Strides Towards Better Application Security

Sathyaraj Balasubramanian

Utah State University

Follow this and additional works at: https://digitalcommons.usu.edu/etd

Part of the Computer Sciences Commons

Recommended Citation
https://digitalcommons.usu.edu/etd/210

This Thesis is brought to you for free and open access by the Graduate Studies at DigitalCommons@USU. It has been accepted for inclusion in All Graduate Theses and Dissertations by an authorized administrator of DigitalCommons@USU. For more information, please contact digitalcommons@usu.edu.
STRIDES TOWARDS BETTER APPLICATION SECURITY

by

Sathyaraj Balasubramanian

A thesis submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

in

Computer Science

Approved:

________________________________________  _______________________________________
Chad Mano                                   Stephen J. Allan
Major Professor                             Committee Member

________________________________________  _______________________________________
Robert F. Erbacher                           Byron R. Burnham
Committee Member                            Dean of Graduate Studies

UTAH STATE UNIVERSITY
Logan, Utah

2008
ABSTRACT

Strides Toward Better Application Security

by

Sathyaraj Balasubramanian, Master of Science
Utah State University, 2008

Major Professor: Dr. Chad Mano
Department: Computer Science

Static analysis tools analyze source code for vulnerabilities. However, these types of tools suffer from various problems that limit their effectiveness. This thesis examines these static analysis tools and suggests techniques for making them more efficient at detecting different types of vulnerabilities.

The thesis further analyzes possible causes for these vulnerabilities by examining the source code written by programmers of various categories. Finally, this thesis discusses solutions and techniques to improve general security awareness as well as the importance of secure coding among the students and software developers.

(131 pages)
ACKNOWLEDGMENTS

I would like to thank my advisor, Dr. Chad Mano, for giving me the opportunity to work under him on this thesis. I would also like to thank him for the great knowledge, support, and guidance he provided me throughout my research. I would also like to thank my other committee members, Dr. Steve Allan and Dr. Robert Erbacher, for their valuable time in evaluating this thesis and providing feedback.

I would like to thank Mr. Dean Mathias and Dr. Brandon Eames for teaching courses in which I gained valuable knowledge. They both have been a great source of inspiration to me.

I am really grateful to my grandmother, my parents, and my brothers for their unbounded love, care, and support. I would also like to thank my friends and family for all their help and support.

Finally, I would like to dedicate this thesis to my beloved little niece, Anusha, who is the only reason holding me from going back to my country.

Sathyaraj Balasubramanian
CONTENTS

ABSTRACT ....................................................................................................................... iii
ACKNOWLEDGMENTS ................................................................................................. iv
LIST OF TABLES ............................................................................................................. vi
LIST OF FIGURES .......................................................................................................... vii

CHAPTER

1 INTRODUCTION ...................................................................................................1

2 RELATED WORK ..................................................................................................7
  2.1 Common Static Analysis Tools ...................................................................8
    2.1.1 ITS4.....................................................................................................8
    2.1.2 RATS ..................................................................................................8
    2.1.3 ARCHER ............................................................................................9
    2.1.4 CQual ..................................................................................................9
    2.1.5 Splint ...................................................................................................9
    2.1.6 Sparse ..................................................................................................9
    2.1.7 Coverity Audit ....................................................................................9
    2.1.8 FindBugs ...........................................................................................10
    2.1.9 Fortify ...............................................................................................10
    2.1.10 Most Prominent Static Analysis Tools ...........................................10
  2.2 Common Vulnerability Types and How They Are Exploited ...................11
    2.2.1 Buffer Overflow ................................................................................11
    2.2.2 Integer Overflow ...............................................................................11
    2.2.3 Cross-Site Scripting ..........................................................................12
    2.2.4 SQL Injection ....................................................................................12
    2.2.5 Format String ....................................................................................13
    2.2.6 Other Common Types of Vulnerabilities ..........................................14

3 SOURCE CODE ANALYSIS ...............................................................................16
  3.1 Analysis Reports ........................................................................................16
    3.1.1 Analysis I (Student Code) ..............................................................16
3.1.1.1 Freshmen Level ................................................................. 18
3.1.1.2 Sophomore Level ............................................................ 20
3.1.1.3 Junior Level ................................................................. 22
3.1.1.4 Senior Level ............................................................... 23

3.1.2 Analysis II (Tech Interview Forum) .................................... 24

3.1.3 Analysis III (Open Source Community) ............................ 26
  3.1.3.1 gets() ................................................................. 26
  3.1.3.2 strcpy() ............................................................ 27
  3.1.3.3 Searching for Other Types of Vulnerabilities ............ 28

3.1.4 Analysis IV (Bugtraq) ...................................................... 28

3.2 Why Does Software End Up Having Vulnerabilities? ............ 30
  3.2.1 Limitations of Open Source Software ............................. 31
  3.2.2 Insufficient Software Specifications ............................... 32
  3.2.3 The Programming Language .......................................... 32
  3.2.4 Awareness of Secure Coding Practices .......................... 33

3.3 Building It Safe ..................................................................... 34
  3.3.1 Offering Secure Coding Courses ................................... 34
  3.3.2 Better Software Design and Development Practices ........ 35
  3.3.3 Policies for Borrowing Code from Open Sources .......... 36
    3.3.3.1 Run Static Analysis Tool .................................... 37
    3.3.3.2 Manual Verification of Code ................................. 38
    3.3.3.3 Check for Unwanted Functionalities ...................... 38
    3.3.3.4 Analyze the Effects of Integration on Security ...... 38

3.4 Using Static Analysis Tools ................................................. 38

4 REPORT ON CURRENT STATIC ANALYSIS TOOLS ............... 40
  4.1 Finding Vulnerabilities in Software .................................. 40
  4.2 Static Analyzers Types .................................................... 41
  4.3 Experiments ..................................................................... 42
    4.3.1 Test for Null Pointer Dereference ............................. 42
      4.3.1.1 Presence in Commercial/OpenSource Programs .... 43
      4.3.1.2 Source Code .................................................... 43
      4.3.1.3 Problems ....................................................... 43
4.3.2 Test for Taint Flag Propagation and Removal ..................................44
  4.3.2.1 Presence in Commercial/Opensource Programs .................45
  4.3.2.2 Source Code ..............................................................45
  4.3.2.3 Problems ..................................................................46

4.3.3 Test for Buffer Overflow, Format String, and String Length Update ............................................47
  4.3.3.1 Presence in Commercial/Opensource Programs .............48
  4.3.3.2 Source Code ..............................................................49
  4.3.3.3 Problems ..................................................................49

4.3.4 Test for Pointer Reference Update and Memory Leak ............50
  4.3.4.1 Presence in Commercial/Opensource Program ...............50
  4.3.4.2 Source Code ..............................................................51
  4.3.4.3 Problems ..................................................................51

4.3.5 Test for Taint Propagation through Stack Referred Variables ....52
  4.3.5.1 Presence in Commercial/Opensource Programs ...............53
  4.3.5.2 Source Code ..............................................................53
  4.3.5.3 Problems ..................................................................54

4.3.6 Test for Integer Overflow .......................................................55
  4.3.6.1 Presence in Commercial/Opensource Programs ...............55
  4.3.6.2 Source Code ..............................................................55
  4.3.6.3 Problems ..................................................................56

4.3.7 Test for Indirect User Data Assignment to Variable ...............57
  4.3.7.1 Presence in Commercial/Opensource Programs ...............57
  4.3.7.2 Source Code ..............................................................57
  4.3.7.3 Problems ..................................................................58

5 FALSE RESULT RESOLUTION .................................................................60
  5.1 Introduction .................................................................................60
  5.2 Design .........................................................................................62
    5.2.1 Package - Lexical Analyzer ..................................................62
      5.2.1.1 Interface Tokenizer .......................................................62
      5.2.1.2 Class Token .................................................................62
5.2.1.3 Class-C Tokenizer ..............................................................63

5.2.2 Package - ProgramElements .................................................63

5.2.2.1 Class “Class” .................................................................63
5.2.2.2 Class Function ...............................................................64
5.2.2.3 Class Variable ...............................................................64

5.2.3 Package Parser ......................................................................65

5.3 Logic and Implementation ..........................................................65

5.3.1 Test for Null Pointer Dereference ...........................................65

5.3.1.1 Logic. .............................................................................65
5.3.1.2 Implementation. .............................................................68
5.3.1.3 Result. .............................................................................68

5.3.2 Test for Taint Flag Propagation and Removal .........................69

5.3.2.1 Logic. .............................................................................69
5.3.2.2 Implementation. .............................................................69
5.3.2.3 Result. .............................................................................70

5.3.3 Test for Buffer Overflow, Format String, and String
Length Update ..............................................................................70

5.3.3.1 Logic. .............................................................................70
5.3.3.2 Implementation. .............................................................71
5.3.3.3 Result. .............................................................................71

5.3.4 Test for Pointer Reference Update and Memory Leak ..........72

5.3.4.1 Logic. .............................................................................73
5.3.4.2 Implementation. .............................................................73
5.3.4.3 Result. .............................................................................73

5.3.5 Test for Taint Propagation through Stack Referred Variables ...74

5.3.5.1 Logic. .............................................................................74
5.3.5.2 Implementation. .............................................................74
5.3.5.3 Result. .............................................................................75

5.3.6 Test for Integer Overflow ......................................................75

5.3.6.1 Logic. .............................................................................75
5.3.6.2 Implementation. .............................................................75
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.3.6.3 Result.</td>
<td>76</td>
</tr>
<tr>
<td>6 SUMMARY AND CONCLUSION</td>
<td>78</td>
</tr>
<tr>
<td>6.1 Summary</td>
<td>78</td>
</tr>
<tr>
<td>6.2 Future Work</td>
<td>81</td>
</tr>
<tr>
<td>6.3 Conclusion</td>
<td>82</td>
</tr>
<tr>
<td>REFERENCES</td>
<td>84</td>
</tr>
<tr>
<td>APPENDICES</td>
<td>88</td>
</tr>
<tr>
<td>Appendix A</td>
<td>89</td>
</tr>
<tr>
<td>Appendix B</td>
<td>92</td>
</tr>
</tbody>
</table>
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>List of Static Analysis Tools</td>
<td>42</td>
</tr>
<tr>
<td>2</td>
<td>Results of Running Test Code over Various Static Analyzers</td>
<td>59</td>
</tr>
<tr>
<td>3</td>
<td>Comparison of Various Static Analysis Tools</td>
<td>77</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>--------</td>
<td>------------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>1</td>
<td>Levels of security.</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>Number of lines in student code.</td>
<td>18</td>
</tr>
<tr>
<td>3</td>
<td>Student code analysis.</td>
<td>19</td>
</tr>
<tr>
<td>4</td>
<td>Vulnerabilities per KLOC.</td>
<td>19</td>
</tr>
<tr>
<td>5</td>
<td>Vulnerability chart for freshmen level courses.</td>
<td>20</td>
</tr>
<tr>
<td>6</td>
<td>Vulnerability chart for sophomore level courses.</td>
<td>21</td>
</tr>
<tr>
<td>7</td>
<td>Vulnerability chart for junior level courses.</td>
<td>22</td>
</tr>
<tr>
<td>8</td>
<td>Vulnerability chart for senior level courses.</td>
<td>25</td>
</tr>
<tr>
<td>9</td>
<td>Tech interview forum analysis.</td>
<td>25</td>
</tr>
<tr>
<td>10</td>
<td>Vulnerability chart for tech interview forum.</td>
<td>27</td>
</tr>
<tr>
<td>11</td>
<td>Vulnerability chart for open source project – Audacity.</td>
<td>29</td>
</tr>
<tr>
<td>12</td>
<td>Vulnerabilities from Bugtraq database (March &amp; April 08).</td>
<td>30</td>
</tr>
<tr>
<td>13</td>
<td>Design of our static analyzer.</td>
<td>63</td>
</tr>
<tr>
<td>14</td>
<td>Structuring of a program.</td>
<td>64</td>
</tr>
<tr>
<td>15</td>
<td>Class design of package lexical analyzer.</td>
<td>65</td>
</tr>
<tr>
<td>16</td>
<td>Class design of package program elements.</td>
<td>65</td>
</tr>
</tbody>
</table>
CHAPTER 1
INTRODUCTION

Security is a continuously evolving field. New techniques and technologies are rapidly being developed in order to ensure continued data security. Ironically, just as new technologies guide developers/programmers and system administrators in maintaining the security of a system, they also help hackers find new techniques to break-in to a system. The security of a system can be viewed at various levels (see Figure 1). The first of these, the physical level, is a situation in which the hacker has hands-on access to the system. Physical access is a fairly easy way of gaining access to any information stored in a system. At the next level, the network level, hackers compromise data moving through a network in which the target system resides, hence gaining access to information going in and out of the system. For example, the man-in-the-middle attack and DNS poisoning are two examples of network level attacks that are well protected by good encryption and message source verification techniques. At the operating system (OS) level, specific vulnerabilities are exploited in order to gain access to the system. Any information about the target system, such as the type of operating system, the kernel version, etc., is valuable to the hacker and can be used for breaking-in to the target system. OS level security is getting better with good data execution prevention techniques, and with companies releasing immediate patches for any new vulnerability. The top level of security of a system is the application level. This includes situations wherein the hacker targets a specific application running on the system and exploits its vulnerabilities to obtain sensitive information or even to take
control of the entire system. At some point, all of these layers are interdependent; vulnerability in one layer leads to vulnerability in a different layer. Thus, a simple bug in any of these levels can lead to the compromise of the entire system.

As the increasing use of firewalls and other kinds of protections make network-level security and OS-level security tighter and harder to exploit, hackers are moving towards exploiting specific application-level vulnerabilities. Unfortunately, while hackers are moving towards application vulnerabilities, not many software developers are focusing on secure coding and other application security techniques. More and more vulnerabilities are being reported to Bugtraq [1]. In fact, the National Institute of Standards and Technology (NIST) reports that 92% of vulnerabilities are in software [2]. With the vulnerabilities list increasing, it is necessary to find the source of such problems, look for patterns of vulnerabilities, and find new tools and techniques that can help in
reducing the number of vulnerabilities in software. Such an attack plan will help the developers produce high quality software.

To that end, static source analyzers are great tools for finding vulnerabilities/bugs in source code. For example, using a static source analyzer can help reduce the number of bugs, thereby increasing the overall quality of the software. However, there are many drawbacks to static analyzers keeping developers from using them. Key drawbacks include usability, false positive rates, and long analysis time. Nevertheless, because of static analyzers’ potential in computer security, addressing these drawbacks is very important.

This project is divided into two major sections. The first section identifies and analyzes the understanding of computer security among undergraduates and developers. This analysis includes identifying common vulnerabilities in programming assignments, thereby giving a view of how students come to understand the security issues in software. The first section of this project investigates some of the common, current problems in analyzers that prevent developers from using them. The second section of this project includes suggestions for techniques to overcome common problems found in static analyzers, thereby increasing the popularity and usage of static analyzers among developers.

The rest of this thesis is divided into the following chapters:

- Chapter 2 – Related Work:

  This chapter presents an overview of related research in static analysis tools and common vulnerability analysis. It discusses various static analysis tools,
their performance analysis, common vulnerabilities in software, and secure coding principles and techniques.

- Chapter 3 – Source Code Analysis:

  Chapter 3 deals with the first section of this project, identifying and analyzing undergraduates’ and developers’ understanding of computer security. Chapter 3 discusses analysis performed on source code written by students and developers. First, an analysis was performed on student source code in order to understand students’ perspective on security. The aim was to identify and understand secure coding techniques and principles among students, as well as determine students’ level of concern regarding writing secure code. The next subsection in this chapter presents the results and analysis of source code from a technical interview forum. This was done because an analysis of the awareness of secure coding practices among the students who are about to join work fleshes out our understanding of security practices, in general. Both analysis on student code, and code from an interview discussion forum were used to study the mistakes made by students while coding, given that students probably make similar mistakes when they join a company.

