Engineering Degree Project

Code Review Application

- Simplifying code review through data flow visualization

Authors: Viktor Möllerström, Jesper Roos
Supervisors: Oscar Henriksson, Torbjörn Jonsson
Lnu Supervisor: Jonas Lundberg
Semester: Spring 2023
Subject: Computer Science

Abstract

From a security standpoint, manual code review is widely regarded as a dependable practice, particularly in systems with heightened security needs. However, it is also a time-consuming and laborious task that requires careful consideration. To address this issue, this project aims to explore the feasibility of an application that would present graphical presentations of data flow, which would simplify the manual review process. Input data is an excellent starting point when searching for security vulnerabilities in a program. For that reason, input data traversal is of significant interest when conducting code review with respect to security. The application will track the input data flow through function calls in the program to facilitate the task of identifying which functions require closer examination. The development of such an application is a significant undertaking, and therefore, the decision is made to limit the scope of the project to a proof of concept that will function on smaller programs. The findings indicate that the developed application possesses the capability to perform input data backtracking across function calls. However, it is important to note that a functional forward tracking algorithm has not been integrated into the application at present. Despite this limitation, the feasibility of fully realizing the project is perceived to hold promising potential within the code review market.
Keywords: data flow, visualization, code review, security

Preface

We extend our sincere appreciation to Combitech for graciously affording us the privilege to conduct this project within their esteemed organization. Additionally, we would like to express our heartfelt gratitude to Oscar Henriksson and Torbjörn Jonsson, who served as our supervisors at Combitech and provided invaluable guidance and support throughout the course of this project.

Contents

1 Introduction

The introductory section of this report provides an overview of the research objectives, the problem statement, and the significance of the study. It highlights the motivation behind the research and introduces the context and background of the topic. Additionally, it outlines the problem formulation and structure of the report. The chapter aims to set the stage for the subsequent chapters by establishing the relevance and importance of the research and providing a clear roadmap for the reader.

1.1 Background

In an era characterized by increasing digitalization, the escalation of potential cyber threats necessitates heightened security awareness and robust solutions. Consequently, ensuring the meticulous examination of developed software prior to its deployment in production is paramount in countering these threats. Among the crucial steps in safeguarding the integrity of a program, performing a comprehensive code review assumes significance [1]. This process entails the systematic scrutiny of a program's code to identify, eliminate, and mitigate any vulnerabilities that may compromise its security.

The topic of code review is extensive, and a software program may encompass millions of lines of code. Conducting a code review holds significant importance for various reasons. It fosters consistency in design and implementation, enhances code performance, guarantees project quality, and ensures adherence to application requirements. The specific approach taken during code review can vary significantly depending on the purpose of the review and the aspects being assessed. In the context of code review with a focus on programs with high security demands, reviewers must undertake a thorough, manual examination of the code within the program to identify any flaws that could potentially jeopardize its security. The article authored by Katerina Goseva-Popstojanova and Andrei Perhinschi, evaluates three contemporary commercial static analysis tools [2]. The results of their assessment indicate that these tools are susceptible to generating false-negatives, highlighting the importance of manual review as the most dependable practice [3]. However, these automatic tools are a great support for the code review process. Consequently, the manual review process can be a laborious and time-consuming task. As such, there is a desire to expedite this process where possible to save time and resources.

This project focuses on providing assistance for static code review, specifically with regard to security vulnerabilities. Hence, factors other than code quality or performance are taken into account. In accordance with the request made by Combitech, the project aims to investigate the development of an application that can identify flows and functions in a program that may pose potential security risks, and present these findings in an accessible manner for the reviewers. Combitech is a subsidiary of the Swedish company Saab, and primarily operates in the realm of defense and military [4]. The company is involved in the development of a range of products such as Jas Gripen and the digital air traffic tower. It is a large corporation with a significant presence in Sweden and Scandinavia, employing a substantial number of individuals.

1.2 Related work

OWASP is a frequently used website in all matters that regard security on the internet [5]. They are especially famous for their top ten security vulnerabilities to be aware of, which is updated every year. In an article available on their site, input data validation is explained, its importance, and how it can be exploited [6]. To summarize, there are numerous types of input data, and many cases where input data validation is important. If handled poorly, it can result in severe security vulnerabilities.

Ari Kesäniemi has written an article, which focuses on examining and contrasting manual and automatic analysis methodologies within the domain of application security document [3]. The content delves into a comprehensive evaluation of the merits and drawbacks associated with both approaches. Moreover, it underscores the importance of achieving a harmonious integration of manual and automatic analysis techniques to proficiently detect and address security vulnerabilities prevalent in software applications.

Rakan Alanazi, Gharib Gharibi, and Yugyung Lee has written a study that explores the possibility of increasing comprehension of large call graphs [7]. Providing call graphs for comprehension becomes difficult as the program increases in size and complexity. With no stopping point, the graph can become too large to be visually readable. Therefore, a way of dividing the graph into sub graphs is necessary in order to increase readability.

An academic thesis, conducted by Nico L. de Poel in 2010, explores the assessment of Static code analysis tools and their suitability in the context of PHP web applications' security [8]. The research investigates and evaluates the effectiveness of these tools in addressing security concerns. The primary objective is to assess the applicability of Static code analysis tools for enhancing the security of PHP web applications. The findings of this study provide valuable insights into the efficacy and potential benefits of utilizing such tools in the development and maintenance of secure PHP web applications.

David Evans and David Larochelle, has authored a paper that focuses on the topic of 'Improving Security Using Extensible Lightweight Static Analysis' [9]. The authors address the significant challenges posed by security risks and emphasize the importance of mitigating vulnerabilities to either eliminate them entirely or minimize the potential damage they can cause. In this context, the authors conduct an evaluation of a specific tool called Splint, which employs extensible and lightweight static analysis techniques. The study critically examines the effectiveness of Splint in enhancing security measures.

Gary McGraw has authored a paper that explores the efforts of three prominent vendors in automating the code review process with a focus on security [10]. The paper emphasizes the importance of automating code review as a means to enhance software security. The three vendors discussed in the paper aim to develop tools that can automatically analyze code for potential security vulnerabilities. By utilizing various techniques and algorithms, these tools assist developers in identifying and addressing security issues in their code.

In an article authored by Katerina Goseva-Popstojanova and Andrei Perhinschi, the focus is on empirical evaluation of the ability of static code analysis tools to detect security vulnerabilities with an objective to better understand their strengths and shortcomings [2]. The authors have conducted a comprehensive evaluation of three state-of-the-art static code analysis tools by subjecting them to benchmarking tests using the Juliet suite.

The results of their study indicate that none of the tools were capable of detecting all vulnerabilities during the tests. In fact, numerous vulnerabilities were overlooked by all three tools. It is crucial to acknowledge that this study was conducted in 2015, and since then, significant advancements have been made in the capabilities of such tools, rendering them more proficient today.

1.3 Problem formulation

In the context of code review from a security vulnerability perspective, while automated tools are used as support, manual code review is necessary when considering programs with high demands on security [2, 3, 10]. This is due to the fact that the reliability of available applications and tools cannot be completely trusted, especially when security is of great importance. These tools are often created with the purpose of automating the whole review process. The aforementioned paper authored by Gary McGraw, highlights three prominent vendors in the domain who aim to automate the code review process specifically for security purposes [10]. In contrast, this project will instead attempt to realize a prototype, which will serve as a proof of concept, that can assist manual code review. To accomplish this, the objective is to develop an application capable of leveraging information extracted from a code parser to identify data flow within the code. This information will be presented in the form of a call graph, which will offer an overview of the code and identify specific areas requiring manual code review.

The following subsections provides the project's objectives to be addressed, with a brief explanation of what they will entail.

1.3.1 Create a protoype

The primary objective of this research project is to develop a prototype application that aims to identify input functions within a program and subsequently track the information flow originating from these inputs. The ultimate goal is to visually represent this information in the form of a call graph. The call graph generated by the application illustrates the sequence of function calls along with the corresponding code lines that pertain to the handling of input data.

1.3.2 Evaluation of the prototype

In order to assess the potential value of further developing the application, it is imperative to ascertain its utility for code reviewers. To achieve this objective, a series of experiments are conducted utilizing the completed prototype. These experiments are designed to evaluate the performance of the prototype by specifically examining its key features and assessing their efficacy in fulfilling their designated tasks. Through this evaluation process, the experiment results will provide insights into the overall performance and accuracy of the prototype, thus informing the decision-making process regarding the feasibility of advancing its development.

1.4 Expected Result

The development of an application capable of generating visual representations of program code of any size and type is a highly intricate undertaking. The project timeline will not permit the full realization of this objective. Therefore, the object of the project will be to present a proof-of-concept demonstration using smaller-scale program code. This approach will enable the possibility to validate the core functionalities of the application and assess its potential for future expansion and refinement. Within this context, Fig. 1.1 exemplifies a useful and easily comprehensible graph of a small program, which would significantly assist in the manual code review process.

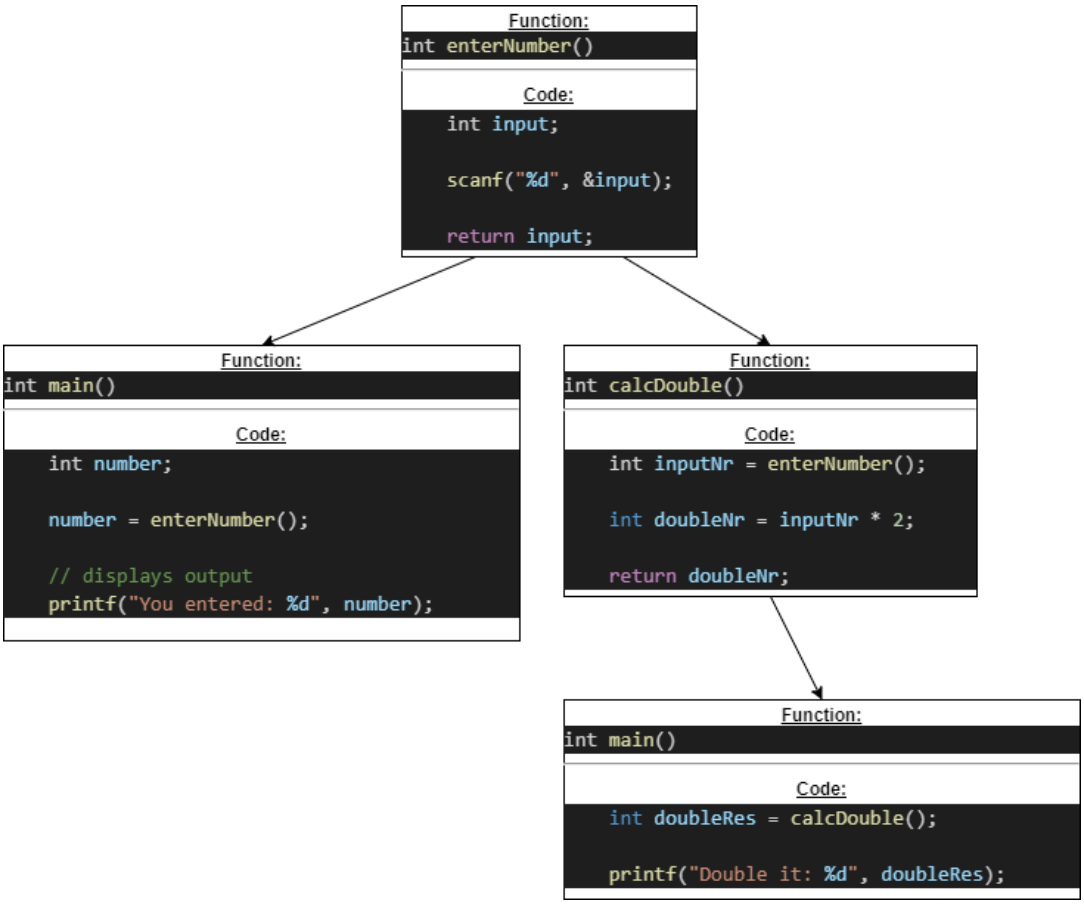

Figure 1.1: Example of a graphical representation of a small program.

Fig. 1.1 illustrates the tracking of the input data function `scanf()` in a small program and tracing the path of data throughout the program. Given that `scanf()` is a function that captures input from the keyboard, it is a potential security vulnerability in the program.

The top node of the resulting tree depicts the sole function that employs `scanf()`, namely `enterNumber()`. The node displays the function's name, along with pertinent lines of code that relate to the variable associated with the `scanf()` function, and the function's return value. The left and right branches of the tree illustrate the functions that call `enterNumber()`. Specifically, the branches show that `enterNumber()` is used in `main()` and `calcDouble()`. Furthermore, the right branch displays another node that represents how `main()` calls `calcDouble()`. When a function utilizes data for a specific task without returning anything, the branch terminates.

By reviewing this tree, code reviewers can effectively track the flow of input data, its manipulation, and the functions that employ it. This makes the manual review process less arduous and more efficient, as reviewers can pinpoint where to focus their attention.

1.5 Motivation

This project is driven by the growing digitization of society and the ubiquitous presence of software in various domains. Consequently, the imperative to develop secure software has become a top priority. A significant aspect of ensuring software security lies in conducting thorough code reviews, and any advancements in this field can be highly advantageous. However, the code review process is inherently time-consuming, making it essential to explore means of expediting the process without compromising security.

The primary emphasis of this project lies in the realm of input data as it represents the prevailing vulnerability in terms of security. Several noteworthy examples of such vulnerabilities include injection and buffer overflow, both of which present significant risks within a program. Further elaboration on these vulnerabilities can be found in Chapter Two. Furthermore, the comprehension of visual representations of programs, particularly those of significant size, can pose challenges. The graphical illustrations typically depict function calls alone, without presenting information regarding data exchange between functions. This information is of interest to code reviewers, who require a holistic understanding of a program. With respect to security vulnerability, there is information that is not relevant in the program. Consequently, the development of an application capable of providing crucial program information, such as input data traversal, would be immensely beneficial to Combitech and code reviewers in general. Successful implementation of this proof of concept could pave the way for further study and potentially offer substantial time-saving capabilities.

