

**T.R.**  
**GEBZE TECHNICAL UNIVERSITY**  
**FACULTY OF ENGINEERING**  
**DEPARTMENT OF COMPUTER ENGINEERING**

**STATIC OVERFLOW ANALYSIS AND TESTS**

**TUGAY APAYDIN**

**SUPERVISOR**  
**PROF. İBRAHİM SOĞUKPINAR**

**GEBZE**  
**2022**

**T.R.**  
**GEBZE TECHNICAL UNIVERSITY**  
**FACULTY OF ENGINEERING**  
**COMPUTER ENGINEERING DEPARTMENT**

**STATIC OVERFLOW ANALYSIS AND  
TESTS**

**TUGAY APAYDIN**

**SUPERVISOR**  
**PROF. İBRAHİM SOĞUKPINAR**

**2022**  
**GEBZE**

 <p><b>GEBZE</b> TECHNICAL UNIVERSITY</p>	<p>GRADUATION PROJECT JURY APPROVAL FORM</p>
--	--

This study has been accepted as an Undergraduate Graduation Project in the Department of Computer Engineering on 31/08/2021 by the following jury.

**JURY**

Member

(Supervisor) : Prof. İbrahim SOĞUKPINAR

Member : Prof. Erchan Aptoula

Member : M.Sc Melike İlteralp

# ÖZET

This study presents a static analysis tool for C language that can detect some buffer overflows that are generally called integer and stack overflows. Buffer overflows occur when a program tries to access to outside of an allocated memory area. In most cases, this causes the program to crash. In a worse scenario, a buffer overflow can provide the access an attacker needs to gain remote code execution. To provide secure code against buffer overflows, programmers need methods for analyzing source code of a program to detect buffer overflow vulnerabilities. One way to implement such a method is to perform static analysis on the program. This involves looking at the source code to find the errors in the program. This study uses syntax analysis and then value flow analysis to find overflows in a source code of a program.

# CONTENTS

<b>Özet</b>	<b>iv</b>
<b>Contents</b>	<b>v</b>
<b>List of Figures</b>	<b>vi</b>
<b>List of Tables</b>	<b>vii</b>
<b>1 INTRODUCTION</b>	<b>1</b>
1.1 PROBLEM DEFINITION . . . . .	1
1.2 MOTIVATION . . . . .	2
<b>2 RELATED STUDY</b>	<b>3</b>
2.1 STATIC ANALYSIS APPROACH . . . . .	3
2.2 EXISTING BUFFER OVERFLOW DETECTION METHODS . . . . .	5
2.2.1 ARCHER (Array Checker) . . . . .	5
2.2.2 BOON (Buffer Overrun Detection) . . . . .	5
2.2.3 UNO (Uninitialized Variables, Nil-Pointer Dereferencing, And Out of Bound Array Indexing) . . . . .	6
2.2.4 Value Set Analysis . . . . .	6
<b>3 PROJECT DESIGN</b>	<b>7</b>
3.1 INTRODUCTION . . . . .	7
3.2 STATIC ANALYSIS ENGINE . . . . .	7
3.3 DIAGNOSIS ENGINE . . . . .	8
3.3.0.1 Integer Overflow Analysis . . . . .	8
3.3.0.2 Stack and Heap Overflow Analysis . . . . .	9
3.4 REQUIREMENTS . . . . .	11
<b>4 IMPLEMENTATION AND TESTS</b>	<b>12</b>
<b>5 CONCLUSION</b>	<b>14</b>
<b>Bibliography</b>	<b>15</b>

# LIST OF FIGURES

1.1	The Catalogued Vulnerabilities Data from CERT Coordination Center from 1995 to 2007 [1]. . . . .	1
2.1	Generalized view of the process of a static analysis . . . . .	4
3.1	System Architecture . . . . .	7
3.2	System Architecture . . . . .	8
3.3	Integer Overflow Conditions . . . . .	9
3.4	The sample unsafe standard C library functions [1] . . . . .	9
3.5	Class Diagram . . . . .	10
4.1	The test file . . . . .	12
4.2	An example result list from the GUI . . . . .	13
4.3	CppCheck Results . . . . .	13

# **LIST OF TABLES**

# 1. INTRODUCTION

## 1.1. PROBLEM DEFINITION

Buffer overflows are common vulnerabilities in software. In the languages that offer direct, low-level access to read and write memory such as C and C++, buffer overflow vulnerabilities deal with buffers or memory allocations. In the best cases, this type of access causes the program to crash. In the worst case, a buffer overflow can be exploited to hijack control of a program.