  This chapter also presents the results of the analysis performed on code from open source software/projects and the results from the Bugtraq database. This project analyzed code from open source software as a means of learning common vulnerabilities found among developers within the industry. Incidentally, the common types of vulnerabilities found in open source software help
demonstrate that open source does not guarantee vulnerability-free software.

Because of the obvious link between the training students receive and the type of developers they become, this project analyzes computer security courses from various universities, and a suggested list of topics to be covered in computer security courses is provided.

- **Chapter 4 – Report on Current Static Analysis Tools:**

  Chapter 4 addresses the second major section of this project, suggestions for techniques to overcome common problems found in static analyzers. Chapter 4 presents analyzes of various source code analysis tools to ascertain their false positive and false negative problems. Common test cases were written to find the weaknesses of these analyzers. The test code was run on all the static analyzers, and the output was analyzed. These test cases and the output produced by the static analysis tools are presented in this chapter.

- **Chapter 5 – False Results Resolution:**

  Chapter 5 also addresses the second major section of this project. This chapter discusses the proposed logic which, when implemented in source code analyzers, significantly reduces false positives and false negatives. False results are one of the major drawbacks in source analysis tools, and reducing them will entice more developers to use them [3]. The logic discussed in this section was implemented on a custom parser written for this research work. The results of running the same test cases over this custom-built parser are also discussed in this chapter. The results clearly show that implementing these logics would definitely
improve the quality of the static analysis tools.

- Chapter 6 – Summary and Conclusions:

  Finally, Chapter 6 summarizes the work done in this project, suggests directions for future work in this area, and provides concluding remarks.
CHAPTER 2

RELATED WORK

There has been a lot of research done in source code analysis and static analyzers. Static analyzers started getting attention in early 2000 when the first static analyzer, the ITS4, was introduced [1]. Since then, various other static analyzers have been introduced, each one improving on the previous ones. Improvements include better features, algorithms, and methodologies to scan source files and perform vulnerability analysis on them. Consequently, there has been a significant increase in how these tools perform analysis and identify vulnerabilities and bugs.

In [4], the authors provide a list of static analysis tools, along with a small description of these tools and their basic purpose. Also, a graph showing the running time of the analysis of various static analyzers is provided, but information such as the code used with these static analyzers, size of that code (i.e., number of lines) is not given. Also, the authors do not discuss some of the false positives and false negatives associated with performing the source analysis on the test code, a necessary component for evaluating a static analyzer’s effectiveness.

McGraw provides a list of source code analyzers and briefly discusses the rule sets and programming languages that these tools support [1]. He terms old static analyzers as first generation and latter ones as second generation static analyzers. A very similar description of various source code analyzers can be found in [5]. The paper also discusses the need for static analysis tools, but still, it does not show the false positive/false negatives of these tools. A variety of journal articles [6, 7, 8, 9, 10, 11] have been
written on static code analyzers and finding vulnerabilities in source code, but none of them emphasize techniques for writing a source analyzer, reducing the number of false positives and false negatives, and improving their runtime.

2.1 Common Static Analysis Tools

The following are the most common static code analysis tools that have either been released or are currently under development/research. Some of these static analysis tools are open source, while others are commercial tools.

2.1.1 ITS4

ITS4 is one of the oldest source analyzers [1] around. It merely searches the source code for unsafe function calls, and flags them as security vulnerabilities. ITS4 performs some basic lexical analysis [9] on the source code i.e., it breaks down the source code into a set of tokens, and matches these tokens with predefined rule sets. It suffers from lot of false positives.

2.1.2 RATS

RATS is a rough auditing tool for security that is very similar to ITS4, the primary difference being that ITS4 was targeted towards C and C++, whereas in addition to C/C++, RATS supports Perl, PHP, Python, and open SSL. It is an open source application and one of the earliest built static analyzers. RATS identifies the places that need attention and aids in finding security-related errors in the source code. Like ITS4, RATS produces a lot of false alarms [1].
2.1.3 ARCHER

ARCHER [20] is an array bounds checker program written in C [1, 12]. It helps in locating index access problems in C programs, thus eliminating the possibility of segmentation faults [12].

2.1.4 CQual

CQual detects format string vulnerabilities in programs written in C [4]. In order for CQual to carry out this function, the program needs to be annotated [1].

2.1.5 Splint

Splint is an extension of LINT. It was created in 2002 for detecting vulnerabilities and coding mistakes in C/C++ applications [4]. Splint needs annotation to find buffer overflows and race conditions. Without annotation, it can perform some basic analysis on the source code [4]. However, for better results and analysis, it needs some kind of annotation in the source code.

2.1.6 Sparse

The Sparse project was started by Linux Torvalds in 2003 [13]. It was developed for kernel developers for finding bugs in kernel code and was initially designed to detect errors in mixing pointers between user space and pointers in kernel space.

2.1.7 Coverity Audit

A product of Coverity, Inc., Coverity Audit [14] is a commercial application that can detect various types of problems in C/C++ code, although Coverity Audit is also available for Java. Coverity Audit detects and lists errors in the source code. For
information on the location of the error, an additional product named Coverity Prevent needs to be installed.

2.1.8 FindBugs

FindBugs is a static analysis tool for programs written in Java. FindBugs was initially started as a research tool at the University of Maryland [1].

2.1.9 Fortify

Fortify static analyzer [15] is a commercial product that supports various programming languages, including .Net, Cold Fusion, and PHP. Because Fortify can identify almost any type of software vulnerability, it has been an important part of the current project. The fact that it has some false positives and false negatives far outweighs the hassle involved in manually auditing the source code.

2.1.10 Most Prominent Static Analysis Tools

Coverity and Fortify are the two most prominent names in static analysis tools. Both of these companies have released a commercial product for static source analysis. These tools are more advanced than other static analyzers, and they support various languages. Both of these vendors are actively involved in source code analysis, improving the overall quality, and reducing false positives. Further, both companies have published numerous white papers that discuss next generation attacks, the future of source code analysis, and new techniques to improve the source analysis [6, 16]
2.2 Common Vulnerability Types and How They Are Exploited

This section discusses the most common types of vulnerabilities found in software. There are several types of vulnerabilities, and an explanation of all types of vulnerabilities is simply beyond the scope of this document. This project focuses only on a subset of the most commonly found vulnerabilities in software applications.

2.2.1 Buffer Overflow

One of the most common software vulnerabilities found, buffer overflow has been in existence for a long time. Even though the buffer overflow problem has been around for years and its cause and consequences have been published in various books and papers, it still remains at the top of the list of vulnerabilities. Buffer overflow can result in memory corruption which, in turn, can lead to an arbitrary execution of code [11]. Buffer overflow occurs due to improper handling of memory or improper bounds checking [11]. Both stack and heap memory are susceptible to buffer overflow. When the program does not check for bounds or does not limit the size of the input based on allocated buffer space, the program can end up overwriting other parts of memory [17, 18], not allocated to or intended for storing input data. Buffer overflow can be exploited by the attacker by feeding the program with malicious input data, or by inputting data that is bigger than what the program could handle.

2.2.2 Integer Overflow

Similar to the buffer overflow problem, integer overflow is constrained to a particular data type, integers. In a normal X86 architecture, the size of an integer is 32
bits; there are also 64-bit processors in X86. Integer overflow occurs when a large integer value i.e., value greater than 2147483647 is stored in an integer variable [19]. Such an overflow causes memory corruption, thus leading to misinterpretation of data [19, 20]. Integer overflows are exploited by inputting large integer values where a normal integer value is expected.

2.2.3 Cross-Site Scripting

Cross-site scripting has recently started gaining more attention, probably because of the increase in the number of web-based applications and the rapidly growing online population. The cross-site scripting vulnerability is most commonly found in web applications and web pages [4, 21]. Cross-site scripting allows hackers to embed their own scripts as a part of an organization’s website; hence, if a user visits the said webpage, the malicious script is executed in the client's web browser as if it were a part of the original webpage [21]. During most cases, users are unaware of this happening in their computer [4].

Cross-site scripting occurs as a result of bad input processing or bad input validation [4]. Cross-site scripting can be exploited by writing a malicious script and saving it in a Java script file (JS file), by writing smart script tags that take in this JS file, or by feeding the unsecure website with malicious input [4, 22]. A simple example is an HTML page that gets a value from a user's form, passes the value to a script, and the script echoes back the form’s data to the user's screen without validating the value [22].

2.2.4 SQL Injection

SQL injection is another commonly found vulnerability in web-based
applications. SQL injections occur when the application does not validate input data but, instead, directly use the input data as a part of an SQL statement [4, 22]. Thus, when the SQL statement is executed without validating user input, a disaster may well result. A recent analysis showed that around half-a-million Microsoft-powered websites are vulnerable to SQL injection attack [23]. SQL injection is not restricted to web applications. Even the normal desktop applications that interact with the SQL servers are susceptible to this attack. A simple example is as follows:

```cpp
Output<< "Enter User Name ";
Input >>uname;
query1 = "select * from USERDB where user_name = " + uname;
odbc_object.execute(query1);
```

In this example the input from user is stored in the variable “uname”, the value in this variable is concatenated to the string variable “query1”, which is passed as a parameter to the execute function. When the user enters “user123; Delete * from USERDB;” as username, query1 will have the following value “select * from USERDB where user_name = user123; Delete * from USERDB;” and when this query is executed in the database server, all the data in USERDB will be lost.

### 2.2.5 Format String

Format string vulnerability occurs when the program does not handle the input string properly. Printf and similar printing routines have format string options that tell the function call how to interpret the data or the variable that needs to be printed. These function calls are susceptible to the format string vulnerability [17]. If a string that needs
to be printed is directly given into one such call without formatting options, i.e.,
printf(userinput); instead of printf("%s",userinput);, a format string vulnerability that
exposes the data in the program's stack can result [17].

2.2.6 Other Common Types of Vulnerabilities

Other types of vulnerabilities include arbitrary file overwrite, DoS attack,
privilege escalation, HTTP response splitting, memory leaks, double free, etc. Not
checking the return value of a function call is a bad practice, especially when function
calls, such as malloc or similar memory allocation functions, are used. The program
should validate the return address before doing anything with it [4]. A recent paper
published by IBM [24] gives an example in which the Flash player, a call to the calloc
function is not checked against null, and the program just continues to write some values
to the return address. As a result, an attacker could input some value to the program,
thereby controlling the address to which the program writes the data [22, 24].

Arbitrary file overwrite is a vulnerability in which the program starts writing to an
arbitrary file, and the file to which it writes is controlled by the attacker. Arbitrary file
overwrite can have disastrous consequences when a program is running in the root mode
[17, 22]. The attacker can write anything to any file and, hence, can execute arbitrary
code in the vulnerable system. DoS(Denial of Service) is a pretty common attack. In
DoS, the attacker creates a situation in which a program does not respond to any user-
generated events/request. This is generally achieved by inputting malicious data that
make the program crash or by putting the program into a busy state so that it does not
respond to a user's request. HTTP response splitting occurs when a web page's http
response header contains an invalidated user input. The attacker can split the response into multiple HTTP responses by adding some extra data to the response. Memory leak occurs when the program calls a memory allocation function call and does not free that allocated memory [17, 22]. This might result in a system running out of memory, thereby causing the system to not function properly. Double free is the opposite of memory leak. In this case, free is called on an address that was previously freed. This might result in unpredictable consequences. The authors of [22, 25] give a complete list of vulnerabilities that could be found in any software program.

Although configuration errors and weak access control could lead to security vulnerabilities, problems like buffer overflows, format string errors, SQL Injections, and memory leaks can occur due to bad programming practices or bugs in code. Such problems can be identified with the aid of static analysis tools. One real world example is, when a static analysis tool was used on SAMBA, an open source file sharing software, it identified about 915 vulnerabilities, which is definitely a huge number. More case studies and surveys prove that usage of static analysis tools helps in identifying security vulnerabilities in software [26, 27, 28, 29].
CHAPTER 3

SOURCE CODE ANALYSIS

This chapter presents an overview of the common types of vulnerabilities found in various software. Also, as a part of this research, analyses on various source codes have been performed, the results of which are presented in this chapter.

3.1 Analysis Reports

To have a better understanding of vulnerabilities, it was necessary to perform an analysis on various source codes that were available to us. These source codes include student source code, source code from some of the open source search engines, student discussion forums, and open source projects. Some parts were analyzed manually by going over the source code, but most of the source codes were analyzed with the help of the Fortify source code analyzer.

3.1.1 Analysis I (Student Code)

Currently, little research exists on analyzing student programming assignments with respect to the security factor. Indeed, very few research papers/journal articles are dedicated towards analyzing student code, ascertaining students' perspective on security issues, and students' knowledge of secure coding and source analysis tools. The analysis for this project started with student programming assignments. Students are taught how to program, but all too often, students do not learn how not to program [14]. So, it was necessary to analyze how students code their assignments, what type of mistakes they generally make while programming their assignments, and try to answer questions of
whether students consider security at all while coding. Because students ultimately become programmers and developers in industry, the basics of secure coding must be taught to students in order for students to carry those patterns with them when they code for a company.

As the first step of analysis, student code was obtained from the department of computer science at Utah State University. Since this is not a funded project, it did not require any special approval from IRB. In order to protect the student’s privacy, a script to delete the comments from the top section of the source code, where students generally include their name, was written and was run over the students’ assignments. Later, courses from each level (freshmen, sophomore, junior, and senior) were selected for source analysis. Fortify software analyzer was used for scanning these source files, and parts of the source code were analyzed manually in order to verify the results produced by Fortify.

Figure 2 shows the number of lines of code analyzed in each course level. The freshmen level course had assignments only in C/C++ language, and hence it had the highest number of C/C++ files. The sophomore level had the lowest number of lines of code. It had approximately 190,000 lines of code written in C/C++ and nothing in Java or C#. The junior level had a lot of code written in C/C++ and a few thousand lines in Java and C#. The senior level had an even higher number of C/C++ files and around 50,000 lines of code written in Java.

Figure 3 shows the analysis report on student source code, including the total number of critical vulnerabilities at every course level. Figure 4 shows the number of
critical vulnerabilities present in students’ assignments per thousand lines of code (KLOC), categorized based on course level. The patterns in Figures 3 and 4 are very similar. The overall mean number of errors per assignment is 1.4 and the standard deviation is 0.768.

3.1.1.1 Freshmen Level. Freshmen level courses had about 900 vulnerabilities out of 1579 assignments analyzed. Obviously, these assignments are from a basic course, such as an introduction to C/C++ programming, wherein students are taught how to code in these languages and are given assignments including using if statements, writing switch cases, writing loops, basics input and output functions, file, and IO processing. Figure 5 shows the vulnerabilities found in freshmen level courses categorized based on the type of the vulnerability. Buffer overflow and string termination errors constitute the

![Figure 2. Number of lines in student code.](image-url)
Figure 3. Student code analysis.

Figure 4. Vulnerabilities per KLOC.
highest number of vulnerabilities.

3.1.1.2 Sophomore Level. Sophomore level courses had the lowest number of vulnerabilities compared to the other levels. They also had the lowest number of programming assignments, about 610 assignments. Typical courses at this level include data structures and algorithms, introduction to software engineering, computer organization/architecture, etc. Typical coding assignments include basic data structures implementations. The number of vulnerabilities here is less because there are fewer lines of code (see Figure 3). With both freshmen and sophomore level courses, buffer overflow and usage of dangerous functions, such as gets, unrestricted cin, etc., top the list with significant numbers. The category-based chart in Figure 6 shows that around 30% of the vulnerabilities found in this level are due to buffer overflow issues.

