Bachelor Degree Project

Secure Application Development

Author: Alaa Alwan
Supervisor: Jesper Andersson
Semester: VT 2021
Subject: Software Technology
Abstract

Security testing is a widely applied measure to evaluate and improve software security by identifying vulnerabilities and ensuring security requirements related to properties like confidentiality, integrity, and availability. A confidentiality policy guarantees that attackers will not be able to expose secret information. In the context of software programs, the output that attackers observe will not carry any information about the confidential input information. Integrity is the dual of confidentiality, i.e., unauthorized and untrusted data provided to the system will not affect or modify the system’s data. Availability means that systems must be available at a reasonable time. Information flow control is a mechanism to enforce confidentiality and integrity. An accurate security assessment is critical in an age when the open nature of modern software-based systems makes them vulnerable to exploitation. Security testing that verifies and validates software systems is prone to false positives, false negatives, and other such errors, requiring more resilient tools to provide an efficient way to evaluate the threats and vulnerabilities of a given system. Therefore, the newly developed tool Reax controls information flow in Java programs by synthesizing conditions under which a method or an application is secure. Reax is a command-line application, and it is hard to be used by developers. This project has its primary goal to integrate Reax by introducing a plugin for Java IDEs to perform an advanced analysis of security flaws. Specifically, by design, a graphical plugin performs advanced security analysis that detects and reacts directly to security flaws within the graphical widget toolkit environment (SWT). As a second important goal, the project proposed a new algorithm to find the root cause of security violations through a graphical interface. As a result, developers will be able to detect security violations and fix their code during the implementation phase, which reduces the costs.

Keywords: secure development, application security, static application security testing, SAST
1 Introduction

In this 15-credit BSc thesis, we examine software security and secure programming aspects. More specifically, software security when programming in the programming language Java. Leakage and data integrity are classes of application vulnerabilities and a massive problem for security. Attackers can utilize such vulnerabilities to steal sensitive information and hack companies and users, e.g., stealing the personal records of 233 million users of eBay [1]. The damage caused to any organization could be severe. For instance, it could affect revenue and reputation and expose companies to lawsuits. This problem should be considered during the life-cycle, especially in the development phase (by secure code development). Two aspects of security are:

- Controlling information flow to a system to block and validate untrusted data.
- Controlling information from a system to avoid information leakage.

Considering some of the most catastrophic security incidents in the past years, it explains why developers need a more helpful tool to understand the vulnerabilities made by developers and administrators. There will be advances along these lines the suggestion that security vulnerability would be helped an active defense, employing a specifically designed application security testing tool, a vulnerability detection plugin, and a detection algorithm in this security direction. It is not intended to be presented as a standing model but rather as an application of Static Application Security Testing (SAST), allowing scans at the code level and finding flaws early in the development process. Developers are too often neglected in support tools compared to what is available to ordinary users. Developers have more complex security challenges than ever and need all the help they can get.

Reax is developed as a specific security tool to control information flow in Java programs by synthesizing conditions under which a method or an application is secure. The problem is that Reax does not provide enough information about the root cause of security violations. Moreover, Reax produces a complex security guard (conditions under which a method or an application is secure). The main goal of this project is to produce a root cause analysis algorithm to find the root cause of violations and simplify the produced security guard to be usable by developers. To demonstrate the result, we produce a plugin to integrate the engine with Java IDEs and help developers check the security of their code employing the core engine. The plugin is called Secure_Development_Plugin, which works with an engine that is implemented using Soot Code Analyzer and Reax to analyze Java programs at the byte-code level.

As a result, the project develops a tool (as a prototype) that assists developers engaged in the most varied tasks in a dynamic environment in performing SAST operations using Reax. The project implemented the prototype by integrating Reax with Java IDEs. As further assistance on tasks that will always remain a work in progress, one step ahead of failure, a root cause analysis will be advanced to find the reasons for security violations. Hopefully, this will give developers a deeper understanding of the nature of the security violations and make finding solutions more efficient.

1.1 Background

In information technology, a vulnerability is a flaw or weakness in code or design that creates a potential point of security compromise for an endpoint or network [2]. One or more attackers exploit a vulnerability to violate the system’s security policy [2].
In the early 2000s, personal computers became an essential part of our lives. Similarly, the Internet has become widely used. Thus, the use of internet applications has become widespread. This fact has led to an increase in the number of attackers. These attackers exploit vulnerabilities in applications. As a result, secure development has become an integral part of the program development process [3].

Web applications are vulnerable to violations. Those programs are highly vulnerable to hacking through many attacks [4]. In many cases, developers fail to protect confidential information from leakage. Moreover, developers fail to protect their applications from untrusted inputs in many cases. Here are the most critical web application security risks for 2021 [5]:

- Broken Access Control
- Cryptographic Failures
- Injection
- Insecure Design
- Security Misconfiguration
- Vulnerable & Outdated Components
- Identification & Authentication Failures
- Software & Data Integrity Failures
- Security Logging & Monitoring Failures
- Server-Side Request Forgery

Secure development helps in avoiding such violations by performing security testing. The testing should be performed during the development phase, not as the final development step. The testing confirms that the security controls run as expected. The testing includes the original security requirements, common security issues, and specially developed security controls [6]. Secure development helps developers detect vulnerabilities in the early stages of development and avoid the high costs of later security flaws detection and repair.

Secure Development Lifecycle (SDL) [7] is a way to avoid the introduction of security vulnerabilities, defects, bugs, and logic flaws which are the primary cause of software vulnerabilities. Attackers commonly exploit these vulnerabilities [7]. SDL is a standard approach where security should be a top priority during the software lifecycle [8]. An SDL is a process that consists of many phases. Companies produced a lot of SDLs. Most SDLs include the same primary security phases [8]. The standard SDL includes requirements, design, implementation, test, and release. The standard SDL is similar to the waterfall approach [8].

As a result of Bill Gates’s memo, Microsoft unleashed SDL. SDL became mandatory at Microsoft in 2004 [8]. Many other companies used the same approach as Microsoft’s SDL processes or created their own.

1.2 Related work

Several previous research has been conducted on cyber security and software vulnerability to find effective ways to help programmers code securely. Studies proposed various algorithms and models to help identify insecure code. Zhang and Myers proposed a new algorithm called SHErrLoc to help programmers identify errors [9]. SHErrLoc introduces a general way to find programmer mistakes by performing static and information flow analysis. The analysis searches for unsatisfiable constraints in programmers’ code. Then it tries to identify the program expressions most likely to cause unsatisfiability. A
The developers need to enforce security constraints at the coding level to control the information flow [10]. Pullicino proposed a java extension tool called JIF to help the developer identify common security issues during the software development life cycle. JIF is a security-typed programming language that extends Java. It supports information flow control and access control, enforced at compile and run time. The developer can specify confidentiality and integrity policies for the various variables in their code.

During coding, developers need to perform optimizations and remove unnecessary synchronization and perform stack-based allocation of objects. This approach helps understand what object addresses that reference variables store. Giiffhorn and Hammer introduced JOANA to perform Points-to analysis, which optimizes the Java compilers [11]. JOANA is a program analysis infrastructure for the Java language. It is available as a plugin for Eclipse. The developer can navigate through dependence graphs for the entire Java bytecode. It uses algorithms for language-based security to check programs for information leaks.

Applications developed by various programmers often integrate services from different providers using third-party code. However, this opens a loophole as the third-party code may not respect the developed application’s security and privacy. As a result, information flow tracking for secure information flow. In their study Hedi, Bellow and Sabelfeld proposed [12]. The tool was used to track the use of sensitive information in web applications such as the browser. It is a security-enhanced JavaScript interpreter for fine-grained tracking of information flow. The purpose of JSFlow is to enforce information-flow policies for the whole JavaScript language and track information in the presence of libraries.

The main problems of these tools:

- They often produce false-positive or false-negative results. E.g., a study of the use of SherLock to identify synchronizations found that the feedback information showed incorrect synchronizations [13]. Every false synchronization recorded represents an actual problem that the tool missed to identify. Similarly, JSFlow requires a balance of design complexity and usability, allowing permissive enforcement and library complexity. The algorithm is imprecise; hence it rejects too many secure programs [14]. Likewise, JOANA has the limitation of giving out false positives when the Information Flow Control (IFC) technique is used to evaluate interference [15]. An evaluation was conducted by FILHO [15], using a System Dependence Graph (SDG) in JOANA to conduct information flow in Java programs. The evaluation determined that in 64% of the cases evaluated, there was information flow in merged programs from same-method contributions by developers, but in manual analysis of 35 of these cases with information flow, interference was recorded in 15 of them only.

- They cannot find every class of vulnerability. E.g SHErrLoc uses Bayesian prediction and constraint graph hence less precision in error identification [16]. The model is more likely to identify only the compiler-generated constraints.

- Most of these categories of approaches introduce no information flow. The above algorithm does not predict violations that cause a leak of sensitive information.
As a result, it is challenging to enforce countermeasures to avoid leakages [17]. The complexity of tools such as SHErrLoc, and JOANA, among others, makes it difficult for the developers to predict the sources of security violations. In contrast, this project uses SCFG (a result of SootCodeAnalyzer) to find the root cause of security flaws via the proposed algorithms.

- As mentioned, some of them use algorithms for language-based security to check programs for information leaks. For instance, JOANA only analyses Java programs. JID is a Java programming extension, while JSFlow tracks information in JavaScript. While the proposed algorithm in this project is not related to a specific language, the same approach can apply to all languages.

For that, an additional tool is built to detect vulnerabilities. The tool is called Reax. Reax is built on a proven approach to detect vulnerabilities [18]. Reax is produced to compute a maximally permissive controller by converting all variables to Boolean. It generates security guards to monitor code using symbolic control flow graphs. Moreover, Reax can compute controller handling numerical aspect. The problem is that Reax does not provide enough information about the root cause of security violations. Moreover, Reax produces a complex security guard (conditions under which a method or an application is secure).