The attacks caused by buffer overflow continue to be the major computer security threats. As a traditional exploit, buffer overflow allows attackers to inject malicious codes in the application at run-time. Those injected codes can help attacker to gain the access privilege of the host machine maliciously.

Even though buffer overflow has been intensively studied and researchers have proposed various mitigation techniques, it is still the most critical security threat these days. According to statistics from CERT Coordination Center, the number of identified and catalogued vulnerabilities has increased over 213% in years between 2004-2006 as shown in Figure 1.1.

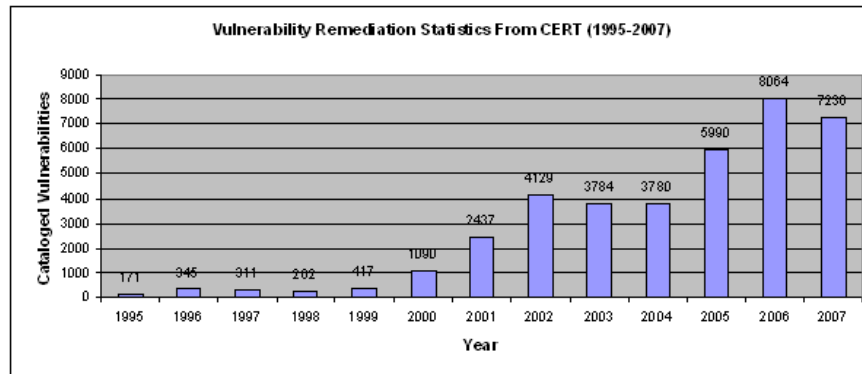


Figure 1.1: The Catalogued Vulnerabilities Data from CERT Coordination Center from 1995 to 2007 [1].

Due to its importance, researchers have proposed various approaches to defend against buffer overflow attacks. However, each of them has its own limitations and cannot be applied universally to prevent overflow attacks.

In this project, it is aimed to detect and test buffer overflow vulnerabilities in a source code using static analysis techniques. [2]



## 1.2. MOTIVATION

Being able to detect overflows would improve software assurance. Static analysis can be used as a baseline for determining the susceptibility of a program to cause buffer overflows.

When a new buffer overflow vulnerability is first exploited, a primary challenge is to diagnose the vulnerability. The diagnosis results include which buffer was overflowed; under what conditions the buffer will be overflowed.

This research investigates the viability of a buffer overflow detection method to statically analyse source files for buffer overflows. The research focuses on the discovery of stack-based, heap-based and integer overflows that occur in C programs.

## 2. RELATED STUDY

The aim of software security is that software continues to function correctly even while under attack. Memory errors do not always expose a way for an attacker to exploit them but even in these cases an attacker could potentially exploit these problems in order to perform a very effective denial of service attack. However, when memory errors in programs exposed to the internet are exploitable, they can lead to the attackers gaining control over an entire system . So from a safety viewpoint it is also very important to make sure these types of errors are found and fixed. Or mitigated some other way, for example with dynamic checks around memory access. This means that not using static analysis may make software much more vulnerable to attacks, than if a static analysis tool was used that can detect various issues.

C and languages similar to it employ manual memory management. These types of languages are not memory safe languages as they require the programmer to explicitly request and free memory when needed instead of the language taking care of it automatically. C and C++ are examples of these kind of, memory unsafe languages.