Figure 5. Vulnerability chart for freshmen level courses.
Figure 6. Vulnerability chart for sophomore level courses.

The following code snippet for unrestricted input illustrates a common vulnerability found at the sophomore level:

1.  
   ```
   char screename[30]; //declared inside a struct
   cout <<"Enter new data" << endl;
   cout << "Screen name :";
   cin >> screename;
   ```

2.  
   ```
   char fileName[100];
   cout << "Please enter input file name: ";
   cin >> fileName;
   ```
3.1.1.3 Junior Level. Next, junior level courses had the highest number of vulnerabilities, approximately 3500, which is directly proportional to the number of assignments, 1640, the highest among all the other levels. Junior level courses include operating systems, Java/internet, C# .NET. Although buffer overflows and memory leaks comprise a big share of these vulnerabilities, another interesting type of vulnerability that was pretty dominant was command injection, which is about 22% of the total (see Figure 7). The rise in command injection vulnerability is probably because students are not aware of the fact that passing invalidated input to function calls, such as system(), execv(), is a bad programming practice. As Hoglund and McGraw strongly emphasize [17], do not trust user input.

Figure 7. Vulnerability chart for junior level courses.
The following code snippet for command injection illustrates a common vulnerability found at the junior level:

1.

```c
int main(int argc, char **argv) // input comes from user through argv
    printf(ng,"mount | grep -e "/dev/fd" | %s -readfloppy ", argv[0]); // input is passed to string ng
    system(ng); // ng is given to system() function call
```

2.

```c
execvp(argv[0], argv); // input from user is directly given to execvp
```

3.1.1.4 Senior Level. Next up, in the senior level, the number of vulnerabilities dropped to around 1900. The total number of assignments analyzed in this level was 973. Senior level courses typically include computer security, network programming, object-oriented programming, advanced algorithms, databases, etc. Enrollment in computer security courses and use of programming languages like C# and Java for object-oriented development probably account for the drop in the number of vulnerabilities from the junior level. Nonetheless, as shown in Figure 8, the buffer overflow problem remains dominant in senior level courses as well as lower level courses.

3.1.2 Analysis II (Tech Interview Forum)

The next analysis performed is very much related to student source code. Just about to finish school, what is the next thing a student probably does: look for a job. Going with that assumption, a popular interview-questions discussion forum [30] was
analyzed. Because students post their interview experiences there, and some even post solutions in terms of source code or in algorithm form, students make use of these websites to prepare for their interviews. These source codes were analyzed for security vulnerabilities.

As the thesis is focused on improving the overall security of an application and the security awareness among the students, it was necessary to perform this analysis which could give a better idea about students’ knowledge on secure coding. Although this analysis could reveal the weaknesses in educational system, it may not be possible to rectify them by merely teaching the student about various static analysis tools. The usage of such tools should also be made mandatory in every assignment submission. This could help the students in knowing the importance of application security and also make them better prepared for industry.

There is absolutely no guarantee that only students discuss and post solutions, and there is absolutely no information on where and when people did their undergraduate work; nonetheless, it is still interesting and worthwhile to analyze these source codes and gather some information about their vulnerabilities. Figure 9 shows the result of the analysis of source codes posted on a technical interview discussion forum. Most of these were mostly written in C, C++, or Java. As can be seen in Figure 9, about 17% of these source codes had some kind of security vulnerability, and about 50% of those source codes having security vulnerabilities were posted in the Microsoft section. It should be noted that there is absolutely no relevance between this result and security vulnerability
Figure 8. Vulnerability chart for senior level courses.

Figure 9. Tech interview forum analysis.
or the quality of Microsoft developed products in general. The results were categorized based on the company just to make the analysis and results more interesting.

Around 450 different source codes from the tech interview forum were analyzed, and various types of vulnerabilities were found in these source codes. Figure 10 shows the breakdown of the types of vulnerabilities found. Buffer overflow, missing null check, and integer overflow were the three major types of vulnerabilities observed, with these three having almost an equal share in the total number of vulnerabilities.

3.1.3 Analysis III (Open Source Community)

The purpose of this analysis was to find common vulnerabilities in open source projects. Search engines that search through various open source projects were used to perform the code search. These include Google Code Search, Krugle, and Codase. My specific goal was to check how many open source codes had some of the most obvious problems and followed very bad practices. Source code engines are not guaranteed to have indexed the latest version of the open source project, and hence, the results and analysis presented here are based on the version of source code indexed by these search engines. The following functions are considered to be insecure and were searched in open source code using all three search engines mentioned above.

3.1.3.1 gets(). Gets is a function call that gets the input data from stdin and saves it in a string. No matter how big the string buffer, any gets call is vulnerable. In general, when a search is made for a particular function call, the search engine lists more than 1000 results. However for this project, only the first 50 results were analyzed for vulnerabilities. This applies to the following function calls, too. Out of the first 50 results,
most were an implementation of the gets call in various versions of glibc or other relevant projects. Still, any project that had this function call with a destination buffer was vulnerable to buffer overflow attacks.

3.1.3.2 strcpy(). The strcpy function copies the data from the source to the destination string. Strcpy is considered an insecure call because it does not check if the destination buffer is big enough to hold the data [17]. Instead, it starts copying the data from the source to the destination irrespective of how big or small the destination is. Hence, it is susceptible to buffer overflow, although it should be noted that strcpy can still be used safely if the programmer is pretty sure that the destination buffer is bigger than the source buffer. Again, a lot of results (especially from Google) were the actual implementation of the strcpy function from various libraries. But analyzing the other
source code reveals that every one of the seven strcpy calls was vulnerable.

3.1.3.3 Searching for Other Types of Vulnerabilities. Searching for SQL injection, off-by-one, or cross-site scripting errors is difficult in open source engines. Searching for regular expression like “select \* from “, or similar expressions yielded some results; however, most of these were irrelevant to SQL queries. Out of those that were useful to this discussion, most had a module that gets a parameter and passes that parameter into an SQL query. It may be that the parameter was already validated in some other module and was consequently a safe module inside that project. Nevertheless, a problem arises when that same module is used as a part of another project. In such cases, it probably ends up being vulnerable to SQL injection attacks.

Also, it was necessary to pick one of the most popular open source projects and see if it had any vulnerability. The Source Forge website lists the top ten open source project based on number of downloads and popularity. The popular open source project, Audacity, an open source program for editing audio files, was used for source code analysis. This program was chosen solely because it is in the top ten open source programs used. The Fortify source analyzer was used to scan the Audacity source code. Figure 11 shows the various types of vulnerabilities found in Audacity. Nearly 65% of the vulnerabilities were due to memory leaks. Buffer overflow and missing checks against null were also frequent.

3.1.4 Analysis IV (Bugtraq)

Finally, I looked at the Bugtraq database for vulnerabilities being reported, and for whether those vulnerabilities were associated with open source or closed source software.
The results, shown in Figure 12, were collected over a period of two months. A longer collection period was beyond the scope of this project. However, it would be interesting to see if a longer collection period resulted in different results. Figure 12 shows the vulnerabilities reported to the Bugtraq database categorized by the type of vulnerability. It is evident from this figure that open source and closed source applications contribute equally to the vulnerabilities list. Be it buffer overflow or SQL injection, there is not much difference between number of vulnerabilities reported in open source applications and the number of vulnerabilities reported in closed source applications.
3.2 Why Does Software End Up Having Vulnerabilities?

Vulnerabilities in software have been reported via various means for many years. Nonetheless, there are still a lot of applications with lots of security bugs in them. The Bugtraq forum is updated daily with numerous vulnerabilities in almost all types of software applications, from normal paint applications to the complex software firewalls.

There are a few vulnerabilities that have remained fairly dominant for the past several years, and buffer overflow is one of them. All software developers should be aware of the buffer overflow vulnerability, yet many buffer overflow vulnerabilities are continually being reported to Bugtraq. As software becomes more and more sophisticated and complex, it carries more vulnerability. Additionally, people find new ways of exploiting these vulnerabilities. Exploiting vulnerability can result in various things, of which the arbitrary code execution and privilege escalation are the most evil. In these, the
attacker can take the complete control of the system.

3.2.1 Limitations of Open Source Software

Some people argue that having open source software by its very nature is very secure [7, 31]. But according to the Bugtraq database, there are an equal number of vulnerabilities in open source and closed source applications. In fact, Ranum, the innovator of firewall, states: “I don't think making software open source contributes to making it better at all. What makes good software is single-minded focus” [9:2]. Also, Spafford of Purdue asserts: “What really determines whether it is trustable is quality and care, was it designed well? Was it built using proper tools? Did the people who built it use discipline and not add a lot of features? A lot of the open source software out there is built by people who don't have experience, the tools, the time, or the resources to do it as carefully as you would want in a highly trusted environment” [9:2].

Also, many people have the misconception that open source software is being continuously watched by several people; hence, the software is well written. While it might be true that many people take a look at these open source codes, it is not certain that all of these people contribute to the project [1]. For example, Lawton uses Ranum’s firewall to illustrate the same point [9]. When Ranum released the firewall software, there were about 2000 sites using it to some extent, but only around 10 people had written back to Ranum and even less had submitted patches. Ranum was unimpressed with the open source community. Another example [1] is the ITS4 software, which is actually a static checker. When ITS4 was released as an open source software, there were around 15,000 downloads during the first year, yet during the same time, not even a single rule was
added to its knowledge-base by the open source community [1]. So, open source software
does not always guarantee that the software is secure enough.

3.2.2 Insufficient Software Specifications

There are various reasons for software to be vulnerable. Writing secure software is
probably a rarely mentioned requirement of a product. Mistakes or bugs could occur
during any phase of development. McGraw indicates that as high as 50% of
vulnerabilities are due to design issues [14]. Given that bugs due to design issues are
harder and more costly to rectify than other types of vulnerabilities, this is an alarming
number. Hence, the initial design should be strong enough to withstand attacks and
should be free from bugs.

Most software designers give a higher priority to functional requirements than to
the unspecified security requirements. This might be another reason why software
applications suffer from vulnerabilities.

3.2.3 The Programming Language

The programming language used for developing the software might actually play
a significant role in the security of the application. Programming languages like Java and
C# enforce more security in the code compared to C and C++ [4]. Indeed, it is probably
easier to exploit software written in C or C++ compared to software written in any other
programming language. This is probably because both C and C++ libraries are written
with more focus on performance rather than security. One obvious example is the strcpy
function which does not check the bounds before copying the source into the destination
buffer. Another example is the realloc function call which dynamically reallocates a
memory segment which was previously allocated using the malloc function call. The realloc function takes the memory address as one of its parameters and the new size as another parameter. So, if the new size is less than the old size, there is a possibility of buffer overflow occurring in the heap, because the realloc function does not check if the new size is less than the old size but, instead, immediately starts copying the data from old segment in to the new segment.

Also, another primary reason why Java and .Net are more secure is because programs written in these languages run on top of a virtual machine. So, even if the program itself is exploited, the hacker gains control only over the virtual machine and not the actual system in which the virtual machine is installed.

3.2.4 Awareness of Secure Coding Practices

Another important contribution to application-based security problems is university-level secure coding courses or the lack thereof. Probably not many universities offer courses on secure coding. For example, A Google search for "secure coding courses" does not return any information about universities that offers secure coding courses. McGraw sees this as one of the major problems in computer security [1]. Since secure programming courses are not typically a part of standard curricula, one cannot expect students to be aware of all types of vulnerabilities and good programming practices. McGraw lists some of the universities that offer courses related to software security [1]. There are only a handful of such universities, and most of these are highly ranked universities. This lack of computer security courses becomes even more troubling when one considers the industry itself.
Big organizations typically have lots of ongoing projects and enough resources to hire many good programmers. Because of their size, large organizations can afford to have a separate software security team that checks source code against all types of vulnerabilities. But consider small organizations, especially start-ups. Such organizations obviously are not able to have a separate team for software security. Hence, the programmers themselves might spend some time on doing the source analysis. Also, small organizations might end up borrowing open source code in order to get their work done.

As analysis II shows, open source search engines might end up throwing in some vulnerable code that will result in a vulnerable application. Also, most big organizations have secure code training programs for their employees. So, before the employee starts coding s/he has to undergo training, learn good programming practices, and get some knowledge about vulnerabilities and how to avoid them. Thus, they often end up doing a better job at creating secure software. Small organizations might not have these kinds of training programs for their employees.

3.3 Building It Safe

3.3.1 Offering Secure Coding Courses

Analysis-I provides a clear picture of security awareness among students. It is evident that not many students are aware of secure coding practices/standards. As mentioned earlier, not many universities offer courses on software security, secure coding, etc. Among those universities that do offer software security and secure coding courses, the following are some of the common topics found among the various security
courses offered:

- Secure software engineering
- Cryptographic protocols
- Access control techniques
- Memory protection techniques
- Buffer overflow attacks and defenses
- DoS attacks and defenses
- Format string vulnerabilities
- Web security, XSS attacks and defenses
- Case study on real world software vulnerabilities
- Database security
- Programming language-based security issues
- Source code analysis tools

Offering such types of courses or at least covering these topics in security courses at every university would definitely improve the awareness of secure coding and software security vulnerabilities among the students. Having a case study on real world software vulnerability would guide students in understanding the consequences of software vulnerabilities in an application and would have an impact on students’ awareness of the importance of secure coding.

3.3.2 Better Software Design and Development Practices

Every organization has its own practices and follows the software life cycle model pertinent to its projects. There is no one-size-fit-all for software engineering practices or
software security. But irrespective of the various methodologies followed by various organizations, building secure software requires good practices right from phase one [9], i.e., security has to be built into the software right from the design phase.

McGraw also [1] suggests some software practices, known as touch points, for developing secure software. These software touch points work for iterative approaches, but a lot of companies now follow agile development practices. This is particularly true for many tech start ups. They start with a scratch idea, and they build software without really having a proper software requirement specification (SRS) document. Once the developers have a basic idea, they start coding. In such cases, it is easy for the developers to miss security features, as their main focus is developing a basic product initially and then improving the product. Adding security on top of an existing project can be tricky. An interesting example of a touch point is designing abuse cases [1]. This is a very important practice. Designing abuse cases makes the programmers think from the attacker’s perspective and helps them write some extra code to prevent security breaches from happening.

3.3.3 Policies for Borrowing Code from Open Source

There are indications that many large organizations, including Google, Microsoft, and Amazon, borrow code from open source. For example google’s latest product, Chrome, a browser which is completely built with open source libraries, Microsoft uses open source code like jQuery, zlib, and MPI, and Amazon.com is entirely built over open source libraries [32, 33, 34, 35]. Analysis III and Analysis IV show that even open source software suffers from various types of software vulnerabilities; making software open
source does not guarantee a completely secure or vulnerability free software. Hence, organizations must follow a standard policy for adapting open source code and merging it into their proprietary product. Most companies/organizations might already have their own set of policies or practices for using open source code, but each one differs from the other. The following is a set of steps that an organization could take in order to use open source software as a part of its product. These steps are relevant only to security aspects; there might be other aspects of using open source code in an organization, such as legal issues, coding standards etc.. Such matters are beyond the scope of this project and are not covered in the following steps.

3.3.3.1 Run Static Analysis Tool. Static analysis tools simply make the job of source code analysis easier. It is a good idea to run the static analysis tool over the open source code to find the vulnerabilities in it. This will give a clear idea whether it is safe to use the open source code in the software product that the organization is developing. However, the company should be aware of the false positives and false negatives of the static analysis tools that are being used for the analysis.

3.3.3.2 Manual Verification of Code. Results from static analysis tools are obviously going to include false positives and false negatives. These can be eliminated by manual analysis of the source code. As mentioned earlier, static analysis tools are going to make this job a little easier because they spot the places that need a closer look. Manual auditing of the source code is very necessary for safe usage of the open source code.