1.3 Problem formulation

It is often not efficient to depend on developers to fix applications’ security vulnerabilities. For instance, it is time-consuming to find and fix each vulnerability, and the lack of experience of developers. Reax is a synthesis tool that synthesizes security guards under which a method/application is secure. Reax does not provide enough information about the reasons for security violations. This project aims to introduce a root cause analysis algorithm to find the reasons for violations and demonstrate the result by integrating Reax and the introduced algorithm with Java IDEs. This project produces a plugin (as a prototype) to help developers check their code’s security. The plug-in are implemented and called SecureDevelopmentPlugin. The plugin works with an engine that uses SootCodeAnalyzer and Reax to analyze Java programs at the byte-code level. The plugin will help perform advanced checks on developer code to identify the source of security flaws using a graphical interface. The graphs make it easier for developers to understand the security flaw and fix it easily.

The engine is already implemented and consists of SootCodeAnalyzer and Reax. The engine analyzes the security of programs and source code. Two of the analysis results are SCFG and the security guard. This project analyzes the engine’s results. The project integrates this engine with Java IDEs, simplifies the produced security guard to be usable by developers, and analyzes SCFG (a result of SootCodeAnalyzer) to find the root cause of security flaws via the proposed algorithms.

1.4 Motivation

During the software development lifecycle, it is important to perform static application security testing (SAST), the efficiency of which depends on the tools used. For this purpose, Reax is developed. Reax makes use of formal methods to provide formal guarantees making them dependable and promising. The problem is that Reax does not provide enough information about the root cause of security violations. Moreover, Reax produces...
The main goal of this project is to produce a root cause analysis algorithm to find the root cause of violations and simplify the produced security guard to be usable by developers. Another contribution of this project is to integrate Reax with Java IDEs. As a result, developers will be able to perform SAST using Reax without leaving the development environment.

### 1.5 Objectives

| O1 | Develop and implement a new algorithm to find the root cause of security violations. |
| O2 | Find a solution to simplify the result of Reax (the security guard) to be readable and usable by developers. |
| O3 | Demonstrating the solutions by Integrating Reax and the algorithm in the development environment, i.e., as a Plug-in. |

### 1.6 Scope/Limitation

There are many approaches and algorithms to detect software security flaws. Therefore, it is impossible to implement and compare all possible methods and algorithms. The project is essentially built to be compatible with Eclipse. Then it is extended to be compatible with IntelliJ. However, extending the plugin to be compatible with any Java IDE is easy. Therefore, the plugin is limited to Java IDEs.

Unfortunately, within the period allocated to the project, it is not easy to provide a stable version in the practical field and software building. Therefore, the work that is done in this project can be tested only with simple code. However, this report explains how developers can extend that work to run the tool on more complex code.

The input for the introduced algorithm (that finds the root cause of security violations) is SCFG generated by SootCodeAnalyzer (the tool that prepares Reax’s inputs). Therefore, the algorithm is limited to using tools that generate such SCFG for its outputs. SCFG is explained in Section 2.

### 1.7 Target group

The primary target group of this project is software development companies concerned with improving software security or producing software that is immune to hackers. This target group could also include individual researchers and developers interested in this field.

### 1.8 Outline

Section 2 defines the various tools and methodologies employed in this project, including Security Control Flow Model (SCFG), Reax, and Eclipse. Section 3 outlines the specific methodology employed to complete the project. Section 4 illustrates the proposed algorithm to find the root cause of security violations, in addition to illustrating the solution to simplify the result of Reax (the security guard). Section 5 illustrates the specific architecture and implementation of a Reax support Secure_Development_Plugin. Section 6 outlines the result and analysis of the project. Section 7 provides a discussion of the project. Section 8 Conclusion.
2 Background

Information flow control (IFC)

Two aspects of Security are confidentiality and integrity. A confidentiality policy guarantees that secret information will not be exposed to attackers. In the context of software programs, this means that the output observed by the attackers is not influenced by the input confidential information [19]. Integrity is the dual of confidentiality, i.e., the data is not modified by unauthorized and untrusted data provided to the system [20].

Information flow control (IFC) is producing and applying a policy that regulates information flow to prevent attackers from inferring or discovering confidential input data by observing system outputs [19]. In addition to enforcing integrity by tracking the tainted and untainted status of variables throughout the control flow of the application [20]. Furthermore, several techniques have been proposed to enforce information flow policies.

SAST-tools can detect many types of vulnerabilities. For instance, SAST-tools perform an analysis to track the tainted and untainted status of variables throughout the control flow of the application [20]. A vulnerability will be reported if a tainted variable is used in a sensitive statement. Similarly, SAST-tools can also track the confidential and non-confidential status of variables throughout the application’s control flow and report a vulnerability if a confidential variable is used in a sink statement (function or method that could cause an information leakage to its parameters).

Currently, there are multiple approaches to restrict the unintended disclosure of data. E.g., access control and cryptography. These mechanisms limit the release of the information [19]. Mainly, two classes of approaches exist [21]. The first approach is a run-time artifact that connects the data with information flow labels developed in programming languages as well as the operating system level [21]. The second mechanism is a static mechanism for analyzing data flow [21].

The tool Reax has been developed to perform static analysis. Reax is a tool to perform a discrete controller synthesis for logico-numerical programs via abstract interpretation [18]. To analyze security, Reax follows a new approach. The new approach has proposed utilizing a permissive monitor using boolean supervisory controller synthesis that observes a Java program at certain checkpoints and predicts security flaws [17]. A permissive monitor is required for the following reason. Consider a method with two branches where the first branch is secure, whereas the second is not secure. Security-type systems, one of the main techniques for static analysis, reject this program completely. As opposed to the new permissive monitor, which shows that the method is secure under specific circumstances [17].

The new approach relies on boolean supervisory controller synthesis [22] to produce a monitor that monitors a program written in Java or its subset languages at certain checkpoints. In addition to producing an executable model of the monitored program. The executable model includes only observation points and checkpoints [17]. It is proved that the method is correct and enforces localized delimited release [23]. The new approach can expect future information flow for Java programs at a predefined checkpoints [17].

2.1 Security Control Flow Graph (SCFG)

The new approach follows the Jimple IR generated by Soot and assigns a security type to each variable. The security type of a variable may change based on the information flow (i.e., statements in Jimple). The set of security type is \{H, L\} or \{HIGH, LOW\}. SCFG shows the security semantics of a program is represented by $G = (L, V, I, l_0, v_0, \Delta)$ as
illustrated in [17]. $L$ are the set of configurations. $V$ the state variables. $I$ the uncontrollable inputs. $\Delta$ is defined using specific rules [17]. Configurations is defined as a stack $\sigma_0: \ldots: \sigma_n$ of currently active contexts. $\sigma_k$ where $0 \leq k \leq n$ represents a statement in a method or block of instructions. An uncontrollable input is an input that not be prevented from occurring in a system. Controllable inputs are inputs that the controller issues to control the system behavior.

Figure 2.1 is an example of simple code and the generated SCFG. According to the generated SCFG, $x$ inherits $parA$, and then $y$ inherits $x$. As a result, if $parA$ is a confidential parameter, $y$ will carry sensitive information. Therefore, at location 2 (print($y$)), will be a leakage for sensitive information.

![Figure 2.1: SCFG and Security updates.](image)

In the proposed algorithm in this project, SCFG is one of the algorithm’s primary inputs. The algorithm will utilize security updates in SCFG to find the root cause of security flaws.

### 2.2 Monitor Synthesis

In this section, the two steps of Monitor Synthesis will be explained.

#### 2.2.1 Step 1- Generating Checkpoint Security Guards

The program is insecure if it arrives at an observation point and the policies have been violated at that point [17]. An observation point is either a third-party method call or the exit point of an un-executed branch where the other branch contains a third-party method call [17]. Each branch guard should be propagated along its path to its controlling checkpoint to obtain the security guards for a program or a method, [17].
2.2.2 Step 2- Monitor Construction

In the checkpoints, if the security guards produced in the first step allow the execution program to continue its execution to the next checkpoint. Otherwise, action will be applied to prevent the violation [17]. The new approach aims to develop a tool, Soot Code Analyzer (SootCodeAnalyzer), to monitor the execution of Java programs at specific checkpoints dynamically and statically. **Dynamically**, the tool detects the violations and unsafe branches during the execution of the program and applies suitable countermeasures. The tool can apply the countermeasure automatically if there is a preferred solution. Otherwise, it asks the user to make a decision. **Statically**, the tool analysis is done after coding and before executing the assigned codes. Along the lines of this thesis and its limitations, given that there follow coding and its verification and not a system in dynamic use. At issue, therefore, is the source code, looking for errors and such non-complying rules to ensure proper coding standards and conventions are used to construct the program.

The main dependencies for SootCodeAnalyzer are Reax and Soot. The steps of the static analysis (figure 2.2 presents the steps):

- Transform the java code to Jimple code using “Soot”.
- Use “Soot” again to produce a control flow graph of each method.
- SootCodeAnalyzer produces a Security Control Flow Graph (SCFG) for each method.
- SootCodeAnalyzer produces the input for the Reax tool.
- SootCodeAnalyzer sends the input to the Reax tool and receives the result.
- The result will be saved in a map object as “Propagated Guards”.

![Figure 2.2: Steps of the static monitoring shown as a graph.](image-url)
2.3 Code Handling

To understand how the tool manipulates methods, the methods will be represented as boxes and arrows connecting them. Each box represents a basic block (a group of statements where there is no jump statement).

To illustrate that, consider the method and its flow chart 2.3. We can identify two jump statements (“if” statements). We can represent the method by keeping the boxes and writing the conditions of the “if” statements on the tracks directly, as in Figure 2.4. The execution can follow one of four tracks. By looking at the last figure, it is clear that there is no jump statement in the boxes anymore. Therefore, it is possible to compile all statements of each track in one box as in Figure 2.5.

```java
public void method1(int a, int b) {
    int x=1;
    int y=2;
    int z=a+b;
    if (z>2000) {
        print(z);
    }
    if(b<1000) {
        print("no enough money");
    }
    x=a+z;
    a=b;
}
```

![Flowchart](image-url)

Figure 2.3: Method with jump statements, four branches.
There are four tracks. The first three tracks have at least one "print" statement. The question is; assuming that "a" and "b" are confidential information, is the given method secure?

The print statements in the first three tracks give the attacker valuable information. For instance, if the program prints "no enough money" without printing the value of z, an attacker would be able to conclude that a+b<2000 and b<1000. This behavior leads to another question Is the fourth path secure?

