# Comparing Detection Ratio of Three Static Analysis Tools

Hanmeet Kaur Brar
Student, UIET, Panjab University
Chandigarh
India.

Puneet Jai Kaur
Asstt. Professor, UIET, Panjab University
Chandigarh
India

## ABSTRACT

Static code analysis is a software verification activity in which source code is scrutinized for quality and security. In a Software Development Lifecycle, timely detection of flaws is beneficial and static analysis tools help us to detect flaws at a very early stage. Both commercial and open source static analysis tools are available today. Due to diverse user requirements and capabilities of the tools, a comparison between tools is required. Three open source static analysis tools for security are evaluated in this paper. These are Cppcheck, RATS and Flawfinder. They have been studied and compared to each other on the basis of detection ratio. For the purpose of obtaining the detection ratio, the vulnerabilities were categorized and intentionally introduced into the demo codes.

## General Terms

Security.

## Keywords

Software development life cycle; Static analysis; Static analysis tools; Detection Ratio; Vulnerabilities; Security; Assessment.

## 1. INTRODUCTION

Software security has become vital due to high inclusion of software applications in almost every sphere of our lives. Code with compromised quality may become functional but can be a great threat to security of the software. The problem of software security emerges from vulnerabilities in software. The main cause of these vulnerabilities is either improper coding done by the coder [1] or a deficiency in the language in which coding is being done. In both the cases, figuring out and then eliminating these vulnerabilities is quite important and that too in early stages of Software Development Life Cycle (SDLC). Otherwise this may lead to disastrous results [2]. Avoiding investment for this purpose at earlier stages may lead to great damages at later stages and some of these may become irreparable at that time [3].

Nearly 90% of detected hacking attacks related to security arise because of flawed coding [4]. So finding the inadequacy in software development and then working upon it is very important and should be done. There are numerous ways for improvement of security of software and those include firm model usage for the purpose of design development, raising awareness related to security among the programmers, safe environment for running software, etc. [5].

Any kind of vulnerabilities can be discovered either manually examining the source code or with usage of tools which are automated. In the first case, that is manual examination of the code, the time taken may reach unbearable limits. Moreover it is a tedious job. In few cases, examining manually is not even possible [6]. So here we take the help of static analysis tools. These tools provide support to manual approach of examining code, by pointing out the vulnerabilities or the potential risks. Thus they save energy as well as time.

Due to immense effectiveness of these tools in code analysis, many tools are being designed and used. The significance of these tools can be estimated by the truth that static analysis tools have been made a vital part in SDLC of many companies including Microsoft [7]. Although these tools detect vulnerabilities automatically, these need to be operated manually. Deciding that a vulnerability detected by the tool is really a risk has to be done manually. But still these tools can be of great help if operated intelligently to aid the manual approach [8].

Various studies and research have already been done on static analysis tools and their utility. Many comparative studies have also been done. It is quite important to comparatively analyze the tools separately for each field. It is important to judge their utility, strengths and weaknesses as compared to other tools in the same field.

These tools can uncover various categories of vulnerabilities but their ability is not regular across the spectrum of the vulnerabilities. One tool can detect a particular category while the other tool cannot and vice versa. Sometimes even this happens that a tool detects a particular category but skips few of its variants [9]. These things are not mentioned by all user manuals and hence require research.

In this paper, the selected 3 static analysis tools have been studied and compared to each other on the basis of detection ratio. For the purpose of obtaining the detection ratio, there was a need to categorize the vulnerabilities and indicate the detection and ignorance of each category and the same has been done in this paper.

Arrangement of the rest of the paper is as follows. Section 2 gives a brief a brief introduction to static code analysis. In section 3 static analysis tools used in this paper and the criteria for their selection is discussed. Section 4 explains Detection Ratio and Section 5 includes the results and their analysis. Section 6 and 7 contain the conclusion and any future work possible respectively. Section 8 contains the references.

## 2. STATIC CODE ANALYSIS

Static code analysis is an activity invloving the inspection of a source code for quality and security [10]. It helps the software developers and testers in detecting and making out several types of flaws—e.g. dead code, divide by zero, overflow of bufffer boundaries, out of bounds read/write, etc.—essentialy without running the code. The flaws thus detected can easily be taken out by the programmer. Thus, a more efficient code can be generated with the help of static code analysis.