3.3.3.3 Check for Unwanted Functionalities. Unwanted functionalities can
introduce vulnerability into the program. Open source code might have a lot of features that are not related to the product the company is developing, and it is good idea to remove these unwanted functionalities from the source code. Also, open source code might have malicious code in it. Because the source code is modified by hundreds of people online, the source code is not always to be trusted. So, checking for unwanted functionalities and malicious code and removing them helps eliminate the possibilities of vulnerabilities in the company's software product.

3.3.3.4 Analyze The Effects of Integration on Security. Adding a new module to the current project is always going to introduce uncertainty into the project. The new module might affect the current code in any way. This is not only with respect to the functionality of the program but also with respect to the security aspect of the program. When an open source code is added to the current project, it is necessary to analyze the effect of adding the new piece of code. The open source code might be completely void of vulnerabilities and be totally secure as a separate module, but merging the open source module into the current project might introduce new vulnerabilities into the program. Hence, it is necessary to analyze the effects of integration of new modules on security.

3.4 Using Static Analysis Tools

Use of a static code analyzer is a must in all projects. Though static analyzers will not pinpoint the exact spots of vulnerabilities and suggest methods to rectify them [1, 4, 10], they definitely give an overview of the most probable vulnerable components in code and bring them into the view. Using static analyzers definitely improves the overall quality of the software product. As mentioned earlier making the static analyzers as
standard part of student’s assignments will help the students understand various coding mistakes and vulnerabilities caused by it, and also ways to avoid these vulnerabilities.
4.1 Finding Vulnerabilities in Software

Finding vulnerabilities in software is not an easy task, although there are some tools that can make it a little easier [6]. Although often very tedious and time consuming, going through the source code manually is good option, when the source code of the software is available. There are some static code analyzers available to perform this job. A static code analyzer helps in identifying security related bugs by looking at the source code of the application, but it does not run the application, nor does it need its binary [6, 9] (although there are static analyzers that analyze binaries, too [4, 8]). This helps in eliminating bugs before the application is built and shipped. There are various static code analyzers available, some specialized towards finding one type of vulnerability, some specialized towards a particular programming language, and some very generic ones that can be used against a wide range of programming languages. But most static code analyzers suffer from a high false positive rate [6]. Consequently, some type of human intervention is necessary to verify the results of these static checkers. Another concern is that some static code checkers expect programmers to add some annotations to the code in order to find the vulnerabilities.

When source code for an application is not available, finding vulnerabilities becomes a harder task. Running the program and entering random inputs and invalid inputs, or doing random operations is one type of testing for finding vulnerabilities. But this can end up being unfruitful, after a lot of time and effort have been invested [9]. It is
possible to not find even single vulnerability after spending many hours on testing.

Another way of finding vulnerabilities with binary files is reverse engineering the binary, analyzing the assembly code in order to find how things have been implemented in the software [24]. Once an overview of the software is obtained, it is a little easier to guess some of the weaker points in an application. The authors of [36] provide a very good example of how reverse engineering can be used to reframe parts of source code and find vulnerabilities in the software. Disassemblers and decompilers are some of the tools that are used for reverse engineering. For carrying out reverse engineering, one needs to have a thorough idea of the underlying OS and the architecture, since assembly instruction can vary widely depending upon the architecture [24]. There are certain tools available [4] that can check the disassembled code for some common vulnerability patterns and highlight them. Reverse engineering is a huge area, and it is outside of scope of this paper.

### 4.2 Static Analyzers Types

Most early versions of static analyzers were written specifically for one language, and some were written for specific type of vulnerability. But more recent static analyzers are better written and carry support for source code written in various languages. Thus, they can detect a huge range or types of vulnerabilities in source code.

Table 1 provides a list of static analysis tools that are used as a part of this research. The intention of the analysis was not to evaluate a product or any of the source code analyzers. Each analyzer is good in its own way and performs well for the specific reason for which it was written. The main intention was to analyze their false positives,
false negatives, and their time of operation. The tools used as a part of this analysis were chosen because they support source code written in C and C++.

4.3 Experiments

This section describes the tests that were conducted on the static analyzers and the result produced. These test cases were written based on the observations of running some of the static analyzers on student code. Manual verification of the static analysis results against student assignments revealed some false positives and false negatives with the static analyzers, and some of these test cases are based on those results. Table 2 at the end of this chapter provides a summary of the result.

4.3.1 Test for Null Pointer Dereference

Null pointer dereference is a potential security issue. It can lead to failure of program, DoS or could even be exploited to write into arbitrary memory location [36]. It is more commonly found in C/C++ programs compared to other programming languages, and it is rated as medium in open web application security project (OWASP).
4.3.1.1 Presence in Commercial/Open Source Programs. The null pointer dereferencing problem is pretty common among various software products. It occurs either because of missing a null check after a memory allocation function call, such as malloc or realloc, or it occurs because of invalid/wrong pointer assignments. A search for null pointer dereference in Bugtraq lists problems in various software, including some of the more popular ones like Apache, Internet Explorer, Linux Kernel, etc.

4.3.1.2 Source Code. The source code for the first test case is as follows:

```c
1. int main()
2. {
3.   int *p, *c;
4.   p = (int*) malloc(4);
5.   p = c;
6.   *p = 100;
7.   return 0;
8. }
```

4.3.1.3 Problem. As is clearly seen in the above code, pointer p is allocated some space with malloc call, and later p is assigned c's value which does not point to anything. In this test case, the null pointer is being tried to dereference. There is also a memory leak in this case wherein p is not freed before it is assigned c's value. The following are results from various static analyzers that were run on this code.

Sparse: No defects found.

Splint: Variable c used before definition; and Memory leak detected;

Flawfinder: No defects found.
RATS: No defects found.

Fortify: Missing checking against null and memory leak;

Neither Sparse, nor Flawfinder, nor RATS found any defects in the code. Splint, however, found both the coding mistakes. It reported using c's value before initialization, though it did not specify that the problem was a null pointer dereference. Also, Splint detected the memory leak in the code.

Fortify identified the memory leak problem correctly, but it was not able to track down the null pointer dereference problem for this case. The other error it reported was missing checking against null, which is probably, in reality, the null pointer dereference problem. Fortify gave this error for the statement p=c, with the comment that p is used before it is checked against null. This is probably a false negative because in the statement p=c, p is assigned c's value, and p's value (which was originally returned from malloc) is not used.

4.3.2 Test for Taint Flag Propagation and Removal

An input to the program from the external world either directly or indirectly is considered to be tainted data, and passing tainted data to a sensitive function call like system(), setenv() will result in either command injection vulnerability, environment manipulation vulnerability, or some other security issue. All static analyzers should track tainted data propagation and flag a variable as tainted if it is used to store tainted data. Also, it is necessary for the source analyzer to remove the taint flag from the variable if the input data in it is overwritten with a constant data. Not updating the taint flag on a variable could either result in a false negative or a false positive.
In the sample code below, the input from a user is obtained and stored in a variable named “value”. Later, the contents of this are overwritten with constant data. So, the taint flag should be removed from the variable, as it is safe to pass this variable to the setenv function.

The purpose of this test was to first check if the source analyzers assign a taint flag to variables that store data from the external world and next to check if that taint flag is removed later when the variable is assigned a constant value.

4.3.2.1 Presence in Commercial/Open Source Software. The code pattern here, i.e., storing external input into a variable and later storing a constant data into the same variable, can be found in lot of open source software. It is a way of saving some memory by reusing an already declared buffer instead of having some extra space. A regular expression search in an open source code search engine revealed at least six projects that followed the same pattern. If these codes were to be run over the static analyzers, a lot of false positives related to command injection vulnerability or similar ones might be produced.

4.3.2.2 Source Code. The source code for the second test follows:

1. #include<stdio.h>
2. #include<string.h>
3. #include<stdlib.h>
4. int main()
5. {
6.  char name[30];
7.  char value[20];
8. scanf("%19s",value);
9. printf("\n%s\n",value);
10. strncpy(name,"sathya-090-",6);
11. strcpy(value ,"Hrt");
12. setenv("myname",value,1);
13. return 0;
14. }

4.3.2.3 Problem. In the above code, an input is obtained from the user and is stored in a char array. The value obtained from the user is directly passed to the setenv() function call which manipulates the value of an environment variable. It should be noted that the above code was split into two versions, one in which line 11 is commented and one in which line 11 is uncommented and is part of the code. When line 11 is uncommented the following was found:

Sparse: No defects found.
Splint: Return value of scanf is ignored;
RATS: Warning of usage of scanf
Flawfinder: 4 Warning of buffer overflow. Flagging the use of static-sized array, use of scanf,strcpy, strncpy.
Fortify: Setting manipulation error on setenv() line

Sparse did not find any defects with the code. Splint resulted in a warning that specified that the return value of scanf was being ignored which probably was not relevant to the current test case. RATS resulted in a false positive. It gave a warning of the usage of scanf, although scanf is, in fact, being used properly in the above code.
Flawfinder gave four false positive results for a buffer overflow error. It gave a warning about declaring a static-sized array, using scanf to obtain input into the char array, using strcpy, and using of strncpy, although all the functions were used in a safe manner and did not affect the security of the source code in any way. So, except for Fortify, all these tools produced the same result, irrespective of line 11 being present or commented out in the code.

When the source code with line 11 commented out was run over Fortify, it identified passing the user input value directly to setenv as a security vulnerability and gave a setting manipulation error. But even after that line was uncommented, Fortify still gave the setting manipulation error, which is a false positive. Line 11 is assigning a completely new value to the “value” variable which initially had the user input stored in it. As a result, passing “value” to setenv is safe usage, since it has the hardcoded value in it. Nevertheless, Fortify still gives the setting manipulation error.

4.3.3 Test for Buffer Overflow, Format String and String Length Update

The buffer overflow problem has been in existence for a long time, and almost all static analyzers are capable of detecting simple buffer overflow vulnerabilities in source code. The risk of buffer overflow is rated high. Consequently, it is very important for a source analyzer to detect buffer overflow and format string problems in the code.

Also, for a static analyzer, it is important to have meta-information about a variable declared in the source code. Unfortunately, not all source code analyzers maintain meta-information about the variables that are used in the source code. One such piece of information that is necessary is the length of the string. It is not always possible
to have such information, but wherever constant data is being written into a character array, the length of the string should be updated. Having this type of information reduces some of the false positives.

4.3.3.1 Presence in Commercial / Open Source Software. Various types of software have buffer overflow vulnerabilities or format string vulnerabilities. A search for buffer overflow in the Bugtraq database listed around 1000+ results, indicating that these vulnerabilities are very much in existence. Since buffer overflow has been well handled by the new generation source code analyzers, the main intention of this test was actually to find if static analyzers catch format string vulnerability and if the source analyzer keeps track of string length to the extent possible. Keeping track of the string length could reduce a false positive in this test case.

In this test code, user input is obtained in str1 at line 7, and a buffer overflow problem is found there. However, because str1 is terminated to the appropriate length in line 8, the strcpy in line 9 is actually a safe call in this case, since str1 and str2 are variables having same buffer size. Hence, if a string length is correctly updated, line 9 will not be flagged as buffer overflow vulnerability.

Using a printf without a format string would probably be noticed in a lot of debugging code. Some debugging code is not taken away from the original source code prior to release, and this could lead to format string vulnerability. A search in a code search engine produced a lot of results related to this problem. However, not all of these could be a format string problem, because the data passed to printf may not be tainted data. Nevertheless, a similar pattern is seen in several source codes among which Mozilla
internationalization code and the open source version of Broadcom wireless driver are notable projects.

**4.3.3.2 Source Code.** The source code for the third test follows:

```c
1. #include<stdio.h>
2. #include<string.h>
3. int main()
4. {
5. char str1[40], str2[40];
6. scanf("%43s",str1);
7. str1[39] = '\0';
8. strcpy(str2,str1);
9. sprintf(str2,"%s",str1);
10. printf(str2);
11. return 0;
12. }
```

**4.3.3.3 Problem.** Two strings str1 and str2 are declared with same size. User input at line 6 does not limit the input size to 39; instead, 43 is specified as the input limit, so there is a buffer overflow vulnerability in line 6. In line 7, str1 [33] is set to null so that, the following two lines are a safe usage of strcpy and sprintf function calls. In Line 10, a string in which user input is stored,str2 is directly fed to the printf call without any format string specified. This could result in format string vulnerability. Here are the results.

Sparse: No defects found

Splint: Return type of scanf ignored, usage of sprintf leads to buffer overflow, format string in printf.
Flawfinder: 5 warnings. Flagging the usage of scanf, strcpy, sprintf and declaration of static size arrays.

RATS: Fixed size buffer, usage of scanf, strcpy, sprintf and printf without format string.

Fortify: Format string vulnerability and buffer overflow.

Although two explicit coding errors were present, Sparse did not flag those mistakes. While Flawfinder resulted in many false positives, it flagged the usage of all dangerous functions. These functions could be easily misused, but in this instance they were used in a safe way in the above code, with the exception of scanf which introduces buffer overflow vulnerability in the above code. Flawfinder did not catch the usage of printf, without specifying the format string option. RATS did find the usage of printf function which could lead to format string vulnerability. The other warnings RATS produced were just false positives. Fortify caught both the buffer overflow problem and the format string vulnerability in the above source code.

4.3.4 Test for Pointer Reference Update and Memory Leak

This test code was intended to verify if the source analyzer tracks pointer assignment and detects memory leak.

4.3.4.1 Presence in Commercial/ Open Source Software. Memory leaks occur due to not freeing up memory that was allocated dynamically in the program. Memory leaks can lead to denial of service attacks. Memory leaks are present in lot of commercial and open source applications, including Microsoft Word, Internet Explorer, and Firefox.

In this test code, there is only one memory leak. At line 5, variable b is assigned
a's value. If source analyzers keep track of this, it will not flag line 8 as a memory leak problem since free(b) is called there, thus releasing the memory which was declared at line 4 (since b actually points to a). A very similar code pattern can be observed in lot of open source projects, such as Syallable and Diet Libc.

### 4.3.4.2 Source Code.

```c
1. void func_mleak()
2. {
3. char *a, *b;
4. a = malloc(10);
5. b = a;
6. a[9] = '1';
7. a = realloc((void *)a, 2);
8. if(a == NULL) free(b);
9. }
10. int main()
11. {
12. func_mleak();
13. return 0;
14. }
```

### 4.3.4.3 Problem. This case was written to check if the static analyzers are able to determine a memory leak and no null check after malloc problems. In the above code, the function fun_mleak() has both the specified problems. Variable 'a' is malloced, and the return value is never checked against null before 'a' is de-referenced. Also, there is only one case wherein 'a' is being deallocated. When call to realloc fails to allocate memory,
the memory allocated initially by malloc calls is deallocated, or if realloc succeeds, the memory is not deallocated, hence a memory leak in the program. Here are the results.

Sparse: No defects except for few false positives

Splint: Index of possibly null pointer, dead storage b passed to free, storage a not released before return

Flawfinder: No defects found.

RATS: Warning about usage of realloc on sensitive memory

Fortify: Two memory leaks, missing check against null, usage of realloc on sensitive memory.

Sparse did not find any defects with this code, except for a few false positives about function declarations. Splint flagged the missing null check against the return value of malloc and also found the memory leak flaw in the code. In addition, it had one warning related to passing dead storage 'b' to free function call. This is probably a false positive, since that statement is executed only when the realloc fails, at which point b will have the initial value of 'a'. No luck with Flawfinder, in this case. Fortify had two memory leaks in which one memory leak was valid one and the other memory leak was flagged at the realloc line, thus it was probably a false positive since the next line checks the return value of realloc and calls free if the realloc fails. Fortify also detected the missing null check flaw. Other than this, it gave a warning about usage of realloc on memory that stored sensitive information. The same was returned by RATS.