The fourth path does not have a print statement. The fourth path is secure if the other paths do not exist. However, because of the other paths, the fourth path also becomes insecure. In the given method, if the program did not print anything, the hackers will be able to conclude that a+b<2000 and b>1000. As a result, the given method as a whole is insecure.
To motivate the developer to use the static monitoring tool, the developer should be able to integrate the tool with the commonly used java programming programs like Eclipse.

2.4 The Provided Engine (Reax)

At the beginning of this project, the provided engine consisted of two tools (SootCodeAnalyzer and Reax). **SootCodeAnalyzer** is the tool that is used to prepare Reax’s inputs. In this module, SootCodeAnalyzer generates SCFG for the given method. SootCodeAnalyzer analyzes each statement in the method. If the statements are not an invocation of another method, SootCodeAnalyzer analyzes the statements according to their behavior (the behavior differs according to the statements). Whereas if the statement is a method (i.e., an invocation to another method), SootCodeAnalyzer will replace the method with its body. As a result, the generated SCFG will not contain any invoked method.

There is much-related work in this area. The main problems with these tools are that (i) They often produce false-positive or false-negative results. In addition, they cannot find every class of vulnerability. In contrast, Reax is a tool that is built on a proven approach to detecting vulnerabilities. (ii) Most of these categories of approaches introduce no information flow. In comparison, this project uses SCFG (a result of SootCodeAnalyzer) to find the root cause of security flaws via the proposed algorithms. (iii) Some of them use algorithms for language-based security to check programs for information leaks. While the proposed algorithm in this project is not related to a specific language, the same approach can apply to all languages.
2.5 Objectives of a solution

This project has three objectives:

- Develop and implement a new algorithm to find the root cause of security violations. That is because Reax does not produce enough information about the root cause of security violations in a program 1.3. The development of this new algorithm will address this.

- Find a solution to simplify the result of Reax (the security guard) to be readable and usable by developers. That is because Reax produces a complex security guard 1.2. This project will make the newly produced security guard simple to use by program developers.

- Demonstrating the solutions by Integrating Reax and the algorithm with Java IDEs produces a plugin (as a prototype) to help developers check and identify the security of their code during coding. The plug-in are implemented and called SecureDevelopmentPlugin.

As a result, the project will produce a Static Analysis Security Testing (SAST). SAST tools are automated tools developed to test the code during the implementation phase [8]. SAST tools perform automated testing and analysis of program source code to identify security flaws in applications [8]. SASTs help programmers follow some guides in SDL to ensure the security of their code. There are many advantages to this type of testing. For example, it does not rely on run time environment. In addition, it is possible to plug SAST-tools directly into IDE to help developers to find security flaws without leaving the IDE environment. SAST-tools solve many problems. Some of these problems, according to Chris Romeo [8] are:

- SAST-tools help developers to detect security flaws in the early stages of development. These tools help developers avoid later security flaws detection and repair costs.

- Avoid repeating known mistakes.

- Customers will have an assurance that the product is secure.

- In addition to catching bugs and viewing the security flaws directly to developers will encourage them to perform additional analysis. As a result, developers will learn to produce secure code.
3 Method

This chapter will explain the scientific approaches to achieving the objectives. Moreover, this chapter discusses the reliability and validity of the project.

This research follows the Design Science method in order to reach the research results. The main goal of design science research is to develop knowledge that the discipline professionals in question can use to design solutions for their field problems. According to Hevner, the primary purpose of design science research is achieving knowledge and understanding of a problem domain by building and application of a designed artifact[24]. To apply the Design Science method, the project follows the design science research process (DSRP). DSRP is a model for producing and presenting information systems research[25]. This project will cover the following six steps in DS that are illustrated in the figure 3.6.

- **Problem identification and motivation:** the project will start with the identification of the problem. The project will review some security problems and highlight the most important of these problems. Practical problems will be identified at this stage. Then the existing models to solve these problems will be evaluated to establish their problems and why they do not perform as required.

- **Objectives of a solution:** the project will embark on solving the identified problem in the previous phases. This requires a solution to be developed. The solution to be developed must have objectives to be met to measure the success of the solution in solving the problem.

- **Design and development:** the proposed solution to the problem will be designed and developed in line with the set objectives. In the beginning, the project will design and develop a new algorithm that will be used to find the root cause of security violations in programs. Then the project will come up with a solution to simplify the result of Reax (the security guard) to be readable and usable by developers.

- **Demonstration:** the developed solution will be demonstrated to show how its efficacy. This will establish how the solution achieves the set objectives and how it solved the already identified problem. To achieve that, the project will integrate Reax and the new algorithm with Java IDEs by implementing a new tool (as an Eclipse plugin) to test the produced solutions.

- **Evaluation:** the project will check the results of the demonstration step to see how well the solutions solve the problem. In addition to evaluating the solution’s efficiency and effectiveness.
• **Communication:** at the end, the project will share information about the problem, the set objectives, design and development, analysis, results, discussion, and conclusion for this report. In addition to suggesting future work to be carried out on the project.

As a result, the project will propose and implement an algorithm to find the root cause of security violations. The algorithm will find the root cause of security violations by analyzing the generated SCFG that was illustrated in the background section 2.1. Moreover, the project will produce a new tool (as an Eclipse plugin) called SecureDevelopmentPlugin. The plugin SecureDevelopmentPlugin will be compatible with (Reax). The plugin will integrate Reax in the development environment. The plugin will use a third tool that is called SootCoodeAnalyzer. SootCoodeAnalyzer is responsible for preparing the input for Reax. The architecture for SecureDevelopmentPlugin is illustrated in the section 5.3.

The scientific method that is used to evaluate the results of this project is **Verification and Validation**. The verification is achieved by performing manual tests for all requirements. On the other hand, to performing the validation, the plugin was tested in its first version in the presence of the external company Omegapoint.

### 3.1 Reliability and Validity

As already mentioned, SCFG is produced by SootCodeAnalyzer. SootCodeAnalyzer are reliable. This means that each time the plugin is running for the same conditions, it will produce the same results. As a result, the plugin is reliable as well.

All terms in this report have been explained and clarified to avoid confusion or misunderstanding of this thesis and construct validity.

The performance of the proposed algorithms in this thesis relates to how these algorithms are implemented and the programming language that is used. For this project, all proposed algorithms are implemented in JAVA. The time complexity for the algorithm is illustrated in section 4.1.3).

The inputs for the engine are generated by Soot [26]. Soot is a tool to translate the Java bytecode into Jimple IR. As a result, the use of this project is limited to Java applications (.class, .jar, and .apk). This plugin is built essentially for Eclipse. However, it could be extended to any Java IDE, as it is illustrated in section 5.6.
4 Implementation

This chapter discusses the implementation of the proposed solution. The chapter is divided into two parts where each relates to the set objectives. The first section discusses the implementation of the first objective. The second section discusses the implementation of the second objective. The third objective will be implemented in the next chapter 5 to demonstrate the first and second objectives.

4.1 Objective 1: Develop a new algorithm to find the root cause of security violations

As already discussed, the existing algorithms do not show the root cause of security violations. The root cause can be established by using a graphical interface that allows the developers to identify security violations and fix them during the coding phases of an application. This project proposes and implements an algorithm to find the root cause of security violations. This section illustrates the algorithm’s inputs, rules, pseudocode, complexity, and performance. The algorithm tracks the security vulnerability from the location of the vulnerability (for instance, a leakage location) backward until the beginning of the method. The algorithm finds how sensitive information has been transferred from one parameter to the point of vulnerability. The result is displayed as a tree. This tree is called a dependencies tree. As a result, the first node in the dependencies tree is the location of the vulnerability. For instance, Figure 4.7 views an example for a dependencies tree that is readable like this; at the leakage-point, there is leakage for \( z \). \( z \) inherits the confidential-level for both \( x \) and \( y \) at the location \( z = x + y \). \( x \) inherits \( \text{parA} \) at the location \( x = \text{parA} \). \( y \) inherits \( \text{parA} \) at the location \( y = \text{parA} + 7 \). \( \text{parA} \) is \( \text{param0} \). I.e. \( \text{parA} \) is the first parameter of the checked-method.

The algorithm has the following inputs

- SCFG that was illustrated in the background section 2.1.
- Set of all possible paths in SCFG. The set of paths is produced as explained in Section 2.3 and Figure 2.5.
- The parameters’ initial confidential level for the method the user wants to check.

The algorithm has the following rules

- The algorithm supposes that each path could have loops with multiple branches.
- Each location in SCFG has a list of security updates.
- Each variable in the given method has a confidentiality level. This confidentiality level could be changed only according to the security updates in SCFG.
- At the execution time of a method, each variable at each transition in the method will have a specific confidentiality level. The confidentiality level is TRUE: T (confidential) or FALSE: F (not confidential). To make it easier to follow the explanation of the algorithm, the set of high confidentiality-level variables before each transition is called the input at that transition. Similarly, the set of high confidentiality-level variables after each transition is called the output at that transition.
- All transitions in the path are one of two types
Figure 4.7: the dependencies tree that leads to security violation (i.e., The way in which confidential information has been transferred from one parameter to the point of violation)

- **Normal transition**: that does not have a body (i.e single transition between two different locations)
- **Loop transition**: that has a body (i.e multi transition where first_location = last_location)

### 4.1.1 Algorithm’s Phases

In this section, the project will try to divide the detection of leakage’s root cause into multiple phases. The root cause analysis algorithm aims to define whether there is information leakage. In case of information leakage, the algorithm should define the leakage location and find the root cause of leakage (i.e., what information is leaked and why the leakage location has sensitive information).

The first phase in the algorithm is to define the candidate leak locations in the code and the security policy at each leak location. **SootCodeAnalyzer** will provide this information according to the given policy by the user. e.g., assuming the code have the statement `print(y)`, it is a candidate leakage location since it prints information. The security policy at that location will be something like `(y ≠ TRUE)`. I.e., `y` should not be or hold any confidential information.

The algorithm should define whether any security policies are broken or not. The algorithm should know the confidential level for all variables at the leakage location to define that. Therefore, the algorithm should perform a virtual execution that simulates the confidential-level changes for all variables. This could be achieved by executing all security updates in **SCFG** from the first to the last location in the function. From the
algorithm’s inputs and rules, the initial confidential levels of all parameters are predefined by the user in addition that the confidential level for all variables could be changed only according to the security updates in SCFG. Therefore, the algorithm will be able to calculate the actual confidentiality level for all variables at each transition. As a result, the confidentiality level for all variables will be known at all candidate leakage locations. Therefore, the algorithm will be able to check whether any of the security policies are broken or not. Consequently, the algorithm will be able to tell whether a leakage happens or not.