1.6 Scope/Limitation

The ambit of the project has been narrowed down to serve as a proof of concept. The examination of programs shall not be extensive. In order to restrict the ambit, the project shall concentrate on functions that employ input data, specifically `scanf()`, `getchar()`, `getch()` and `gets()`. Encompassing all categories of input data would constitute an overly extensive endeavor given the time constraints of this project. The application is only able to search C code.

1.7 Target group

The target group is code reviewers that work in the security sector. The application is designed for security vulnerabilities, and since this is a small area of code review, it is rather focused in its usage and probably not useful to other code reviewers.

1.8 Outline

Chapter Two, entitled *Theory*, serves to present the theoretical underpinnings of the research problem, wherein a review of pertinent literature and theoretical frameworks that inform the research problem is provided. The chapter also introduces and explicates relevant concepts and tools that are significant for the project.

Chapter Three, named *Method*, provides a detailed account of the research methodology adopted to address the research problem. The chapter describes the research design, data collection methods and data analysis techniques employed to achieve the research objectives. Each method is elaborated upon, and its application in the project is discussed.

Chapter Four, entitled *Implementation*, explicates how the software was designed and implemented. The chapter provides a schematic overview of the software architecture and showcases code snippets to further explicate the code. The design and development of the software are described, with a focus on highlighting the key features and functionalities.

Chapter Five, named *Experimental Setup, Results and Analysis*, details the experimental setup and presents the results. The chapter provides comprehensive details of the experimental design, including research hypotheses and variables under investigation. The results obtained from the experiments are presented, analyzed, and interpreted.

Chapter Six, named *Discussion*, focuses on the findings and compares them with what others in the same field have accomplished. The chapter identifies the strengths and weaknesses of the research and discusses the implications of the findings for theory and practice.

The final chapter, Chapter Seven, entitled *Conclusion*, presents a conclusive summary of all the findings in the project. The chapter considers the problem formulation and summarizes the main findings of the study. It also discusses the relevance of the findings for the industry and what further work should be done. The chapter highlights the contributions of the study and its limitations.

2 Theory

The following pages will provide a brief explanation of different concepts and tools being used in this project. It will provide a broad overview with the relevant information concerning each concept or tool.

2.1 Static code review

Static code review, also known as static code analysis or source code analysis, is the process of analyzing computer software without executing it [11]. It involves examining the code structure to detect bad coding style, potential vulnerabilities, and security flaws in a software's source code. This process is a common method for detecting bugs and issues in the code before it is executed. Static code analysis can be done using automated tools that examine the code for violations of rules and conventions that affect program execution and non-functional quality aspects of a software system such as complexity and maintainability. The process could involve transforming the code into an Abstract Syntax Tree (AST) and applying analysis rules to find potential issues. Static code analysis is generally good at finding coding issues such as programming errors, coding standard violations, undefined values, syntax violations, and security vulnerabilities.

2.2 Injection

An injection flaw represents a vulnerability that enables an assailant to transmit malevolent code through an application to an alternate system [12]. This encompasses the compromise of both backend systems and other clients connected to the susceptible application. Numerous web applications rely on operating system functionalities, external programs, and the processing of data queries submitted by users. When a web application incorporates information from an HTTP request as part of an external request, the possibility arises for an attacker to insert specialized (meta) characters, malicious commands/-code, or command modifiers into the message. These techniques empower an attacker to acquire, manipulate, or annihilate the contents of a database, compromise backend systems, or launch attacks against application users. Successful injection attacks possess the potential to entirely compromise or destroy a system.

The effects of these attacks include:

- Allowing an attacker to execute operating system calls on a target machine.
- Allowing an attacker to compromise backend data stores.
- Allowing an attacker to compromise or hijack sessions of other users.
- Allowing an attacker to force actions on behalf of other users or services.

2.3 Buffer overflow

Buffer overflow errors are a class of vulnerabilities wherein memory fragments within a process are overwritten, resulting in unintended modifications either intentionally or unintentionally [13]. This form of error typically manifests through the overwriting of critical registers such as the Instruction Pointer (IP), Base Pointer (BP), and other associated registers. The consequences of such overwriting are the generation of exceptions,

segmentation faults, and various other types of errors. These errors often lead to the abrupt termination of the application's execution in an unpredictable manner. It is important to note that buffer overflow errors specifically arise when manipulating buffers of character (char) type.

2.4 Abstract Syntax Tree (AST)

An abstract syntax tree (AST) is a tree representation of the abstract syntactic structure of source code written in a programming language [14]. It represents the structure of the code without including every detail of how it is executed. The AST for C code is built by parsing the code and then creating a tree structure that represents its syntactic structure. The nodes of the tree represent different elements of the code such as expressions, statements, and declarations. The edges between nodes represent how these elements are related to each other in the code.

The AST provides a high-level representation of the program that can be used for various purposes such as code analysis, optimization, and transformation. For example, compilers use the AST to generate machine code, while program analysis tools use the AST to extract information about the program's structure and behavior. In summary, the AST is a crucial data structure in the compilation process of C programs, providing a structured representation of the program's syntax that can be used for various purposes. Fig. 2.1 illustrates a graphical representation of the Abstract Syntax Tree.

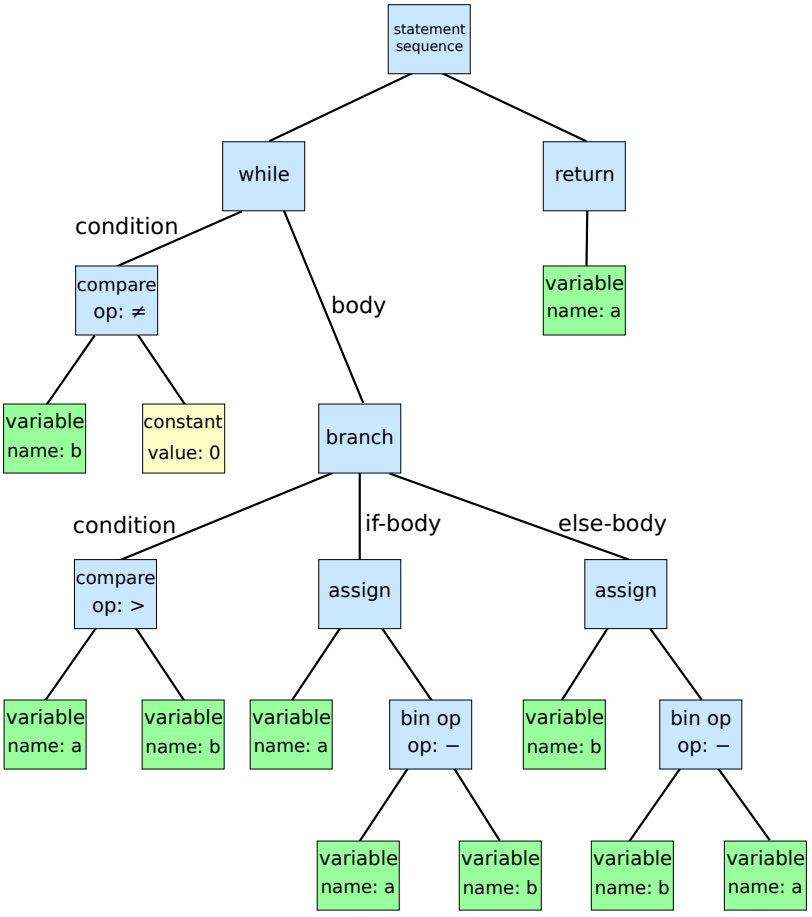

Figure 2.1: A graphical representation of an Abstract Syntax Tree. Image by Dcoetzee, licensed under CC0, via Wikimedia Commons [15].

2.5 Pycparser, library for parsing C code in Python

Pycparser is a Python module that provides a complete parser for the C language [16]. It is used to analyze C source code and extract information about the code's structure, such as the types of variables, the names of functions, and the structure of control flow statements.

Pycparser can parse C code represented as a string or read directly from a C file. For the file to be read without being transformed to a string first, Pycparser interacts with the C preprocessor. The C preprocessor handles preprocessing directives like `#include` and `#define`, removes comments, and performs other minor tasks that prepare the C code for compilation. If the user choose to just have the code for the C file as a string, preprocessing directives like `#include`, `#define` and comments needs to be removed manually before parsing the code. The most basic use case is to parse a C file, create an Abstract Syntax Tree (AST) and traverse it. The AST generated by pycparser can be used for a variety of purposes, such as code analysis, transformation, optimization, or generation. The library is widely used in various domains, including software testing, reverse engineering, and security analysis. It is an open-source project and is actively maintained by a community of contributors.

Fig. 2.2 illustrates a textual representation of a function definition in an abstract syntax tree (AST) generated by Pycpaser.

```
FileAST: (at None)
  FuncDef: (at test:2:9)
    Decl: add_numbers, [], [], [], [] (at test:2:9)
      FuncDecl: (at test:2:9)
        ParamList: (at test:2:25)
          Decl: a, [], [], [], [] (at test:2:25)
            TypeDecl: a, [], None (at test:2:25)
              IdentifierType: ['int'] (at test:2:21)
          Decl: b, [], [], [], [] (at test:2:32)
            TypeDecl: b, [], None (at test:2:32)
              IdentifierType: ['int'] (at test:2:28)
        TypeDecl: add_numbers, [], None (at test:2:9)
          IdentifierType: ['int'] (at test:2:5)
    Compound: (at test:3:1)
      Return: (at test:4:9)
        BinaryOp: + (at test:4:16)
          ID: a (at test:4:16)
          ID: b (at test:4:20)
```

Figure 2.2: A part of an Abstract Syntax Tree generated by Pycpaser.

2.6 DOT, language for describing graphs in Graphviz

The DOT language is a simple, text-based language used to describe graphs and networks [17]. It was developed as part of the Graphviz project, which is an open-source graph visualization software package. Each DOT file is started with `digraph G {}`. Between the brackets, the nodes and edges are written. To write an edge, the format is as follows: node1 -> node2. Fig. 2.3 displays a graph description. It contains a small call graph of two functions, where enterNumber has an edge towards main. This can be visualized with a tool called Graphviz, which is explained in the next subsection.

```
digraph G {
enterNumber [label="enterNumber
    int input;
    scanf('%d', &input);
    return input;
"]

main [label="main
    int number;
    number = enterNumber();
    printf('You entered: %d', number)
"]

enterNumber -> main
}
```

Figure 2.3: Example of a graph description written in DOT.

2.7 Graphviz library for visualising data

Graphviz is an open-source software tool that offers a variety of libraries and tools for generating diagrams and visualizing data structures [18]. It provides support for creating different types of diagrams, such as flowcharts, trees, and directed and undirected graphs. To generate an image in various formats, including PNG, SVG, and PDF, the software uses a simple text-based language referred to as DOT, to specify the structure of the graph or diagram. The appearance of the graph can be customized by specifying various attributes, including colors, fonts, and shapes. The software's automatic layout capabilities enable the visualization of complex data structures in an efficient and user-friendly manner.

Graphviz has numerous applications in software engineering, data science, and network analysis, and can be used for visualizing code dependencies, project timelines, decision trees, social networks, and more. Overall, Graphviz is a powerful and valuable resource for a wide range of applications, with its simple text-based syntax and automatic layout capabilities. Fig. 2.4 shows a visual representation of the graph description in Fig. 2.3.

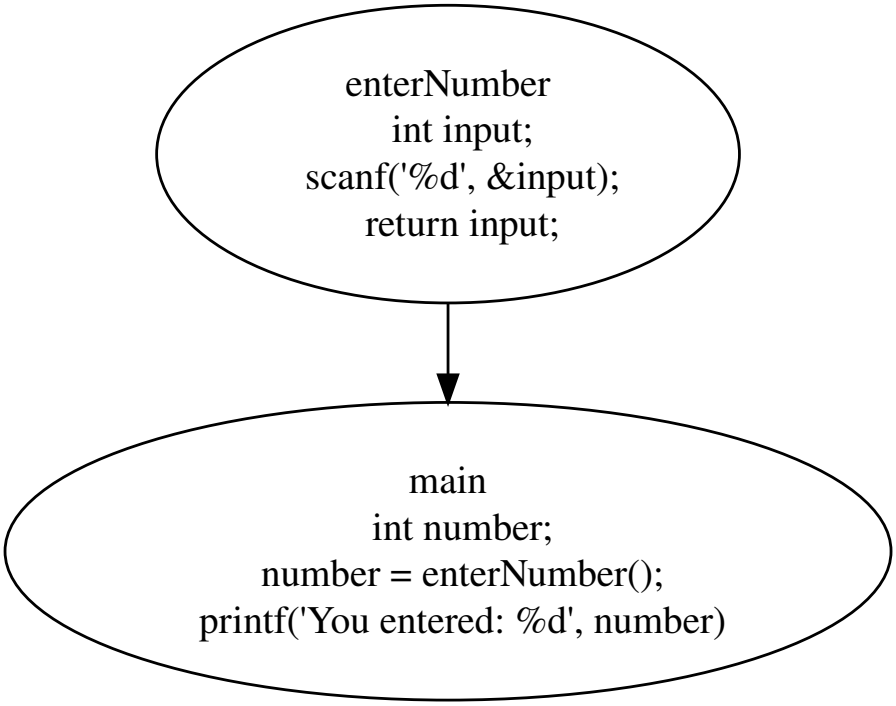

Figure 2.4: Graph produced with Graphviz.

2.8 Static Backtracking

In the realm of program analysis, static backtracking pertains to the process of scrutinizing a program's behavior and tracing its execution path, all while abstaining from the actual execution of the program itself. Rather than dynamically running the program, static backtracking relies on the utilization of static analysis techniques to investigate the program's source code or intermediate representation.

Static backtracking encompasses the examination of the program's control flow, data dependencies, and variable values without resorting to program execution. It delves into different paths and execution branches by thoroughly analyzing the program's structural elements, including conditionals, loops, and function invocations. Throughout the course of static backtracking, diverse analysis techniques are employed to monitor the flow of data and control within the program. Such techniques encompass symbolic execution, abstract interpretation, data-flow analysis, and constraint solving. By applying these techniques, static backtracking is capable of unveiling potential execution paths, identifying dead code, detecting unreachable states, and inferring plausible values of variables at various program junctures.