4.3.5 Test for Taint Propagation Through Stack Referred Variables

Taint propagation is forwarding the taint flag from one variable to another
variable, i.e., when the source variable is tainted, if its value is assigned to any other variable, the taint flag must be propagated to the destination variable as well. Propagating the taint flag properly between variables reduces the number of false positives and false negatives.

The main intention of the test code was to check if the source code analyzer understands the relative addressing of variables. Instead of addressing a variable directly by its name, it is indirectly addressed by using the offset from a different variable in the stack.

4.3.5.1 Presence in Commercial/Open Source Project. The code pattern followed here (referring to a variable by its offset from a different variable in stack) is a common practice in C, C++ programs. This type of code is seen in a lot of open source projects, including big projects like PHP, Perl, FFMPEG, etc.

4.3.5.2 Source Code. Below is this source code for this test:

```c
1. int main()
2. {
3.   char a[20];
4.   char b[20] = "ls";
5.   scanf("%19s", a);
6.   strcpy(b, b+20);
7.   printf("%s\n", b);
8.   setenv("PATH","/usr/local/bin",1);
9.   system(b);
10.  return 0;
```
4.3.5.3 Problem. This program has command injection vulnerability. User input is obtained and is stored in variable ‘a’. Then strncpy function is used to copy the contents of ‘a’ into variable ‘b’. Note that the strncpy function call does not take the source as variable ‘a’ explicitly, rather it specifies b+20 which is the location of the first byte of ‘a’. It was written in this way so as to learn if the static analyzer really follows what is happening in the program. Later, after copying the value of ‘a’ into ‘b’, the PATH variable is set, and the system function is called with ‘b’ as its parameter. However, this program was also tested with parameter ‘a’ going directly into system() function call.

Sparse: No Defects found.
Splint: Return value of scanf ignored, unrecognized identifier setenv, return value system ignored.
Flawfinder: Usage of static sized buffer, usage of scanf, strncpy on static-sized buffer, usage of system() could be dangerous
Rats: Warnings about usage of fixed size buffer, scanf, and system() function calls
Fortify: Warning about command injection, unchecked return value of system()

Sparse did not find any defects with this code. Flawfinder, RATS, and Splint gave similar warnings related to the usage of the static size buffer, usage of scanf, and strncpy function calls, as well as ignoring the return value of the system function call. None of these programs were able to trace that variable ‘b’ holds the value of variable ‘a’ which is nothing but the user input.

However, the program was modified to make the system call directly take variable ‘a’ as its argument. Even after changing the argument to ‘a’, except for Fortify, none of
the tools was able to trace the input data path. The rest of the static analyzers produced the same result. Further, Sparse did not find any defects, even though the command injection was very explicit in the program. Fortify noticed that the input variable was directly sent to the system() function, flagged that line as a high command injection vulnerability, and showed a small trace of how user input reached the system function() call.

4.3.6 Test for Integer Overflow

The purpose of this test code was to verify whether the static analyzers detect integer overflow problems. Integer overflow can lead to denial of service attacks. While integer overflow problems can be easily overlooked, detecting them could save developers considerable time and debugging effort in the long run.

4.3.6.1 Presence in Commercial /Open Source Projects. Integer overflow problems are widespread across both commercial and open source applications. GIMP and Internet Explorer are some of the popular applications found to have integer overflow vulnerabilities.

4.3.6.2 Source Code. The source code for this test follows:

1. int main(int argc, char *argv[])
2. {
3.   int a, i ;
4.   long long b;
5.   scanf("%lld",&b);
6.   a = b;
7.   for(i=0;i>a;i++)
4.3.6.3 Problem. This program has an integer overflow problem that could lead to a denial of service attack. A long long variable which is generally double the size of an int is used to get some input from the user. In the later part of the program, this long long value is copied into an integer variable. This assignment can cause the integer variable to overflow. Later in the program, there is a loop wherein variable ‘i’ runs from 0 to the value of ‘a’ (which is negative due to overflow) and assumes that the loop does some heavy operations. Now, if the value of 'a' turns out negative, the loop will run for a very long time and can potentially lead to a denial of service attack. Here are the results:

Sparse: No defects found
Splint: Assignment of long long to int variable, return value of scanf ignored.
Flawfinder: No defects found
RATS: No defects found.
Fortify: No Defects found.

Except for Splint, no tool was able to find anything wrong with the code. Splint gave a warning about the assignment of long long to int type.

4.3.7 Test for Indirect User Data Assignment to Variable

This test code was adopted from a strategy in [21]. The intent was to see if static analyzers understand the logic of the source code. Although it is very difficult for static
analyzers to detect program logic, doing so helps them reduce false positives by a huge factor.

4.3.7.1 Presence in Commercial /Open Source Application. As mentioned earlier, this test code idea is purely adopted from a research paper and does not correspond to any particular vulnerabilities or pattern that could be observed or searched for in any commercial or open source applications.

4.3.7.2 Source Code. The source code for this problem follows:

1. int main()
2. {
3. char a[10], b[10];
4. int i = 'a', j = 0;
5. scanf("%9s", a);
6. for( j = 0; j < strlen(a); j++){
7.     for( i = 97 ; i < 122; i++)
8.     {
9.         if(a[j] == i)
10.            {
11.                b[j] = i;
12.                break;
13.            }
14.     }
15. }
16. b[j] = '\0';
17. printf("%s\n", b);
18. system(b);
19. return 0;
20. }

4.3.7.3 Problem. This program has command injection vulnerabilities, but the main intention of this program was not to see whether static analyzers are able to detect the command injection vulnerability. Input obtained from the user is stored in variable ‘a’ and is copied to ‘b’ by comparing each and every character against all the ASCII values. The ascidia values are copied to the destination string. Certainly no program will have a function, similar to this, for copying one string to another; this was merely written to see if static analyzers follow the code.

Sparse: No defects found.
Splint: No defects found.
Flawfinder: No defects found.
Rats: No Defects found.
Fortify: A warning about command injection vulnerability.
Table 2. Results of Running Test Code over Various Static Analyzers.

<table>
<thead>
<tr>
<th>Test Case</th>
<th>Sparse</th>
<th>Splint</th>
<th>Flawfinder</th>
<th>Rats</th>
<th>Fortify</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test for Null Pointer Dereference</td>
<td>F -</td>
<td>F -</td>
<td>F -</td>
<td>F -</td>
<td>F -</td>
</tr>
<tr>
<td>Test for Taint Flag propagation and Removal</td>
<td>F -</td>
<td>F +</td>
<td>F +</td>
<td>F +</td>
<td>F +</td>
</tr>
<tr>
<td>Test for Buffer Overflow, Format String and String length update</td>
<td>F -</td>
<td>F +</td>
<td>F +</td>
<td>F +</td>
<td>N.F.R</td>
</tr>
<tr>
<td>Test for Pointer Reference Update and Memory Leak</td>
<td>F -</td>
<td>F +</td>
<td>F -</td>
<td>F +</td>
<td>F +</td>
</tr>
<tr>
<td>Test for Taint Propagation through Stack Referred Variables</td>
<td>F -</td>
<td>F -</td>
<td>F -</td>
<td>F -</td>
<td>F -</td>
</tr>
<tr>
<td>Test for Integer Overflow</td>
<td>F -</td>
<td>N.F.R</td>
<td>F -</td>
<td>F -</td>
<td>F -</td>
</tr>
<tr>
<td>Test for Indirect User Data assignment to Variable</td>
<td>F -</td>
<td>F -</td>
<td>F -</td>
<td>F -</td>
<td>F -</td>
</tr>
</tbody>
</table>

F - = False Negative, F + = False Positive, N.F.R = No False Result
CHAPTER 5
FALSE RESULTS RESOLUTION

5.1 Introduction

The current set of static analyzers has both false positive and false negative problems. Each of the static analyzers either misses vulnerability or flags a proper or valid statement as a security bug. Sure, it takes a long time to completely eliminate false positive and false negative problems -- assuming it is even possible to do so. However, it would be highly beneficial to lower the number of false positives and false negatives. This chapter presents techniques developed to lower the number of false results by static analyzers.

The results of a vulnerability scan of various source files revealed various problems in the current set of static analyzers. To overcome this set of problems, a set of logical rules/statements was written. The initial plan was to convert these logical rules into a set of rule packs for a static analyzer. These rule packs are similar to a plug-in for software. These rules sets were written in XML with a proper syntax so that software analyzers recognize the format and can perform some extra operations based on the specifications written inside the XML file. This helps the static analyzer to more efficiently detect problems and reduces the number of false positives. The following are the set of logics or rules that were designed to overcome the false positive/negative problems.

Fortify, a static analyzer was the initial target since it has options for writing customized rule sets, including those that detect additional types of vulnerabilities or
discard a particular type of vulnerability. Also, the Fortify software analyzer has an inbuilt custom rule set generator that guides the developer in creating rule sets. In fact, Fortify provides documentation on how to write custom rule sets. The custom rule set generator is designed to generate rules that are customized to a specific project rather than a generic rule set. For example, one can specify the function within the program that data from the external world enters, which function is going to use this input, and other similar things.

Unfortunately, the documentation is not complete. For example, there is no documentation on how to remove taint flags from the variables, there is no list of the different type of taint flags that are used by the program, one is unable to ascertain if variables are dynamically allocated, etc. Thus, while Fortify has access to this type of information, sparse documentation exists, hence restricting access to these data for use by third party developers.

Because of Fortify’s lack of documentation, in order to implement the logic for scanning and finding the vulnerabilities in the source code, it was necessary to write a custom tokenizer and parser for a language. Sure, rewriting the parser and tokenizer was not a good option to start with, as, other software exists that already includes a parser and does a great job of parsing source code. For example GCC Frontend includes a great parser for many languages, but the problem comes in adopting the source code to the custom-built source analyzer project in order to perform all the required tasks. One needs a deep understanding of the GCC source code or the source code of other software for modification, recompiling with all the dependencies, and making it work. Such a process
can take a tremendous amount of work and time. Hence, a custom-built tokenizer and parser were written in order to perform the job that was required for this project. The following section provides a little information about the design and implementation of the tokenizer and parser for the C language, the logical rules that were written to overcome the problems, and how these logics were implemented and tested with the source code.

5.2 Design

Figure 13 shows the steps involved in analyzing the source code. First, the source code is given to the tokenizer which splits up the program into tokens. Next, the parser takes these tokens, validates them, and passes them to the analyzer which checks for security issues and outputs the result to the user. To know more about the member variables and the member functions of each and every class presented here, refer to Appendix B.

5.2.1 Package - Lexical Analyzer

This package has a tokenizer implementation. The tokenizer reads the source code character-by-character and forms appropriate tokens. It tokenizes the whole source code file and provides these tokens to the parser. This package, lexical analyzer has the following interface/classes inside it.

5.2.1.1 Interface Tokenizer. This interface specifies the necessary methods that need to be implemented for a valid tokenizer. A tokenizer should be specific to a programming language, and it needs the grammar file for that language to be in a particular format. Please refer to Appendix A for more details on this.

5.2.1.2 Class Token. This class defines the member variables for a token.
Token types are pretty generic with respect to programming languages [37]. For example, each programming language might have a different set of keywords. Nevertheless, they belong to same token type/category, namely, KEYWORDS [37].

5.2.1.3 Class - CTokenizer. This is the tokenizer for the C programming language. This class implements the tokenizer interface. The grammar file for the C programming language was adopted from a YACC grammar file written by Lee in 1985 [38, 39]. This grammar file was modified in order to get it to work with this tokenizer implementation.

5.2.2 Package - ProgramElements

This package contains the classes that represent the basic elements of a program. The approach in structuring a program is shown in Figure 14, and this package contains all the classes discussed in the subsections below.

5.2.2.1 Class Class. Classes form the topmost layer in the program hierarchy. Any C program is considered to have only one class, the “MAIN” class. Other than that, there are no other classes involved in a C program.
5.2.2.2 **Class Function**. This class is going to maintain meta-information about any function that is declared inside the source code. This class has two main members, blocks and a variable stack. A block has variables inside it. The variable stack is updated whenever a variable declaration statement is found in the function.

5.2.2.3 **Class Variable**. This class has a huge number of member variables inside it. This class maintains a lot of information about each and every variable declared in the program source code. Whenever a new variable is declared in a program source, a new variable object is created, is added to the function in which it was declared, and is stored under the appropriate block. Initially, the size of the variable is just hardcoded inside the constructor. The size is set to 1, 4, or 8 based on the type of string. But the variable could be changed easily just by having a configuration file that contains these data. There are variables to keep track of array size and also malloc size of a variable. The array size is updated if the variable is declared as a static array, and the malloc size is updated if the
variable is declared as pointer or if it is later assigned a return value of malloc function call. This class has a string variable named ‘pointsto’ which is used to keep track of whether the variable points to a null location, malloc, realloc, or another variable. There is a variable named ‘flag’ which helps in maintaining the taint flags and source of taint information, i.e., whether is it from user input, a file, or from an environment variable, etc. Figures 15 and 16 illustrate the class design of the package lexical analyzer and the package program elements, respectively. Refer to Appendix B to know more about what type of metadata are maintained for variable and how it is used for identifying the problems.

5.2.3 Package Parser

This package contains only one class CParser. Initially, the parser is implemented for the C language and has minimal internal functions. It was mainly written just to implement and test the logic or the rules. Hence, it cannot be used with any C file. It works for all the source code discussed in Section 4.3, but it is not expected to work with any other source code. Also, this class assumes that the program is syntactically correct, i.e., it is compilable with a standard C compiler.

5.3 Logic and Implementation

5.3.1 Test for Null Pointer Dereference

Refer to Section 4.3.1 for details about this test case and the source code used for testing the static analysis tools.

5.3.1.1 Logic.

- Before being used, the pointer variable should be malloced or assigned a constant
Figure 15. Class design of package lexical analyzer.
Figure 16. Class design of package program elements.
memory location.

- If the pointer is assigned another variable in rhs, the variable should be malloced or should be assigned to a constant (memory location).

5.3.1.2 Implementation. To implement this logic, the CParser class has two different functions, one to handle the variable declaration statements and the other to handle the assignment statements. In this case, the pointer ‘p’ is declared first. Next, the declaration statement handler inside the CParser creates a new variable object and adds it to the function. Then, at line 6, variable ‘p’ is assigned to the return value of the malloc function call ((int *) malloc (4)). Once this statement is parsed, the assignment statement handler updates the pointer variable's (p) mallocsize to 4. At line 5, ‘p’ is assigned to variable ‘c’. The assignment handler now checks if the rhs of the assignment statement is a variable or a constant. Since it is a variable, the assignment handler checks if ‘c’ has been malloced and to where c is pointing (mallocsize and pointsto are member variables of the class ‘Variable’). Since neither of them is assigned, the CParser throws an error saying variable c is not initialized. Now, at the next statement *p = 100;, the assignment handler checks the malloc size of variable ‘p’, and it throws an error that ‘p’ might point to null, which could end up in a segmentation fault.

5.3.1.3 Result.

WARNING : Variable c might not be initialized or checked against NULL

ERROR - THIS VARIABLE (var name=p) WAS MALLOCED AND NOW REFERENCE IS BEING CHANGED, THIS WILL LEAD TO MEMORY LEAK
ERROR - Var p might point to NULL , line no = 8

5.3.2 Test for Taint Flag Propagation and Removal

Refer to Section 4.3.2 for details about this test case and the source code that was used for testing the static analysis tools.