As already mentioned, if there is a leakage, the algorithm should define the root cause of leakage. Therefore, the algorithm should define why the variables at the leakage location have sensitive information. According to algorithm rules, the confidentiality level for any variables could be changed only according to the security updates in SCFG. That means the variables at the leakage location gain a high confidentiality level at one or many of the security updates at the previous transitions. For that, if the algorithm detected a leakage at the location \( n (\text{LOC}_n) \), the algorithm will trackback all security updates for leaked variables from that location. First, the algorithm will go back one transition up in SCFG, and check all security update at the previous location \( (\text{LOC}_{n-1}) \) to define the inherited sensitive variables at the location \( (\text{LOC}_{n-1}) \). Then the algorithm will trackback all involved sensitive variables at the location \( (\text{LOC}_{n-1}) \) by moving one transition up to the location \( (\text{LOC}_{n-2}) \) and check all security updates at that location. The algorithm will continue moving up one transition until it reaches the beginning of the function \( (\text{LOC}_0) \), where it finds the confidential parameters that are leaked.

From the above, the algorithm can be divided into four phases.

1. **Find Candidate Leakage Points.** This could be achieved according to the given policy by the user and generated policies by SootCodeAnalyzer.

2. **Analyze boolean changes.** That means the algorithm should define the high confidential level for each variable at each transition. The confidentiality level for each variable could be changed according to the security updates in SCFG. The initial levels at the beginning of the tested method are known. All confidential levels should be initiated as LOW except the method’s parameters that the user marked it as HIGH. Then the algorithm goes through the transition in the path sequentially. At each transition, the new confidential level for each variable will be determined according to the security update at that transition.

3. **Check if there is a leakage.** This could be achieved by comparing the confidentiality level for each variable with the policy. If any variable security label does not comply with the policy (i.e. if the variable has a high confidential level where the policy says it should be at a low level), that means there is leakage at this transition for this variable.

4. **Track back all leakages** from the leakage-point until the beginning of the method. That means if there is leakage for a variable at a specific transition (according to the previous step), the algorithm tries to find why this variable is at a high confidential level at that transition. This is achieved by going back one transition and checking if the variable inherits any other variable with a high confidentiality level. The algorithm continues to go back one transition until finding all variables were inherited. This approach is guaranteed since it is impossible that a confidential level is changed without being affected by a secure update at a specific transition (unless
this variable is a high confidential parameter from the beginning of the method). This way, the algorithm will be able to define at which transition a variable inherited sensitive information and what the inherited variables are.

To understand the algorithm, consider the simple method in Figure 4.8. The results of applying the four phases of the algorithm are as the following:

1. **Find Candidate Leakage Points.** A policy in this program is that all parameters of the method (print) will be exposed to observers. Therefore, location (2) is a leakage point, and \( y \) should not carry sensitive information.

2. **Analyze boolean changes.** The algorithm goes through all locations and defines the confidentiality of each variable. At location 0, the developer should define the confidential parameters. In our example, \( \text{parA} \) is confidential parameter. According to the security-updates, \( x \) inherits \( \text{parA} \) (i.e at location 1, \( x \) and \( \text{parA} \) will have confidential information). Similarly, according to the security-updates, \( y \) inherits \( x \) (i.e at location 2, \( x \), \( y \) and \( \text{parA} \) will have confidential information).

3. **Check if there is a leakage.** According to step 1, \( y \) should not carry confidential information at the location (2). Nevertheless, according to step 2, \( y \) carries confidential information. As a result, the algorithm decides that there is a leakage at location 2.

4. **Track back all leakages.** The algorithm will trackback the leakage from location (2) until the beginning of the method to define where \( y \) inherited confidential information. Therefore, the algorithm goes back one location and finds that \( y \) inherits \( x \). Then, the algorithm continues to go back until defining where \( x \) inherited confidential information. The algorithm will find that \( x \) inherits \( \text{parA} \). Finally, the algorithm finds that \( \text{parA} \) is the method’s first parameter. As a result, the dependencies tree for that method will be as in Figure 4.9.

\begin{verbatim}
public int test(int parA, int parB) {
  int x = parA;
  int y = x + 1;
  print(y);
  return y;
}
\end{verbatim}

Figure 4.8: SCFG and Security updates.
Figure 4.9: The dependencies tree that leads to security violation in the method in Figure 4.8

4.1.2 Algorithm’s Pseudo-code

This section discusses and analyzes the four phases of the root cause analysis algorithm to generate the corresponding pseudo-code.

The first phase in the algorithm is **Find Candidate Leakage Points**. As already mentioned, SootCodeAnalyzer provides the security policy at each leakage point. Therefore, the pseudo-code for this phase is straightforward and illustrated at pseudo-code phase 1.

<table>
<thead>
<tr>
<th>pseudocode phase 1: Find Leakage Points</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>foreach</strong> leakage location lk ∈ securityPolicies do</td>
</tr>
<tr>
<td>Mark the corresponding transition in the path as leakage point;</td>
</tr>
</tbody>
</table>

The second phase in the algorithm is **Analyze boolean changes**. In this phase, the algorithm will execute SCFG for each path virtually. As a start, the algorithm considers all variables in SCFG to have a low confidential level. Therefore, all variables will have the value **FALSE** except the function’s parameters and global variables that appeared in the security guard, and the user has defined them as sensitive variables. These variables will have a high confidential level. I.e., they will have the value **TRUE**. The user is not
required to define the confidential level for the parameters or the global variables that do not appear in the security guard because if they do not appear on the security guard, their confidential level does not affect the security of the function. After initiating all variables with their initial values, the algorithm will move to the first transition \((\text{tra}_1)\) in \(\text{SCFG}\) and process all security updates at that transition to calculate the new values for the involved variables at that transition. Then the algorithm will move to the next transition \((\text{tra}_2)\) and do the same. The algorithm will keep moving through transitions until it reaches the last transition in the path. The initial pseudo-code for this phase will be like the pseudo-code phase 2.a.

**Pseudocode phase 2.a: Initial pseudo-code for phase 2**

Initiate all Boolean variables as false;
Define the involved variables in the security guard;
Let the developer define the sensitive variables in the security guard;
Give all sensitive variables the value true;

\[\text{while the path has next transition do}\]

\[\text{tra} = \text{next transition in the path};\]
\[\text{tra}_\text{input} = \text{set of the high variables at the beginning of the transition};\]
\[\text{tra}_\text{output} = \text{tra}_\text{input};\]
Handel the security updates at the current transition;
Update the set \(\text{tra}_\text{output}\);

According to the algorithm’s rules, the algorithm will counter two types of transitions (**Normal transition** and **Loop transition**). It is straightforward how the algorithm will handle normal transitions. The algorithm will iterate through all security updates at normal transitions and add all variables that turned in a high confidential level to the transition output \(\text{tra}_\text{output}\). Vise versa, the algorithm will remove all variables that turned in a low confidential level from the transition output \(\text{tra}_\text{output}\). Pseudo-code phase 2.b illustrates an improved pseudo-code for phase 2 after adding how the algorithm will handle normal transitions.

**Pseudocode phase 2.b: initial pseudo-code for phase 2 with normal transitions handling**

Initiate all Boolean variables as false;
Define the involved variables in the security guard;
Let the developer define the sensitive variables in the security guard;
Give all sensitive variables the value true;

\[\text{while the path has next transition do}\]

\[\text{tra} = \text{next transition in the path};\]
\[\text{tra}_\text{input} = \text{set of the high variables at the beginning of the transition};\]
\[\text{tra}_\text{output} = \text{tra}_\text{input};\]

\[\text{if tra is single transition then}\]

\[\text{Security Update } \text{s}_\text{u} = \text{first Security Update in } \text{tra} =;\]
\[\text{while } \text{s}_\text{u} \neq \text{null do}\]

\[\text{Confidentiality Level } \text{conf} = \text{solve( Right Side Of } \text{s}_\text{u});\]
\[\text{if } \text{conf} \text{ is High then}\]
\[\text{add the variable at the left side Of } \text{s}_\text{u} \text{ to } \text{tra}_\text{output};\]
\[\text{else}\]
\[\text{remove the variable at the left side Of } \text{s}_\text{u} \text{ from } \text{tra}_\text{output};\]
\[\text{s}_\text{u} = \text{Next Security Update in } \text{tra} =.\]
The second type of transition the algorithm could encounter is **loop transition**. To understand how the given algorithm handles loops, consider a program with a set of variables $G$ and a loop with two branches. The input for the loop is a set $X \subseteq G$. The algorithm will generate the output for each branch in the loop separately. For instance, the output for the first branch is a set $Y$, and the output for the second branch is a set $Z$. In this case, the output $R$ for the loop will be the union of the three sets $X$, $Y$, and $Z$.

$$R = X \cup Y \cup Z$$

Then, the algorithm will compare the loop’s input and output (i.e., $X$ and $R$). If $X == R$, then the final result is $R$. Else, the algorithm repeats the loop with the input $R$. After many iterations, the system will reach a fixed point where $INPUT = OUTPUT$. The worst case is $INPUT = OUTPUT = G$.

The purpose of merging the results is to find the worst case. I.e., if a variable reaches a high state at the end of any iteration or branch, it will stay high in the last result. As a result, all variables that are candidates to be at a high level after this loop will be included in the final result $R$. To make it easier to understand, consider a method that has the loop in Code 1:

```
1 while (i < 10) {
2     if (z < i) {
3         a = b + 1;
4     } else {
5         b = c + 1;
6         d = a + 1;
7     }
8     i ++;
9 }
```

The loop has two branches. Figure 4.10 illustrates how the algorithm reaches a fixed point. In the given example, after four iteration $R = X = \{c, b, a, d\}$.