Static code analysis uncovers "hard" bugs before runtime which may be impossible to detect during rumtime e.g. memory leaks which increase memory footprint but do not affect the functioning of the programs. Finding flaws in a very long code is not possible manually [11].

Thus to detect security vulnerabilities early in a software development lifecycle, automatic static analysis is used [12]. The objective of doing this is to decrease the time and work required to do a code review. The quest to automate the code reviews began with simple program checkers [13], but more complicated and competent tools for different languages were developed later on. These tools help the programmer in making their code secure, stable, efficient and dependable.

Staic analysis has many advantages. The program to be analyzed does not have to be complete. Static analysis can be used very early in the software development lifecycle. Thus, an early report on software quality is received. This reduces the rework cost and development productivity increases. Test cases do not need to be designed and "hard" bugs like memory leaks can be detected [14]. Tool has access to the entire code i.e. it has full access to all of the software's possible behaviors. Thus, it does not need to guess or understand behavior [15].

Static analysis also has its fair share of disadvantages like production of false positives which have to be inspected later on. Tools like Flawfinder, RATS, ITS4 report a large number of false positives [16]. In order to perform a build, complete access to source code or at least binary code is required. Proficiency running software builds are characteristically needed. Flaws connected to operational deployment environments will not be detected [15].

## 3. STATIC ANALYSIS TOOLS FOR SECURITY

Static analysis tools made for security are kind of programs written to statically review the code for security-centric analysis. These tools are used for automation of analysis work to save energy as well as time. They may not always point out actual defects and may not find all the defects but warn about the presence of risk in some or the other form. They basically aid manual approach of code [17][18].

### 3.1 Selection Criteria

Cppcheck, Flawfinder, and RATS have been selected for this research. The thought behind their selection is to present distinct approaches towards the same problem. The tools used are open source/free as no financial budget is involved. These tool selected are the latest ones amongst the open source static analysis tools for security.

RATS and Flawfinder focus primarily on detecting security vulnerabilities but on the other hand, Cppcheck offers much wider analysis potential than both RATS and Flawfinder.

### 3.2 Description of Tools

The tools used for research are: RATS, Flawfinder and Cppcheck. Little more discussion of the tools is done here.

#### *3.2.1 RATS*

Rough Auditing Tool for Security (RATS) is a tool for auditing the source code developed in C, C++, Python, Perl, and PHP. Secure Software Inc. developed it originally. It figures out potential flaws related to security [19].

It does not uncover all the vulnerabilities present in the code as well as it may point things as potential risks that are not actually problems. This means it gives prominent number of false positives as well as false negatives. So it is mainly to aid manual inspection [19].

#### *3.2.2 Flawfinder*

Flawfinder is a static analysis tool for C/C++ programming languages, mainly meant for security. It reports the potential security vulnerabilities. It is compatible with CWE, officially. This means along with the warnings it reports, Flawfinder also reports CWE error code for that vulnerability. Author of this tool is David A. Wheeler [20].

Flawfinder presents the vulnerabilities as "hits" which are then sorted in descending order by their risk level. Risk level can have integer values stretching from 0 to 5, where 0 indicates the minimum risk and 5 indicates very high risk [20].

#### *3.2.3 Cppcheck*

Cppcheck is a tool used for static code analysis for programming languages C/C++. It does not uncover the syntax errors as the compiler does. It uncovers the errors and warnings that the compilers generally skip. It is platform independent and only requirement is enough memory space and CPU to work. Daniel Marjamaki is the creator of this tool as well as the lead developer [21].

It is generally not wrong about the errors it report but the chances are there that it reports less number of errors that the actual number of errors present in the code. This means it aims at minimal false positives but can have many false negatives [21].

## 4. DETECTION RATIO

Detection Ratio can be defined as a measure to judge the effectiveness of an error detection tool. It is extremely useful in scenarios where the error detection tools have to be compared and rated on the basis of number of error categories detected. Detection Ratio helps to give the programmer an idea about which among the error detection tools being used has a broader spectrum. Therefore it can be said that, Higher the detection ratio, wider the scope of error detection tool being considered. The following is the formula for detection ratio:

$$Detection\ Ratio = \frac{No. of\ Vulnerabilities\ Detected}{Total\ Vulnerabilities\ Introduced} \quad (1)[22]$$

Its value varies from 0 to 1. The detection ratio is effective when the error detection tools have to be determined regardless of the type or category of error detected but on the basis of the area of coverage for a tool. For example, if a person wants to choose an error detection tools and he is clear about a particular category he wants to detect, then detection ratio may not help him. But if the person wants to choose a tool that covers maximum range of errors or vulnerabilities, then detection ratio can be of great help for him and he should go by higher value of detection ratio.