The methodology employed for backward tracking in this project can be outlined as follows: the algorithm is implemented recursively, whereby it identifies functions that utilize a specific input data function, such as `scanf()`, and subsequently traces the sequence of functions invoking this particular function. This process extends further to include subsequent functions that call the previously identified functions, forming an iterative exploration of the complete function call tree. The algorithm is presented in a form of pseudocode in Fig. 2.5.

```
function searchFunctions(inputFunction):
    foundFunctions ← [];
    examineAllFunctions();
    for each function in allFunctions do
        if function calls inputFunction then
            foundFunctions.append(function);
        end
    end
    for each foundFunction in foundFunctions do
        searchFunctions (foundFunction);
    end
end
```

Figure 2.5: Back tracking algorithm.

2.9 Static Forwardtracking

Static forwardtracking, within the realm of program analysis, pertains to the systematic examination of a program's behavior and the anticipation of its execution path. This approach involves commencing from the program's initial state and progressing along the flow of control in a forward manner. Static forwardtracking concentrates on scrutinizing the program's forthcoming states and projected outcomes, all while abstaining from actual program execution.

The methodology employed for forward tracking within the framework of this project can be outlined as follows: Firstly, an examination of the program's functions is conducted to locate instances of input functions, paying particular attention to pertinent code sections. Subsequently, the variables associated with the input functions and their respective values are extracted. Next, an analysis is performed to determine whether these variables are utilized as arguments in other function calls. In the event that a variable is indeed passed to another function, an exploration is initiated within that function to trace the progression of the transmitted variable. Within this new function, the relevant code sections and variables of interest are identified, and the process is repeated to ascertain if the variable is further transmitted to subsequent functions. This iterative procedure is continued until a point is reached where the variable ceases to be passed along any further. The algorithm is presented in a form of pseudocode in Fig. 2.6.

```
function forwardTracking(inputFunction):
    examineFunctions();
    locateInputFunctionInstances(inputFunction);
    extractVariablesAndValues();
    for each variable in inputFunction.variables do
        exploreVariableProgression(variable);
    end
end
function exploreVariableProgression(variable):
    relevantCodeSections ← identifyRelevantCodeSections();
    for each functionCall in relevantCodeSections do
        if variable is passed as an argument to functionCall then
            exploreFunction(functionCall);
        end
    end
end
function exploreFunction(function):
    relevantCodeSections ← identifyRelevantCodeSections();
    for each variable in function.variables do
        if variable is passed as an argument to anotherFunction then
            exploreVariableProgression(variable);
        end
    end
    if no more variables are passed to subsequent functions then
        return
    end
    exploreFunction(nextFunction);
end
```

Figure 2.6: Forward track algorithm.

2.10 False-negatives, False-positives

False-negative and false-positive are concepts commonly used in the context of testing, evaluation, and classification [19]. These terms are crucial in understanding the accuracy and reliability of tests, evaluations, or classification systems. Minimizing both false-negatives and false-positives is important to ensure effective decision-making and accurate assessments.

A false-negative occurs when a test or evaluation incorrectly indicates a negative result for a condition or attribute that is actually present. In other words, it is a failure to detect or identify something that should have been detected. False-negatives can lead to missed opportunities, overlooked risks, or incorrect conclusions. For example, in medical testing, a false-negative result would suggest that a person does not have a particular disease or condition when they actually do. Conversely, a false-positive happens when a test or evaluation incorrectly indicates a positive result for a condition or attribute that is not actually present. It is a false alarm or an erroneous identification of something that should not have been detected. False-positives can lead to unnecessary actions, wasted resources, or incorrect assumptions. For instance, in security screening at an airport, a false-positive result would indicate the presence of a prohibited item when there is none.

2.11 Fortify, an automated code review tool

Fortify, developed by Micro Focus, is an esteemed security tool widely employed for static code analysis purposes, specifically targeting the identification of security vulnerabilities in software code [20]. Esteemed for its efficacy, Fortify offers the Static Code Analyzer feature, which automatically scans source code and delivers comprehensive analysis data, enabling developers to promptly identify and address violations. Facilitating swift remediation, Fortify strategically prioritizes the identified violations.

Furthermore, Fortify's offline functionality allows users to conveniently install the tool on their local machines, thereby enabling offline code analysis. Execution via the command line can be accomplished utilizing the "sourceanalyzer" command. Notably, Fortify also encompasses an IDE plugin for Visual Studio 2017, delivering real-time security analysis while developers engage in code composition. Fortify further extends its utility by offering integration capabilities with other tools, including Software Security Center, Audit Workbench, and WebInspect. With its robust capabilities, Fortify emerges as a formidable security tool, equipping developers with the means to effectively identify and remediate security vulnerabilities in their code.

3 Method

This chapter presents a detailed account of the sequential progression of the project, along with an explanation of the varied methodologies utilized and their implementation towards achieving the project's objectives.

3.1 Research Project

At the beginning of the project, no prior knowledge existed regarding the relevant tools and libraries for the subject at hand. For this reason, a preliminary literature search is being conducted to identify the tools relevant to the project, both for parsing code and presenting graphs, as well as what others have already done in the field.

In order to evaluate the applications performance, controlled experiments will be conducted using the implemented application on several different test programs. Key features will be selected, which will serve as a measurement of how well it is performing.

3.2 Method

The following subsections will provide a description of the methodologies used, and how they are applied to the project. These will involve a literature study, and controlled experiments.

3.2.1 Literature study

A literature study involves an initial exploration of scholarly sources, such as research articles, books, conference proceedings, and relevant publications, with the aim of acquiring a comprehensive understanding of existing knowledge and identifying research gaps within a specific field of study. This endeavor serves as a crucial starting point for any academic or research project, as it facilitates the identification of pertinent literature, establishes contextual background, and refines research questions or objectives. Throughout a preliminary literature search, researchers employ a variety of search strategies and techniques to locate and access relevant publications. The primary purpose of conducting a preliminary literature search is to obtain an overview of existing literature, ascertain key themes, theories, methodologies, and findings associated with the research topic. It is important to note that a preliminary literature search differs from a comprehensive literature review, as the former is not exhaustive in nature.

For the purpose of conducting the preliminary literature search in this particular project, both Google Scholar and regular Google search were utilized. Google Scholar was employed to identify related work and gain insights into previous endeavors within the field. The search strings utilized on Google Scholar included "code review security thesis" and "code review security." The search yielded a substantial number of results, which are presented in Table 3.1. On regular Google search, the search strings employed encompassed "c parser," "code to callgraph," and "code review open source tools." The number of results obtained from these searches is presented in Table 3.2.

Table 3.1: Search results on google scholar.

Date	Search string	Results	Search engine
2023-04	code review security thesis	1 470 000	Google scholar
2023-04	code review security	3 270 000	Google scholar

Table 3.2: Search results on google search.

Date	Search string	Results	Search engine
2023-04	c parser	256 000 000	Google search
2023-04	code to callgraph	480 000	Google search
2023-04	code review open source tools	681 000 000	Google search

3.2.2 Controlled experiments

Controlled experiments are a scientific methodology employed for the purpose of hypothesis testing and informed decision-making [21]. They involve the selection of one variable, from among several, to serve as the independent variable, while all other variables are carefully controlled or held constant to prevent their influence on the independent variable. By adopting this approach, it becomes easier to ascertain the effect of the independent variable on the system under investigation. This experimental technique is widely employed across numerous scientific disciplines, including but not limited to biology, psychology, physics, and engineering. Its significance lies in its ability to establish a cause-and-effect relationship between variables, thereby enhancing the accuracy of research conclusions.

Through systematic variation of test programs, the application will serve as a dependant variable and the test programs are independent variables. The output of the application will present how well it is performing with regard to a number of key features.

- Ability to find all functions in the test program.
- Ability to find all important lines of code in the test program.
- Ability to find all input data functions in the test program.
- Ability to track the flow of data regarding each input function in the test program.

3.3 Reliability and Validity

Reliability and validity are two important concepts in research methodology, which are used to evaluate the quality of data and the extent to which it is accurate, consistent, and trustworthy. Reliability refers to the consistency and stability of data over time and across different observers, instruments, or settings. In other words, it measures the degree to which the same results can be obtained repeatedly under the same conditions. Validity, on the other hand, refers to the extent to which a research study measures what it is intended to measure. It assesses the accuracy and appropriateness of the research design, methods, and instruments used to collect and analyze data.

The reliability of the project cannot be precisely determined due to various factors that preclude an unambiguous implementation. While it is possible for another person to select the same tools and employ them in the same manner, numerous code parsing- and visualization tools are available, and their use may yield diverse end results. Such divergence is primarily attributed to differences in the code written to execute these tools.

In the initial stages of the project, it was determined that a customized code parser would be developed instead of utilizing an existing one. This decision was driven by the constraints of a tight timeline, which hindered the acquisition of comprehensive knowledge regarding available code parsers. As a result, it is likely that others would not replicate the exact implementation approach due to these unique circumstances. Nonetheless, the key aspects of the project that are pertinent to code reviewers will remain unchanged, and another implementation will likely provide the same information. However, the specific tools used and how they are applied may vary, resulting in marginally different outcomes.

In conclusion, the project's cornerstone comprises of a parser and a graphical presentation tool. The employment of the DOT language and Graphviz to produce graphs is a probable choice. Although the program's logic and tool utilization may differ, the final outcome should be comparable since the problem formulation stipulates the graph's appearance. It is crucial to emphasize that the project is a proof of concept and is limited in its application scope. Specifically, it is applicable solely to small programs that are not intricate and encompass a maximum of 250 lines of code. Furthermore, the project concentrates on tracking input data; however, it does not encompass all the methods available for obtaining user input in C. To validate the project, thorough testing is carried out on multiple small programs written in C code that have a predetermined anticipated graph. The precision of the result is monitored throughout the testing process. However, it is crucial to acknowledge that the project serves solely as a proof of concept and is not feasible for large-scale programs. Therefore, its validity is restricted to a narrow scope of input data and cannot not be used in larger and more complex programs.

3.4 Ethical considerations

This project does not encompass any ethical considerations as it does not involve any research directly involving individuals, nor does it involve the handling of sensitive personal information that may pertain to individuals' confidentiality or pose potential harm. Furthermore, all results generated in the project are presented in their accurate form and have not undergone any modifications.

4 Implementation

The project encompasses an implementation comprising a software program that undertakes the parsing of code, partitioning it into distinct functions, and subsequently scanning these functions for function calls and critical lines of code. Ultimately, the program presents the obtained information in the form of a call graph. In order to determine the intended outcome of the application, a process of collaborative decision-making is undertaken with code reviewers at Combitech. Within these discussions, a consensus was reached regarding a preferred exemplar output, as illustrated in Fig. 1.1.

The software program can be segmented into three main components, each of which is composed of various subcomponents. Initially, all functions are extracted from the C file. Within this phase, a dictionary is constructed, containing code lines alongside their corresponding row numbers. `Recursive_function_call_backtrack` is utilizing the backtracking algorithm to track all functions commencing from a designated starting point, such as `scanf`. This process is complemented by the utilization of `code_lines_search` to extract pertinent lines of code from each function. Fig. 4.1 provides a simplified overview of the program's structure.

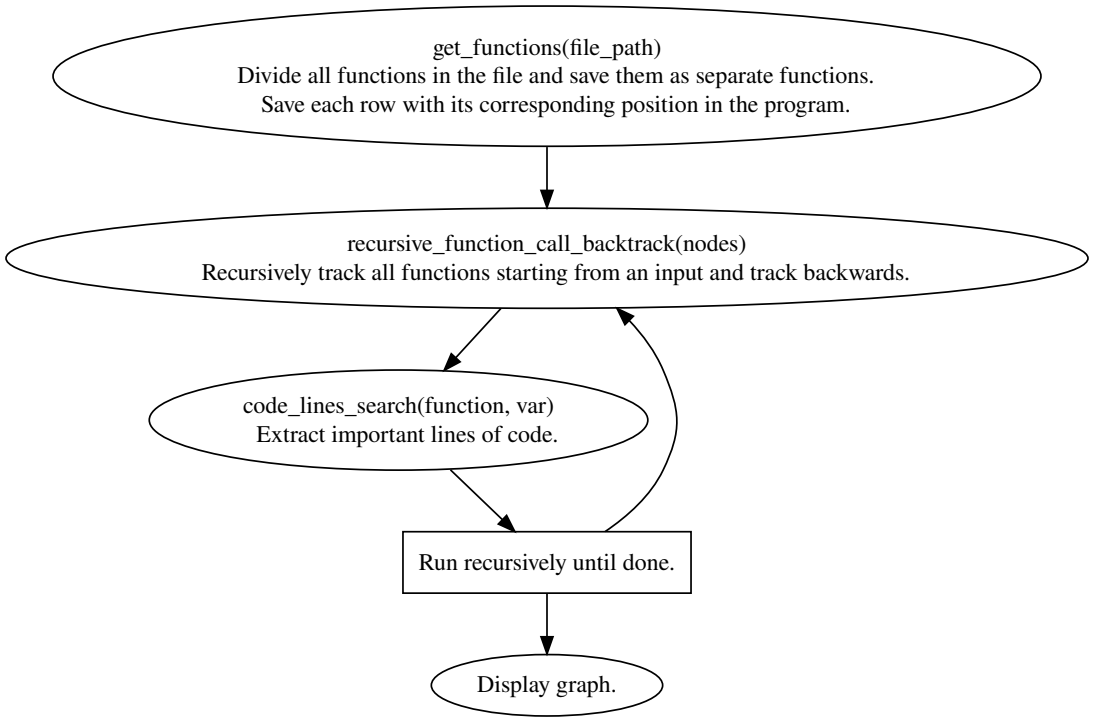

Figure 4.1: Simplified overview of the program.

The application has been developed employing an exploratory programming approach. Following the formulation of a set of requirements shown in Table 4.1, a sequential implementation approach was adopted. Among these requirements, certain ones posed greater challenges, with particular emphasis on those necessitating recursion algorithms. Notably, the backtracking and code line search algorithms fell into this category. An endeavor was made to devise a forward tracking algorithm, which would have constituted the most formidable challenge. Regrettably, due to project time constraints, the implementation of this algorithm could not be realized. However, its implementation is certainly feasible.