5.3.2.1 Logic. During an assignment statement, a string copy function, or any other equivalent function, if the source is a constant or if the source variable is not tainted, the taint flag (if any) should be removed from the destination variable.

• Pseudo rule:

<DataflowRule>

<Function name = strcpy>

if <TaintFlagCheck(source) == false> OR < Source == CONSTANT>

then <TaintFlagRemove(DestinationVariable)>

</DataflowRule>

• Since there was not much documentation on how to retrieve the taint flag of a variable nor on getting the parameters of function, it was difficult to implement this dataflow rule in Fortify.

5.3.2.2 Implementation. A function handler was written to implement this logic. In the current version of the CParsen class, the function prototype and format of some of the standard functions are hardcoded. For example, the function handler goes through an if-else loop and compares the function name found in source code against a set of strings to identify the function call. If it is the scanf function call, the function handler retrieves the format string and identifies the variable where input is stored. Also, the function handler
adds a taint flag to the variables whose values are obtained from user. The scanf at line 13 receives input from the user and stores it in the variable “value”. Later, at line 16, the user input is overwritten by a constant value. The function handler checks to see the source parameter of strcpy and whether it is a constant or a variable. Since in this case it is a constant, the function handler removes the taint flag from the destination variable (the variable “value”, in this case). Now, if the value is used in the setenv function, it should be completely safe as the variable is no more tainted. Hence, the function handler checks if the parameters of setenv are tainted. If they are, the function handler raises an error; otherwise, it does not.

5.3.2.3 Result.

With Line 11 commented:
ERROR Variable name = value seems to be tainted, flag = 1

With Line 11 Uncommented:
strcpy detected
Removing Taint Flag on variable value

5.3.3 Test for Buffer Overflow, Format String, and String Length Update

Refer to Section 4.3.3 for details about this test case and the source code that was used for testing the static analysis tools.

5.3.3.1 Logic.

- The main logic is to keep track of the length of the string (char array).
- A null character appended to the char array is a signal to update the length of the
string to the index wherein null was stored.

5.3.3.2 Implementation. This test has logic pretty similar to the previous one. There are two interesting things in the source code. First, there is a buffer overflow in the scanf function call at line 7. Some of the source code analyzers did not identify this vulnerability. But the custom parser that was written for this project identified this buffer overflow error. The next line appends a string terminator symbol to the appropriate position in order to mark the end of string. At line 9, the strcpy copies string 1 in to string 2. So now logically, string 1 and string 2 should be tainted. But no buffer overflow problem with string 2 exists, since string 1 is now terminated at the right index. However, assume a case wherein line 8 is commented out, i.e., string 1 is not appended with a null character. Now, copying string 1 to string 2 should result in a buffer overflow condition.

This problem was not detected by most of the source code analyzers, but the custom parser identified this error and reported it. Detection is done by having a ‘length’ member variable inside the variable class. This length is updated based on scanf or a null char assignment. If the length of the source string is greater than the array size of the destination variable, the parser detects the buffer overflow error.

The second flaw in this code is the format string vulnerability at line 11. Since string 2 is tainted, passing it directly to printf results in format string vulnerability. The function handler inside CParser checks if the parameter 0 to printf is a variable or a constant. If it is a variable and if the variable is tainted, the function handler raises format string error.

5.3.3.3 Result.

(With line no 8 uncommented)
DETECTED BUFFER OVERFLOW in line 7
ERROR ARRAY INDEX > ARRAY SIZE .. BUFFER OVER FLOW, Line NUmber =7
PRINTF FUNCTION CALL
ERROR- LOOKS LIKE VARIABLE str2 IS TAINTED AND PASSING IT INTO PRINTF() WITHOUT FORMAT STRING COULD LEAD TO FORMAT STRING VULNERABILITY
(With line no 8 commented out.)
DETECTED BUFFER OVERFLOW in line 7
ERROR ARRAY INDEX > ARRAY SIZE .. BUFFER OVER FLOW, Line NUmber =7
Strcpy detected
ERROR Source Variable length is > dest variable size in line =9
SPRINTF detected
ERROR Source Variable length is > dest variable size in line =10
PRINTF FUNCTION CALL
ERROR- LOOKS LIKE VARIABLE str2 IS TAINTED AND PASSING IT INTO PRINTF() WITHOUT FORMAT STRING COULD LEAD TO FORMAT STRING VULNERABILITY

5.3.4 Test for Pointer Reference Update and Memory Leak

Refer to Section 4.3.4 to know more about this test case and the source code that was used for testing the static analysis tools.
5.3.4.1 Logic.

- The static analyzer should keep track of aliases to some level.
- Pseudo rule:

```<ControlflowRule>
Variable f;
State start, allocated, safe, leak
start->allocated { f = alloc(…) }
allocated->safe { return(f) | free(f) | free(alias(f)) | ifblock(f, null, true) }
allocated->leak { end_scope(f) }
</ControlflowRule>
```

5.3.4.2 Implementation. This logic has to deal with memory leaks. The main thing is to keep track of aliases. In the source code at line 5, variable ‘a’ is copied into variable ‘b’. While parsing the statement, variable ‘b’ is updated with the value of ‘a’, and the pointsto variable of ‘b’ is set to ‘a’. Now, ‘b’ points to ‘a’. In line 8, free (b) is called if ‘a’ is null. So, if the realloc function call fails, variable ‘a’ is released. Thus, logically there is no memory leak if ‘a’ is null. Because ‘b’ points to variable ‘a’, ‘a’ is freed in that statement. The function handler takes care of this. Hence, another false positive is reduced by the proposed logic.

5.3.4.3 Result.

(With line 8 commented out)

End of function func_mleak
This function had 2 variables
Analyzing variable a
MEMORY LEAK DETECTED. Variable name = a

Analyzing variable b

(With line 8 uncommented)

free(b);

varname = b is being freed

varname = a is being freed

(It recognized that variable b points to a in this case and frees up variable a and thus reducing a false positive.)

5.3.5 Test for Taint Propagation Through Stack Referred Variables

Refer to Section 4.3.5 to know more about this test case and the source code that was used for testing the static analysis tools.

5.3.5.1 Logic.

- The main concept here is to track the variable based on its relative position in the stack.
- A variable can be addressed by its name directly, or it can be addressed by its position in stack.
- It is important to construct a virtual stack while going through the source code [40]. When a variable is addressed by its relative position, the virtual stack should be checked to see which variable is actually referred to it. This will help in detecting more problems in program source code.

5.3.5.2 Implementation. The solution for this test case cannot be obtained by writing a rule file for the source code analyzer. The source code analyzer should maintain
a stack that mimics the program’s stack when it is run. This was implemented in the CParses the class Function has a stack object in it. So, whenever a variable is added to a function, it is pushed onto the stack. In line 7 of the source code, strcpy copies the value from b+20 into ‘b’. b+20 is nothing but variable ‘a’. This information can be obtained only if the source analyzer builds a pseudo stack. Hence, the CParses class identifies that b+20 is nothing but variable ‘a’. And since variable ‘a’ is tainted, the taint flag is now forwarded to variable ‘b’. So, at line 9 when variable ‘b’ is infused into the system’s method, the parser throws a tainted variable in System () function call error.

5.3.5.3 Result.

Expression b + 20 at line = 6, actually points to variable a.

SYSTEM FUNCTION CALL -

ERROR- LOOKS LIKE VARIABLE b IS TAINTED AND PASSING IT INTO SYSTEM () COULD LEAD TO COMMAND INJECTION VULNERABILITY

5.3.6 Test for Integer Overflow

Refer to Section 4.3.6 for details more about this test case and the source code that was used for testing the static analysis tools.

5.3.6.1 Logic. Copying data from a variable of bigger size to a variable of lower size will result in an integer overflow or misinterpretation of data.

5.3.6.2 Implementation. There is an integer overflow problem here. Long long variable ‘b’ stores user input that is copied into variable ‘a’. Consequently, variable ‘a’ is loaded with a value that is more than what variable ‘a’ can store. The variable class has a variable named ‘size’ inside it. The value of this variable depends on the data type of the
variable. So, having a destination size less than the source size prompts an overflow error.

5.3.6.3 Result.

Var source = b dest = a

OVERFLOW: DATA SIZE OF DATATYPE long long IS > DATATYPE int

Table 3 shows the result of running various static analyzers over the test source code. As one can see, the custom parser that was written for this project detects the vulnerabilities in the entire test source code, whereas the other static analyzers were not able to find the problems present in the source code. This proves that implementing the above discussed logic in other static analyzers will improve their efficiency, thus making for better static source code analysis. The accuracy in detecting these errors is limited by the functionality of the parser that implements these logics. If the parser extracts all necessary information from the source code and updates the metadata of the variable then there won’t be any place for false results. Since our custom parser has various limitations such as it cannot preprocess header file, does not handle macros, cannot handle variables that are extern or static etc., the accuracy is limited to finding vulnerabilities in a source code written inside a single source file. Although implementing these logics in an advanced parser could yield better results.
Table 3. Comparison of Our Source Analyzer with Other Tools.

<table>
<thead>
<tr>
<th>Test Case</th>
<th>Our Source Analyzer</th>
<th>Sparse</th>
<th>Splint</th>
<th>Flawfinder</th>
<th>Rats</th>
<th>Fortify</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test for Null Pointer Dereference</td>
<td>✓</td>
<td>✘</td>
<td>✘</td>
<td>✘</td>
<td>✘</td>
<td>✘</td>
</tr>
<tr>
<td>Test for Taint Flag propagation and Removal</td>
<td>✓</td>
<td>✘</td>
<td>✘</td>
<td>✘</td>
<td>✘</td>
<td>✘</td>
</tr>
<tr>
<td>Test for Buffer Overflow, Format String and String length update</td>
<td>✓</td>
<td>✘</td>
<td>✘</td>
<td>✘</td>
<td>✘</td>
<td>✓</td>
</tr>
<tr>
<td>Test for Pointer Reference Update and Memory Leak</td>
<td>✓</td>
<td>✘</td>
<td>✘</td>
<td>✘</td>
<td>✘</td>
<td>✘</td>
</tr>
<tr>
<td>Test for Taint Propagation through Stack Referred Variables</td>
<td>✓</td>
<td>✘</td>
<td>✘</td>
<td>✘</td>
<td>✘</td>
<td>✘</td>
</tr>
<tr>
<td>Test for Integer Overflow</td>
<td>✓</td>
<td>✘</td>
<td>✓</td>
<td>✘</td>
<td>✘</td>
<td>✘</td>
</tr>
<tr>
<td>Test for Indirect User Data assignment</td>
<td>✓</td>
<td>✘</td>
<td>✘</td>
<td>✘</td>
<td>✘</td>
<td>✘</td>
</tr>
</tbody>
</table>
CHAPTER 6
SUMMARY AND CONCLUSION

6.1 Summary

Initially, a study on software vulnerabilities was carried out. Why various types of vulnerabilities occur and how can they be exploited are discussed in the thesis. A literature survey of current list of static analysis tools was done. A list of static analysis tools and a brief description about these tools are listed in Chapter 2. This survey discusses various tools and how these static analysis tools have grown over the years. Most first generation static analysis tools were geared towards a particular language, but new, or second generation, static analysis tools tend to be geared towards more than one programming language. They support various languages and are able to detect a wider range of vulnerabilities.

Analysis on various types of source code was performed. Students’ code from each undergraduate level was analyzed, and the results were shown. The analysis and results are novel contribution as this type of research on student’s assignment has not been done. It is clear from the analysis results that students are not very aware of secure coding concepts and are not worried about the vulnerabilities present in their coding assignments. The results also show how programming languages can help in reducing the number of vulnerabilities in software. A survey on security-related courses offered in various universities was performed. The topics covered in the security courses of different universities were analyzed, and the common topics among several security courses are discussed in this thesis. Offering these types of security courses will, improve
students’ knowledge of security vulnerabilities. Analysis on student code and identifying the course list from various universities was one of the major contributions of this thesis.

The next analysis was performed on the source code posted on a popular open technical interview forum. The main idea behind this analysis was to find out to what extent students who are ready to join a company are aware of security issues related to programming. Although there were not many source codes posted in the forum, around 20% of the ones that were posted had vulnerabilities in them. Although 20% seems low, the fact that these source codes were not big enough to have much vulnerability should also be considered. Again analysis of source code present in tech interview forums and trying evaluate student’s perspective of security is something which is has not been worked on previously.

The analysis on open source code and the Bugtraq database demonstrates that even open source software suffers from a lot of vulnerabilities. From the Bugtraq database, it is clear that there is not much difference between the number of vulnerabilities reported on commercial applications and the number of vulnerabilities reported on open source software. It is necessary for organizations to have policies that ensure that the code they borrow is safe and can be used without any problem (with respect to security) as a part of their existing code. Some of these policies for borrowing code from open source are discussed in the thesis. These are a very generic set of policies or steps, and implementing them might differ from one organization to the other.

Based results of the various above mentioned analyses, test cases were written that could expose the false positives and false negatives of common static analysis tools.
Running these test source codes over various tools revealed some of their false positives and false negatives, and the result of these tests are shown in the thesis.

To lower the rate false positives and false negatives of the static analysis tools, some rules/logic were designed. When implemented, these logical rule sets will lower false rates of most static analyzers. To test the logic, it was necessary to write a custom tokenizer and parser for C programs. Although the custom parser is not very generic and was not a great contribution to the parser world, it served the purposes of this project well. It did parse the test code and validated the rules/logic that was framed for reducing the false rate. Additionally, this parser could be used for validating such rules written in the future. Running the test cases over the static analysis tool that was written for this thesis proved that the designed logic were correct and had far fewer false positives and false negatives compared to the other tools. A project in this area would have been more like implementing the existing logic, or bringing in support for a new language for an existing source analyzer etc., but framing new logic or rule sets to improve the efficiency of the static analyzers is another novel contribution of this thesis.

Contributions:

- Student code Analysis. Identifying problems in student’s assignments.
- Assessment of open source code.
- Identified problems in static analysis tools by using appropriate test cases.
- Framed rulesets for false result resolution.
- Implemented and validated the rule set using the custom source analyzer.
6.2 Future Work

This thesis opens up a lot of opportunities for future work. The student code analysis section especially has a lot of scope for further research. Since there has not been much research done on this area, it is a good one to explore. The student code analysis that was performed as a part of this research is constrained to the assignments that belong to one particular university and department (the Computer Science Department at Utah State University). It would be interesting to study the vulnerabilities patterns in student assignments from other universities. In particular, analyzing assignments from universities wherein secure coding is already taught to see how effective these courses are would be worthwhile. Further, a comparison of students’ perspective of security from one university to the other would be interesting.

Analyzing the assignments obtained from various universities and comparing them would show which university has better results. Courses from the university with the best results could potentially serve as a model for other universities.

The interview forum that was analyzed in this thesis was just one of many. Possibly, the analysis could be extended to other similar forums to find out more about the vulnerabilities in source code found in those forums. Also, it would be an interesting option to collaborate with the domain administrators of these forums to learn the geographical location of the author, and find out more about the schools present in those regions to analyze the vulnerabilities pattern made by people of a particular region or belonging to schools from a particular region.

The policy for borrowing code from the open source community described in this
thesis is a very generic one. Further research can be done in this area to learn more about organizations that borrow a lot of code from open sources; a literature survey of policies used in these organizations for adopting code from open source could be taken. Such an undertaking would certainly help small organizations get an idea of the processes followed in bigger organizations, which would help them in safely borrowing code from the open source community.

The simple static analyzer that was written in order to validate the logic is very closely coupled to the test source code that was written. Further development of the current tool in order to learn various other vulnerabilities and extending the parser to parse several other languages is a whole new exciting area and has a lot potential for future work.