The most common issues, in not memory safe languages, are array bounds checking problems (buffer overflow and underflow), not releasing memory (memory leak), releasing memory too soon (also called dangling pointers), and problems with using uninitialized data.

This study aims to create a static analysis tool for detecting buffer overflow vulnerabilities.

### 2.1. STATIC ANALYSIS APPROACH

Sophisticated techniques couple static code analysis with formal methods. Formal methods apply theoretical computer science fundamentals to solve difficult problems in software, such as proving that the software will not fail with a run-time error. The combination of static code analysis and formal methods enables developers to detect difficult to find errors and prove the absence of certain types of bugs/errors. E.g. these techniques can prove that the following line of code will never fail with a divide by zero run-time error. Most of today's tools for static analysis function in basically the same way. They receive code as input, build a model that represents the program, perform some sort of analysis in combination with knowledge about what to search for and finally present the result to the user. Figure 2.1 shows the process that all static analysis tools that target security make use of.

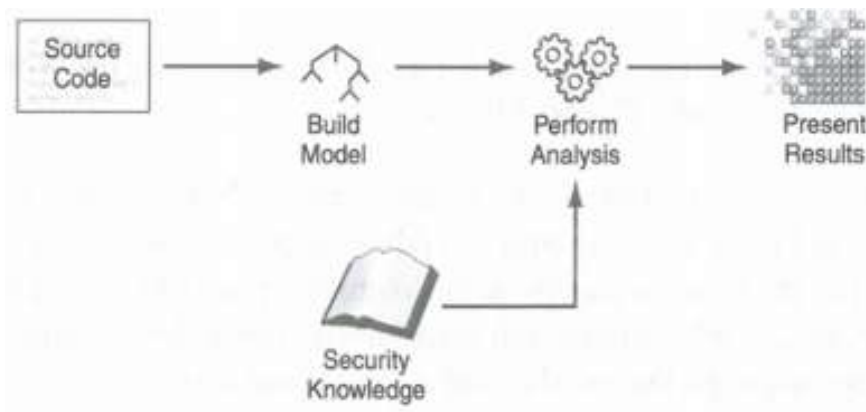


Figure 2.1: Generalized view of the process of a static analysis

The first step taken by the tools when performing an analysis is to build a model of the program. In this way an abstract representation of the program is created, which is better suited to be used for the actual analysis and what kind of model that is created depends largely on what kind of analysis that is to be performed. The simplest model, used in a lexical analysis, is a stream of tokens. This is created by taking the source code first. Then discarding all unimportant features such as white spaces and comments and finally translates all parts of the code into tokens.

This stream can then be used to perform a simple lexical analysis on. The next step is to translate the stream of tokens into a parse tree by using a parser that matches the token stream against a set of production rules. The parse tree contains the most direct representation of the code just as the programmer wrote it and can thus be used as a good base to perform analysis.

However, since the production rules might introduce various symbols in order to make the parsing easy, a parse tree is not the best representation to perform more complex analyzes on. The next step is instead to translate the parse tree into an AST as well as creating a symbol table alongside it. These can then be used as input to a semantic analysis of the program. Another thing that the AST might be used for is to scan it by using predefined patterns in order to find errors that require data-flow analysis of the code. This is not possible to perform on for example a token stream. The procedure of performing a static analysis is until now very alike the procedure taken by a compiler when compiling a program.

But as a compiler as a next step will use the AST to generate an intermediate representation of the code, which can be used for optimization and later on translation to platform specific code, the processes now separates. Most of the tools instead continue by building a control flow graph, which can be used to inspect the various paths in the program or to track data flow, on top of the AST. The way the graph is used by a tool

depends largely on which techniques the tool makes use of and this has a big impact on efficiency, speed and result of a scan.

Another thing that influences the result of a scan is how the tool makes certain simplifications of the program. The very essence of a static analyzer is that the program being analyzed is not run. However, in order to perform a through scan the tool, as already described, translates the code into an abstract representation and then “executes” the program on an abstract machine with a set of non-standard values replacing the standard ones. This is where a concept referred to as states is introduced. States are a collection of program variables and the association of values to those variables.