Table 4.1: Requirements.

R1	Separate all functions in the program and save them as strings.
R2	Identify function calls inside functions.
R3	Find variables that use input data.
R4	Store all lines of code that are relevant to the input data variable(s).
R5	Follow the function calls that use the data.
R6	Present the flow as a call-graph that is similar to Fig. 1.1.

The project began by examining available tools for code parsing and graph visualization, with the realization that utilizing an existing code parser would require a comprehensive understanding of its inner workings. Consequently, the decision was made to develop a custom method for parsing C code, as it was deemed more practical for the specific requirements of the project. This made python a logical choice of programming language for the application, due to its superior text processing capabilities. To streamline the implementation process, individual requirements were tackled and tested in isolation. This approach facilitated task delegation, with one team member focusing on backtracking and another on code line searching, both of which presented notable challenges due to their recursive algorithmic nature.

In order to implement the recursion algorithms, an understanding of the available information during each iteration and its utilization for graph creation was imperative. Time and effort was dedicated to conceptualizing and strategizing these processes, which facilitated a clear vision and enabled a systematic step-by-step implementation. Once these components were successfully implemented, minor adjustments were made to facilitate their integration. The final phase involved leveraging the accumulated information to generate a graph. To achieve this, proficiency was attained in the DOT language and Graphviz. The final step was enabling the automation of the graph generation process.

In the following sections the functions of the application will be explained, some in detail while others are small and self explanatory.

4.1 Extract and store all functions

The purpose of the function `get_functions(file_path)`, is to open a C file specified by the provided path argument and extract the functions within it, while also recording the line numbers associated with each code line. This is necessary for displaying the line numbers in the final output. The function returns a list of functions and a dictionary that maps each function to its corresponding code lines, and their line numbers.

4.2 Recursively backtrack function calls

Fig. 4.2 illustrates the recursive backtrack function, which operates iteratively until all input data functions and their data traversal have been backtracked.

```
def recursive_function_call_backtrack(nodes):
    subnodes = []

    for functionName in nodes:
        subnodes = function_call_search(functionName)
        add_nodes(functionName, subnodes)
        recursive_function_call_backtrack(subnodes)
```

Figure 4.2: Recursive Backtrack Search of Function Calls.

An initial list of `nodes` is created, which includes four specific input functions: `scanf`, `getchar, getch`, and `gets`. To commence the backtracking algorithm, the function is invoked with the argument `nodes`. Subsequently, the function initiates the search process by iteratively traversing the `nodes` list and examining each function for any calls to the corresponding node. In the event that such calls are discovered, they are appended to the list of nodes and serve as subnodes for the subsequent iteration. Conversely, if no calls are found, the program proceeds to the next iteration.

During the first iteration, the `function_call_search` method is invoked, returning a fresh list of nodes that have invoked `scanf`. Then, the `add_nodes` function is called, which employs the subnodes to collect significant code lines and write the collected information to the DOT file. In the first iteration, the `functionName` argument corresponds to `scanf`. In the resulting DOT file, this function will serve as the top node, while the subnodes will represent the originating points of the arrows. Subsequently, the subnodes are passed as arguments to `recursive_function_call_backtrack`, and the entire process is repeated for each subnode.

4.3 Search for function calls

Upon program initialization, the C file is read, and all the functions contained within are extracted, these are stored in a global `functions` list. To identify functions that invoke a particular function, the `functions` list is iterated, and a check is performed to determine if the specified `functionName` is present. If the `functionName` is found within a function, and the function name is not identical to `functionName`, the function is included in the list of returned functions. Fig. 4.3 illustrates the `function_call_search` function.

```
1 def function_call_search(functionName):
2     global functions
3     newFunctions = []
4     for function in functions:
5         if f'{functionName}(' in function and get_function_name(
              function) != functionName:
6             newFunctions.append(get_function_name(function))
7     return newFunctions
```

Figure 4.3: Function to search for function calls.

4.4 Add nodes to graph

Fig. 4.4 illustrates the `add_nodes` function. This function performs an iteration over the subnodes, each referred to as `caller` and representing a function name. Subsequently, the function `get_function_by_name` is invoked to retrieve the code lines associated with the `caller` function. The `call` variable is then prepared for transmission to the `create_node` function. In the context of the DOT language, this variable is a string used to establish a graphical connection between two functions. Following this, the function `code_line_search` is called to extract code lines from the `caller` function that are relevant to the specified `functionName` (in the initial iteration, `scanf` is used as the `functionName`). The extracted code lines are stored in the global variable `tmp_code_lines` which must be sorted to ensure their proper sequence in the resulting graph. Finally, the function `create_node` is invoked, responsible for writing the node to the DOT file.

```
def add_nodes(functionName, subnodes):
    global tmp_code_lines
    code_lines = []
    for caller in subnodes:
        function = get_function_by_name(caller)
        call = f'{functionName} -> {caller}'
        code_lines_search(function, f'{functionName}(')
        for i in function:
            if i in tmp_code_lines:
                code_lines.append(i)
        create_node(caller, code_lines, call)
        code_lines.clear()
        tmp_code_lines.clear()
```

Figure 4.4: Function to add nodes.

4.5 Create a node

The purpose of this function is to process the information gathered in the `add_nodes` function so that it can be appropriately written to the DOT file. The function accepts several arguments, including the name of the new node, important code lines, and the `call` string generated in the `add_nodes` function. These inputs are manipulated within the function to ensure they are ready for writing to the DOT file. As part of the processing, each call is stored in a global list. This list is necessary because the DOT language does not permit multiple functions with the same name to be written. In cases where there are functions with the same name, a number is appended to the function name to differentiate them. Once the node text is prepared, it is then passed as an argument to the `write_to_dot_file` function for further handling and writing to the DOT file.

4.6 Write to DOT file

The purpose of this function is to write the information about a node to the "graph.dot" file. It begins by opening the "graph.dot" file and proceeds to write the `node_text` to it. This variable is passed as an argument as the function is invoked. Additionally, the function includes a check to determine if the code lines mentioned in the `node_text` already exist in the file. This situation may arise when a function is called by multiple functions. If the code lines are already present in the file, it indicates that the node has been previously written, and as a result, the function omits writing the node to the DOT file to avoid duplication.

4.7 Get a function by name

The purpose of this function is to iterate through the global list of functions and locate the desired function based on its name, using the assistance of the `get_function_name` function. Once the correct function is found, the function retrieves and returns the code lines associated with that particular function, identified by the name `functionName`.

4.8 Get a function name

This function utilizes the regular expression library in Python. It employs a predefined pattern to search the code lines of the `function` parameter. If a match is found, the

name of the function is returned as a string. For example, if the function definition line follows the format: `int add_numbers(int num1, int num2) {`, the function will return the string `add_numbers`, which corresponds to the name of the function.

4.9 Extract variable names

This function serves the purpose of verifying the feasibility of extracting a variable from a given line of code. The corresponding code is depicted in Fig. 4.5. The function `get_variable` accepts a code line as its input argument. Initially, the line is examined for the presence of '`scanf(`'. If '`scanf(`' is detected within the line, the variable is extracted through a series of operations. This entails eliminating all spaces and the symbol '`&`' by replacing them with an empty string, removing the closing parenthesis followed by a semicolon ('`);`'), and subsequently splitting the remaining string at each occurrence of a comma (,). Consequently, a `scanf` statement resembling `scanf("%d",&input);` would ultimately result in a list containing `["%d", input]`. The first element of this list is then discarded, leaving only the variable behind. In the event that '`gets(`' is found within the line, an alternative splitting mechanism is employed to extract the variable. Subsequently, an examination is conducted to identify the presence of '`getchar(`', '`getch(`', or '`=`' in the line. This analysis is performed using a function named `split_line`, which is called upon to execute the necessary splitting operation for variable extraction.

If none of the aforementioned patterns are identified within a given code line, the function returns None, signifying the absence of a valid variable extraction.

```
def get_variable(line):
    variables_list = []
    if 'scanf(' in line:
        variables = line.replace('␣', '').replace('&', '').strip
            (');').split(',')
        variables.pop(0)
        variables_list.extend(variables)
    elif 'gets(' in line:
        variable = line.replace('␣', '').split('(')[1].split(')'
            )[0]
        variables_list.append(variable)
    elif 'getchar(' in line or 'getch(' in line or '=' in line:
        variable = line.split("=")[0].strip()
        if '␣' in variable:
            variable = variable.split('␣')[1]
            variables_list.append(variable)
        else:
            variables_list.append(variable)
    else:
        return None
    return variables_list
```

Figure 4.5: Function to extract variables.

4.10 Find variables containing input data

This function is designed to systematically examine each line within a given function, aiming to extract variables using the assistance of the `get_variable` function. Initially, an empty list is initialized, which will subsequently serve as a repository for the identified variables. Subsequently, each line in the function is scrutinized for a specific keyword. If the designated keyword is detected within a line, the `get_variable` function is invoked to extract the variable present in that line. A subsequent verification process is conducted to ensure the accurate extraction of the variable. This involves checking if the extracted variable is not None. In the event that a valid variable is obtained, it is appended to the list that was created earlier. Once all lines have been inspected, the resulting list of variables is returned. The code exemplifying this functionality can be found in Fig. 4.6.

```
def find_variable(function, search_word):
    variable_list = []
    for line in function:
        if search_word in line:
            var = get_variable(line)
            if var is not None:
                for vv in var:
                    if vv is not None:
                        variable_list.append(vv)
    return variable_list
```

Figure 4.6: Function to find variables containing input data.

4.11 Recursive code line search

This recursive function is implemented with the objective of extracting significant code lines, leveraging the assistance of other functions. The code snippet for this function is presented in Fig. 4.7. The function accepts two arguments: "function" and "var." The "function" parameter represents the target function within which the search is conducted, while "var" denotes the search term.

Initially, the function invokes `find_variable(function, var)` to extract variables that interact with the search term. The obtained variables are stored in the variable "variables." Subsequently, a loop iterates through these variables, and the important code lines containing these variables are extracted. The extracted lines are also subjected to a loop. The primary aim is to identify lines containing the "=" symbol. Since these lines involve variables related to the search term, the presence of "=" signifies the assignment of another variable to the previously discovered variable. In such instances, the additional variable is extracted by invoking the `get_variable` function. Following this extraction, a check is performed to ensure that the newly found variable is distinct from the previous variable and is not identical to the search term. If these conditions are met, a recursive call is made, initiating a new search using the newly discovered variable as the search term.

By employing this recursive approach, all significant code lines are extracted. The utilization of recursion facilitates the continuation of the search process whenever a new variable is discovered, thereby ensuring the comprehensive identification of all relevant lines.

```
def code_lines_search(function, var):
    variables = find_variable(function, var)
    for variable in variables:
        lines = add_code_lines(function, variable)
        for line in lines:
            if '=' in line:
                newVar = get_variable(line)
                for vv in newVar:
                    if vv != variable and var != variable:
                        code_lines_search(function, vv)
```

Figure 4.7: Recursive code line search function.

5 Experimental Setup, Results and Analysis

This chapter aims to present the outcomes derived from a series of experiments conducted on a set of fifteen distinct test programs written in the C programming language. These are available in a GitHub repository [22]. Each of these programs incorporates at least one of the following input methods: `scanf`, `gets`, `getch` and `getchar`. Program 1, 2, 3 and 4 was developed specifically for this research, and are written by the authors of this report. Program 9 is obtained from a github repository [23] and program 10 is obtained from an educational programming website [24]. Program 5, 6, 7, 8, 11, 12, 13, 14 and 15 were generated using an AI language model based on the GPT-3.5 architecture provided by OpenAI [25].

To generate these programs from the AI, the queries were of this type:

- Design a program written in c, which encompasses at least 10 functions, makes use of at least one of the following input data functions: scanf, getchar, getch or gets, and where some functions are calling other functions.
- Write a simple implementation of a hangman game, written in c.

5.1 Experimental setup

Table 5.1 provides an overview of the computer specifications utilized for the experiments. The test programs employed in the study are characterized by their relatively small size, ranging from 32 to 258 lines of code. Detailed descriptions and implementations of all test programs can be found in the aforementioned GitHub repository, while the corresponding graphs are included in the appendix A section.

Table 5.1: Experimental setup

OS	Windows 11 64 bit
CPU	Intel(R) Core(TM) i7-9750H CPU @ 2.60GHz 2.59 GHz
RAM	16 Gb

The experiments were conducted using the developed application, wherein each test program was executed individually. In order to assess the performance of the application, four key features have been selected.

- Ability to identify functions.
- Ability to identify input data functions.
- Ability to identify relevant code lines.
- Ability to identify the flow of input data.

Tables were constructed to capture various metrics. These metrics encompassed the manual enumeration of relevant code lines, functions, and input functions within each test program, as well as an analysis of the expected flow. Subsequently, a comparison was made between the anticipated output and the actual output generated by the application.

5.2 Results and Analysis

The findings of the study will be presented through the utilization of three distinct tables. Specifically, these tables represent 3 out of the 4 key features that determine the performance of the application. The fourth feature regards the flow of input data within each program. To evaluate the flow a manual review of each program is made to determine the actual flow of the program, and is then compared to the output of the application. A few of these were selected to be displayed in this chapter, but for the interested reader, all flows can be manually followed by looking at the code corresponding to each graph that are present in the appendix section.

5.2.1 Identified functions

Table 5.2 provides an evaluation of the application's performance in effectively identifying all functions within each respective test program. The ability to accurately detect functions is a crucial aspect of the application, as each function may potentially encompass an input data function. In the context of the experiments conducted using small-scale test programs, the application demonstrates a commendable accuracy rate of 100 % in identifying all functions. It is important to note, however, that this accuracy should be interpreted within the context of the application's proof-of-concept nature and the limited size of the test programs used.

While this level of accuracy is satisfactory for the purposes of this project, it should be acknowledged that the diverse range of function writing styles in C has not been exhaustively tested. Hence, the accuracy achieved may be misleading. Nevertheless, given the objectives of the project, the observed performance of the application is deemed sufficient and satisfactory for a proof-of-concept implementation.