![Figure 4.10: Two branches in Code 1.](image)

Pseudo-code phase 2 illustrates the final pseudo-code for phase 2 after adding how the algorithm will handle loop transitions.
**pseudocode phase 2:** Final pseudo-code for phase 2 with loop transitions handling

Initiate all Boolean variables as false;
Define the involved variables in the security guard;
Let the developer define the sensitive variables in the security guard;
Give all sensitive variables the value true;

while the path has next transition do

    tra = next transition in the path;
    tra_input = set of the high variables at the beginning of the transition;
    tra_output = tra_input;

    if tra is single transition then
        Security Update s_u = first Security Update in tra =;
        while s_u ≠ null do
            Confidentiality Level conf = solve( Right Side Of s_u);
            if conf is High then
                add the variable at the left side Of s_u to tra_output;
            else
                remove the variable at the left side Of s_u from tra_output;
            s_u = Next Security Update in tra =.
    else if tra is a loop then
        loop_input = tra_input;
        loop_output = empty_set;
        while loop_input ≠ loop_output do
            Result_List = empty_list;
            foreach Branch bra ∈ loop do
                branch_output = execute analyzeBooleanChanges for bra
                with branch_input = loop_input;
                Add branch_output to Result_List;
            loop_output = merge loop_input and Result_List;
            tra_output = loop_output;

The third phase of the algorithm is to define whether there is any leakage or not. In phase 1, the algorithm defined the candidate leakage locations and the security policies at these locations. From phase 2, the algorithm knows the sensitive variables at each location. Therefore, the algorithm can define if the policy at any location is broken by assigning the values to variables in the policy and checking if the result is TRUE or FALSE. Pseudo-code phase 3 illustrates the Pseudo-code for phase 3.

**pseudocode phase 3:** Find Leakage Points

foreach Leakage location LOC_i do ;  // Marked at first step

    policy = policy at LOC_i;
    Boolean isbreaked = Check policy at LOC_i;
    if isbreaked then
        Transition Leakage status loc_leak_status = true;
        Leaked Variables Set loc_leaked_Variables = Variables that break the policy;

In phase 4, the algorithm will define the root cause of leakage for each leakage de-
ected in phase 3. The algorithm will find why the variables that break the policy in phase 3 are at a high confidential level at the leakage location. The algorithm will go back one location and check if the sensitive variable inherits any other sensitive variables. The algorithm continues to go back one location until finding all variables are inherited. This way, the algorithm will be able to define the locations where the variable inherited sensitive information and what the inherited variables are. Then recursively, for each sensitive inherited variable, the algorithm will trackback these variables until the algorithm reaches the beginning of the function and define the inherited sensitive parameter. Pseudo-code phase 4 illustrates the Pseudo-code for phase 4.

### Pseudocode phase 4: Track Back All Leakages

// First, define the recursive track function

```markdown
define track(var, loc):
    while there is a previous location before the location loc do
        loc = the previous location;
        Security Update s_u = security update ∈ loc where var is the left side;
        if s_u ≠ null then
            inherited_variables = all high variables in the right side of s_u;
            foreach Inherited Variable inh_var ∈ inherited_variables do
                track(var, loc);
        // Start tracking from the leakage locations
        foreach Location loc Where loc_leak_status = true do
            foreach Variable var ∈ loc_leaked_Variables do
                // Track the variable var from location loc.
                track(var, loc);
```

### 4.1.3 Algorithm’s Complexity

As described in the Pseudo-code, the algorithm traverses all method’s paths and solves the security updates in each path. Then, the algorithm generates a tree for each parameter leaked in each path. For each tree, the algorithm should traverse all security updates from the leakage point to the beginning of the method. In the worst case, all paths have leakage for all parameters, and the leakage points in all paths are at the end of the path (i.e., the algorithm should traverse all security updates in the path). Assuming the number of parameters is P, the number of paths in the method is T, the number of the security updates in the method is S, each path has nearly the maximum number of the security-updates (i.e., S). The number of trees in the worst case will be \( P * T \). Therefore, the algorithm will traverse the security updates \( (1 + P * T) \) times. I.e. the algorithm will visit in the worst case \( (1 + P * T) * S \) security-updates. As result, \( complexity = O((1 + P * T) * S) = O(P * T * S) \).
4.2 Objective 2: Simplify The Result Of Reax

Reax gives a complex security guard that makes it hard for developers to follow up and understand the conditions under which a method or an application is secure. One of thesis’s objectives is to simplify the security guard. To show the complexity of the generated security guard by Reax, consider a java method `Program4.test` with two parameters (param0 and param1), two paths (path1 and path2), and 6 locations (statements). The method follows path1 in case a condition CON1 is true. Otherwise, the method follows path2. path1 has the locations LOC1, LOC2, LOC4, LOC5, and LOC6. While path2 has the locations LOC1, LOC2, LOC3, and LOC6. Depending on the code, the security guard for the method before any simplification could be something similar to the code

2. The security-guard is complex and hard to read by the developers. And as much the method has statements, as much the complexity of the security guard.

The generated security guard by Reax is for the whole method. To simplify the security-guard, the plugin views the security-guard for each path in the method separately. As a result, all variable in the security-guard that has the form "LOC[number]" can be assigned a value (ether TRUE or FALSE). That is because the locations for each path are well known (i.e., if the location belongs to the path, the plugin replaces it with TRUE; else, it will be replaced with FALSE). Similarly, the plugin replaces the conditions in the security-guard with TRUE or FALSE depending on the path (i.e for path1, CON1 will be TRUE. While for path2, CON1 will be FALSE). The generated security guard is a set of boolean expressions. To simplify these expressions, the following rules are used [27].

- Literal removal
- Negation simplification
- AND/OR de-duplication/flattening
- Child expression simplification
- Propagating AND
- De Morgan’s law

As a result, the security-guard for the path1 will be `Program4_test = not param0`. While the security-guard for the path2 will be `Program4_test = TRUE`. That means, path1 is secure in case param0 is not a confidential variable. While path2 is always secure.

```
<table>
<thead>
<tr>
<th>Listing 2: Security-Guard without simplification</th>
</tr>
</thead>
<tbody>
<tr>
<td>1133 = ( not (LOC1 and LOC2) or not param0 ) ;</td>
</tr>
<tr>
<td>1134 = not (LOC4 and LOC5) ;</td>
</tr>
<tr>
<td>Program4_test = ( if CON1 then 1133 else 1134 ) ;</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>Listing 3: Security-Guard for path1 after simplification</th>
</tr>
</thead>
<tbody>
<tr>
<td>// Secure if param0 is not confidential.</td>
</tr>
<tr>
<td>Program4_test = not param0 ;</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>Listing 4: Security-Guard for path2 after simplification</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program4_test = TRUE ; // Always secure .</td>
</tr>
</tbody>
</table>
```
5 Demonstration (Design and development the plugin for Eclipse)

IBM started the development of Eclipse in 1990. IBM decided to produce a new development platform [28]. The extensibility is the most important property of Eclipse. The plugin system makes it easy to extend the functionality of Eclipse. Eclipse became an open-source project in 2001. Similarly, many of the Eclipse plugins are free and open source. However, there are some commercial tools.

5.1 Basic Architecture

To understand the basic architecture of Eclipse and where plugins can extend Eclipse, Figure 5.11 shows workbench objects and how they look. Figure 5.12 shows how the workbench owns views and editors. The workbench windows are the leading window where users can open more than one workbench. Each workbench could have many editors. The editor is associated with one workbench and works on a specific input. View belongs to the Workbench Window. However, they do not work on a specific input. Generally, the view-part displays information about the selected editor. Lastly, the perspective specifies the arrangement of Views. When Eclipse starts, the last arrangement is restored.

![Figure 5.11: Workbench objects and what they look like](image)

5.2 Eclipse Plugin Development

This section provides a brief introduction to connecting a plugin with Eclipse. Java Class Files, Extensions, and Extension Points should be defined to produce a plugin. The Java Class Files allow the implementation of plugin functionality. Eclipse has a class loader as a module. The class loader does not load plugin classes if the plugin is not used. The other item that the plugin should declare is the extension. The extensions should be well defined and at known points. The extension points are also needed for the plugin’s work and its functionality. A plugin can define any number of extension points. Each plugin contains
a manifest file (Manifest.mf) and an XML configuration file called (plugin.xml). The plugin declares the extension points in the manifest. In comparison, it declares extensions in the XML configuration file. The XML configuration file contains information about the plugin, such as the plugin’s name is required plugins.

Lazy Loading is a critical feature in Eclipse. The XML and Manifest files contain information about the extensions and the extension points. Such information and requirements are loaded on start-up. In contrast, the classes are only loaded when the plugin is used. For example, if the plugin adds an item to the context menu, the XML file contains all information, such as the name and the icon to add the item. The plugin classes will not be loaded until the user uses the plugin.

### 5.3 SecureDevelopmentPlugin Design

Figure 5.13 shows the plugin architecture. The interaction between the four layers will generate the result. Figure 5.14 Shows how layers interact with each other.

![Diagram of plugin architecture](image)

**Figure 5.12: Ownership of views and editors[29].**

The first development for the plugin is designed for Eclipse. The Eclipse Platform builds upon SWT. Therefore, the user interface of the plugin is built upon SWT. Considering that it is possible to add the SWT library to other Java IDEs, it will be easy to extend the plugin to be compatible with other platforms. The algorithms presented in this report are made independent of any other library.
5.4 Dependencies

5.4.1 Dependencies From The Given Engine (SootCodeAnalyzer & Reax)

This project depends heavily on SootCodeAnalyzer & Reax. Therefore, an essential part of this project is the analysis of their components and mechanisms. SootCodeAnalyzer is a Java project that prepares the input for Reax. SootCodeAnalyzer consists of a set of classes. Figure 5.15 shows the basic classes and the attributes that are used by the plug-in.

- **stsHelper.reaxResult**: has the security guard that is generated by Reax.
- **stsHelper.securityPolicies**: contains information about the locations where policy should be tested.
- **stsHelper.securityVariables**: contains all boolean variables that reflect the confidentiality level for the corresponding variable in the original code.
• **stsHelper.propagatedGuardMaps**: contains the guards that are propagated until the first previous checkpoint.

• **stsHelper.sts**: presents the generated security control follow graph.

• **stsHelper.unit2SrcLocation**: contains information about each unit and its context (i.e. to which loop or method it belongs).

• **monitor.sts**: presents the generated security control follow graph after merging the transitions that belong to one path and propagating guards up to the nearest checkpoint.

### 5.4.2 Other Dependencies

The plugin depends on the other three types of dependencies (Extensions, Extension Points, and external libraries).