For every program statement the state of a variable may change in some way and the aim for a static analysis is to associate the set of all possible states with all program points.

The use of states and how a tool makes the approximations and the simplified descriptions of sets is something that makes the tools differs greatly from one and another. Some make very sophisticated and accurate decisions in order not to make unjustified assumptions about the result of an operation whilst others resort to a more simple approach. This incremental approach goes against what almost everyone in the industry tries to do, especially with security. As the code is being developed, it needs to be brought under the security policy, quality policy, whatever in every single day. If they keep on top of it, it hardly impacts the team’s workflow at all. In fact, it actually saves them time when you factor in the benefits of the error prevention it delivers. But if they allow it to lapse, catching up becomes too overwhelming. [3]

## **2.2. EXISTING BUFFER OVERFLOW DETECTION METHODS**

### **2.2.1. ARCHER (Array Checker)**

ARCHER is a constraint solver tool that checks the bounds of the variables and memory sizes of various objects. ARCHER statically analyzes the source code for memory constraint violations such as array accesses, pointer dereferences, or calls to a function that expects a size parameter.

### **2.2.2. BOON (Buffer Overrun Detection)**

BOON was developed to detect buffer overflows using integer range analysis. This is achieved by parsing through the program looking for string variables. When found,

string variables are assigned two integers. The first integer is the string's allocated size, and the second is the number of bytes currently in use. Then, the algorithm checks the usage of each string to determine if the length of the string is greater than the size allocated for the string.

### **2.2.3. UNO (Uninitialized Variables, Nil-Pointer Dereferencing, And Out of Bound Array Indexing)**

In other buffer overflow detection techniques, false positives are common. UNO is a tool that aims to reduce the number of false positive reported and to allow users to define their own application-dependent properties for checking for flaws. As its name indicates, the tool checks for errors such as uninitialized variables, nil-pointer dereferencing, and out of bound array indexing. UNO is an extension of a tool called ctree, which produces a parse tree of a program. UNO runs the ctree program to get a parse tree, then converts the tree into a control flow graph. Once the graph is created, UNO performs its buffer overflow analysis on the control flow graph. Overall, UNO requires two passes through the code. The first pass is to build the tree and graph, and then check for errors.

### **2.2.4. Value Set Analysis**

Value set analysis is the technique of recovering “information about the contents of machine registers and memory locations at every program point in an executable”. This technique was used by Kindermann to perform buffer overflow detection via the static analysis <sup>7</sup> of executables. This buffer overflow detection method focused on the detection of buffer overflows caused by loops. [4].

## 3. PROJECT DESIGN

### 3.1. INTRODUCTION

Many different buffer overflow attacks use different strategies and target different pieces of code. Below are the best-known buffer overflow vulnerabilities:

Types of buffer overflows:

- Stack Overflow
- Heap Overflow
- Integer Overflow
- Unicode Overflow

The design of this program will include static analysis of Integer and Stack based overflows.

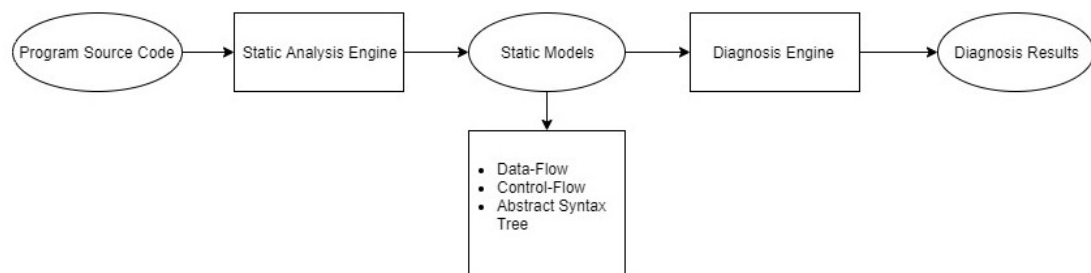


Figure 3.1: System Architecture

The static analysis engine builds models by parsing the source code of the computer software and creates static models. The models consist of abstract syntax trees.