Table 5.2: Functions found.

Program	Number of functions	Found functions	Accuracy
1	3	3	100 %
2	3	3	100 %
3	5	5	100 %
4	3	3	100 %
5	11	11	100 %
6	19	19	100 %
7	22	22	100 %
8	20	20	100 %
9	11	11	100 %
10	5	5	100 %
11	4	4	100 %
12	10	10	100 %
13	12	12	100 %
14	6	6	100 %
15	5	5	100 %
Average			100 %

5.2.2 Identifiable input data functions

Table 5.3 illustrates the effectiveness of the application in identifying all input data functions within each test program. This aspect holds paramount importance as the application aims to trace the data flow within these functions. However, it is crucial to note that the accuracy rate of 100 % achieved in the small-scale test programs may not be indicative of the application's performance in more complex scenarios. It is pertinent to emphasize that the mere identification of input data functions is insufficient; the primary objective of the project lies in accurately tracking the data flow originating from these functions throughout the program. The extent to which the application achieves this objective can be best assessed by examining the resulting graph for each test program.

Table 5.3: Input functions.

Program	input functions	found input functions	Accuracy
1	2	2	100 %
2	3	3	100 %
3	2	2	100 %
4	5	5	100 %
5	1	1	100 %
6	1	1	100 %
7	10	10	100 %
8	2	2	100 %
9	1	1	100 %
10	1	1	100 %
11	3	3	100 %
12	3	3	100 %
13	3	3	100 %
14	3	3	100 %
15	1	1	100 %
Average			100 %

5.2.3 Identifiable relevant code lines

Table 5.4 illustrates the comparison between the manually counted relevant code lines within a program and the corresponding code lines detected by the developed application. Additionally, it provides information on the number of correctly identified code lines. The accuracy metric is computed by dividing the count of correct lines by the count of relevant code lines.

These are the features that determine the relevance of a code line.

- Input relevance: A code line is considered relevant if it involves an input. Any code line that directly handles or processes the input is relevant.
- Variable assignment relevance: If a variable is assigned the value of an input, then each subsequent code line that refers to this variable is considered relevant. This includes both direct usage of the variable and any operations performed on it.
- Variable interaction relevance: If a variable interacts with an input variable, either through an operation or a comparison, the code line involving this interaction is relevant. This includes cases where the interaction is with a variable that was assigned the input value.
- Function call relevance: When a relevant variable is passed as an argument to a function, all the code lines within that function that reference the input variable become relevant. This encompasses both direct usage and any further interactions with the input variable.

Table 5.4: Relevant code lines.

Program	Total code lines	Relevant code lines	Found lines	Correct lines	Accuracy
1	32	14	15	14	100 %
2	46	18	18	18	100 %
3	46	7	10	7	100 %
4	43	23	21	21	91.3 %
5	82	10	13	10	100 %
6	183	5	7	5	100 %
7	202	58	66	58	100 %
8	258	64	26	25	39 %
9	221	5	4	4	80 %
10	146	4	3	3	75 %
11	40	18	21	18	100 %
12	68	27	28	21	78 %
13	96	32	30	24	75 %
14	59	17	21	16	94 %
15	70	7	10	5	71.4 %
Average					86.9 %

This aspect of the study is of significant interest, as one of the primary objectives of the application is to accurately identify relevant code lines while excluding irrelevant ones.

The graphical output generated by the application is designed to enhance comprehensibility compared to manual inspection of the code. Furthermore, it is crucial for the output graph to be reliable and trustworthy. Since the current implementation only incorporates backtracking, the application accurately displays programs that do not forward input. In order to ensure a comprehensive display, it is deemed preferable to show a slightly larger number of lines rather than omitting potentially relevant ones.

5.2.4 Evaluation of the flow

Throughout the experimental phase, the application generated a total of 15 graphs as output. The evaluation of these graphs involved a manual inspection of the code for all test programs, discerning their flow, and subsequently comparing them to the corresponding generated graphs. In this section, four of these graphs are presented and discussed.

Fig. 5.1 illustrates the graph generated by the application for test program 2. The graph exhibits accurate call sequences and produces the expected output. This can be attributed to the absence of any forward calls within the program. The input function, `scanf`, is encountered in two functions: `main` and `enterNumber`. Consequently, these functions are positioned at the top of the graph. Additionally, the `main` function includes a call to `enterNumber`, which is depicted in the graph. Furthermore, the function `calcDouble` invokes `enterNumber`, and main calls `calcDouble`, resulting in the representation of this flow within the graph as well.

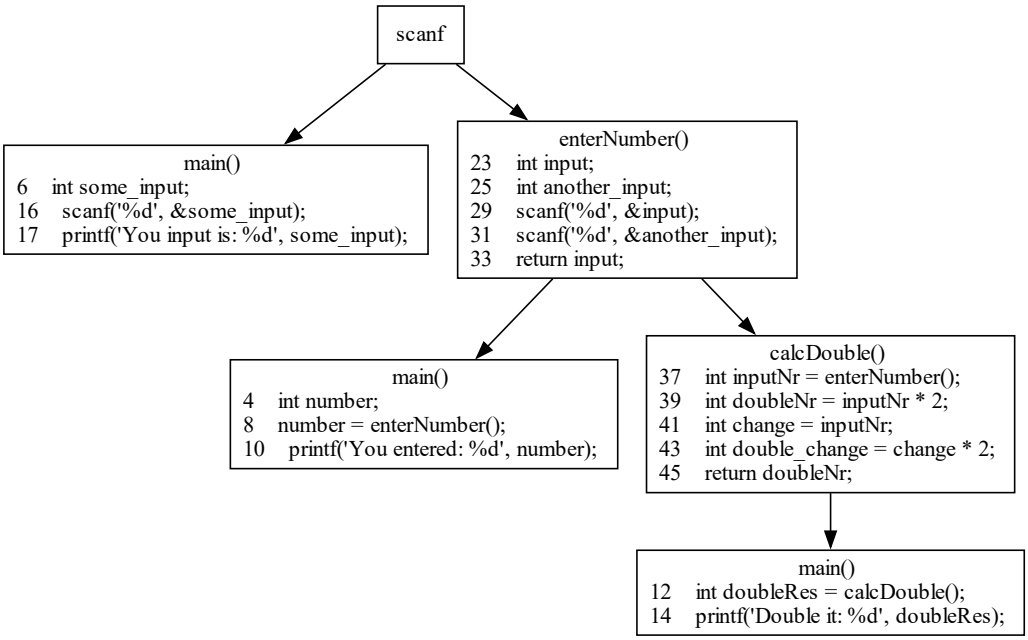

Figure 5.1: Graph generated for test program 2.

Fig. 5.2 portrays the output generated by the application when running test program 11. This graph accurately represents the function calls within the program, which can be attributed to the absence of any forward calls. The presence of multiple input functions in this program demonstrates the capability of the application to identify and handle such scenarios. Specifically, this program encompasses three distinct input functions. The top nodes of the graph highlight the presence of `scanf`, `gets`, and `getch`. The function `number` contains the input function `scanf` and is invoked by the `main` function. The function `string` incorporates the input function `gets` and is invoked by the `main` function. Similarly, the function `character` involves the input function `getch` and is also called by the `main` function.

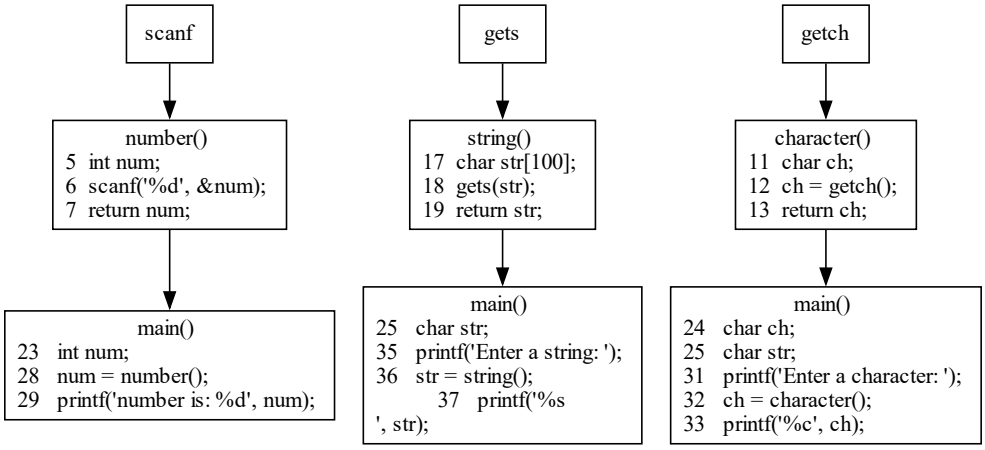

Figure 5.2: Graph generated for test program 11.

Fig. 5.3 illustrates the output generated by the application for test program 9. The resulting graph represents a concise structure wherein the input function `gets` is solely present in the `main` function. Upon reviewing the code within the `main` node, it becomes apparent that the input is forwarded to the `ManipulateCurrent` function, indicating the necessity for a forward call representation. While the graph accurately depicts the backtracking process, it lacks the implementation of forward tracking, thereby failing to capture the forward call connection between the `main` and `ManipulateCurrent` functions.

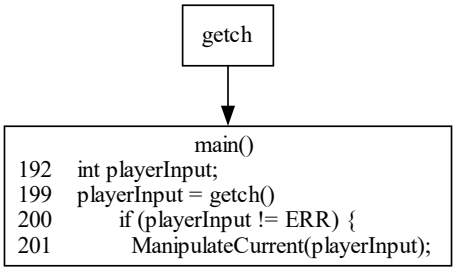

Figure 5.3: Graph generated for test program 9.

Fig. 5.4 provides an example representation of the desired output achieved through the implementation of the forward track algorithm. The presence of a red arrow denotes the flow originating from the forward track, indicating the direction of data propagation. Notably, the parameter `action`, derived from the `playerInput` function in the preceding function, is accurately tracked, ensuring the inclusion of the relevant code line for a comprehensive understanding of the information flow.

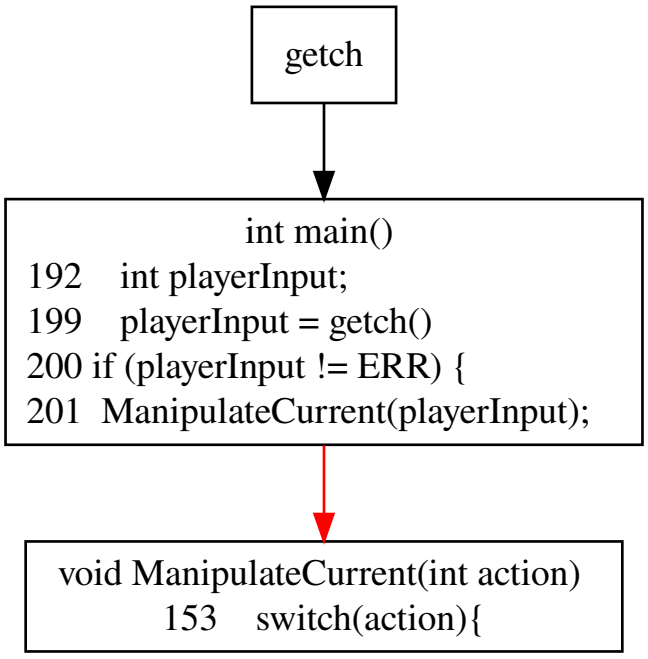

Figure 5.4: Complete flow of test program 9.

Fig. 5.5 exhibits the output graph generated by the application for test program 12. This particular program incorporates several forward calls; however, these connections are not displayed in the graph due to the absence of the forward track algorithm's implementation. Nonetheless, the backward track algorithm functions as intended.

In the program, the input function `scanf` is encountered in the `get_integer` function and is subsequently called by the `main` function. Additionally, `scanf` is also present in the `get_float` function, which is invoked from the `main` function. The function `get_char` involves the input function `getchar` and is similarly called by the `main` function. Upon closer examination of the code within the `main` node, it becomes apparent that forward calls are present, where variables containing input values are passed forward.

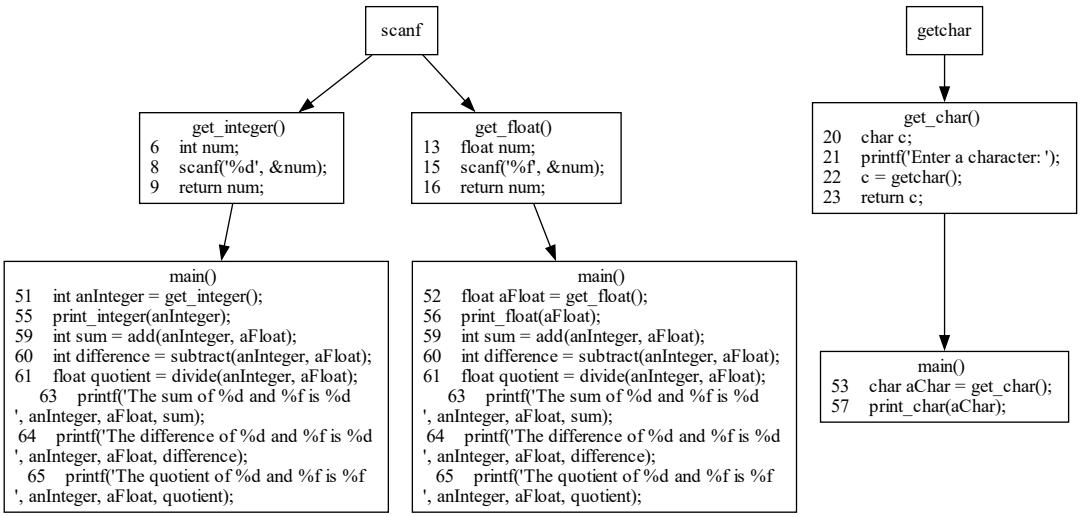

Figure 5.5: Graph generated for test program 12.

