerability Detection in C/C++ and JAVA Source Code," Procedia Comput. Sci., vol. 171, no. 2019, pp. 2023–2029, 2020, doi: 10.1016/j.procs.2020.04.217.

[12] J. Zheng, L. Williams, N. Nagappan, W. Snipes, J. P. Hudepohl, and M. A. Vouk, "On the value of static analysis for fault detection in software," IEEE Trans. Softw. Eng., vol. 32, no. 4, pp. 240–253, 2006, doi: 10.1109/TSE.2006.38.

[13] J. Herter, D. Kästner, C. Mallon, and R. Wilhelm, "Benchmarking static code analyzers," Reliab. Eng. Syst. Saf., vol. 188, no. March, pp. 336–346, 2019, doi: 10.1016/j.ress.2019.03.031.

[14] S. Shiraishi, V. Mohan, and H. Marimuthu, "Test suites for benchmarks of static analysis tools," 2015 IEEE Int. Symp. Softw. Reliab. Eng. Work. ISSREW 2015, no. November, pp. 12–15, 2016, doi: 10.1109/ISSREW.2015.7392027.

[15] D. Stefanović, D. Nikolić, D. Dakić, I. Spasojević, and S. Ristić, "Static code analysis tools: A systematic literature review," Ann. DAAAM Proc. Int. DAAAM Symp., vol. 31, no. 1, pp. 565–573, 2020, doi: 10.2507/31st.daaam.proceedings.078.

[16] J. Novak, A. Krajnc, and R. Žontar, "Taxonomy of static code analysis tools," MIPRO 2010 - 33rd Int. Conv. Inf. Commun. Technol. Electron. Microelectron. Proc., no. March, pp. 418–422, 2010.

[17] J. S. Delmas David, "Astrée: from research to industry," Int. Static Anal. Symp. Springer, pp. 437–451, 2007, doi: 10.1007/978-3-540-74061-2_27.

[18] "Clang-Static Code Analyzer." https://clang-analyzer.llvm.org/ (accessed Nov. 22, 2020).

[19] "CodeSonar." https://www.grammatech.com/codesonar-cc (accessed Nov. 22, 2020).

[20] D. Marjam¨aki, "CppCheck." Cppcheck - A tool for static C/C++ code analysis (sourceforge.io) (accessed Oct. 22, 2020).

[21] D. Wheeler, "FlawFinder." Flawfinder Home Page (dwheeler.com) (accessed Oct. 22, 2020).

[22] F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski, "Under consideration for publication in Formal Aspects of Computing Frama-C A Software Analysis Perspective," 2012, [Online]. Available: https://frama-c.com/.

[23] C. Calcagno et al., "Moving Fast with Software Verification – Facebook Research," [Online]. Available: https://research.fb.com/publications/moving-fast-with-software-verification/

[24] J. Viega, J. T. Bloch, Y. Kohno, and G. McGraw, "ITS4: A static vulnerability scanner for C and C++ code," Proc. - Annu. Comput. Secur. Appl. Conf. ACSAC, pp. 257–267, 2000, doi: 10.1109/ACSAC.2000.898880.

[25] H. Chen and D. Wagner, "Mops," p. 235, 2002, doi: 10.1145/586110.586142.

[26] "Parasoft." https://www.parasoft.com/ (accessed Oct. 19, 2020).

[27] "RATS."https://github.com/andrew-d/rough-auditing-tool-for-security (accessed Sep. 15, 2019).

[28] "Sparse."https://man7.org/linux/man-pages/man1/sparse.1.html (accessed Aug. 19, 2020).

[29] D. Evans and D. Larochelle, "Splint," no. October 2001, 2002.

[30] "Visual Code Greeper." https://security.web.cern.ch/recommendations/en/codetools/vcg.shtml (accessed Aug. 08, 2018).