## 5. RESULTS AND ANALYSIS

The current section of the paper portrays the comparative analysis of the tools selected. Detection ratio has been chosen as the main parameter to compare these tools. To calculate detection ratio, there was a need to first categorize the vulnerabilities and check if a particular tool detects that category of vulnerability or not. So vulnerability categorization has been done and marked for each tool and then detection ratio has been calculated.

For the purpose of comparative evaluation, the environment had to be identical for all selected tools. Ubuntu 14.0.LTS was chosen as the Operating System and C++ was chosen as the programming language. The version of the tools used are the latest ones and are: Cppcheck 1.69, RATS 2.4 and Flawfinder 1.31.

Different categories of vulnerabilities were intentionally introduced in different C++ applications. These applications were fed as input to all selected tools separately and outcomes were marked and are shown in this section.

Table 1 displays the different categories of vulnerabilities that were introduced in different applications. It has been tried to introduce as many categories as possible from the domain of each tool. Care has been taken that categories of any particular tool were not concentrated upon.

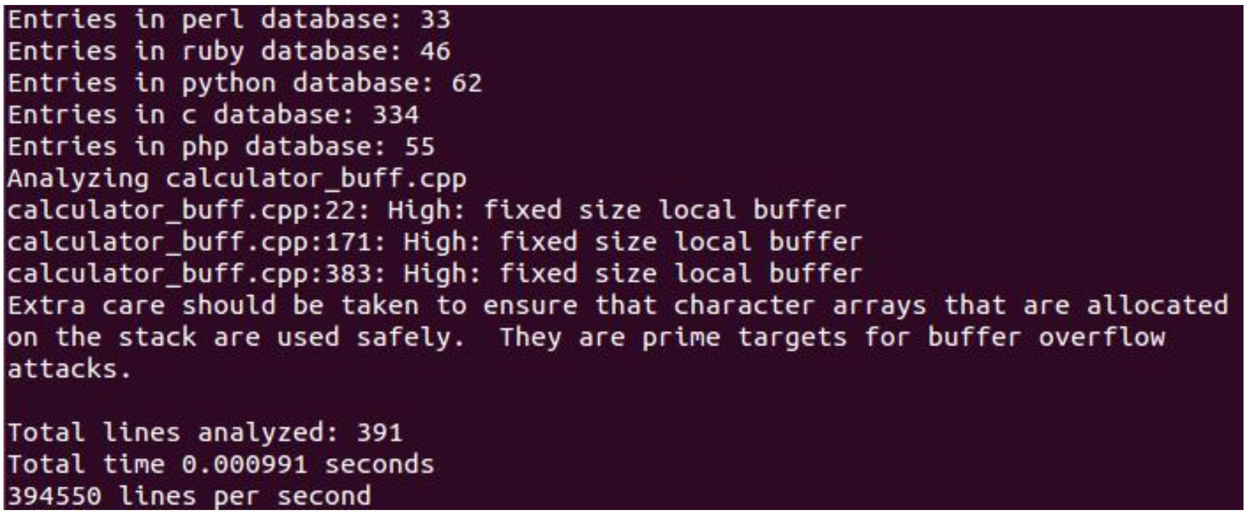
After the vulnerabilities were introduced in the applications and those were executed on each tool, the outcomes were recorded. The outcomes are furnished with the help of tick mark for detection of vulnerability and a cross mark of non-detection.

Figure 1, Figure 2 and Figure 3 are the screenshots of the tools. All the screenshots for all the categories have not been shown here due to the limitation of space. So one screenshot for each tool has been shown to give an idea that what kind of errors or warnings the tools give and in what form.