**External libraries**: simpleGraph.jar, sootclasses-trunk-jar-with-dependencies.jar, log4j-1.2.17.jar, jbool_expressions-1.17.jar, antlr-runtime-3.5.2.jar, ucanaccess-4.0.4.jar, soot-Flowdroid-2.6.jar, graphviz-java-0.8.0.jar.

### 5.5 SecureDevelopmentPlugin Use-Cases

The implemented plugin has the following use-cases, which are also illustrated in the figure 5.16 below.

1. Run the plugin for a specific class.

2. Select a set of methods from the selected class to check the security.

3. Define whether the security analysis will be for Explicit-Confidential, Implicit-Confidential, or Integrity

4. Analyze the security and simplify the result (security guard) to be readable for all possible paths for the selected method

5. Find all violation points according to predefined confidential levels for the method’s parameters.

6. Define to which path each violation point belongs.

7. View the dependencies tree that leads to a security violation (i.e., How confidential information has been transferred from one parameter to the point of violation)

8. Allowing the user to apply the fix that the engine provides.
5.5.1 Implementation Of starting the plugin (Use-Case 1)

<table>
<thead>
<tr>
<th>Description</th>
<th>Run the plugin for a specific class.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pre-Conditions</td>
<td>Eclipse is running and the plugin is installed.</td>
</tr>
</tbody>
</table>
| Main flow               | • The user selects the class from the package explorer window and open the context menu.  
                          | • the context-menu is opened with the option "Check Security". Figure 5.17.  
                          | • the user clicks on "Check Security".  
                          | • SecureDevelopmentPlugin is started for the selected class. |
| Post-Condition          | the start window (option window) of the plugin is opened. Figure 5.18. |

On Eclipse, to add the item "Check Security" to the context menu in the package explorer, should add the following extension to the file plugin.xml. Code5.

Listing 5: To adding the context-menu in the package explorer

```xml
<extension
   point="org.eclipse.ui.menus">
   <menuContribution
      allPopups="false"
      locationURI="popup:org.eclipse.jdt.ui.PackageExplorer?endof=group.edit">
      <command
         commandId="MainActionHandler"
         icon="src/view/icons/secureIcon.gif"
         label="Check Security"
         style="push"
         tooltip="Check Security">
         <visibleWhen>
         <iterate operator="and" ifEmpty="false">
```
5.5.2 Implementation Of selecting methods (Use-Case 2)

<table>
<thead>
<tr>
<th><strong>Description</strong></th>
<th>Select a set of methods from the selected class to check the security.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Pre-Conditions</strong></td>
<td>Use-Case 1.</td>
</tr>
<tr>
<td><strong>Main flow</strong></td>
<td>• The user selects a method from the box &quot;All Methods&quot;.</td>
</tr>
<tr>
<td></td>
<td>• the user click the bottom &quot;→&quot; to add the method to the box</td>
</tr>
<tr>
<td></td>
<td>&quot;Methods To Check&quot;</td>
</tr>
<tr>
<td></td>
<td>• the method will be added to the box &quot;Methods To Check&quot;</td>
</tr>
<tr>
<td><strong>Post-Condition</strong></td>
<td>the methods that the user want to check are added to the box</td>
</tr>
<tr>
<td></td>
<td>&quot;Methods To Check&quot;. Figure 5.18.</td>
</tr>
</tbody>
</table>
5.5.3 Implementation Of setting the options of an analyzing (Use-Case 3)

<table>
<thead>
<tr>
<th>Description</th>
<th>Define whether the security analyzing will be for Explicit-Confidential, Implicit-Confidential, or Integrity.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pre-Conditions</td>
<td>Use-Case 1.</td>
</tr>
<tr>
<td>Main flow</td>
<td>• The user selects security check options in the start window.</td>
</tr>
<tr>
<td>Post-Condition</td>
<td>the desired-options are selected. Figure 5.18.</td>
</tr>
</tbody>
</table>

The configuration in this window will be transferred to the object "SynthesisConfigurations" in SootCodeAnalyzer. Code 6.

Listing 6: Transferring options to SootCodeAnalyzer

```java
if (confidentialityRadio.getSelection()) {
    exConfidentiality = explicitRadio.getSelection();
    imConfidentiality = implicitRadio.getSelection();
} else {
    integrity = integrityRadio.getSelection();
}
isNewModelRunFreeVersion = freeVersionCheck.getSelection();
Tool main = new Tool(exConfidentiality, imConfidentiality, integrity, isNewModelRunFreeVersion);
```
5.5.4 Implementation Of viewing and simplifying the security guard (Use-Case 4)

**Description**
Analyze the security and simplify the result (security guard) to be readable for all possible paths for the selected method.

**Pre-Conditions**
Use-Case 2.

**Main flow**

- In the start window, the user clicks the button "Start".

- The plugin opens the first result window (Security-Guard Result Window).

- The plugin views the simplified-result (the security guard for each path in the method).

**Post-Condition**
the security guard for each path is viewed. Figure 5.19.

Figure 5.19 illustrates the Security-Guard for the first path for the method test(...). The security guard for this path is "!param0 & !pc". param0 is the first parameter for the selected method. I.e., param0 is parA. Pc is the confidentiality level of the location where this method is invoked. Therefore, this path is safe if and only if the confidentiality level of parA is low and the confidentiality level of the location from which this method is invoked is low. Otherwise, this path is not secure.

The original security guard is complex and hard to read by the developers. For instance, consider the simplified security guard for the method that is shown in figure 5.19. The original security guard for the same method before any simplification is illustrated in Code 7. This security guard is for the whole method. To simplify the security guard, the plugin views the security guard for each path in the method separately. As a result, all the security guard variables with the form "LOC[number]" will be removed. That is because the locations for each path are well known (i.e., if the location belongs to the path, the plugin replaces it with TRUE; else, it will be replaced with FALSE). On the other hand, in

---

![Security-Guard Result Window](image-url)

Figure 5.19: Security-Guard Result Window. The Security-Guard for the first path for the method test(...) after simplification.
some cases, the security guard will be more complex since it contains some guards from
the checked method (i.e., Omega1, Omega2, etc.). By reproducing the security guards for
each path in the method, the plugin can define the guard’s value, whether it is TRUE or
FALSE.

Listing 7: Security-Guard without simplification

```
1133 = (not (LOC2 and LOC0) or not L1Ctes0param0);
1134 = not (LOC2 and LOC3 and LOC4 and LOC0);
Program4_test_0 = (if pc then 1134 else 1133);
```

To simplify the security-guard, Jbool library is used [27]. Code 8 illustrate how Jbool
simplify a boolean-equation that is given as a string.

Listing 8: simplify a boolean-equation by Jbool

```
String simplify(String securityGuardsAsOneString) {
    Expression<String> parsedExpression = ExprParser.parse(
        securityGuardsAsOneString);
    parsedExpression = RuleSet.toSop(parsedExpression);
    return parsedExpression.toString();
}
```

To solve boolean-equations. Code 9 illustrate how Jbool solve boolean-equations
according to given values for its variables.

Listing 9: Solving a boolean-equation by Jbool

```
boolean solveBooleanExp(String booleanExp, Map<String, Boolean> variables) {
    Expression<String> expression = ExprParser.parse(booleanExp);
    Expression<String> solved = RuleSet.assign(expression, variables);
    return solved.toString().equals("true") ? true : false;
}
```

5.5.5 Implementation Of finding violation points (Use-Case 5)

<table>
<thead>
<tr>
<th>Description</th>
<th>Find all violation points in a method according to predefined confidential levels for its parameters.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pre-Conditions</td>
<td>Use-Case 4.</td>
</tr>
</tbody>
</table>
| Main flow | • From Security-Guard Result Window, the user selects a method and chooses to perform an addition analyze. 5.20.  
• the plugin opens a new window (advanced analysis window).  
• the user select under which conditions wants to performs the additional analysis (i.e. initial confidentiality level for method’s parameters and pc). 5.22.  
• the plugin performs the analysis according to the selected conditions and view the results. |
| Post-Condition | Advanced-Analysis Window is opened and results is viewed. Figure 5.21.                      |
**Note:** the plugin gives the user the ability to select the confidential levels for the parameters that only appear in the security-guards since other parameters do not affect the result.

Figure 5.20: The user selects a method and chooses to perform an addition analyze

Figure 5.21: Advanced-Analysis Window

Figure 5.22: The user select under which conditions wants to perform the additional analysis
5.5.6 Implementation Of defining paths (Use-Case 6)

<table>
<thead>
<tr>
<th>Description</th>
<th>Define to which path each violation point belongs.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pre-Conditions</td>
<td>Use-Case 5.</td>
</tr>
<tr>
<td>Main flow</td>
<td>• The user navigates to &quot;path statements&quot; to view the path as a tree that reflects the source code. Figure 5.23.</td>
</tr>
<tr>
<td>Alternative flow</td>
<td>• The user chooses to highlight the source code that belongs to a specific path. Figure 5.24.</td>
</tr>
<tr>
<td>Post-Condition</td>
<td>The path is shown in the same window as a tree, or the source code is highlighted. Figure 5.23.</td>
</tr>
</tbody>
</table>

```
public int test(int parA, int parB, Program4 prog) {
    Path1: Omega1
    Security Guards
    ⊨UCles0(param0 & pc)
    Path Statements
    this := @this; Program4
    parA := @parameter0; int
    parB := @parameter1; int
    prog := @parameter2; Program4
    x := 0
    y := 0
    z := 0
    x := parA
    y := parA + 7
    z := x + y
    parB := 2
    x := x + 1
    ⊨virtualinvoke this.<@Program4: void print(int)>[2]
    return z
```

Figure 5.23: Statements Tree

To reflect the source code as it is illustrated in figure 5.23, the statements that belong to a sub-method or a loop should be grouped together. To achieve that, the plugin use the object `(SootCodeAnalyzer:stsHelper.unit2SrcLocation)` which contains the context for all transitions in SCFG.