Fig. 5.6 serves as an illustrative example demonstrating the implementation of the forward track algorithm. This depiction underscores the non-trivial nature of combining both forward and backward tracking algorithms. The graph effectively portrays the forward calls made to the functions `print_integer`, `print_float`, `print_char`, `add`, `subtract`, and `divide`. Each of these functions contains a critical line of code that warrants inclusion for a comprehensive understanding of the input data flow. Within the graph, the red arrows signify forward calls without return values, while the blue arrows represent the returned values, facilitating a clear visualization of the information flow.

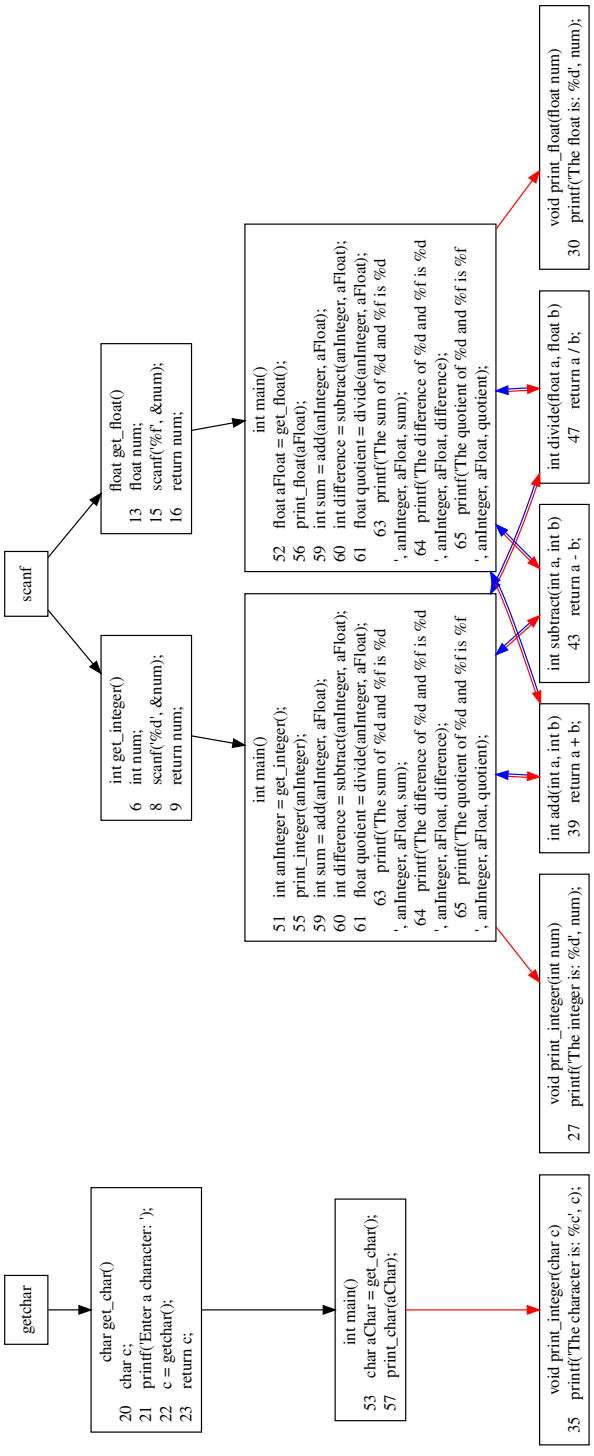

Figure 5.6: Complete flow of test program 12.

5.2.5 Result discussion

The application's performance can be inferred from the results presented in the tables, with each component manually counted within the test programs, as previously mentioned.

Tables 5.2 and 5.3 exemplify two fundamental functionalities of the application. The accuracy of the application in these specific aspects is of paramount importance, as any deviation from a 100 % accuracy rate would significantly undermine its reliability. The application's proficiency in identifying all functions within the code holds substantial significance, as any oversight in this process may result in the omission of crucial input data functions. Consequently, the ability to identify and capture all input data functions within the codebase is essential for ensuring the detection of potential vulnerabilities. As indicated in the tables, the application demonstrates excellent performance in this regard, specifically in the context of the small-scale programs on which it has been tested.

Table 5.4 exhibits certain outliers concerning the count of code lines. This is attributed to the absence of a functional forward track for data traversal in the application. However, the important code lines that would be presented if forward tracking were implemented were still taken into account during the manual counting process. This discrepancy is highlighted to acknowledge the flaw in this particular aspect of the application. Upon reviewing the results in Table 5.4, it becomes evident that the experiments with a 100 % accuracy rate pertain to test programs where the data traversal can be achieved solely through backtracking. The accuracy calculation disregards the fact that some lines are detected even if they are not considered relevant. This approach was chosen to account for the situation where the application may identify more lines than necessary, yet still successfully detect all relevant code lines. Conversely, the experiments with an accuracy rate lower than 100 % correspond to test programs requiring forward tracking to identify all relevant code lines. As the forward tracking functionality is not yet implemented, the accuracy falls short in these cases.

One of the crucial aspects of the application is accurately representing the flow of data. In this regard, the application exhibits strong performance in terms of backtracking, effectively capturing the backward flow of data. However, it becomes evident that the inclusion of forward tracking is essential to provide a comprehensive depiction of the data flow. The presented graphs vividly illustrate this requirement. Through manual inspection of the code for each program, it becomes evident that the backward flow is accurately represented by the application. However, the absence of forward track implementation results in the incomplete representation of the forward flow of data.

The conducted experiments reveal the potential of the application. It consistently achieves 100 % accuracy in detecting all functions and input functions within the small test programs. The reduction in the number of lines that need manual examination by a code reviewer is substantial in most cases. Across all test programs, comprising a total of 1,592 lines of code, the application successfully identifies 293 lines, resulting in an overall reduction of 81.6 % in the manual examination effort.

6 Discussion

This project had a primary objective of developing a proof-of-concept application aimed at streamlining code review processes. The main focus of the project was to investigate the feasibility of implementing a comprehensive tool and explore its potential applications. Specifically, the emphasis was placed on input data functions and the traversal of data within a program. The importance of input data in terms of program security became apparent when considering known vulnerabilities. Input validation emerged as a crucial factor in mitigating security risks, thereby underscoring the project's focus on input data functions.

The literature search section discussed the significance of OWASP (Open Web Application Security Project) as a prominent resource for documented security vulnerabilities. The OWASP cheat sheet provided valuable insights into input data validation, its importance, and recommended implementation strategies [6]. With this objective in mind, the application was designed to identify code sections that may require input validation and subsequently trace the flow of data by traversing backward through function calls. The application presents the findings in a comprehensive call graph format, accompanied by relevant lines of code associated with each function. It is crucial to note that the aim of the application is to provide assistance for manual code review rather than replacing it entirely.

In the subsequent sections of this discussion chapter, the performance of the proof-of-concept application will be evaluated, and a comparison will be drawn with existing code review tools. The limitations encountered during the development process will also be addressed. Through these discussions, the achievements of this project and its potential impact on code review practices will be elucidated.

6.1 Evaluation of the implementation

A backtracking algorithm was implemented to identify function calls within the codebase. Furthermore, a mechanism was devised to locate the relevant code lines associated with these function calls. The visualization of the results utilized the DOT language in conjunction with Graphviz, facilitating the creation of clear and comprehensive representations. The rationale behind this approach was to enable code reviewers to easily understand the flow and dependencies of data within the program, thereby facilitating efficient and effective code review processes. By presenting a visual representation of the data flow as a tree structure, the aim was to enhance the reviewer's comprehension of the code and its complexities. In the aforementioned paper, titled '*Facilitating program comprehension with call graph multilevel hierarchical abstractions*', the authors examine the limitations of static call graphs and propose mitigating approaches involving multi-level graph abstraction, hierarchical clustering algorithms, and varying levels of granularity [7]. The authors highlight that static call graphs often suffer from size inflation and lack of granular detail. Specifically, the paper emphasizes the challenges associated with comprehending call graphs representing entire programs. To address these concerns, multi-level graph abstraction is employed to reduce graph size by transitioning from package calls to class calls and ultimately to function calls. Additionally, hierarchical clustering algorithms are utilized to diminish the size of the graph by reducing the size of the execution paths and their complexity.

In the context of this project, static call graphs are employed to present the findings. However, an additional layer of granularity is introduced by incorporating relevant code lines for each function call. Furthermore, the graph size is minimized by exclusively tracing the paths of input data within the program. As a result, we are confident that program comprehension, specifically regarding input data, is enhanced, which is the key objective of this project.

6.2 Evaluation of the application

As discussed in the previous chapter's result discussion, the developed application exhibits promising potential. Although the reduction in the number of lines for code reviewers to examine may not be a perfect measure of time reduction, it serves as an indicator that the program effectively reduces the time and effort required for manual program analysis.

The application successfully identifies all input functions and can identify all relevant code lines if specifically requested, providing a focused view that encompasses fewer lines than the entire program. Notably, the backward algorithm performs admirably, yet it possesses a flaw in the absence of a termination mechanism. In its current implementation, the algorithm stops backtracking function calls only when there are no further calls remaining, regardless of the relevance of the return values. Ideally, the algorithm should cease backtracking when the data traversal concludes, such as when a function utilizes an input data function without subsequently returning any relevant value. This refinement would improve the algorithm's efficiency and accuracy. Moreover, the application occasionally identifies more relevant code lines than necessary due to its reliance on string-based algorithms for line gathering. Consequently, the algorithm may include lines that contain the search string as a substring, even if they are unrelated to the intended variable.

For example, if the algorithm searches for lines relevant to variables named `ch`, or `in`, it may capture lines such as:

- `char aChar;`
- `int anInt;`
- `printf("Enter an integer: ");`

The experiments indicate that the inclusion of a forward track algorithm is essential to fully visualize the flow of data and identify relevant code lines. Currently, when an input is passed forward in a program, none of the forward calls or associated code lines within the called function are detected. Forward tracking is a necessary implementation in order to get a complete comprehensive flow of data. However, Fig. 5.6 in chapter 5.2.4 illustrates problems that will surface when combining backward- and forward tracking algorithms. In a complex software program, it is essential to devise an effective approach for presenting the interconnected components in a manner that ensures comprehension without overwhelming the graphical representation. Utilizing distinct colors for arrows is one potential solution, although it may not suffice on its own. An alternative idea involves positioning the forward track to the left or right of the relevant function, as opposed to below it as demonstrated in the example. This arrangement would enable the downward continuation of the backward track, while allowing the forward track to extend

sideways. Nonetheless, implementing this approach may introduce readability concerns and potential complications. Despite this limitation, the application still enhances program comprehension and provides code reviewers with a starting point for analysis.

6.3 Comparisons with other tools

The purpose of this project's application is to facilitate the process of manual code review. Consequently, comparing it with fully automatic tools is not straightforward due to differences in how the application is utilized. Although fully automatic tools offer speed, they also possess the potential for generating false positives and false negatives. In the context of conducting a code review for a highly secure program, such inaccuracies are unacceptable, particularly with regards to false negatives. As a result, fully automatic tools are deemed less reliable, and code reviewers, particularly in the security sector, cannot depend solely on them. Given that our application aims to aid in manual code review, its design focuses on identifying all sections of the code that may pose security risks. Assuming a complete implementation of the application, it is expected to yield a significant number of false positives, but ideally no false negatives. This aspect holds paramount importance in the code review of programs with high security requirements.

Among the currently available tools for code review, Fortify is considered one of the most promising options for detecting security vulnerabilities. As stated in their product information available on their website (Fortify, n.d.[20]), Fortify offers a comprehensive solution capable of identifying 815 distinct vulnerabilities across 27 different programming languages. It incorporates a semi-manual review approach, whereby it highlights potential vulnerabilities within the code and provides explanations regarding their threatening nature. In this context, "semi-manual" refers to the application's ability to automatically identify these threats while offering a manual review process for resolving them. Fortify employs a sequence diagram to visually represent the data flow associated with each identified vulnerability. This mechanism operates in a similar manner to the call graph employed in our application, wherein relevant function calls and data traversal are presented as a sequence. Both sequence diagrams and call graphs have their benefits as presented in Table 6.1.

Table 6.1: Benefits of Sequence Diagrams and Call Graphs.

Sequence Diagram	**Call Graph**
Shows the order of interactions between objects in a system over time	Shows the relationships between different function calls within a program
Focuses on the messages exchanged between objects	Focuses on the functions in a program and how they interact with each other
Typically used to model a single use case or scenario	Provides a more abstract view of the potential execution paths of a program (static call graph)
Can show both the call and the reply	Can be useful for debugging, optimizing, and maintaining code

Both sequence diagrams and call graphs serve as valuable tools for understanding software behavior and structure. While sequence diagrams excel in illustrating specific scenarios, call graphs offer a better overview of a program as a whole. By leveraging call graphs, developers can gain insights into the program's structure, dependencies, and potential areas for optimization, facilitating more efficient and maintainable codebases. The call graphs that are presented with our application are, like sequence diagrams, illustrating specific scenarios. However, we posit that by incorporating crucial code lines and fully backtracked function calls, the comprehension of the program is enhanced.

Although a personal assessment of the Fortify application has not been conducted, available information and reviews suggest its commendable performance. However, it is worth speculating on the reasons why companies with stringent security requirements might choose not to adopt such applications.

We believe that one of the main reasons for not placing complete faith in automated tools, is the inherent lack of trust in the automated vulnerability detection process. Certain vulnerabilities escape detection by these applications that would otherwise be identified through a manual code review. False-negatives must not occur, especially for a company that deals with confidential and sensitive information. Another reason could be that many of these automated applications are hosted and executed within the company's cloud infrastructure. This would necessitate placing the code to be reviewed on the company's server for the purpose of executing the automated process. Considering the presence of confidential and sensitive information, such an approach is likely deemed unacceptable. This arrangement also necessitates regular updates, which entails establishing a connection between the host machine and the company's server to retrieve these updates. Given the sensitive nature of the information involved, relying on external servers for updates may raise concerns about the potential leakage of classified code, or other malicious intentions. These considerations may contribute to the hesitation of companies with high security demands in utilizing applications like Fortify for their code review processes. However, it is imperative to acknowledge that offline-capable automated code review applications serve as valuable support tools within the reviewing process.

6.4 Limitations and Challenges

Within the designated timeframe of the project, the inclusion of forward tracking functionality in the application proved to be unfeasible. As a result, the application currently only supports backward tracking. The theoretical section of this project elucidates the algorithm employed for forward tracking.