6.3 Conclusion

Software security is a very important area to concentrate on no matter what product a company is developing. A small vulnerability could end up in a huge loss to the company. Organizations should definitely have some secure coding standards and training programs that help employees understand the importance of secure coding and provide ongoing training for good practices. Further, if secure coding courses were offered at every university, students would get good exposure to security concepts, thus improving their software development skills. Also, static analyzers should be integrated with compilers so that static analysis becomes a standard part of the software build process rather than a separate process in the software development life cycle. This would definitely improve software security standards across the board.
The source analyzer that was written for this project was able to identify various problems found in the sample source codes that were used as test cases for various other source analyzers. Implementing the logic discussed above will definitely help current source analyzers to reduce their number of false positives and false negatives, thereby ensuring better code quality. Such a source analyzer can improve software quality by reducing the number of bugs. Its lower false positive and false negative rates will potentially draw more people towards using source analyzers, as high false positive and false negative rates are the main drawbacks of using source analyzers.
REFERENCES


APPENDICES
APPENDIX A

CGrammar.txt

The following are the contents of grammar file that was used with the custom tokenizer program.

#KEYWORDS
  "auto"
  "break"
  "case"
  "char"
  "char*"
  "const"
  "continue"
  "default"
  "do"
  "double"
  "double*"
  "else"
  "enum"
  "extern"
  "float"
  "float*"
  "for"
  "goto"
  "if"
  "int"
  "int*"
  "long"
  "long*"
  "register"
  "return"
  "short"
  "short*"
  "signed"
  "sizeof"
  "static"
  "struct"
  "struct*"
  "switch"
  "typedef"
  "union"
"unsigned"
"void"
"void*"
"volatile"
"while"

#OPERATORS
">>="
"<<="
"+="
"-=
"*=
="/=
"%=
"&=
"^=
"|=
">>
"<<
"++
"--
"->
"&&
"||
"<=
">=
"==
"!=
"=
"&
"!
"~
"+
"*=
"/
"%=
"<
">"n
"heroes"

#BRACKETS
LexicalAnalyzer

**Class LexicalAnalyzer**

```java
public class LexicalAnalyzer extends java.lang.Object

Lexical Analyzer is the broker between the parser and the tokenizer. The parser constructs the lexical analyzer object with the language as a parameter. The lexical analyzer will construct a new Tokenizer object that's specific to the particular object.
```

<table>
<thead>
<tr>
<th>Constructor Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>(java.lang.String Language)</td>
</tr>
<tr>
<td>Constructor for Lexical Analyzer.</td>
</tr>
</tbody>
</table>

| java.lang.String Language, java.lang.String grammarfile |
| A constructor that takes in two parameters, first param specifies the language and the second one is the grammar file which contains the grammar for the specified language. |

<table>
<thead>
<tr>
<th>Method Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Tokenizer</strong> getTokenizer()</td>
</tr>
<tr>
<td>Returns the tokenizer object</td>
</tr>
</tbody>
</table>

Methods inherited from class java.lang.Object

copy, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

<table>
<thead>
<tr>
<th>Constructor Detail</th>
</tr>
</thead>
<tbody>
<tr>
<td>LexicalAnalyzer</td>
</tr>
<tr>
<td>public LexicalAnalyzer(java.lang.String Language)</td>
</tr>
<tr>
<td>Constructor for Lexical Analyzer. You need to specify the language for the tokenizer instance. The language can be C, CPP, JAVA, ...... But right now I have</td>
</tr>
</tbody>
</table>
tokenizer only for the C language.

**LexicalAnalyzer**

public `LexicalAnalyzer(String Language, String grammarfile)`

A constructor that takes in two parameters, first param specifies the language and the second one is the grammar file which contains the grammar for the specified language. It has to be a txt file. Please refer to cgrammar.txt in the project folder to understand how a grammar file should look like

**Method Detail**

**getTokenizer**

public `Tokenizer getTokenizer()`

Returns the tokenizer object

**LexicalAnalyzer**

**Interface Tokenizer**

All Known Implementing Classes:

CTokenizer

public interface `Tokenizer`

Interface Tokenizer should contains prototypes of functions that should be implemented by any specific Language tokenizer.

**Method Summary**

<table>
<thead>
<tr>
<th>Token</th>
<th>getnextToken()</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>openGrammarFile()</td>
</tr>
<tr>
<td>int</td>
<td>openSourceCode(String sourcecodefilename)</td>
</tr>
<tr>
<td>int</td>
<td>scanGrammarFile()</td>
</tr>
</tbody>
</table>

**Method Detail**
openGrammarFile
int openGrammarFile()
This Function should open the GrammarFile. Should return 0 on success on 1 on failure

scanGrammarFile
int scanGrammarFile()
Should scan the grammar file and populate the data structures

openSourceCode
int openSourceCode(java.lang.String sourcecodefilename)
Open the source code for reading...
Parameters:
sourcecodefilename - Filename of the source code

g getNextToken
Token getNextToken()
Should provide the caller with next available token

LexicalAnalyzer
Class CTokenizer
java.lang.Object
├──LexicalAnalyzer.CTokenizer
All Implemented Interfaces:
  Tokenizer

public class CTokenizer
extends java.lang.Object
implements Tokenizer
This is tokenizer for C program. This class implements Tokenizer interface.

Field Summary
java.util.logging.Logger CtokenLogger

Constructor Summary
### Constructor Detail

**CTokenizer()**  
Constructor

**CTokenizer(java.lang.String GrammarFile)**  
constructor with a grammar file specified in the parameter

### Method Summary

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>getNextToken()</strong></td>
<td>This function should return the next token from the source code it scans</td>
</tr>
<tr>
<td></td>
<td>the source code and finds the next token from the code and returns it</td>
</tr>
<tr>
<td><strong>openGrammarFile()</strong></td>
<td>This function opens the grammar file and sets the appropriate flags</td>
</tr>
<tr>
<td><strong>openSourceCode(String sourcecodefilename)</strong></td>
<td>This function opens the source code and sets up the buffered reader...</td>
</tr>
<tr>
<td><strong>scanGrammarFile()</strong></td>
<td>This function will scan the grammar file and populate the corresponding</td>
</tr>
<tr>
<td></td>
<td>Data Structures Keywords, Operators, Punctuation, Brackets</td>
</tr>
</tbody>
</table>

### Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

### Field Detail

**CtokenLogger**  
public java.util.logging.Logger **CtokenLogger**

**Constructor Detail**

**CTokenizer**  
public **CTokenizer()**  
Constructor

**CTokenizer(java.lang.String GrammarFile)**  
constructor with a grammar file specified in the parameter

### Method Detail
openGrammarFile

public int openGrammarFile()
    This function opens the grammar file and sets the appropriate flags
Specified by:
    openGrammarFile in interface Tokenizer
Returns:
    Returns 0 on success, -1 otherwise

scanGrammarFile

public int scanGrammarFile()
    This function will scan the grammar file and populate the corresponding Data
    Structures Keywords, Operators, Punctuation, Brackets
Specified by:
    scanGrammarFile in interface Tokenizer
Returns:
    Returns 0 on success Returns -1 if the grammar file had some errors in it.

openSourceCode

public int openSourceCode(java.lang.String sourcecodefilename)
    This function opens the source code and sets up the buffered reader...
Specified by:
    openSourceCode in interface Tokenizer
Parameters:
    sourcecodefilename - Filename of the source code
Returns:
    - it returns 0 if it is able to open the file specified
    -if unable to open the file it returns -1

getNextToken

public Token getNextToken()
    This function should return the next token from the source code it scans the
    source code and finds the next token from the code and returns it
Specified by:
    getNextToken in interface Tokenizer
Returns:
    Returns a token if eof not reached otherwise returns null
**LexicalAnalyzer**

**Class Token**

```java
public class Token extends java.lang.Object
```

Tokens are the basic constructs of a programming language. Tokens are the basic elements that helps in forming the programming Language. There could be various types of tokens. The ones that are included here are KEYWORDS, IDENTIFIERS, CONSTANTS, OPERATORS, PUNCTUATIONS, BRACKETS, and PREPROCESSOR.

### Nested Class Summary

<table>
<thead>
<tr>
<th>static class</th>
<th>Token.TokenType</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Describes the type of token.</td>
</tr>
</tbody>
</table>

### Constructor Summary

- **Token()**
  - Token constructor

- **Token(java.lang.String TokenName, Token.TokenType TokenType, int linenum)**
  - Token constructor

- **Token(Token t)**
  - Token duplicator constructor

### Method Summary

<table>
<thead>
<tr>
<th>type</th>
<th>method</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>getLineNumber()</td>
</tr>
<tr>
<td></td>
<td>Returns the line number where the token was found</td>
</tr>
<tr>
<td>java.lang.String</td>
<td>getName()</td>
</tr>
<tr>
<td></td>
<td>Returns the Name of this token.</td>
</tr>
<tr>
<td>Token.TokenType</td>
<td>getType()</td>
</tr>
<tr>
<td></td>
<td>Returns the type of token It could be KEYWORDS, IDENTIFIERS, CONSTANTS, OPERATORS, PUNCTUATIONS, BRACKETS, and PREPROCESSOR</td>
</tr>
</tbody>
</table>
Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait

Constructor Detail

Token
public Token()
    Token constructor

Token
token(Token t)
    Token duplicator constructor

Token
public Token(java.lang.String TokenName, Token.TokenType Tokentype, int linenum)
    Token constructor

Method Detail

getName
public java.lang.String getName()
    Returns the Name of this token..

getType
public Token.TokenType getType()
    Returns the type of token. It could be KEYWORDS, IDENTIFIERS, CONSTANTS, OPERATORS, PUNCTUATIONS, BRACKETS, PREPROC

getLineNumber
public int getLineNumber()
    Returns the line number where the token was found
public class CParser extends java.lang.Object

This is the parser for the C Language. It is no way a generic parser right now. This parser could be broken easily Go ahead and try it and YOU will know..... This parser is just going to look for specific type of code constructs and specific type of problems..... But it does a good job of what ever it does Its good in its own way.. But if you get in any source code that can be parsed through this , this is definitely going to find out a small set of vulnerabilities (if the source code has any)....... I am confident atleast to that point :)

Field Summary

| int | constructor_error |

Constructor Summary

CParser(java.lang.String sourcecodefile)

Method Summary

<table>
<thead>
<tr>
<th>void</th>
<th>parseSourceCode()</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>This function is going to parse the source code..</td>
</tr>
</tbody>
</table>

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

constructor_error

public int constructor_error

Constructor Detail
public CParse(java.lang.String sourcecodefile)

**Method Detail**

**parseSourceCode**

public void parseSourceCode()

This function is going to parse the source code. I.e it is going to ask the tokenizer to provide tokens one by one and it is going to construct the statement with the provided tokens.... After each statement this calls the Analyze statement function.

**ProgramElements**

**Class Block**

java.lang.Object

public class Block
extends java.lang.Object

Block represent a anything within a { ... } When ever a source code contains a "{" token then a new block should be created A block can have variables in side it.

**Constructor Summary**

<table>
<thead>
<tr>
<th>Constructor Summary</th>
<th>Method Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Block()</td>
<td>addVariable(Variable var) Adds a variable to the current block.</td>
</tr>
<tr>
<td>Block(int blocklevel)</td>
<td>getAllVariables() Returns a list of all the variables declared inside the current block.</td>
</tr>
<tr>
<td></td>
<td>getLevel() Setter for level member variable</td>
</tr>
<tr>
<td></td>
<td>variable getVariable(java.lang.String variablename) Gets a variable from the current block</td>
</tr>
<tr>
<td></td>
<td>noOfVariables()</td>
</tr>
</tbody>
</table>
Returns the total no of variables that was declared inside the block

<table>
<thead>
<tr>
<th>int removeVariable(java.lang.String variablename)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Removes a variable from this block</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>void setLevel(int level)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Setter for level member variable</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>int UpdateVariable(Variable v)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Updates a variable with new values found from in the input parameter</td>
</tr>
</tbody>
</table>

Methods inherited from class java.lang.Object

| clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait |

Constructor Detail

Block

<table>
<thead>
<tr>
<th>public Block()</th>
</tr>
</thead>
<tbody>
<tr>
<td>A constructor with no parameters</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>public Block(int blocklevel)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constructor with one parameter, an integer value</td>
</tr>
<tr>
<td>Parameters: blocklevel -- An integer value that specifies the level of this block</td>
</tr>
</tbody>
</table>

Method Detail

addVariable

<table>
<thead>
<tr>
<th>public int addVariable(Variable var)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adds a variable to the current block. It returns the new size of the variables list</td>
</tr>
</tbody>
</table>

getVariable

<table>
<thead>
<tr>
<th>public Variable getVariable(java.lang.String variablename)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gets a variable from the current block</td>
</tr>
<tr>
<td>Parameters:</td>
</tr>
</tbody>
</table>

variableName -- Name of the variable that needs to be retrieved

Returns:
- Returns a variable if a variable with specified variable name exists in this block,
  otherwise returns null

removeVariable
public int removeVariable(java.lang.String variableName)
  Removes a variable from this block
Parameters:
the - Name of the variable that needs to be removed from the current block
Returns:
Integer value that denotes success or failure of this removevariable call

UpdateVariable
public int UpdateVariable(Variable v)
  Updates a variable with new values found from in the input parameter
Parameters:
Specifies - the Variable that needs t

noOfVariables
public int noOfVariables()
  Returns the total no of variables that was declared inside the block

getAllVariables
public Variable[] getAllVariables()
  Returns a list of all the variables declared inside the current block
Returns:
Returns an array of Variable objects

getLevel
public int getLevel()
  Setter for level member variable
Returns:
- integer value that specifies the block level

setLevel
public void setLevel(int level)
  Setter for level member variable
Parameters:
level -- Integer that denotes the level of this current block

ProgramElements
Class Class
java.lang.Object

public class Class
extends java.lang.Object

class "Class" represents a class in a program. A class can contain variables, functions....

Constructor Summary
Class(java.lang.String classname)

Method Summary
void addFunction(Function f)
   Adds a function to the class...

void addVariable(Variable var)
   Adds a variable to the class..

boolean containsFunction(Function function)
   Tells whether specified function exits in the class

Function getFunction(java.lang.String functionname)
   Given a functionname this method Returns the corresponding function from a class if no function with specified name exists in the class then this returns null

Variable getVariable(java.lang.String variablename)
   Given a variable name this returns the variable object

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail
public Class(java.lang.String classname)
ProgramElements

Class Function

```java
java.lang.Object
   ProgramElements.Function
```

public class Function
extends java.lang.Object

This class represents a Function in program. Function has a name, type, parameters, and various other members in it.