To highlight the code in Eclipse, a marker should be created and declared by adding the text in Code 10 to the file plugin.xml. This will create a marker with the name "org.eclipse.viatra2.slicemarker". The plugin uses the created marker to highlight a specific line using the Code 11.
Figure 5.24: Highlight the source code for a specific path

Listing 10: Declare a marker for Eclipse

```xml
<extension
  point="org.eclipse.ui.editors.annotationTypes">
  <type
    markerType="org.eclipse.viatra2.sliceMarker"
    name="org.eclipse.viatra2.sliceMarker"/>
</extension>

<extension
  point="org.eclipse.ui.editors.markerAnnotationSpecification">
  <specification
    annotationType="org.eclipse.viatra2.sliceMarker"
    colorPreferenceKey="org.eclipse.viatra2.slice.color"
    colorPreferenceValue="192,255,192"
    contributesToHeader="false"
    highlightPreferenceKey="org.eclipse.viatra2.slice.highlight"
    highlightPreferenceValue="true"
    includeOnPreferencePage="true"
    label="GTASM Slice Marker"
    overviewRulerPreferenceKey="org.eclipse.viatra2.slice.overview"
    overviewRulerPreferenceValue="true"
    presentationLayer="0"
    textPreferenceKey="org.eclipse.viatra2.slice.text"
    textPreferenceValue="true"
    textStylePreferenceValue="BOX"
    verticalRulerPreferenceKey="org.eclipse.viatra2.slice.ruler"
    verticalRulerPreferenceValue="true">
  </specification>
</extension>

Listing 11: Use a marker to highlight a specific line

```java
public static void markLine(int lineNumber) {
  IWorkbenchPage page = PlatformUI.getWorkbench().
    getActiveWorkbenchWindow().getActivePage();
  IEditorPart editorPart = page.getActiveEditor();
  ITextEditor editor = (ITextEditor) editorPart;
```
5.5.7 Implementation Of creating the dependencies tree (Use-Case 7)

<table>
<thead>
<tr>
<th>Description</th>
<th>View the dependencies tree that leads to security violation (i.e., How confidential information has been transferred from one parameter to the point of violation)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pre-Conditions</td>
<td>Use-Case 5.</td>
</tr>
</tbody>
</table>
| Main flow | • The user selects a specific violation point.  
• The user chooses to view the dependencies tree.  
• The plugin views the dependencies tree. |
| Post-Condition | the dependencies tree for a specific violation-point is viewed. Figure 4.7. |

Performance of creating the dependencies tree

As section 4.1.3 shows, the algorithm visits security updates many times. Therefore, the visiting time for each security update should be reduced to improve the performance. There are two alternatives to handle security updates

- **JavaScript**: the plugin is implemented by Java. Java has an embedded JavaScript engine that can be initiated and executed to handle the security updates.

- **the external library Jbool** [27].

To define the best alternative, both JavaScript and Jbool were tested to handle a path with 100 update-securities and one thousand update-securities. The result is illustrated in the table 5.1

<table>
<thead>
<tr>
<th>Security Updates</th>
<th>JavaScript</th>
<th>Jbool</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>200 ms</td>
<td>60 ms</td>
</tr>
<tr>
<td>1000</td>
<td>1500 ms</td>
<td>220 ms</td>
</tr>
</tbody>
</table>

Table 5.1: JavaScript VS Jbool
5.5.8 Implementation Of applying a fix (Use-Case 8)

The fixing and the fixing process are a part of SootCodeAnalyzer, which is out of this project. This project is just preparing the input for the fixing process. Therefore, there is no verification for the result of the fixing process. The verification is only to ensure that the input for the fixing process is correct and that the fixing process can start usually.

<table>
<thead>
<tr>
<th>Description</th>
<th>Allowing the user to apply the fix that SootCodeAnalyzer provide.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pre-Conditions</td>
<td>Eclipse is running and the plugin is installed.</td>
</tr>
<tr>
<td>Main flow</td>
<td></td>
</tr>
<tr>
<td>• The user select a project from package explorer window and open the context-menu.</td>
<td></td>
</tr>
<tr>
<td>• the context-menu is opened with the option &quot;Fix Security&quot;. Figure 5.25.</td>
<td></td>
</tr>
<tr>
<td>• the user clicks on &quot;Fix Security&quot;.</td>
<td></td>
</tr>
<tr>
<td>• SecureDevelopmentPlugin is started for the selected project.</td>
<td></td>
</tr>
<tr>
<td>• An window to select the entry point is opened &quot;fixing-window&quot; 5.26.</td>
<td></td>
</tr>
<tr>
<td>• The user define the entry point clicks &quot;Start Fixing&quot;.</td>
<td></td>
</tr>
<tr>
<td>• SootCodeAnalyzer start apply the fixing.</td>
<td></td>
</tr>
<tr>
<td>Post-Condition</td>
<td>SootCodeAnalyzer applied the fixing.</td>
</tr>
</tbody>
</table>

Figure 5.25: Context Menue For A Project.

All project classes are located in a list to the left of the fixing window. When the developer selects a class, all methods in that class will be displayed in a drop-list on the right of the window. The developer should select one method as an entry point. In the fixing window, the user should define the entry point manually. Here are some suggestions for automating this process:

• Search for the method "public static void main(String[] args)". The problem with this approach
  – Some Java projects do not have such a method. For instance, libraries that will be used in other applications.
Sometimes, developers create the main method in many classes for test purposes.

- Generate the call graph for the project (i.e., a graph that illustrates all project’s methods with an edge from each method to all its sub-methods). Then the entry point is the method without any inner edge. The problem with this approach
  - it could be time-consuming for big projects.
  - Some Java projects do not have an entry point. For instance, libraries that will be used in other applications.

- The developers mark the entry points with a specific annotation. The problem with this approach
  - Still needs user input.

To check a method, the plugin invokes SootCoodAnalyzer to generate Reax’s input. Then Reax performs a manipulation and produces an output. The process needs considerable time. Therefore, a database is created to avoid re-running the entire process for a method if no change has been made to the method since the last manipulation.

To keep the installation and the using of the plugin simple, it is decided to save the database locally as access file (i.e., file of the type "*.mdb"). The database has only one table with the following columns.

- **MethodName** (Short Text : 255 char): that should be unique. The best choice is to use the entire declaration for the method (i.e., full package name, class name, method name, return type, and parameters).

- **MethodHash** (Short Text: 255 char): according to this field, the plugin decides whether the method has changed or not. The approach for generating Method-Hash
could be changed simply by modifying the method
model.ClassAnalyzer.MethodWithBody.generateHashSignature() in the plugin.

- **CheckType** (Short Text: 255 char): to save for which options this method was checked (i.e. Explicit-Confidential, Implicit-Confidential, Integrity ...etc).

- **SecureState** (Long Text: \(2^{30} - 1\) bytes): is the complete result of the analysis which is done by the plugin. The result is saved in binary form.

To implement this function, two external libraries are used.

- **jackcess-2.1.2.jar**: to initiate an empty access file.

- **ucanaccess-4.0.4.jar**: depends on Jackcess and HSQLDB. This driver allows the plugin to read/write Microsoft Access databases.

Figure 5.27 illustrate the interaction between components in case database is involved.

![Figure 5.27: Interacts between components in case database is involved.](image)

**5.6 Possible Extensions**

This project is finished on top of Eclipse. It is extended to run on top of IntelliJ. IntelliJ is also an open-source project developed by JetBrains. The plugin can be extended to run on top of any Java IDE. The interfaces of the plugin depend heavily on SWT. Therefore, the view does not need many changes to be compatible with any Java IDE. The significant change is in two classes responsible for the interaction between the plugin and IDE. These two classes are

- **model.SelectedItem**: this class is responsible for getting information from Eclipse about the selected project or class. For instance, project path, package name, compile target, file path, dependencies libraries, etc.

- **model.classAnalyzer.classMoifier**: this class is responsible for sending information from the plugin to Eclipse. For instance, select the source code, highlight the source code, add an annotation to a method, etc.

It is enough to produce a new version of these two classes to make the plugin compatible with any other IDE.
6  Results And Analysis

This chapter collects the results of the demonstration of the implemented algorithm in the previous chapter. The results are then analyzed to establish how well they address the research objectives.

Figure 6.28 shows a method with if-statement. As it is shown, that the highlighted branch has a leakage for both parameters cardNumber and cvc. The dependences tree shows that there is a leakage for the variable \( z \) at the transition print\( (z) \). In addition to showing how \( z \) has inherited the confidential level from cardNumber and cvc. This information is very valuable for developers to solve security flaws. To fix this flaw, developer has to break the flow from \( z \) to both cardNumber and cvc.

![Leakage Diagram](image)

Figure 6.28: Method with an if statement. cardNumber and cvc are highly confidential.

Figure 6.29 shows a method with a loop. As it is shown, the highlighted branch has a leakage for cvc. The dependences tree shows that there is a leakage for the variable \( z \) at the transition print\( (z) \). In addition to showing how \( z \) has inherited the confidential level from cvc. To fix this flaw, the developer has to break the flow from \( z \) to cvc. By taking a look at the while-loop, \( z \) does not inherit confidential information since resb is still at a low confidential level. But in the second iteration, resb has confidential information about cvc. Therefore, \( z \) also inherited confidential information about cvc. It is clear that the plugin is able to handle the loop efficiently.

Figure 6.30 shows a method with a loop. It is nearly similar to the previous case in figure 6.29. However, it has a recursive dependency inside the loop. Therefore, any change for the confidential level for \( z \) or resb inside the loop will affect the confidential level for the other. This result shows that the plugin can handle recursive dependencies efficiently.

Figure 6.31 shows a method with if-statement and more complex inheritance. The result shows that there is leakage for just parA despite of \( z \) inherts information about parB because of the condition inside if-statement (i.e. because of parB < 13). Anyway,
Figure 6.29: Method with a loop. cvc is highly confidential.

partB does not have confidential information because of the statement (parB = 2). The plugin was able to detect that partB will not confidential and there will be no leakage for confidential information by partB.

Figure 6.32 shows how the previous result in figure 6.31 will be affected by removing the statement (parB = 2). In this case, the leakage will be for both parA and parB.
Figure 6.30: Method with a loop. cvc is highly confidential. Note the recursive dependencies.

Figure 6.31: Method with if statement. ParA and ParB are highly confidential.

7 Evaluation And Discussion

This section discusses and evaluates the work that is done in this project regarding the three objectives.
7.1 Objective 1: Develop a New Algorithm to Find The Root Cause of Security Violations

A graphical algorithm was implemented with a path showing how the result is achieved. The graphs take the results of SootCodeAnalyzer and reax, and plot a graph that shows the flow of the steps. The graph may have one or more paths. Each path starts from the point of leakage, which is indicated by the broken security policy. Then it moves upwards to the parameters in the method header. For each path in the graph, a leakage path is drawn. The graph allows the developer to trace the source of information leakage easily. The code is divided into branches, as shown in figure 6.31 with a graph that indicates the status of the data carried. Some branches are secure, while others leak sensitive information. The graphs show which parameters should not carry sensitive information; otherwise, it will be leaked. As a result, the developers can easily know the path to leakages if left unattended. Then developers can identify and correct any leakage of sensitive information and improve the code that leads to the leakage and secure sensitive information. Compared to other algorithms and tools, most of the existing control information tools do not use graphs to identify the root causes of a security violation. The graphical representation gives a clear view of the issues at hand. They summarize and display the result to facilitate analysis of the data leakage. The graphs illustrate the relationship between the leakage point and the root cause of leakage. Visual graphs offer clues allowing. Therefore, it is expected that developers will get a big benefit from this project to produce higher security software. This will reduce losses caused by security flaws and reduce the cost of repairing those defects.

The root cause analysis is achieved by logically analyzing the SCFG diagram and solving all its boolean equations. Solving these equations and expressions is performed using known and proven logical rules to calculate the values of these expressions. Then the algorithm tracks the changes in these values. This makes the solution reliable. Therefore, every time we run the algorithm, it gives the same result.

7.2 Objective 2: Simplify the Result of Reax

The project analyzes the result of Reax. Reax gives developers a complex security guard. The complex security guard of Reax makes it an un-preferred tool amongst various de-
velopers who opt to use the complex result to know if their code is secure. The proposed model ensures the developers can easily understand the result to determine whether the code is secure or has a security issue. Whereas in Reax, developers cannot easily read the result since it is a security guard for the whole code, this project takes each path in the code separately and assign the known values in that path to the variables in the security guard. Then it is simplified using proven laws and rules of Boolean expressions. As a result, the security guard will not contain any confusing variables other than the function’s parameters. In this case, the security guard will tell developers what the parameter will be leaked and should not be a confidential parameter. A simplified result means implementing the secure code, and maintenance will be easier to achieve.

7.3 Objective 3: Integrating Reax and The Algorithm in Development Environment

To demonstrate the proposed solution, a graphical plugin was implemented. The plugin is created, implemented, and tested in Eclipse. Eclipse is a powerful IDE for Java developers. Eclipse sends the project’s information to the plugin when a developer chooses to check a method. The plugin sends this information to the engine. Then the engine analyzes information and sends back the result. Then, the plugin does additional analysis and displays the results in a graphical interface. The plugin views the result of Reax. In addition, it performs advanced analysis on SCFG to produce an advanced result that gives developers beneficial information to detect security flaws and avoid them. To analyze a complex code, the plugin should be improved to be compatible with a more advanced version of Reax that is still under development during the work on this project.

This program was developed primarily to work with Eclipse. In addition to the ability to extend it to work with all Java IDEs by a bit of modification. To achieve that, The graphical plugin depends on SWT environment for interfaces. In addition to isolating functions that send/receive information to/from IDEs in two classes, as is illustrated in section 5.6. To ensure that this plugin could be extended to other IDEs, the plugin is improved to work with IntelliJ.

The plugin and the engine can manipulate the files (.class, .apk, .java). To achieve that, Soot is used to translate the Java bytecode into Jimple IR. Soot could be extended with the paths for third libraries to ensure a successful loading for the project.

The proposed algorithm that produces dependences trees needs to solve many boolean equations (thousands in some cases). To guarantee good performance, four points are considered

- A new structure for boolean equations was implemented.

- A comparison was performed between JavaScript and Jbool. The results of the comparison are illustrated in table 5.1.

- The performance in this project depends on the component’s performance that is responsible for representing and handling graphs. For this purpose, a special structure for the graph component was implemented where the complexity of most functions is $O(1)$. The code in Appendix A

The produced plugin has many advantages. Developers will be able to perform SAST using Reax without leaving the development environment. This will increase the possibility of detecting security flaws in the early stages of development. This helps developers avoid the high costs of later security flaws detection and repair. Moreover, catching bugs
and viewing the security flaws and the root cause to developers will increase their knowledge to produce secure code.

The disadvantage is that developers still need to know the confidential level of the method’s parameters to generate the dependency tree and the root cause for security flaws. This information is not always available to developers. Another disadvantage is that this plugin is limited to Java applications.

Omegapoint is involved in the validation process. Omegapoint is an expert in Secure Digital Transformation and enables secure business for its customers. The plugin was tested in its first version in the presence of Omegapoint. They were interested in the project. They agreed that it is promising work and seemed interested in testing the project practically when it is ready for use in production.

Computer security will always be a work in progress, making this project a small step forward in software security in uncovering security flaws through a graphical plugin.
8 Conclusion and Future Work

In this thesis, we present and discuss the project results that investigate how to evaluate and improve software system security. One of the main contributions of our work is to develop a new algorithm to find security violations in applications and the root cause of these violations. The goal was to produce more resilient security controls for developers.

A discussion on different related works has been provided. One of the related works is Reax. Reax is built on a proven approach to detecting vulnerabilities. Reax generates security guards to monitor code using symbolic control flow graphs. The problem is that Reax does not provide enough information about the root cause of security violations. Therefore, we decided to develop a new algorithm that takes Reax’s output as an input. The algorithm can find security violations and the root cause of these violations.

As a second contribution, a graphical tool was developed in this project to verify the developed algorithm. The tool performs advanced security analysis that detects and reacts directly to security flaws. To address the security needs of developers, a boolean supervisory controller synthesis was developed to observe Java programs at checkpoints. At the same time, a command-line application (Reax) was integrated with the tool and visual aid. As a result, a graphical interface tool was developed to define constraints and the root cause that leads to security flaws.

This project and the developed tool should help developers define the vulnerabilities and security flaws that the engine (Reax) detects. In addition to locating the root cause that leads to vulnerabilities by the proposed algorithm in this project. This should help to produce applications that avoid unintentional data leakage.

Security flaw detection mechanisms are still unable to detect all software code flaws. This project is an additional step in improving this situation. The analysis of the control flow graph offers a way to identify how sensitive information could transfer from one variable or object to others. The control flow graph could be generated successfully for .jar, .apk, and .class files. This makes this approach widely applicable. Researchers and developers can take advantage of this approach by generalizing it to other programs, such as .exe programs, and improving this approach to produce a more robust and accurate analysis software capable of detecting a more comprehensive range of defects.

The new algorithm and the developed tool are tested with simple code. Many different adaptations, tests, and experiments have been left for the future due to a lack of time (the experiments with actual code need a stable and advanced version of Reax under development). Future work concerns a deeper analysis of real applications and code and maybe new proposals to make the algorithm compatible with newer versions of Reax.

This program targets software companies and developers interested in developing secure software. Many aspects can be developed to make the program more beneficial for programmers and developers. A more stable and advanced version of Reax is under development to make the tool more useful for programmers and developers. The tool will help to analyze complex code and programs. The tool should be improved to be compatible with the latest versions of Reax. Also, new tools such as Advanced UI to make graphs and dependencies tree interactive. In the future, there also needs to be improvements and advancements in the documentation mechanism that makes the tool reliable for companies to document results. Moreover, The tool currently works on Java. More work is needed to establish how the tool can use with other programming environments such as .NET or JavaScript. Finally, there is a need to Automate or semi-automate the determination of the entry point of Java applications.
References


A Appendix 1

Listing 12: Special structure for the graph component where the complexity of most functions is $O(1)$