**Table 1. Categorization of Vulnerabilities and their Detection**

| Code | Category | RATS | Cppcheck | Flawfinder |
|---|---|---|---|---|
| CWE -20 | Improper Input Validation | √ | √ | √ |
| CWE -78 | OS Command Injection | √ | × | √ |
| CWE -120 | Buffer Overflow | √ | √ | √ |
| CWE -125 | Array Index Out of Bounds- Read | × | √ | × |
| CWE -134 | Uncontrolled Format String | √ | × | √ |
| CWE -190 | Integer Overflow or Wraparound | × | × | √ |
| CWE -250 | Execution with Unnecessary Privileges | × | × | √ |
| CWE -327 | Use of Broken or Risky Cryptographic Algorithm | × | × | √ |
| CWE -338 | Use of Cryptographically Weak Pseudo-random Number Generator | √ | × | √ |
| CWE -362 | Race Condition | × | × | √ |
| CWE -369 | Divide by Zero | × | √ | × |
| CWE -401 | Memory Leak | × | √ | × |
| CWE -561 | Dead Code | × | √ | × |
| CWE -785 | Use of Path Manipulation Function without Maximum Sized Buffer | √ | × | × |
| CWE -787 | Array Index Out of Bounds- Write | × | √ | × |
| CWE -807 | Reliance of Untrusted Inputs in a Security Decision | √ | × | × |

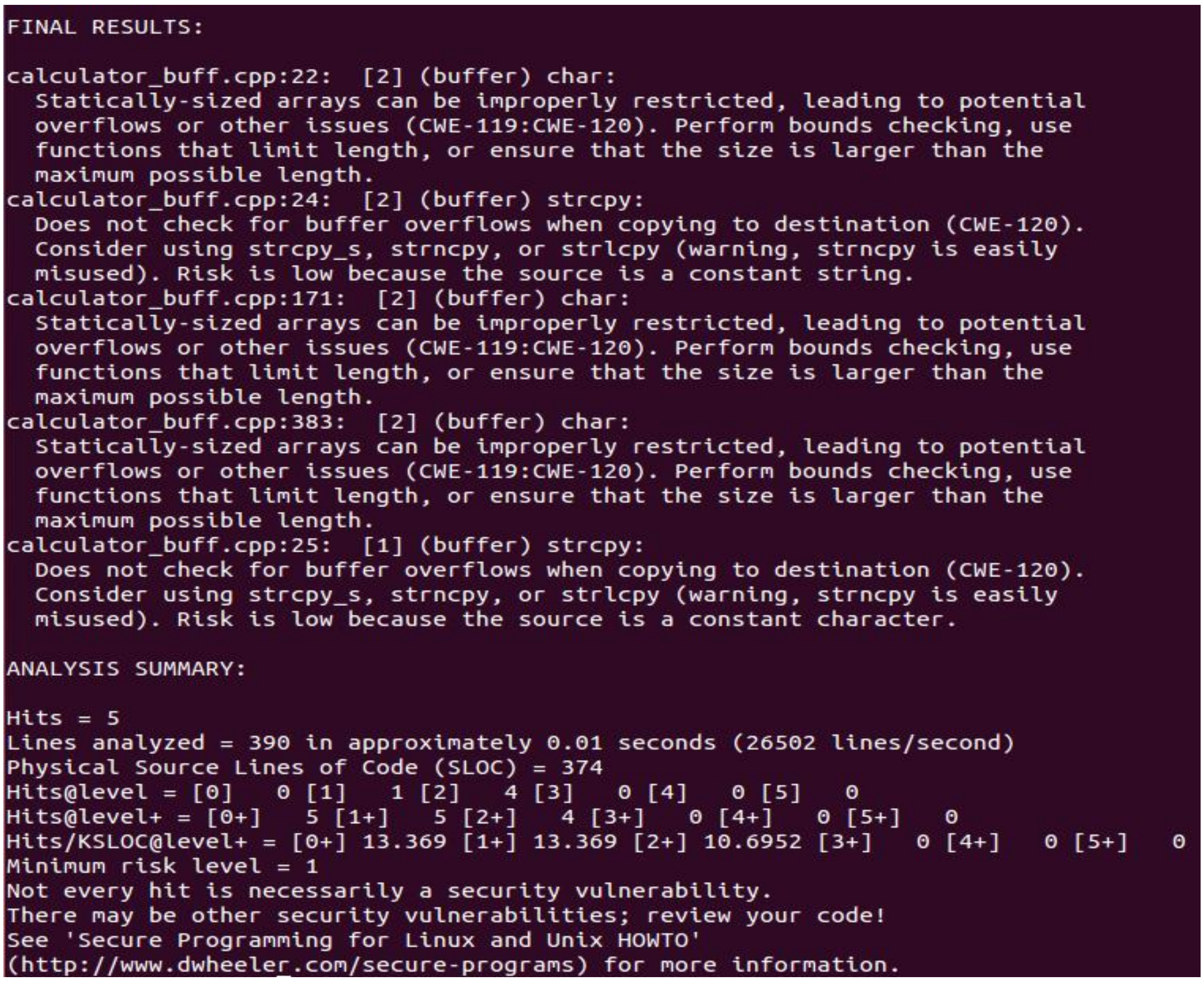
[23]



```
Entries in perl database: 33
Entries in ruby database: 46
Entries in python database: 62
Entries in c database: 334
Entries in php database: 55
Analyzing calculator_buff.cpp
calculator_buff.cpp:22: High: fixed size local buffer
calculator_buff.cpp:171: High: fixed size local buffer
calculator_buff.cpp:383: High: fixed size local buffer
Extra care should be taken to ensure that character arrays that are allocated
on the stack are used safely.  They are prime targets for buffer overflow
attacks.

Total lines analyzed: 391
Total time 0.000991 seconds
394550 lines per second
```