The diagnosis engine analyzes the models created by the static analysis engine. By using the analysis results, it creates results of potential overflows.

### 3.2. STATIC ANALYSIS ENGINE

The static analysis engine firstly parses the source code into tokens. After that, it creates an abstract syntax tree by analysing statements and expressions. In this project, statements and expressions are limited with some types such as if statements, assign statements and while loops. Also data types are limited with primitive int and char

types. Also stack arrays and pointers can be parsed by static analysis engine.

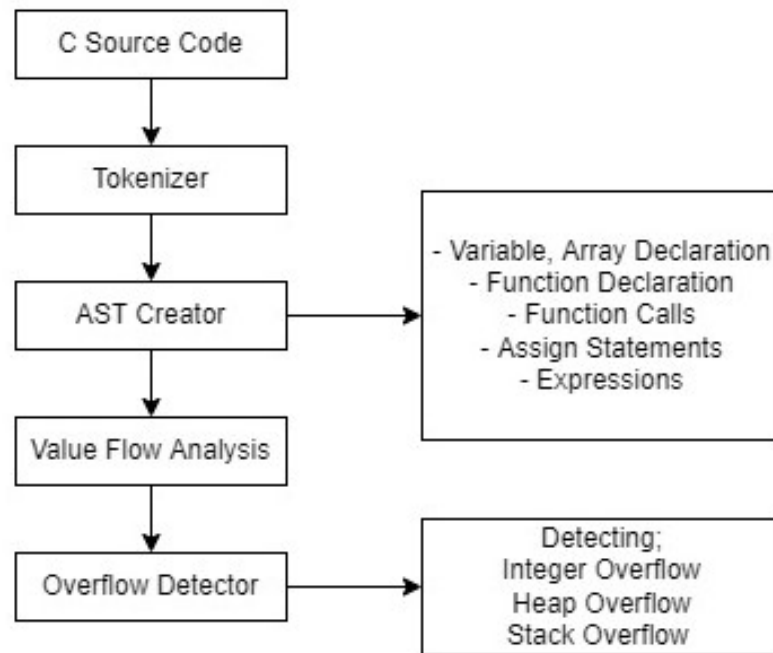


Figure 3.2: System Architecture

### 3.3. DIAGNOSIS ENGINE

The diagnosis engine starts with value flow analysis. The statements inside the abstract syntax tree are executed at first.

While this execution is in progress, the overflow check occurs. For example, the expressions are tested before the execution if any overflow may occur. Also, array bound check and other type of buffer overflows are checked in this step.

#### 3.3.0.1. Integer Overflow Analysis

In the design of the project, each expression are designed to return a result. And while calculating the result of the statements, overflow conditions may occur. That's why, at each result calculation, integer overflow analysis is done.

The integer overflow in a program that is written with C language results in undefined behaviour. That's why, the overflow analysis is done before the statement executions or before calculating expression results.

Integer operation	Sign	Overflow condition
$x + y$	$\langle S, S \rangle$	$x + y \notin [-2^{N-1}, -2^{N-1}]$
	$\langle U, U \rangle$	$x + y \notin [0, -2^{N-1}]$
$x - y$	$\langle S, S \rangle$	$x - y \notin [-2^{N-1}, -2^{N-1}]$
	$\langle U, U \rangle$	$x - y \notin [0, -2^{N-1}]$
$x \times y$	$\langle S, S \rangle$	$x \times y \notin [-2^{N-1}, -2^{N-1}]$
	$\langle U, U \rangle$	$x \times y \notin [0, -2^{N-1}]$
$x/y$	$\langle S, S \rangle$	$(y = 0) \vee (x = -2^{N-1} \wedge y = -1)$
	$\langle U, U \rangle$	$y = 0$
$x \ll y, x \gg y$	$\langle S, S \rangle$	$y \notin [0, N - 1]$
	$\langle U, U \rangle$	

Figure 3.3: Integer Overflow Conditions

The figure 2.3 shows the overflow condition for integer operations. [5]

### 3.3.0.2. Stack and Heap Overflow Analysis

Stack and heap overflows are caused by accessing outside the bounds of the allocated area. That's why, the bound checking is the first criteria for stack and heap overflow analysis.