The present iteration of the application possesses limited capabilities, restricted to the identification of specific input functions, namely `scanf()`, `gets()`, `getch()`, and `getchar()`. However, the application lacks a mechanism to halt backward tracking in instances where a function receives an input but does not return a value. During the analysis of code rows, the program exclusively verifies the presence of a string, which could encompass either a variable or a function name. Consequently, the program is susceptible to identifying redundant rows due to the potential occurrence of the string as a substring within a row.

Considering the designated timeframe, the utilization of an existing parser such as pycparser was not pursued, as it was deemed more convenient to develop a custom string-based parser using Python. Pycparser, being a complex library, necessitates substantial expertise for proper utilization. Additionally, such parsers do not offer an optimal means of collecting pertinent code lines.

The primary objective of this project revolved around the creation of a proof of concept. Hence, these limitations are considered acceptable, as they still facilitate an evaluation of the concept's viability and potential for subsequent development.

7 Conclusion

In conclusion, our proof-of-concept application has yielded satisfactory results, thereby warranting further investigation into its potential for a full implementation. While there are already established vendors in the market offering similar tools, it is worth noting that our application distinguishes itself by serving as a complement to manual code review rather than aiming to fully automate the process. This distinction is particularly valuable in cases where manual code review is necessary. A full implementation of the application would aim to achieve zero false negatives, which holds significant importance.

The current implementation of the application utilizes a custom-built parser. However, it is preferable to employ an existing parser, such as pycparser. Pycparser is designed to parse C code and construct an Abstract Syntax Tree (AST), which would provide a more reliable foundation for traversing function calls and accurately identifying relevant variables. The adoption of pycparser would enable the application to incorporate both backward and forward tracking algorithms, thereby enhancing its reliability and overall performance. Furthermore, it is worth noting that while the application is designed to identify input data functions, its potential applicability extends beyond code review. It could potentially be utilized to locate any function within a program and perform function call backtracking, offering utility in various domains requiring program comprehension.

Overall, the results obtained from the proof-of-concept application demonstrate its viability and merit further exploration for a comprehensive implementation. By leveraging an established parser like pycparser, the application's reliability and effectiveness can be significantly improved. Nonetheless, the supervisors at Combitech expressed satisfaction with the demonstrated outcomes. Consequently, their response serves as validation for the application, affirming its status as a proof of concept that could merit further investigation.

7.1 Future work

The objective of this project was to develop a proof of concept application. However, for the implementation of a fully functional and robust application, it would be advisable to rebuild the application around an existing parser that offers more advanced features, such as the ability to build an Abstract Syntax Tree (AST). Such parsers are more reliable in identifying function calls, input data functions, and variables within the code. Furthermore, the implementation of a forward tracking algorithm is essential for a comprehensive visualization of the data flow. This algorithm was not fully realized in the proof of concept, but it is a crucial component to include in a complete application. While the current implementation utilizes DOT and Graphviz for graph visualization, these tools may prove challenging to use in a larger-scale implementation. For instance, to represent multiple function calls, a strategy was implemented to assign unique names to each node. However, combining this strategy with the attempted forward tracking algorithm led to some difficulties. Exploring alternative visual presentation tools or alternative methods of writing to the DOT file may be beneficial in a full implementation. Expanding the application to detect additional input data functions and other security vulnerabilities would also be necessary to make the application more useful. Nevertheless, the concepts of backtracking and code line search employed in the current implementation still hold value. The AST alone does not provide sufficient information to create a comprehensive graph of function calls or identify all relevant code lines. Therefore, the algorithms utilized in this project could serve as inspiration and be further explored in future work towards a complete implementation.

To develop a complete application, it is recommended to acquire in-depth knowledge of a well-documented and reliable parser. Careful consideration should be given to the overall program execution and thorough planning should be undertaken. While a set of requirements was initially defined, the exploratory programming approach used in this project led to situations where previously functioning components needed to be modified to integrate with other program elements. Such issues increase complexity and reduce flexibility, highlighting the importance of clear structural planning before commencing the implementation phase.

References

[1] G. McGraw, “Software security: Building security in,” in *2006 17th International Symposium on Software Reliability Engineering*, 2006, pp. 6–6.

[2] K. Goseva-Popstojanova, A. Perhinschi, “Capability of static code analysis to detect security vulnerabilities,” *Information and Software Technology*, vol. 68, pp. 18–33, 2015, December.

[3] A. Kesäniemi, “Manual vs automatic analysis,” https://owasp.org/www-pdf-archive/Ari_kesaniemi_nixu_manual-vs-automatic-analysis.pdf, 2023, accessed: May 17, 2023.

[4] Combitech, “Om oss,” accessed: Mar. 28, 2023. [Online]. Available: https://www.combitech.se/om-oss/.

[5] OWASP, “Code review guide,” 2016, accessed: Mar. 27, 2023. [Online]. Available: https://owasp.org/www-project-code-review-guide/.

[6] OWASP, “Input validation cheat sheet,” accessed: Apr. 27, 2023. [Online]. Available: https://cheatsheetseries.owasp.org/cheatsheets/Input_Validation_Cheat_Sheet.html.

[7] R. Alanazi, G. Gharibi and Y. Lee, “Facilitating program comprehension with call graph multilevel hierarchical abstractions,” *Journal of Systems and Software*, 2019, Dec.

[8] N. L. de Poel, “Automated security review of php web applications with static code analysis,” Master’s thesis, University of Groningen, Groningen, Netherlands, 2018, Feb.

[9] D. Evans and D. Larochelle, “Improving security using extensible lightweight static analysis,” *IEEE Software*, vol. 19, no. 1, pp. 42–51, 2002.

[10] G. McGraw, “Automated code review tools for security,” *Computer*, vol. 41, no. 12, pp. 108–111, 2008.

[11] OWASP, “Static Code Analysis,” accessed: Apr. 30, 2023. [Online]. Available: https://owasp.org/www-community/controls/Static_Code_Analysis.

[12] OWASP, “Injection Flaws,” accessed: Apr. 27, 2023. [Online]. Available: https://owasp.org/www-community/Injection_Flaws.

[13] OWASP, “Buffer Overflow Attack,” 2023, accessed: Apr. 27, 2023. [Online]. Available: https://owasp.org/www-community/attacks/Buffer_overflow_attack.

[14] D. Cruz, “AST (Abstract Syntax Tree),” Oct. 2018, accessed on: Apr. 24, 2023. [Online]. Available: https://medium.com/@dinis.cruz/ast-abstract-syntax-tree-538aa146c53b.

[15] Wikimedia Commons, “File:Abstract syntax tree for Euclidean algorithm.svg,” https://commons.wikimedia.org/wiki/File:Abstract_syntax_tree_for_Euclidean_algorithm.svg, 2021.

[16] E. Bendersky, “Pycparser documentation,” 2023, accessed: Mar. 30, 2023. [Online]. Available: https://github.com/eliben/pycparser.

[17] “Dot language,” 2022, accessed: Mar. 30, 2023. [Online]. Available: https://graphviz.org/doc/info/lang.html.

[18] “Graphviz homepage,” 2021, accessed: Mar. 30, 2023. [Online]. Available: https://graphviz.org/.

[19] M. Santos, “False Positives or False Negatives: Which is worse?” accessed: May 10, 2023. [Online]. Available: https://towardsdatascience.com/false-positives-vs-false-negatives-4184c2ff941a.

[20] Micro Focus, “Fortify homepage,” 2023, accessed: May 21, 2023. [Online]. Available: https://www.microfocus.com/en-us/cyberres/application-security.

[21] Khan Academy, “Experiments and observations,” 2023, accessed: Mar. 30, 2023. [Online]. Available: https://www.khanacademy.org/science/biology/intro-to-biology/science-of-biology/a/experiments-and-observations.

[22] viktormollerstrom, “Test-Programs,” https://github.com/viktormollerstrom/Test-Programs/tree/a0c1446057fb39ddcb24bdb21585f4fe6caf2b0f/testprograms, 2023.

[23] G. Najib, “Tetris200lines,” 2021, accessed: May 10, 2023. [Online]. Available: https://github.com/najibghadri/Tetris200lines.

[24] GeeksforGeeks, “Snake game in c,” 2020, accessed: May 10, 2023. [Online]. Available: https://www.geeksforgeeks.org/snake-game-in-c/.

[25] OpenAI, “Gpt-3.5-turbo,” ChatGPT, an AI language model, 2023. [Online]. Available: https://openai.com

A Appendix 1, Output graphs

This appendix encompasses the collection of graphs generated by the application during the execution of the test programs.

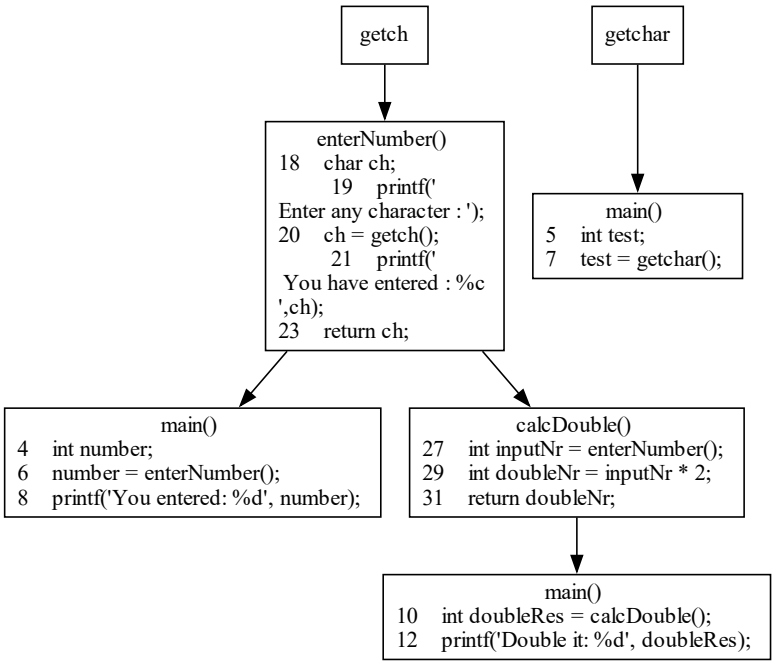

Figure 1.1: Graph for testprogram 1.

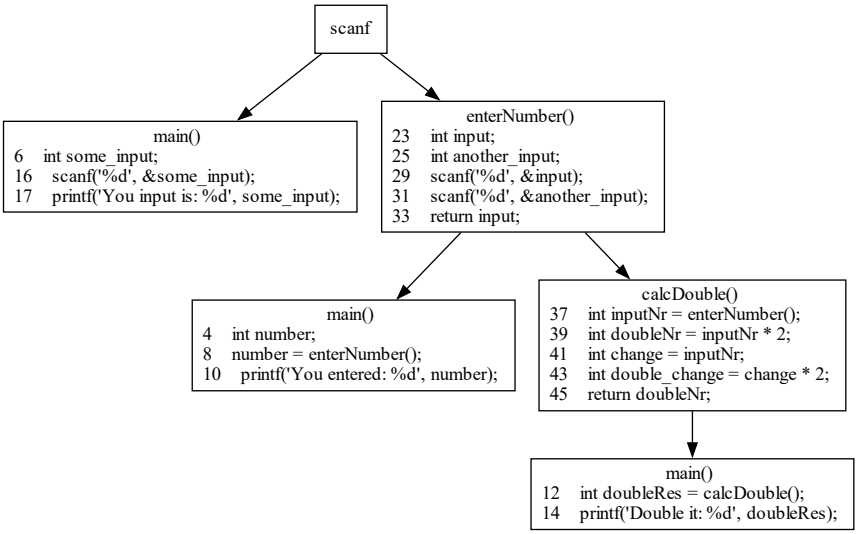

Figure 1.2: Graph for testprogram 2.

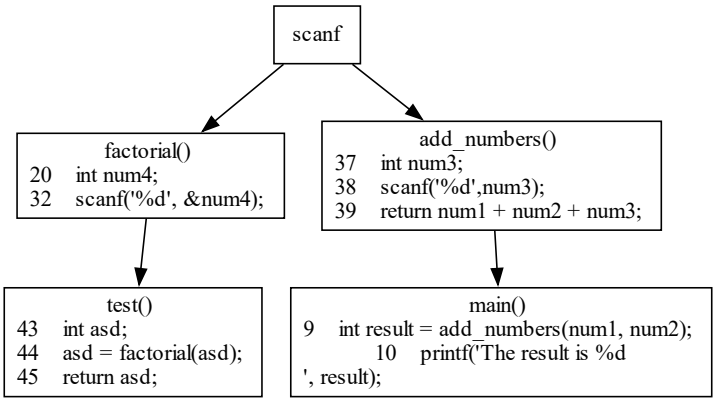

Figure 1.3: Graph for testprogram 3.

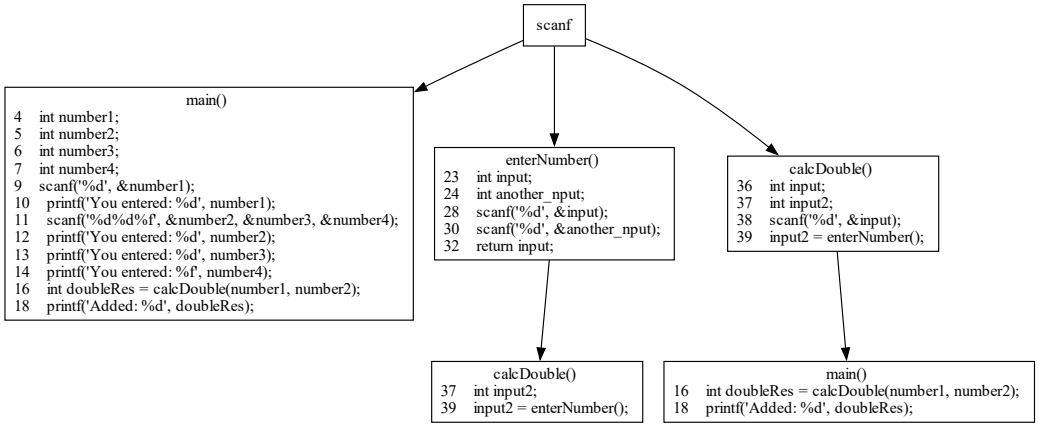

Figure 1.4: Graph for testprogram 4.