**Figure 1. Screenshot of RATS**



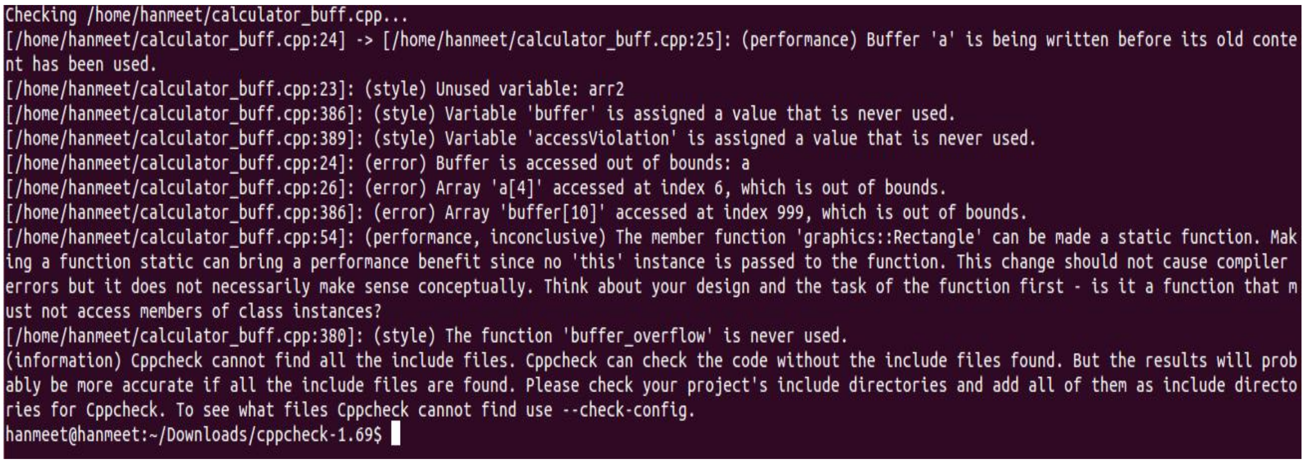
```
FINAL RESULTS:

calculator_buff.cpp:22:  [2] (buffer) char:
  Statically-sized arrays can be improperly restricted, leading to potential
  overflows or other issues (CWE-119:CWE-120). Perform bounds checking, use
  functions that limit length, or ensure that the size is larger than the
  maximum possible length.
calculator_buff.cpp:24:  [2] (buffer) strcpy:
  Does not check for buffer overflows when copying to destination (CWE-120).
  Consider using strcpy_s, strncpy, or strlcpy (warning, strncpy is easily
  misused). Risk is low because the source is a constant string.
calculator_buff.cpp:171:  [2] (buffer) char:
  Statically-sized arrays can be improperly restricted, leading to potential
  overflows or other issues (CWE-119:CWE-120). Perform bounds checking, use
  functions that limit length, or ensure that the size is larger than the
  maximum possible length.
calculator_buff.cpp:383:  [2] (buffer) char:
  Statically-sized arrays can be improperly restricted, leading to potential
  overflows or other issues (CWE-119:CWE-120). Perform bounds checking, use
  functions that limit length, or ensure that the size is larger than the
  maximum possible length.
calculator_buff.cpp:25:  [1] (buffer) strcpy:
  Does not check for buffer overflows when copying to destination (CWE-120).
  Consider using strcpy_s, strncpy, or strlcpy (warning, strncpy is easily
  misused). Risk is low because the source is a constant character.

ANALYSIS SUMMARY:

Hits = 5
Lines analyzed = 390 in approximately 0.01 seconds (26502 lines/second)
Physical Source Lines of Code (SLOC) = 374
Hits@level = [0]   0 [1]   1 [2]   4 [3]   0 [4]   0 [5]   0
Hits@level+ = [0+]   5 [1+]   5 [2+]   4 [3+]   0 [4+]   0 [5+]   0
Hits/KSLOC@level+ = [0+] 13.369 [1+] 13.369 [2+] 10.6952 [3+]   0 [4+]   0 [5+]   0
Minimum risk level = 1
Not every hit is necessarily a security vulnerability.
There may be other security vulnerabilities; review your code!
See 'Secure Programming for Linux and Unix HOWTO'
(http://www.dwheeler.com/secure-programs) for more information.
```