There are many unsafe Standard C Library and system data copy functions that can cause buffer overflow vulnerabilities. By using static analysis, it is not possible to detect overflows without analysing the source code.

Func. Name	Dest. Buffer	Loop_Bound	Unsafe Condition
strcpy(s1, s2)	s1	strlen(s2)	sizeof(s1) < strlen(s2)
strncpy(s1, s2, n)	s1	n	sizeof(s1) < n
memcpy(s1, s2, n)	s1	n	sizeof(s1) < n
memset(s1, s2, n)	s1	n	sizeof(s1) < n
recv(socket, s1, n, flags)	s1	n	sizeof(s1) < n

Figure 3.4: The sample unsafe standard C library functions [1]

In the figure 2.4, some sample of vulnerable standard C library functions are listed. The static analysis tools must be aware of these type of functions and analyse



their parameters to detect buffer overflows.

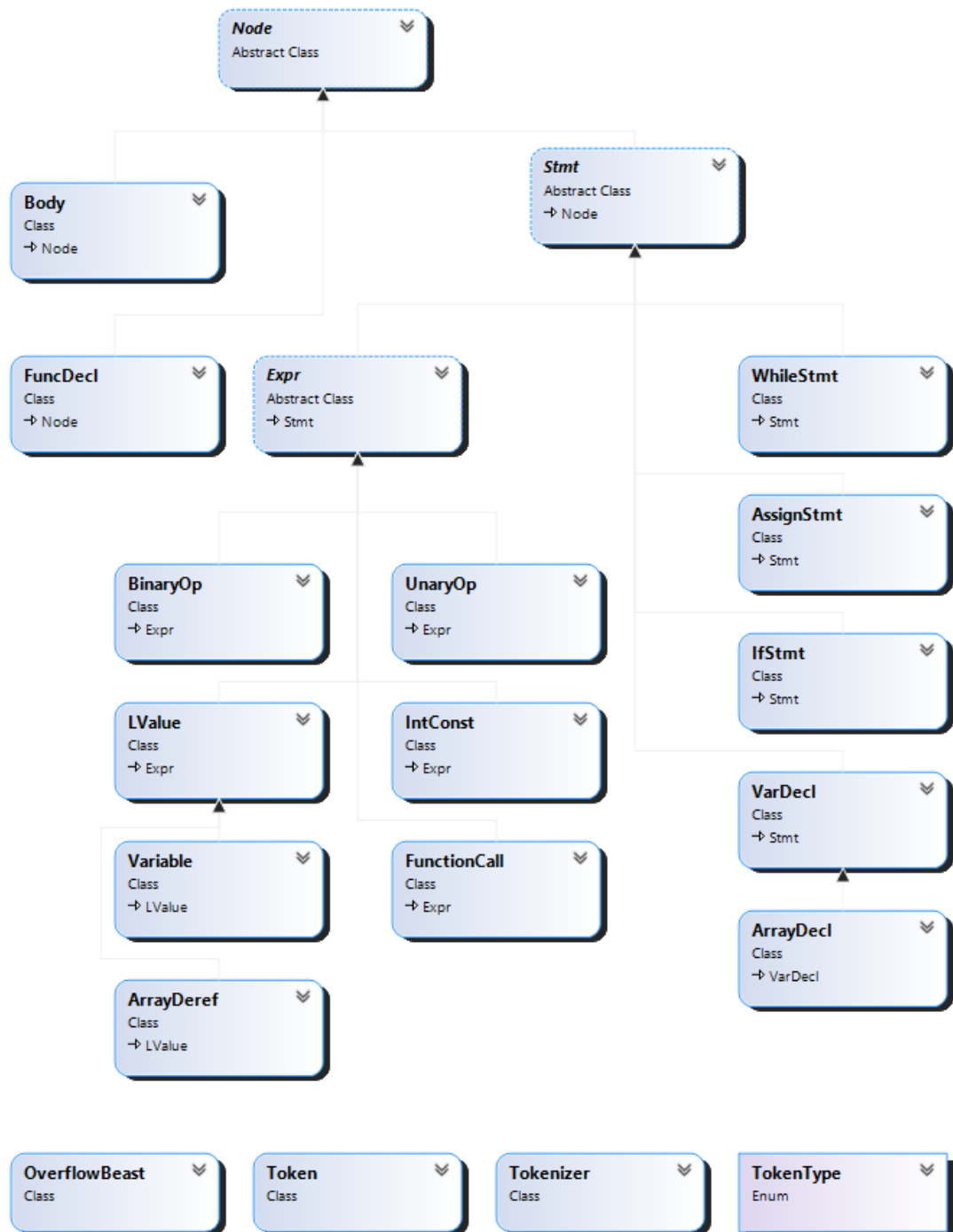


Figure 3.5: Class Diagram

### **3.4. REQUIREMENTS**

The project requirements are listed below:

- The program shall be able to detect integer, stack and heap based overflow vulnerabilities.
- The program shall be able to detect overflows caused by vulnerable C library functions.
- The program will list errors in a Graphical User Interface.
- C Language and libraries.
