Automated Static Analysis: Survey of Tools

Eati Sri Sai Srujan[1], Chaitanya J.V.N.M[2], A. Sai Manikanta[3]

[123]Student, Department of Computer Science, VIT University, Vellore.

ABSTRACT: *Not any error detection tool, is capable of detecting, processing and rectifying all the errors. Our main aim is to increase the level of authenticity that ASA (Automatic Static Analysis) can provide us. These Static analysis tools are used to check for vulnerabilities in systems and programs, as the correctness or authenticity of the program is the greatest concern in developing them and verifying them prior to their release . These tools use a wide variety of functions, to prevent many errors and loopholes from occurring at many different stages of the programs. But still, ASA tools provide a lot of false positives that again require a lot of human involvement to rectify them. Here we review the different techniques, and methods that are primarily used in Automatic Static Analysis, and also that coding concerns that arise in the process.*

Key Words: **Automated Static Analysis, Error Detection, Tools, Refactoring, Metrics.**

1.INTRODUCTION

Now-a-days we use software tools for literally everything and anything, as their ease of use and applications are very high, but there still exits one huge flaw with these tools, namely their security. Low level code, i.e. code written by people who have little to no knowledge about security in programming creates many problems. How so ever the code is not perfect, hence we need to eliminate and rid the code of such imperfections for our code to run smoothly and safely, else it may lead to disastrous results. So, we can justify the investments we put in tackling such security issues. Most of the leaks, and hacks now-a-days occur due to no proper security in the code that is being used by people.

Most of the Huge Companies understand this risk and will stay many steps ahead in maintaining security as, if the security constraints are not met, they know they will lose customers and it will snowball into a huge loss for the company. So, finding and correcting bugs and errors in the code is of the foremost importance for them. And also, we need to educate the programmers in terms of Security on how important it is and ow to achieve it as they play the biggest role in avoiding these, because if they take appropriate measures while writing the code itself all these difficulties will be kept to a minimum.

Now these bugs, to be found take hours and hours of manual work, and as humans are never perfect some bugs might still be present making the whole process don't until now redundant. So, we trust this job to machines to identify and rectify these imperfections that cause security issues in a fraction if the time we take to do the same thing. But even these tools, are not perfect and have their flaws like, for suppose some tools are able to find bugs that are of a certain kind or belong to a certain class, but miss out on some varieties of bugs that the tool can't effectively find and correct. But we can use multiple tools on the same code to minimize the error level. The importance of these ASA tools can be seen clearly from the fact that many big industries and companies like Apple, Microsoft and Google, invest a lot in these to keep their customer base intact.

ASA can also be used to rectify some common problems like garbage values, unused data, compliance with coding, improper declarations, null pointers, infinite loops, etc. So, when a tool can be so flexible and servers so many purposes, it can have some problems. To get the most out of these tools, the tests are run pre-emptively, before the software is rolled out, so that they can get more done before the users report bugs and they start flagging them, and after the roll out other tests are made to make sure al the loopholes are fixed.

Also in a study, it was found out that ASA tools can distinguish between low quality components and high quality components. High quality components of the code are those that were coded or programmed properly with very little to no margin of error and provide maximum security, where are low quality components are those components that pose the utmost threat to security because of the improper coding and structure used in them. As, these low-level components can be distinguished, they can be rectified with the help of the tool or even if the tool can't repair it, it can be repaired manually, and also by that we rare reducing the entirety of problems that we manually have to go through, by us selecting only important and complicated parts that need monitoring.

But as we know these Static Analysis tools have some limitations and some shortcoming like having a very high false positive rate. False positives are those that are classified as not a threat that is positive but actually they are a bug. These false positives are those when the tools alerts us of a fault but in fact it isn't. Similarly, a true positive is when the tool actually gives us an alert when there is actually a bug or a flaw. Generally, developers and users are only interested in detecting true positives and true negatives. So, we need out tools to maintain a very high true positive rate, a low false positive rate and a very low false negative rate. The perfect Static Analysis tool will have zero false positive and false negative rates.