**Figure 2. Screenshot of Flawfinder**



```
Checking /home/hanmeet/calculator_buff.cpp...
[/home/hanmeet/calculator_buff.cpp:24] -> [/home/hanmeet/calculator_buff.cpp:25]: (performance) Buffer 'a' is being written before its old conte
nt has been used.
[/home/hanmeet/calculator_buff.cpp:23]: (style) Unused variable: arr2
[/home/hanmeet/calculator_buff.cpp:386]: (style) Variable 'buffer' is assigned a value that is never used.
[/home/hanmeet/calculator_buff.cpp:389]: (style) Variable 'accessViolation' is assigned a value that is never used.
[/home/hanmeet/calculator_buff.cpp:24]: (error) Buffer is accessed out of bounds: a
[/home/hanmeet/calculator_buff.cpp:26]: (error) Array 'a[4]' accessed at index 6, which is out of bounds.
[/home/hanmeet/calculator_buff.cpp:386]: (error) Array 'buffer[10]' accessed at index 999, which is out of bounds.
[/home/hanmeet/calculator_buff.cpp:54]: (performance, inconclusive) The member function 'graphics::Rectangle' can be made a static function. Mak
ing a function static can bring a performance benefit since no 'this' instance is passed to the function. This change should not cause compiler
errors but it does not necessarily make sense conceptually. Think about your design and the task of the function first - is it a function that m
ust not access members of class instances?
[/home/hanmeet/calculator_buff.cpp:380]: (style) The function 'buffer_overflow' is never used.
(information) Cppcheck cannot find all the include files. Cppcheck can check the code without the include files found. But the results will prob
ably be more accurate if all the include files are found. Please check your project's include directories and add all of them as include directo
ries for Cppcheck. To see what files Cppcheck cannot find use --check-config.
hanmeet@hanmeet:~/Downloads/cppcheck-1.69$
```

**Figure 3. Screenshot of Cppcheck**

Total 16 categories of vulnerabilities were introduced and out of which Cppcheck could detect 7; RATS could detect 7 whereas Flawfinder could detect 9 out of 16. Although the count for both RATS and Cppcheck is same but this does not mean that they detect the same categories. Only the count is same but the detection of particular categories varies for both.

Equation (1) is used to calculate Detection Ratio. Table 2 shows the detection ratio calculated. The table presents the picture of how the detection ratio varies.

**Table 2. Detection Ratio**

| **Tool** | **Number of Vulnerabilities Detected** | **Detection Ratio** |
|---|---|---|
| RATS | 7 | 7/16=0.4375 |
| Cppcheck | 7 | 7/16=0.4375 |
| Flawfinder | 9 | 9/16=0.5625 |

## 6. CONCLUSION

The prime intention of the research was to ease the choice of static analysis tool for the developers or the testers. The main parameter considered was the detection ratio that came out to be highest for Flawfinder among the three tools under consideration whereas for RATS and Cppcheck, it came out to be same. This does not mean that the results will always be same under all circumstances. The results may vary with the variation of categories of vulnerabilities introduced. It has been tried to include maximum categories from the domain of each of these tools. Thus if one wants to go for detection of a particular category of vulnerability, then he may go by vulnerability categorization and detection or if he simply wants the tool to have wide spectrum or wide coverage of categories, then he may go by Detection ratio.