- Static analysis tools such as CppCheck for tests and comparisons.
- Microsoft Visual Studio IDE for creating GUI and developing the program.

## 4. IMPLEMENTATION AND TESTS

First of all, in this project only several syntax and statement models are implemented in the given time. That's why, the tests and experiments are limited.

In the test file, there are some integer overflows, an array bound overflow (which occurs when trying to access outside of an allocated area) and a stack-heap based overflow which occurs in a vulnerable C library function `strcpy()`. Figure 4.1 shows the test code.

```
int a = 5;
int g;
int b = -2147483648;
int z = -1;

int c = a - b;
int e = b * z;
int d = a + g;

if (a + 5)
{
    c = 6;
}

z = 5;

int arr[4] = { 5, 8, 3, 124 };

arr[5] = 6;
arr[2] = 8;

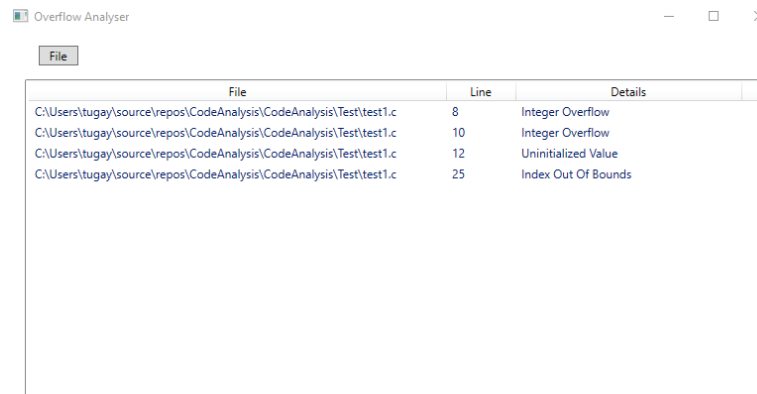
char str1[8] = "hello";
char str2[30] = " world from the mars";
char str3[5] = "world";

strcpy(str1, str3);
strcpy(str1, str2);
```

Figure 4.1: The test file

In figure 4.2, the overflow analysis of our program showed in graphical user interface.

In figure 4.2, the overflow analysis results of our program are showed in a user interface.