```java
public class SpecialGraph {
    public int size = 0;
    public boolean isDirected = true;

    private Map<String, HashSet<MyNode>> labelToNodeSet = new HashMap<String, HashSet<MyNode>>();
    private Map<Integer, MyNode> numberToNode = new HashMap<Integer, MyNode>();
    private Map<String, MyNode> nameToNode = new HashMap<String, MyNode>();
    public Map<Integer, HashSet<Integer>> verticesAndEdges = new HashMap<Integer, HashSet<Integer>>();
    private Map<Integer, HashSet<Transition>> verticesAndOutTransitions = new HashMap<Integer, HashSet<Transition>>();
    private Set<Transition> transitionsSet = new HashSet<Transition>();

    public void addTransition(Transition tra) {
        transitionsSet.add(tra);
        addNode(tra.startNode);
        addNode(tra.endNode);
        addEdge(tra.startNode.number, tra.endNode.number);
        verticesAndOutTransitions.get(tra.startNode.number).add(tra);
        verticesAndInTransitions.get(tra.endNode.number).add(tra);
    }

    public void addNode(MyNode node) {
        if (nameToNode.get(node.name) != null)
            return;

        if (node.number < 0)
            node.number = nodeId++;

        if (addVertex(node.number)) {
            if (labelToNodeSet.get(node.label) == null) {
                labelToNodeSet.put(node.label, new HashSet<MyNode>());
            }
            labelToNodeSet.get(node.label).add(node);
            numberToNode.put(node.number, node);
            nameToNode.put(node.name, node);
        }
    }

    public boolean addVertex(int vertex) {
        if (!verticesAndEdges.containsKey(vertex)) {
            size++;
            verticesAndEdges.put(vertex, new HashSet<Integer>());
            return true;
        }
        return false;
    }

    public void addEdge(int sourceVertex, int targetVertex) {
```
verticesAndEdges.get(sourceVertex).add(targetVertex);

if (!isDirected) {
    verticesAndEdges.get(targetVertex).add(sourceVertex);
}

public Set<Transition> getTransitionsSet() {
    return transitionsSet;
}

public Set<Transition> getOutTransitions(int v) {
    return verticesAndOutTransitions.get(v);
}

public Set<Transition> getInTransitions(int v) {
    return verticesAndInTransitions.get(v);
}

public MyNode getNodeByName(String nodeName) {
    return nameToNode.get(nodeName);
}

public MyNode getNodeById(int nodeId) {
    return numberToNode.get(nodeId);
}

public Set<Integer> getAllIdNodesSet() {
    return numberToNode.keySet();
}

public int size() {
    return getAllIdNodesSet().size();
}

public Transition getTransition(int source, int target) {
    Set<Transition> OutTransitions = this.getOutTransitions(source);
    for (Transition tra : OutTransitions) {
        if (tra.getTarget() == target)
            return tra;
    }
    return null;
}