## 7. FUTURE WORK

Detection Ratio has been considered as a parameter to evaluate three different tools. Many other parameters like performance, accuracy, reliability, precision etc. over a larger number of tools and taking different operating systems can also be evaluated to further help testers in choosing the tool of their choice.

## 8. REFERENCES

[1] McGraw, Gary, and John Viega. "Building Secure Software." In RTO/NATO Real-Time Intrusion Detection Symp. 2002.

[2] R. Jetley, B. Chelf. "Diagnosing Medical Device Software Defects Using Static Analysis." Coverity. Published in MD&DI (2009).

[3] H. K. Brar, P. J. Kaur, “Differentiating Integration Testing and Unit Testing”, 2015 2nd International Conference on Computing for Sustainable Global Development (INDIACom), IEEE, pp. 796-798.

[4] Wang, J. A., Wang, H., Guo, M., & Xia, M., “Security metrics for software system,” Proceedings of the 47th Annual Southeast Regional Conference, pp. 47, New York: ACM, 2009.

[5] P. Li, B. Cui. "A comparative study on software vulnerability static analysis techniques and tools." In Information Theory and Information Security (ICITIS), 2010 IEEE International Conference on, pp. 521-524. IEEE, 2010.

[6] M. Mantere, I. Uusitalo, and Juha Röning. "Comparison of static code analysis tools." In 2009 Third International Conference on Emerging Security Information, Systems and Technologies, pp. 15-22. IEEE, 2009.

[7] M. Howard and S. Lipner, “The Security Development Lifecycle: SDL: A process for developing demonstrably more secure software,” Microsoft Press, 2006, ISBN-13: 978-0735622142.

[8] B. Chess and J. West, “Secure programming with static analysis,” Addison-Wesley, 2007, ISBN-13: 978-0321424778.

[9] “On analyzing static analysis tools”, National security Agency Center for Assured Software, July 26, 2011, pp. 1-13.

[10] A. German, “Software static code analysis lessons learned,” Crosstalk, vol. 16, no. 11, 2003.

[11] Vincenzo Ciriello, Gabriella Carrozza and Stefano Rosati, “Practical experience and evaluation of continuous code static analysis with C++ test,” , In Proceedings of the 2013 International Workshop on

Joining AcadeMiA and Industry Contributions to testing Automation, pp. 19-22, ACM New York, 2013.

[12] S. Lipner, “The trustworthy computing security development lifecycle.” Proceedings of the 20th Annual Computer Security Applications Conference (ACSAC), 2004.

[13] S.C. Johnson, “Lint, a C program checker” Computer Science Tech. report 65, Bell Laboratories, 1978.

[14] Patrik Hellström, “Tools for static code analysis: A survey,” Department of Computer and Information Science, Linköping University, 2009.

[15] Dan Cornell, “Static analysis techniques for testing application security,” OWASP San Antonio, 2008.

[16] Misha zitser, Richard Lippmann and Tim Leek, ‘Testing static analysis tools using exploitable buffer overflows from open source code,” ACM New York, 2004.

[17] S. Wagner, J. Jürjens, C. Koller, and P. Trischberger. "Comparing bug finding tools with reviews and tests." Lecture Notes in Computer Science 3502, 2005, pp. 40-55.

[18] H.H. AlBreiki, and Q. H. Mahmoud. "Evaluation of static analysis tools for software security." In Innovations in Information Technology (INNOVATIONS), 2014 10th International Conference on, pp. 93-98. IEEE, 2014

[19] RATS Information Website. URL: https://code.google.com/p/rough-auditing-tool-for-security/

[20] Flawfinder Website. URL: http://www.dwheeler.com/flawfinder/

[21] Cppcheck 1.69 Manual. URL: http://cppcheck.sourceforge.net/manual.pdf

[22] M. A. A. Mamun, A. Khanam, H. Grahn, and R. Feldt. "Comparing four static analysis tools for java concurrency bugs." In Third Swedish Workshop on Multi-Core Computing (MCC-10). 2010.

[23] Common Weakness Enumeration(CWE) website. URL: https://cwe.mitre.org/