2. TOOLS AND TESTING

The main purpose of automated static analysis tools is to detect anomalies in code and report them. There can be a wide range of anomalies like not following the coding standards, having dead code and unused data, a null pointer or a void pointer being dereferenced, security issues, infinite loops, and other arithmetic problems. These anomalies significantly affect the running of the software and can cause potential failures. Earlier, the lexical analyser was used to deal with static analysis but now we have tools to help us in analysis and they have better functioning capabilities. Going forward we are going to discuss the tools available for automated static analysis and also discuss certain metrics which can be used to compare these different tools.

A static analysis tool must have minimal false positives and false negative while having maximal true positives. The tool must be able to predict code refactoring modifications effectively. The different kinds of tools currently available are RATS, Cppcheck and Flawfinder. These tools have failry recent releases and are not out dated.

2.1 Cppcheck

Cppcheck is a static analysis tool which used for C and C++ languages. It was created by Daniel Marjamaki. This tool can check for anomalies in non-standard code as well. The analysis checks can be performed at a source code level. This tool is more focused towards rigorous code checks. Syntax errors are not detected by Cppcheck.

2.2 RATS

The rough auditing tool for security is an analysis tool developed by Secure Software Engineers. It is an open source tool which is fast and easily integrated without overhead. RATS can be used for various languages like C, C++, PERL, PHP and Python. The tools makes an analysis of the source code and can detect things that are not errors.

2.3 Flawfinder

Flawfinder is a program that examines C and C++ source code and reports the security weaknesses and sorts it by risk level. Flawfinder has a built-in database consisting of known anomalies and this tool searches for problems in the database and then reports them if there is a match. It is run from Command line of the system and its output can be customised. Flawfinder was developed by David Wheeler.

These tools are predominantly used to find anomalies and errors in C, C++, PERL, PHP, Python. Apart from this there are special tools for static analysis of code in JAVA. Three of these tools include IntelliJ IDEA, Jlint and FindBugs.

2.4 IntelliJ IDEA

IntelliJ IDEA is a comprehensive development environment which is used to provide special tools for development including a tool for code inspection. It has a feature for code refactoring . It was developed by the JetBrains company and inspects the code by using 632 concerns which are organized into 49 groups.

2.5 Jlint

Jlint is a free static analysis tool that analyses the Java Bytecode. Syntactic checks and data flow analysis are done by Jlint. It detects synchronization problems by building a lock graph and verifying whether the graph is cycle free or not.

2.7 FindBugs

Again FindBugs is a free tool which uses static analysis to inspect the Bytecode for faults and maps the faults to the java source code. It is java oriented and runs with any virtual machine. It can analyse programs written in any version of java.

To analyse which of the above tools is better and which tool to use, depending on the programming language used, we can use certain metrics to measure the performance capabilities of the tools.

1. Fault detection ratio
   This gives the ratio of faults that are detected by the tool. Detection ratio uses the number of fault fixes fund in the CVS repository.
   Detection Ratio =
   $$\frac{\text{No. of faults detected by ASA}}{\text{Total no. of faults fixed}}$$

2. Refactoring ratio
   This tells us how effective is the tool in finding code that can be later modified to improve the design.
   Refactoring ratio =
   $$\frac{\text{No. of performed refactoring recommended by ASA}}{\text{Total no. of refactorings performed}}$$

3. False Positive ratio
   This gives us the ratio of false positives detected by the tool and the number of actual concerns.
   False Positive Ratio =
   $$\frac{\text{No. of false positives}}{\text{No. of coding concerns}}$$

4. False Negative Ratio
   This gives us the ratio of false negatives and the number of modifications made.

False Negative Ratio = $\frac{\text{No. of false negatives}}{\text{No. of modificaations}}$

3. RESULTS AND CONCLUSIONS: -

This section presents the comparative study of the above mentioned three static code analysis tools for security and the ASA tools for Java code.

Comparison of ASA tools for security can be done on basis of many scales, for example by their execution time, on the premise of classification of vulnerabilities a tool can discover, precision, accuracy of the tools. However, in this paper we only compare on basis of time and classification of vulnerabilities.