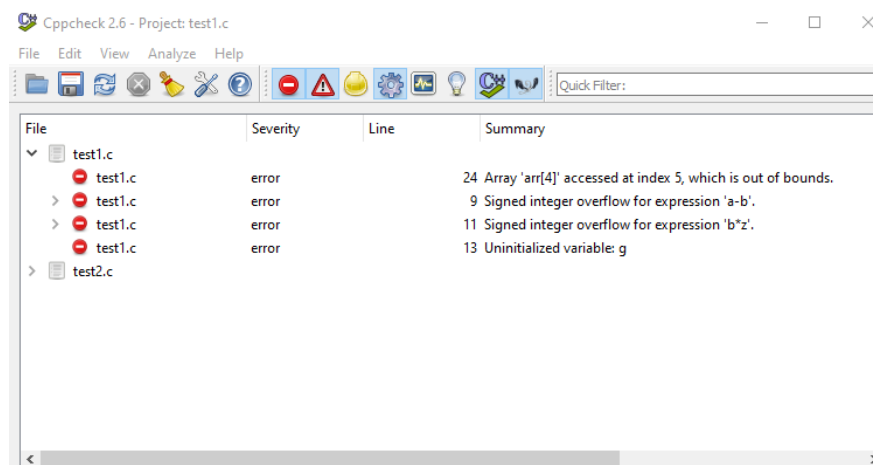


The screenshot shows a window titled "Overflow Analyser" with a "File" button and a table of results. The table has three columns: "File", "Line", and "Details".

File	Line	Details
C:\Users\tugay\source\repos\CodeAnalysis\CodeAnalysis\Test\test1.c	8	Integer Overflow
C:\Users\tugay\source\repos\CodeAnalysis\CodeAnalysis\Test\test1.c	10	Integer Overflow
C:\Users\tugay\source\repos\CodeAnalysis\CodeAnalysis\Test\test1.c	12	Uninitialized Value
C:\Users\tugay\source\repos\CodeAnalysis\CodeAnalysis\Test\test1.c	25	Index Out Of Bounds

Figure 4.2: An example result list from the GUI

In figure 4.3 the same test file is analysed with CppCheck program. It can be seen that the only difference is the overflow that occurs in strcpy() function.



The screenshot shows the Cppcheck 2.6 GUI with a project named "test1.c". The main window displays a list of errors with columns for "File", "Severity", "Line", and "Summary".

File	Severity	Line	Summary
test1.c	error	24	Array 'arr[4]' accessed at index 5, which is out of bounds.
test1.c	error	9	Signed integer overflow for expression 'a-b'.
test1.c	error	11	Signed integer overflow for expression 'b*z'.
test1.c	error	13	Uninitialized variable: g
test2.c			

Figure 4.3: CppCheck Results

There are many vulnerable C library functions that can cause overflows. But, static analysis can only be done on source code. That's why, static analysis has some disadvantages. In this condition, I manually defined vulnerable C library functions. That's why, my program was able to detect overflow caused at strcpy() function call.

## 5. CONCLUSION

In this project, some integer and stack based overflow conditions are handled. The integer overflows for expressions and bound limits for stack arrays are checked. As a result, the possible problems are list using a GUI.

The program generally creates a syntax representation of the source code. After that, it performs value flow analysis to detect overflows. Currently, the syntax of the C language is very limited. After detecting the overflows, the errors are listed in the GUI.

In future study, the syntax analyser implemented in this project should be improved. The expression and statement models should cover all C Language syntax models. The diagnosis engine also modified according to syntax analyser.

# BIBLIOGRAPHY

- [1] J. Zheng, “Buffer overflow vulnerabilities diagnosis for commodity software. M.S., North-Eastern University, Boston, USA,” 2001.
- [2] H. Hyyryläinen, “A Static Analysis Tool for Finding Buffer Overflows in C, University of Oulu,” 24th May 2020.
- [3] M. M. KARIM, “Source Code based Buffer Overflow Detection Technology,” 2015.
- [4] E. C. Wikman, “STATIC ANALYSIS TOOLS FOR DETECTING STACK-BASED BUFFER OVERFLOWS,” 2020.
- [5] F. L. Lili Xu Mingjie Xu and W. Huo, “ELAID: detecting integer-Overflow-to-Buffer-Overflow vulnerabilities by lightweight and accurate static analysis,”