### Constructor Summary

<table>
<thead>
<tr>
<th>Constructor</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Function</strong></td>
<td>(java.lang.String functionname, java.lang.String functiontype)</td>
</tr>
<tr>
<td><strong>Function</strong></td>
<td>(java.lang.String functionname, java.lang.String functiontype, int numparameters) Public construtor for this class</td>
</tr>
</tbody>
</table>

### Method Summary

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>addBlock</strong></td>
<td>(Block b) Adds a block to the current function</td>
</tr>
<tr>
<td><strong>addBlock</strong></td>
<td>(int level) Adds a block to the current function</td>
</tr>
<tr>
<td><strong>addVariable</strong></td>
<td>(Variable v, int blocklevel) This function adds a variable at the specified block level, to this function object.</td>
</tr>
<tr>
<td><strong>findInStack</strong></td>
<td>(Variable var, int offset) Finds a variable in stack.</td>
</tr>
<tr>
<td><strong>getAllVariables</strong></td>
<td>() This function returns an array of all the variables that are declared inside the current function</td>
</tr>
<tr>
<td><strong>getBlock</strong></td>
<td>(int blocklevel) This should return the block with the specified block level</td>
</tr>
<tr>
<td><strong>getName</strong></td>
<td>() getter Method for name member variable.</td>
</tr>
<tr>
<td><strong>getNoofparams</strong></td>
<td>() Getter for no of parameter member variable</td>
</tr>
<tr>
<td><strong>getParameters</strong></td>
<td>() Getter for parameters member variable</td>
</tr>
<tr>
<td><strong>getType</strong></td>
<td>()</td>
</tr>
<tr>
<td>Method</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Getter for the type member variable</td>
<td></td>
</tr>
<tr>
<td><strong>Variable getVariable</strong>(java.lang.String variablename)</td>
<td>This function will return the variable with corresponding variable name. This function will check each and every block in the reverse order.</td>
</tr>
<tr>
<td>void removeBlock(int blocklevel)</td>
<td></td>
</tr>
<tr>
<td>void setName(java.lang.String name)</td>
<td>Setter for name member variable.</td>
</tr>
<tr>
<td>void setNoofparams(int noofparams)</td>
<td>Setter for no of parameters</td>
</tr>
<tr>
<td>void setParameters(java.lang.String parameters)</td>
<td>Setter for parameters member function</td>
</tr>
<tr>
<td>void setType(java.lang.String type)</td>
<td>Setter for type member variable.</td>
</tr>
<tr>
<td>int UpdateVariable(Variable var)</td>
<td>Should update the variable with appropriate new values.</td>
</tr>
</tbody>
</table>

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait

### Constructor Detail

**Function**

public Function(java.lang.String functionname, java.lang.String functiontype, int numparameters)

Public constructor for this class

### Method Detail

**addBlock**

public void addBlock(int level)

Adds a block to the current function
addBlock
public void addBlock(Block b)
    Adds a block to the current function

getBlock
public Block getBlock(int blocklevel)
    This should return the block with the specified block level

removeBlock
public void removeBlock(int blocklevel)

getVariable
public Variable getVariable(java.lang.String variablename)
    This function will return the variable with corresponding variable name. This function will check each and every block in the reverse order. i.e. if there are two variables with the same name, it will always return the instance that was declared more recently.
Parameters:
    variable - name - Name of the variable that you are looking for...
Returns:
    returns the Variable if found else returns null

addVariable
public int addVariable(Variable v, int blocklevel)
    This function adds a variable at the specified block level, to this function object.
Parameters:
    Variable - v - The variable that needs to be added to the function
    int blocklevel - The block to which the variable should be added
Returns:
    integer value that specifies if it was able to or not able to add the variable to this function
    return value of -1 denotes that add resulted in a failure otherwise success.

UpdateVariable
public int UpdateVariable(Variable var)
    Should update the variable with appropriate new values
**getName**

```java
public java.lang.String getName()
getter Method for name member variable.

**Returns:**
returns a string that denotes the name of this variable
```

**setName**

```java
public void setName(java.lang.String name)
Setter for name member variable.

**Parameters:**
- - The name of this variable
```

**getAllVariables**

```java
public Variable[] getAllVariables()
This function returns an array of all the variables that are declared inside the current function

**Returns:**
- Returns a Variable[], an array of variable
```

**findInStack**

```java
public Variable findInStack(Variable var, int offset)
Finds a variable in stack .... Useful in cases where a an operand in a program is specified with its relative address.. For example assume a function has three variables int a,b,c... A is pushed onto the stack, then B and then C, So is on top now.. Now variable a can be represented by a , or b+4 or c+8 ... All of them refers to a. So in order to know what exactly is b+4 the program needs to take a look at function's stack and need to see the variable that is offsetted by 4 bytes from b. So this function will help in dereferencing those types of stuff.......

**Parameters:**
var - - Variable that acts as the starting point ( example if a is referred to as b+4 then b is the variable and 4 is the offset
offset - - Integer value that specifies the offset value. It may be negative or positive

**Returns:**
- Returns a variable that was reffered by expression var+offset ... if there is a variable at specified offset then it returns that variable , otherwise it returns NULL
```

**getType**

```java
public java.lang.String getType()
```
Getter for the type member variable  
**Returns:**  
- Returns the data type of this function as a string

---

**setType**  
public void **setType**(java.lang.String type)  
  
  Setter for type member variable  
  **Parameters:**  
  type - - String that specifies the data type of the Function

---

**getNoofparams**  
public int **getNoofparams**()  
  
  Getter for no of parameter member variable  
  **Returns:**  
  Returns the No of parameters in the function prototype

---

**setNoofparams**  
public void **setNoofparams**(int noofparams)  
  
  Setter for no of parameters  
  **Parameters:**  
  noofparams - - integer value that specifies the no of parameters in function prototype

---

**getParameters**  
public java.lang.String **getParameters**()  
  
  Getter for parameters member variable  
  **Returns:**  
  A string that specifies the parameters of the function

---

**setParameters**  
public void **setParameters**(java.lang.String parameters)  
  
  setter for parameters member function  
  **Parameters:**  
  parameters - - A string that represent the parameters of this function. It could be int a , int b or something similar

---

**ProgramElements**
**Class Statement**

```java
class Statement extends java.lang.Object
```

Statements are the basic building blocks of a program. A program is made up of multiple statements And each statement is made up of multiple tokens.

## Constructor Summary

**Statement()**

Constructor

## Method Summary

**void addToken(Token t)**

Adds token to the end of the statement

**boolean contains(java.lang.String value)**

Check if the statement currents a token with specified name

**Token getLastToken()**

returns a token from the end of the statement

**Token getNextToken()**

returns the next token in the statement

**int getNoOfTokens()**

returns the total no of tokens in the statement

**boolean isAssignmentStatement()**

of course this will not detect a assignment statement if it has a conditional statement inside it.....

**boolean isComparisonStatement()**

This function analyzes the current statement and returns a boolean value that specifies if this statement is an assignment statement or not.

**int occurrences(java.lang.String value)**

returns the no of occurrences of a particular tokenname in a statement

**void printStatement()**

Prints the current statement

**void pushToken(Token t)**

Adds the token to the beginning of the tokens list now when u call statement.getNextToken this will be sent out first
Methods inherited from class java.lang.Object
- clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

public Statement()

Method Detail

addToken
public void addToken(Token t)

   Adds token to the end of the statement

   Parameters:
   Token - that needs to be inserted into the statement

getNextToken
public Token getNextToken()

   returns the next token in the statement

   Returns:
   The next available token in the statement

getNoOfTokens
public int getNoOfTokens()

   returns the total no of tokens in the statement

pushToken
public void pushToken(Token t)

   Adds the token to the beginning of the tokens list now when u call
   statement.getNextToken this will be sent out first

contains
public boolean contains(java.lang.String value)

   Check if the statement currents a token with specified name
occurences
public int occurences(java.lang.String value)
    returns the no of occurences of a particular tokenname in a statement*

Parameters:
Token - name that needs to be found in the statement

Returns:
The no of occurences of the token

isAssignmentStatement
public boolean isAssignmentStatement()
    ofcourse this will not detect a assignment statement if it has a conditional
statement inside it..... for ex this will detect a = b or a= b=c as assignment
statements but it will not detect the following statement as assignment statement a
= (a<=b)?0:1;

isComparisonStatement
public boolean isComparisonStatement()
    This function analyzes the current statement and returns a boolean value that
specifies if this statement is an assignment statement or not.

Returns:
- Boolean value that true if its an assignment statement , otherwise false

getLastToken
public Token getLastToken()
    returns a token from the end of the statement

printStatement
public void printStatement()
    Prints the current statement

ProgramElements
Class Variable
java.lang.Object
    ProgramElements.Variable

public class Variable
    extends java.lang.Object
This class represents a variable object.....

### Constructor Summary

<table>
<thead>
<tr>
<th>Constructor</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Variable(java.lang.String varname, java.lang.String vartype, boolean isUnsigned)</code></td>
</tr>
<tr>
<td><code>Variable(java.lang.String varname, java.lang.String vartype, boolean isUnsigned, int pointerDepth)</code></td>
</tr>
</tbody>
</table>

### Method Summary

<table>
<thead>
<tr>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>void duplicateVariable(Variable sourceVariable)</code></td>
</tr>
<tr>
<td><code>int getArraySize()</code></td>
</tr>
<tr>
<td><code>int getFlags()</code></td>
</tr>
<tr>
<td><code>int getLength()</code></td>
</tr>
<tr>
<td><code>int getMallocSize()</code></td>
</tr>
<tr>
<td><code>java.lang.String getName()</code></td>
</tr>
<tr>
<td><code>int getPointerDepth()</code></td>
</tr>
<tr>
<td><code>java.lang.String getPointsto()</code></td>
</tr>
<tr>
<td><code>int getSize()</code></td>
</tr>
<tr>
<td><code>java.lang.String getType()</code></td>
</tr>
<tr>
<td><code>java.lang.String getValue()</code></td>
</tr>
<tr>
<td><code>boolean isArray()</code></td>
</tr>
<tr>
<td><code>boolean isHas_alias()</code></td>
</tr>
</tbody>
</table>
This function returns a boolean value that specifies if the variable has an alias...

<table>
<thead>
<tr>
<th>boolean isInitialized()</th>
<th>Returns if the variable is initialized or not</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean isNullChecked()</td>
<td>Returns if the variable is Nullchecked after malloc or not...</td>
</tr>
<tr>
<td>boolean isUnsigned()</td>
<td>Getter method of unsigned member variable</td>
</tr>
<tr>
<td>void setArray(boolean isarray)</td>
<td>Sets the array boolean flag</td>
</tr>
<tr>
<td>void setArraySize(int Size)</td>
<td>Sets the array size of the variable</td>
</tr>
<tr>
<td>void setFlags(int Flag)</td>
<td>Sets the flag value of this variable</td>
</tr>
<tr>
<td>void setHas_alias(boolean has_alias)</td>
<td>Setter for has_alias member variable</td>
</tr>
<tr>
<td>void setInitialized()</td>
<td>Sets the initialized flag to true</td>
</tr>
<tr>
<td>void setLength(int length)</td>
<td></td>
</tr>
<tr>
<td>void setMallocSize(int msize)</td>
<td>Sets the Malloc size</td>
</tr>
<tr>
<td>void setName(java.lang.String Name)</td>
<td>Sets the name of this variables</td>
</tr>
<tr>
<td>void setNullCheck(boolean nullCheck)</td>
<td>Sets the null check flag to true</td>
</tr>
<tr>
<td>void setPointerDepth(int pointerDepth)</td>
<td>Sets the pointer depth of the variable</td>
</tr>
<tr>
<td>void setPointsto(java.lang.String pointsto)</td>
<td>Function that sets the value for points to member variable of this class</td>
</tr>
<tr>
<td>void setSize(int Size)</td>
<td>Sets the size of the variable..</td>
</tr>
<tr>
<td>void setType(java.lang.String Type)</td>
<td>Sets the data type of this variable.</td>
</tr>
<tr>
<td>void setUnsigned(boolean unsigned)</td>
<td>Setter for unsigned member variable</td>
</tr>
<tr>
<td>void setValue(java.lang.String Value)</td>
<td>Sets the value of the variable</td>
</tr>
</tbody>
</table>
void toggleNullCheck()
    toggles null check

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

Variable
public Variable(java.lang.String varname,
                java.lang.String vartype,
                boolean isUnsigned)
    constructor

Variable
public Variable(java.lang.String varname,
                java.lang.String vartype,
                boolean isUnsigned,
                int pointerDepth)
    constructor

Method Detail

getName
public java.lang.String getName()
    Returns the Name of the Variable
    Returns:
    - String Name - Name of this Variable

setName
public void setName(java.lang.String Name)
    Sets the name of this variables
    Parameters:
    Name - = Name that needs to assigned to this variable

getType
public java.lang.String getType()
    Returns the type of this variable, May be int, float, short, long, double, char etc.,
Returns:
Type - Specifies the data type of this variable

**setType**
public void **setType**(java.lang.String Type)
Sets the data type of this variable.
**Parameters:**
Type -- The data type of the variable

**getValue**
public java.lang.String **getValue**()
returns the value of the variable
**Returns:**
String representing the value of the variable

**setValue**
public void **setValue**(java.lang.String Value)
Sets the value of the variable
**Parameters:**
Value -- Value that needs to be assigned to the variable

**isInitialized**
public boolean **isInitialized**()
Returns if the variable is initialized or not
**Returns:**
boolean value that is true if the variable is initialized, false otherwise

**setInitialized**
public void **setInitialized**()
Sets the initialized flag to true

**isNullChecked**
public boolean **isNullChecked**()
Returns if the variable is Nullchecked after malloc or not...
**Returns:**
boolean value that is true if the variable is null checked, false otherwise
**setNullCheck**

public void **setNullCheck**(boolean nullCheck)

Sets the null check flag to true

---

**toggleNullCheck**

public void **toggleNullCheck**()

toggles null check

---

**setPointerDepth**

public void **setPointerDepth**(int pointerDepth)

Sets the pointer depth of the variable

**Parameters:**

- int - pointerDepth - value that denotes the pointer depth of the variable if the variable is int *a; then pointerdepth = 1 , if variable is int **a; then pointerdepth should be 2.

---

**getPointerDepth**

public int **getPointerDepth**()

Returns the pointer depth of the variable

**Returns:**

integer value that denotes the pointer depth of the variable

---

**getSize**

public int **getSize**()

Returns the size of the variable

**Returns:**

Returns an integer value that denotes the size of the variable rather the size of the data type of the variable

---

**setSize**

public void **setSize**(int size)

Sets the size of the variable.. Actually variable's size is set in the constructor automatically based on the data type of the variable. But this function is provide in case the size has to be modified manually for some reason

**Parameters:**

- size - of the variable
**getArraySize**

public int getArraySize()

    Returns the array size

**Returns:**

if the variable is an array it returns the size of the array otherwise returns 0

---

**setArraySize**

public void setArraySize(int Size)

    Sets the array size of the variable

**Parameters:**

Integer - value that denotes the size of the array. if its int a[10] then 10 is the size of the array

---

**setFlags**

public void setFlags(int Flag)

    Sets the flag value of this variable

**Parameters:**

Integer - value that denotes the flag value of this variable. As of now i am using only flag value = 1 which is TAINTED FLAG and it is set if the variable receives input from user

---

**getFlags**

public int getFlags()

    Returns the flag value of this variable

**Returns:**

Returns an integer value that denotes the flag value of this variable

---

**duplicateVariable**

public void duplicateVariable(Variable sourceVariable)

    copies the values and other properties of one variable in to the other variable

**Parameters:**

Variable - from this the value and other properties has to be copied in to the current(this) variable

---

**getMallocSize**

public int getMallocSize()

    Returns the malloc size
**setMallocSize**
public void setMallocSize(int msize)
   Sets the Malloc size

**isArray**
public boolean isArray()
   returns boolean value that says if this variable is an array

**setArray**
public void setArray(boolean isarray)
   Sets the array boolean flag

**getLength**
public int getLength()

**setLength**
public void setLength(int length)

**getPointsto**
public java.lang.String getPointsto()

**setPointsto**
public void setPointsto(java.lang.String pointsto)
   Function that sets the value for points to member variable of this class

**isHas_alias**
public boolean isHas_alias()
   This function returns a boolean value that specifies if the variable has an alias...
   (Only for variables with pointer depth >0 or rather Pointers)

**setHas_alias**
public void setHas_alias(boolean has_alias)
   Setter for has_alias member variable
**isUnsigned**

public boolean isUnsigned()

Getter method of unsigned member variable

**Returns:**

True if the variable is an unsigned variable, otherwise false.

---

**setUnsigned**

public void setUnsigned(boolean unsigned)

Setter for unsigned member variable

**Parameters:**

unsigned -- boolean value that specifies if the variable is unsigned or not