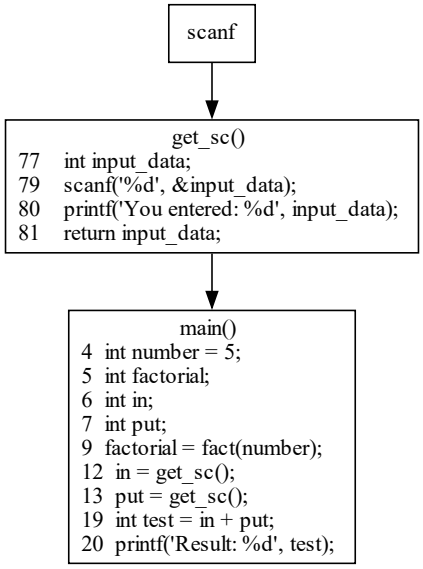

Figure 1.5: Graph for testprogram 5.

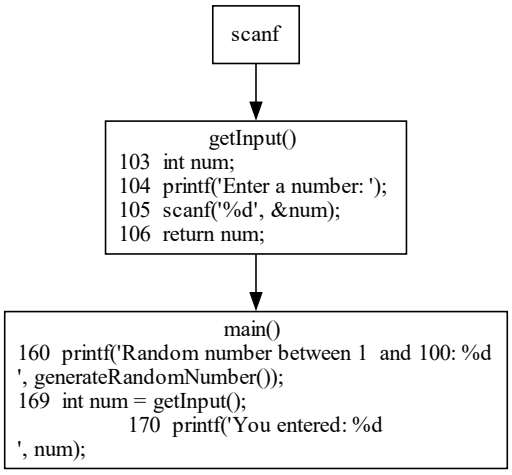

Figure 1.6: Graph for testprogram 6.

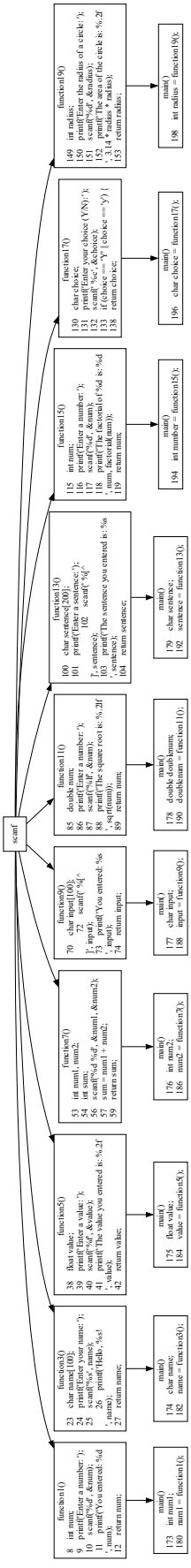

Figure 1.7: Graph for testprogram 7.

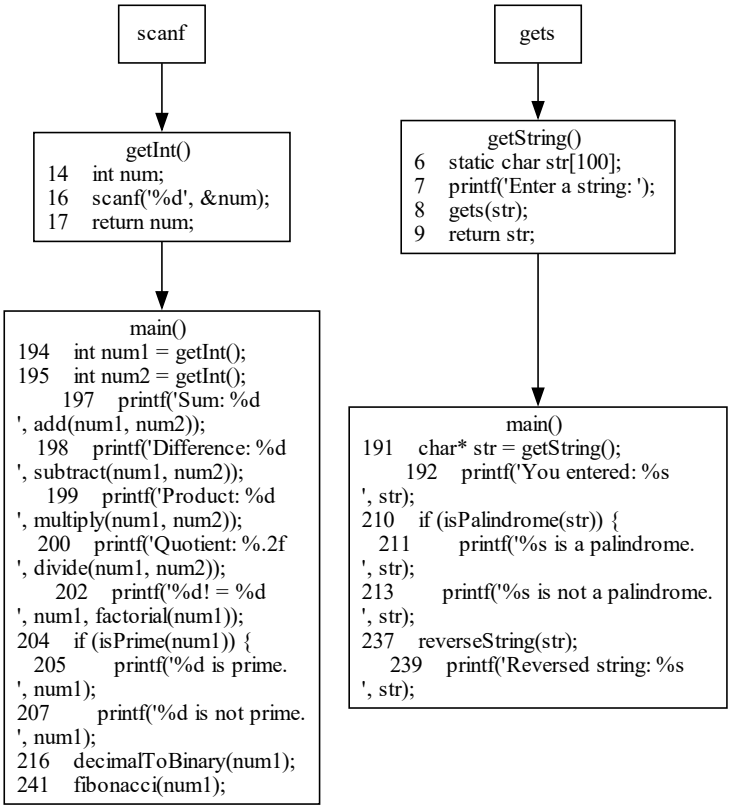

Figure 1.8: Graph for testprogram 8.

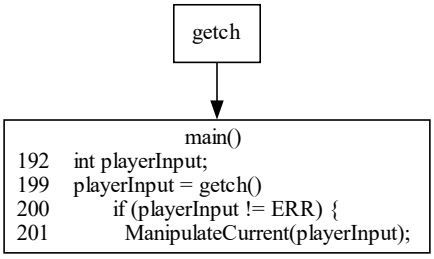

Figure 1.9: Graph for testprogram 9.

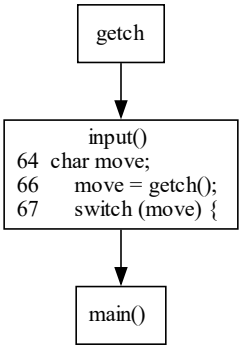

Figure 1.10: Graph for testprogram 10.

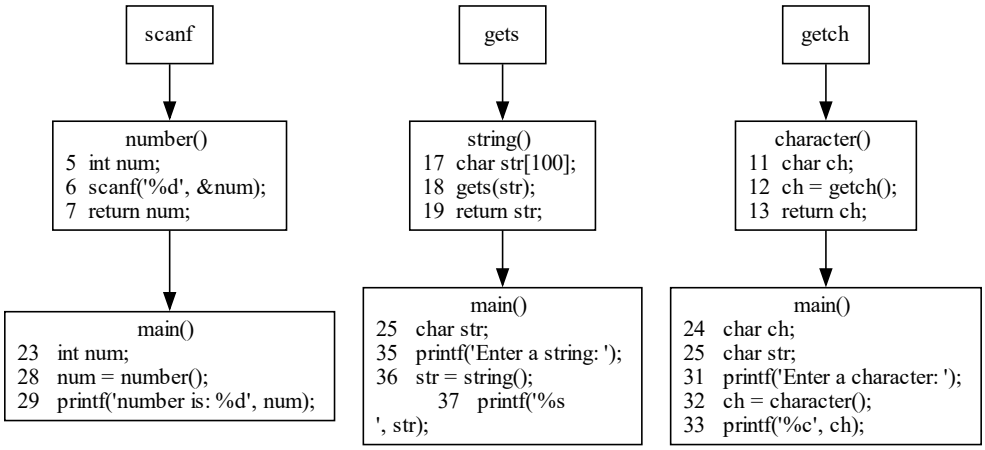

Figure 1.11: Graph for testprogram 11.

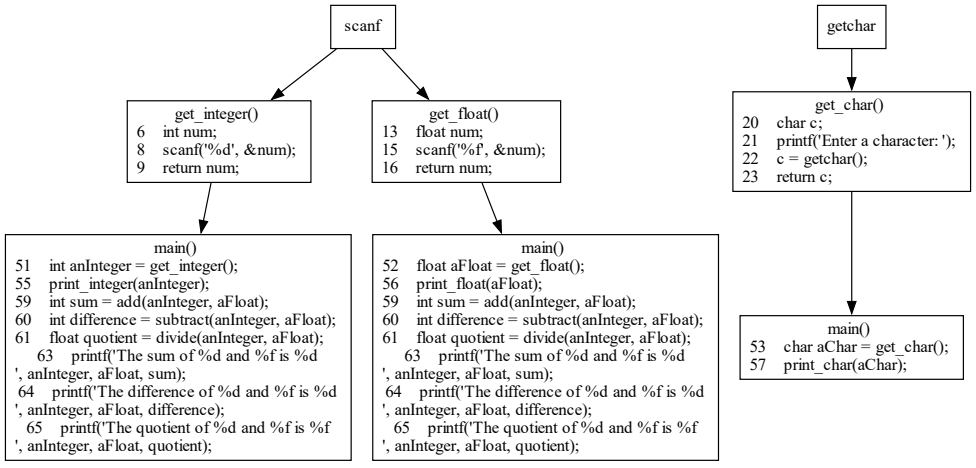

Figure 1.12: Graph for testprogram 12.

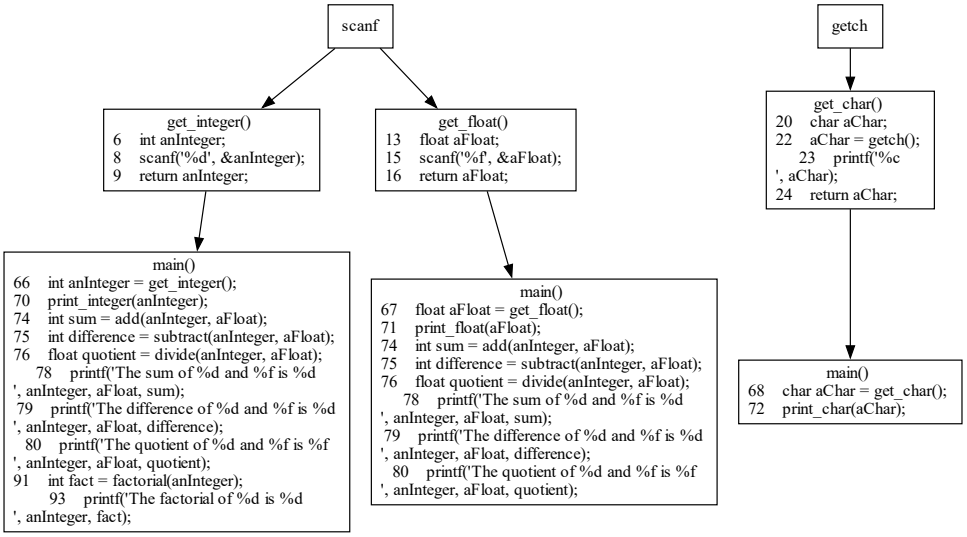

Figure 1.13: Graph for testprogram 13.

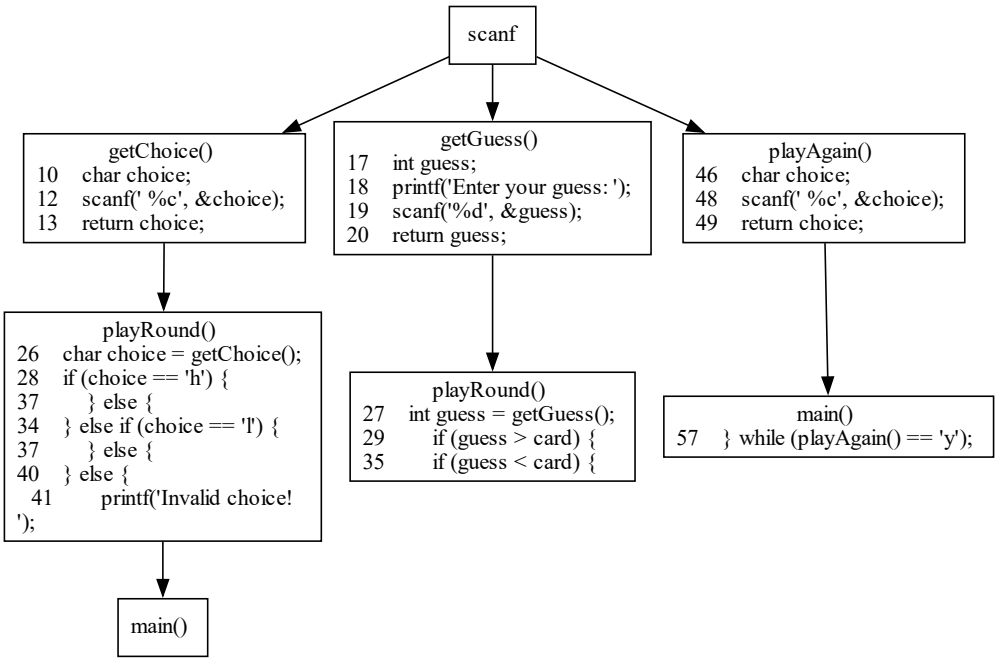

Figure 1.14: Graph for testprogram 14.

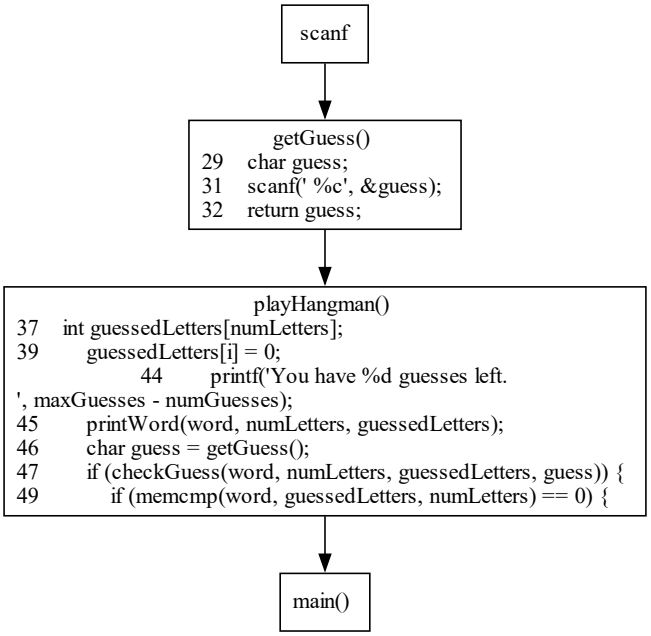

Figure 1.15: Graph for testprogram 15.