3.1 EXECUTION TIME

On the basis of execution time we found that RATS was the quicker than Flawfinder and Cppcheck. Cppcheck was slowest in execution of practically every application. Depending on the application, one can choose between RATS or Flawfinder and Cppcheck.

3.2 CLASSIFICATION OF VULNERABILITIES

The categories of vulnerabilities that we considered for our review are: -

- Improper Input Validation
- OS Command Injection
- Buffer Overflow
- Array Index Out of Bounds- Read
- Uncontrolled Format String
- Integer Overflow or Wraparound
- Execution with Unnecessary Privileges
- Race Condition
- Divide by Zero
- Memory Leak
- Dead Code
- Array Index Out of Bounds
- Reliance of Untrusted Inputs in a Security Decision

Based upon the outcomes after these vulnerabilities are included in the same application, the application is subjected to testing with all the three ASA tools to find the detection ratio.

Detection Ratio =

$$\frac{\text{No. of vulnerabilities detected}}{\text{Total vulnerabilities introduced}}$$

The main parameter to be considered here is detection ratio and among all the ASA tools considered, Flawfinder has the highest ratio while for Cppcheck and RATS it turned out to be the same. This does not mean that using Flawfinder is recommended. This ratio is subject to the types of vulnerabilities introduced. If one has to choose among the tools for particular vulnerability, then he may go by the detection in that particular vulnerability or if its aggregate of categories, he may go by the detection ratio.

Moving to the tools for analysing the Java codes, we are dealing with IntelliJ IDEA, Jlint and FindBugs. Since the more the detection of refractorings implied better tool for ASA, we find that IntelliJ IDEA is more successful in Java documentation, unused variables and data members, redundant casting and some other categories compared to the other tools in concern. Minimization of false positives or false negatives is done only with detection. Abundance of these two will not suffice the needs of developers to work on it.

The false positive ratios for IDEA are higher than for FindBugs with higher ratios in the refactoring class. All of the false positive ratios for Jlint are 100%.

False negative ratios refer to the percent of faults or refactorings that are not detected by the static analysis tools. IDEA again can more successfully detect false negatives compared to FindBugs. Since Jlint missed all of the refactorings and faults, all of the false negative ratios for Jlint are 100%.

The fault detection in the three ASA tools were below the minimum which means all these tools would just insignificantly help the engineers to identify the reasons for future reported failures.

ASA tools sometimes report unnecessary errors that are not real faults or necessary refractorings. So developers utilizing ASA devices must look at numerous false positives to choose which ones are genuine. So we can conclude that ASA tools that we reviewed are not effective in detecting the faults with respect to the cost of performing analysis with these tools.

Whether the tools maybe of open source, commercial or developed by researchers, the application of these tools vary in number and types of concerns they detect and handle, programming languages they support. So selection of tools is to be done with respect to the preferred application.

FUTUREWORKS

For the ASA tools used for security, we have considered execution and categories of vulnerabilities as parameters for comparative evaluation. Different parameters, for example, exactness, precision of the instruments can likewise be computed and looked at by considering false positives and false negatives.

In future work, we plan to look at coding concerns revealed by extra ASA tools, and study programming written in other programming languages. Likewise, we plan to distinguish the sorts of faults that ASA tools can recognize more effectively.

REFERENCES

[1] Vinícius Rafael Lobo de Mendonça, Cássio Leonardo Rodrigues, Auri Marcelo Rizzo Vincenzi, and Fabrízzio Alphonsus A. de M. N. Soares, “Static analysis techniques and tools: a systematic mapping study”.
[2] Vijay D’Silva, Daniel Kroening, and Georg Weissenbacher, “A survey of automated techniques for formal software verification”.
[3] Jiang Zheng, Laurie Williams,
Nachiappan Nagappan, Will Snipes, John P. Hudepohl, and Mladen A. Vouk,
“On the value of static analysis for
fault detection in software”.
[4] Fadi Wedyan, Dalal Alrmuny, and James M. Bieman, “The effectiveness of automated static analysis tools for fault detection and refactoring prediction.”
[5] Hanmeet Kaur Brar and Puneet Jai Kaur,
“ Static analysis tools for security: a comparative evaluation.”